The Oberon Companion

A Guide to Using and Programming Oberon System 3

André Fischer and Hannes Marais

This book was written and typeset by the authors using Oberon System 3. Errors and corrections can be reported to the authors at

Institute for Computer Systems
ETH Zentrum
CH–8092 Zürich
Switzerland

or sent by e–mail to fischer@inf.ethz.ch.

Acknowledgements

Oberon System 3 with Gadgets is the product of exhilarating and, at the same time, hard work by many people. We would like to address our first and heartiest thanks to Prof. N. Wirth and Prof. J. Gutknecht who designed and implemented the original Oberon system. What started as an elegant programming language Oberon and a minimal operating system has now become a grown–up, mature system with powerful object–oriented capabilities, also providing a component framework ideally suited for further extensions and an imposant number of ready–made extensions, in other words "applications". Note in passing, that this book was written and typeset by the authors using Oberon System 3.

Our next warmest thanks go to several generations of assistants who have achieved an outstanding work, most of the time under extreme time pressure and always in search of the clearest and most adequate concepts. From the outside it may not be quite recognizable how these smart guys are constantly torn between diametrically opposed attitudes: that of an engineer preparing an original, novative dissertation and that of a software implementor writing code to be tested and mended, release after release for several platforms as a daily down–to–earth activity. This book is no more than just a reflection, a spin–off from a long haul teamwork by Karl Rege, Ralph Sommerer, Josef Templ, Regis Crelier, Markus Dätwyler, Marc Sperisen, Michael Franz, Thomas Kistler, Andreas Disteli, who have left the team for new horizons by now, and Pieter Muller, Erich Oswald, Patrik Reali, Patrick Saladin, Emil Zeller, who are forming the new core responsible for the production of the latest releases including the latest, in February 1997, which this book documents. We must emphazise that all of them have always offered their competent advices spontaneously, for which we are very grateful.

The section dedicated to LayLa is due to the pen of Jörg Derungs who offered it ready on–a–plate in his diplom work.

The "commented" source code which comes together with the system is like a cherry on the cake. With it, there might be little incentive to write a book entitled "Undocumented System 3", but it is quite conceivable that several users, in the search of a problem solution in their respective disciplines, might publish articles on unexploited capabilities of Oberon System 3.

Finally, it is a pleasure to acknowledge the great help we have received from Günther Sawitzki who provided much useful criticisms and from Dominique Marais, Frederic Rentsch and Roland Vögeli who checked the earlier drafts.

Chapter One

Introduction and Design Principles

1.1 Introduction

Oberon is simultaneously the name of a programming language and of a modern operating system. The Oberon project [WG92] was started at the Swiss Federal Institute for Technology (ETH) in 1985 by Niklaus Wirth and Jürg Gutknecht. In addition to the software, hardware in the form of a general–purpose computer called *Ceres* [Ebe87] (based on the National Semiconductors 32000 processor family) was built to run the new operating system. After a period of internal ETH use for education, the decision was made to document the language and the operating system in a series of books, and to port the Oberon system to popular computer hardware where it would run natively or on top of the native operating system of the host. Today, the original Oberon system is available freely for many computer architectures.

In 1991, Jürg Gutknecht continued the development of the operating system in a newly formed Oberon System 3 group [Gut94, Mar94]. The goal was to exploit the inherent features of Oberon to a much larger degree, upgrade the system by a concept of persistent objects, modernize the user interface and provide support for the ubiquitous network. In 1995, the Oberon System 3 Release 2.0 was finished. Concurrently with the development, the system was documented with a new set of hypertext–based tools. Since then, the system has been constantly improved and extended.

This guide forms part of the documentation effort. It is addressed to users of the system and to programmers with Oberon language experience. The guide covers the current state of the project and is divided into user and programmer guides.

1.2 Design Principles

The underlying dynamic model of Oberon is extremely simple. There exists a single process acting as a common carrier of multiple tasks. This process repetitively interprets *commands* which are the entities of execution in Oberon. Commands are atomic actions operating on the global state of the system. Unlike customary interactive programs, they rigorously avoid direct dialogs with the system user; in other words, the system is completely *non–modal*. The following examples indicate the bandwidth covered by the concept of command: placing the caret, inserting a character into a text, selecting a piece of text or a visual object, deleting a selected piece of text, changing the font of a piece of text, compiling a software module, opening a document, backing up a sequence of files to diskette, displaying a directory, running a simulation or some other application. We emphasize that the execution of a command always results in non–volatile information. For example, a displayed directory is a text that might immediately undergo further processing. Typically, commands report the outcome of their execution in the form of an entry in the *system log*. Therefore, the log provides a protocol of the current session.

Commands are initiated by input actions. Apart from a few universal operations, every input action is connected with a displayed *visual object* to which its further handling is delegated. A visual object in Oberon has a rectangular area that can display any kind of data. Most visual objects feature a thin frame often used for manipulating it. Any mouse–oriented input is handled by the visual object the mouse points to. Data from the keyboard is passed over to the current so–called focus object. An important feature of visual objects is that they are first class citizens, which means that they are deployed wherever required and are not bound to specific applications. As a practical example, we can insert a visual object like a line from a graphic editor into a text document, or vice–versa. Furthermore, we notice that command interpretation is a highly decentralized activity in Oberon and, as such, is a substantial contribution to what we consider as Oberon's most important quality, namely unlimited extensibility.

Implementing a new object type is a very powerful but also quite far–reaching method to extend the Oberon system. A more modest way to increase the system's functionality consists of adding new commands operating on objects of an already existing class. A more ambitious extension could be the construction of a language compiler operating on text for example. We shall see that Oberon's open and coherent modular architecture provides effective support for that. Practically all system ingredients and resources are directly accessible and usable via modular interfaces on as high a level of abstraction as possible. This makes Oberon ideally suitable as a rapid development environment. Commands effectively replace conventional applications which have to be started. As commands operate on the shared system state and can be activated directly when required, it is simple to extend the system with new special–purpose commands. The programmer's guide will provide more insight into this topic.

In addition to commands and visual objects, Oberon also supports *non–visual objects* for storing data, and *documents* for making collections of objects persistent. In fact, one of Oberon's biggest strengths is the large collection of prefabricated *persistent objects*, or *components* as we also refer to them. As the user can customize the system by combining objects together interactively at run–time, we also call our components *end–user objects*. End–user objects are maintained and managed in Oberon by the *Gadgets* framework and toolkit.

We should deduce from the foregoing that there is no symbolic wall in Oberon separating actual users from developers. Users are encouraged to customize the system and tailor it to their individual needs either by modifying the tools and graphical application interfaces delivered with the system or by designing and implementing private commands and facilities. Little is "hardwired" in the system. However, there are several general conventions and existing tools, which are presented in the following user guides.

Historically *text* as input and output medium plays a very important role in the Oberon system. The following chapter covers the *textual user interface* of the Oberon system, whereas the chapter titled the Gadgets User Interface gives more insight into its *graphical user interface*. Notice however that, from a technical point of view, text and text documents are just special cases of non–visual and visual objects respectively.

1.3 A tour through the chapters

Chapter 2 describes the textual user interface. It introduces the notion of command –– a unit of operation in Oberon. The use of the text editor and of the compiler is explained. Together, they form the principal tool for developing new software. Several software development tools and utility programs are described in the rest of this chapter.

Chapter 3 describes the Gadgets user interface. In the Gadgets system, objects called "gadgets" are divided into visual gadgets and non–visual gadgets. In many cases, visual gadgets have the duty of visualizing the non–visual gadgets or models to which they are linked. The central topic is the interactive composition of gadgets. The next topic is devoted to Columbus, an indispensable GUI tool for inspecting and manipulating gadgets. Watson is an example of a composed gadget often needed for inspecting the definitions of modules installed on the system. A presentation of persistent objects and of libraries complete this chapter.

Chapter 4 is devoted to the description of the gadgets delivered with the system.

Chapter 5 explains how to program in Oberon. It starts with a description of the module hierarchy providing a sound basis for the construction of further system extensions. A detailed study of the text manipulation mechanisms follows. The next topic explains Oberon's display space structure and the hierarchy of the object types. The chapter continues with the study of the gadgets manipulation mechanism controlled by a special message protocol. An overview of a variety of design patterns which can be used in the construction of new gadgets of different types concludes the chapter.

Chapter 6 introduces the reader to some of the applications included in the system. One of them enables the construction of graphical user interfaces on the basis of a textual description written in the LayLa layout language. The TextPopups application facilitates the work of Oberon program developers, helping them to locate quickly commands, document names, procedure names and type names in source program texts.

1.4 System implementations

This system, together with its source code (under a general license agreement), is available as freeware. The current release which this book documents is called Oberon System 3 Release 2.2. It can be installed on various platforms, either on top of the operating system (Windows, Linux for Intel–based PC, MacOS for Macintosh) of the host machine or, in the case of Intel–based PCs, as native system. The complete material for all these platforms is available on the CD–ROM included in this book. The platform–specific hardware and software requirements together with installation instructions are described in *readme files*. Details on how to obtain the latest update for a specific platform are given in the Appendix C.

PC Native Oberon is absolutely self–contained and makes no use of any alien software layer on Intel–based PCs. In a way, PC Native Oberon is to the PC what the original Oberon [WG92] was to the Ceres. For instance, it uses the same flat file directory structure found in the original Oberon. The three other implementations use the directory structure of their host operating system and the concept of a path for accessing an external storage medium.

The diversity in the hardware has forced the implementors to adapt a number of system features and functions to the host hardware characteristics. The mouse type and the keyboard layout are of primary concern for the user controlling the system. The Oberon system is fine–tuned toward a 3–key mouse where a large number of functions can be activated by single mouse key clicks and *interclick* combinations. Accordingly, the system description is based on the assumption that a 3–key mouse is used. Not only have most Macintoshes a single button mouse, but many Intel–based PCs have a two–button mouse, so that the missing mouse keys have to be emulated by keyboard keys. The mapping of the mouse keys is depicted in Figure 1.1. In order to abstract the physical differences, the three mouse keys have been named ML, MM and MR.

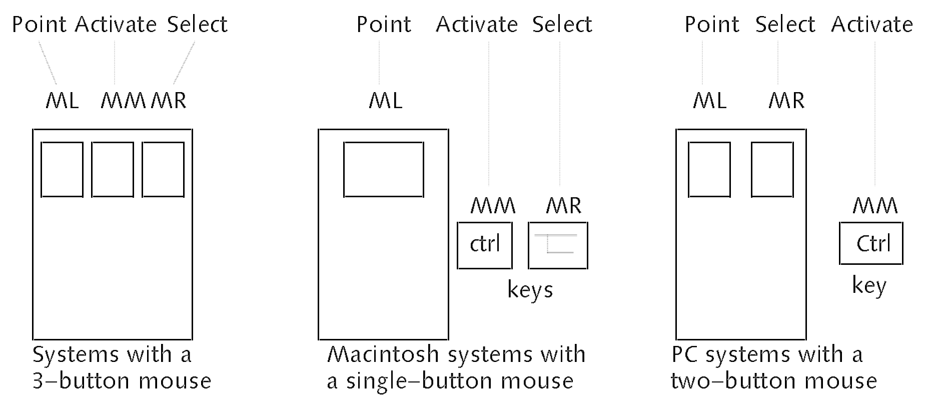

Point Activate Select
ML MM MR
Systems with a 3–button mouse
Point Activate Select
ML
MM MR
ctrl
keys
Macintosh systems with a single–button mouse
Point Select Activate
ML MR
MM
Ctrl
key
PC systems with a two–button mouse

Figure 1.1 Mouse varieties

If you are using a PC with a two–button mouse, use the **Ctrl** key at the left of the keyboard as a substitute for the missing MM key. If you are using a Macintosh with a single button mouse, use the left control key as the MM key and the alt key as the MR key.

Interclicking means clicking (pressing and releasing) a second mouse key at an arbitrary time while the first key is being held down. In general, interclicking is an efficient and versatile tool to multiply the expressiveness of the mouse. In Oberon, interclicking is applied according to a systematic underlying pattern. You will find out more about this pattern soon.

Chapter Two

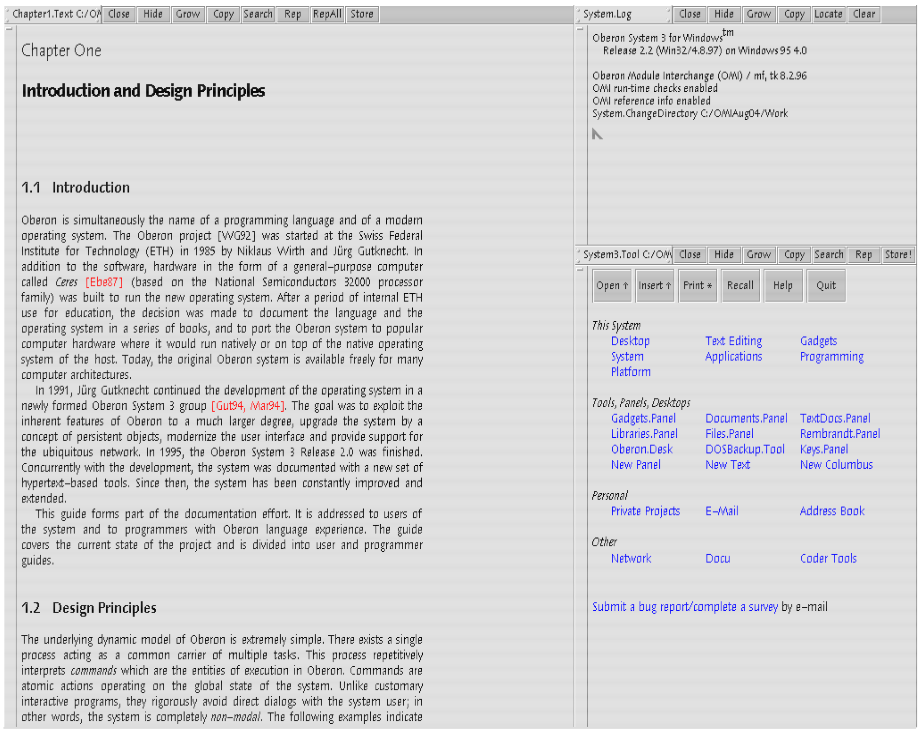
The Basic System

2.1 Introducing the Oberon User Interface

The most remarkable difference between the Oberon user interface and other graphical user interfaces is its philosophy of presenting the user with a large collection of *components* that can be composed in arbitrary ways. Components are the basic building blocks of the Oberon system from which everything else is constructed. This should sound familiar to programmers well–acquainted with object–oriented techniques. A crucial difference is however that all components in the Oberon system are directly accessible and interactively composable by end–users. In fact, Oberon users don't make a distinction between pre–fabricated applications and multimedia documents: everything can be composed interactively, and just as easily taken apart or modified at run–time. To drive this point home, imagine reorganizing the contents of dialog boxes in your programs to your taste – this is possible in Oberon. To distinguish components from those used in other systems, we call our components *gadgets*. Correspondingly, we call the Oberon user interface the *Gadgets System*.

Once composed, collections of components need to be archived for future use (or later modification). To this purpose Oberon introduces *documents*, the storage medium for components. Documents are typically stored as files on your computer, but also might be components composed by program or constructed from HTML. Because of the ubiquitous use of documents and the flexibility of component composition, we call the Oberon system a *document–based system*.

The document–based nature of Oberon is immediately observed after starting the system. Figure 2.1 illustrates the default configuration of the display into a wider vertical *user track* to the left and a narrower vertical *system track* to the right. Each track is further divided into rectangular *viewers*. Viewers correspond to windows in other systems, and are one of the ways in which documents are viewed in Oberon. Each viewer consists of a horizontal *menu bar* at the top, and a larger viewing area at the bottom called the *main* frame. The menu bar contains the name of the document viewed in the main frame and a sequence of buttons that apply to the viewer.



Chapter One

Introduction and Design Principles

1.1 Introduction

Oberon is simultaneously the name of a programming language and of a modern operating system. The Oberon project [WG92] was started at the Swiss Federal Institute for Technology (ETH) in 1985 by Niklaus Wirth and Jürg Gutknecht. In addition to the software, hardware in the form of a general–purpose computer called *Ceres* [Ebe87] (based on the National Semiconductors 32000 processor family) was built to run the new operating system. After a period of internal ETH use for education, the decision was made to document the language and the operating system in a series of books, and to port the Oberon system to popular computer hardware where it would run natively or on top of the native operating system of the host. Today, the original Oberon system is available freely for many computer architectures.

In 1991, Jürg Gutknecht continued the development of the operating system in a newly formed Oberon System 3 group [Gut94, Mar94]. The goal was to exploit the inherent features of Oberon to a much larger degree, upgrade the system by a concept of persistent objects, modernize the user interface and provide support for the ubiquitous network. In 1995, the Oberon System 3 Release 2.0 was finished. Concurrently with the development, the system was documented with a new set of hypertext–based tools. Since then, the system has been constantly improved and extended.

This guide forms part of the documentation effort. It is addressed to users of the system and to programmers with Oberon language experience. The guide covers the current state of the project and is divided into user and programmer guides.

1.2 Design Principles

The underlying dynamic model of Oberon is extremely simple. There exists a single process acting as a common carrier of multiple tasks. This process repetitively interprets *commands* which are the entities of execution in Oberon. Commands are atomic actions operating on the global state of the system. Unlike customary interactive programs, they rigorously avoid direct dialogs with the system user; in other words, the system is completely *non–modal*. The following examples indicate

```
Oberon System 3 for Windows^tm
  Release 2.2 (Win32/4.8.97) on Windows 95 4.0

Oberon Module Interchange (OMI) / mf, tk 8.2.96
OMI run-time checks enabled
OMI reference info enabled
System.ChangeDirectory C:/OMIAug04/Work
```

This System

Desktop	Text Editing	Gadgets
System	Applications	Programming
Platform		

Tools, Panels, Desktops

Gadgets.Panel	Documents.Panel	TextDocs.Panel
Libraries.Panel	Files.Panel	Rembrandt.Panel
Oberon.Desk	DOSBackup.Tool	Keys.Panel
New Panel	New Text	New Columbus

Personal

Private Projects	E–Mail	Address Book

Other

Network	Docu	Coder Tools

Submit a bug report/complete a survey by e–mail

Figure 2.1 The Oberon startup display

The main frame of viewers display a graphical view of a document. Often we will refer to viewers simply as documents, because they are often seen together. The Oberon systems uses several different types of documents to display information. The bulk of this chapter refers to a specific document type called *text documents*, which belong to a predefined class of gadgets named *TextDoc*. As the name indicates, text documents contain mostly text. We say "mostly" – because Oberon texts may contain arbitrary components that float along inside of the text. For example, the bottom viewer in the system track of Figure 2.1 contains a number of buttons. Note that although we emphasize text documents in this chapter, there are many other document classes that are of completely different nature.

We refer to the arrangement of viewers and documents as the *viewer system* or *desktop*. The configuration as sketched is called a *tiled viewer system* because viewers share the screen in a tiled fashion. Oberon also supports an overlapping viewer model, which is discussed in the following chapter. In reality, the structure of viewers is three–dimensional. A new track may in fact overlay one or, more generally, an integral number of existing tracks. The original configuration will be re–established when the overlaying track is later removed.

In order to change the size of an existing viewer, simply point with the mouse to its menu bar, press the ML key and move the mouse up or down. Release the key when the viewer has the desired size. You can also conveniently move a viewer to any different place on the display screen by starting exactly as just explained, then interclicking the MM key, dragging the mouse to the new location, and releasing all keys there.

Text documents are often distinguished further by their content. The top–most viewer in the system track of Figure 2.1 is called the *system log*, *log viewer*, or simply *log*. Status messages that indicate how a computation was completed are always written to the log. The bottom–most viewer in the system track is an instance of a *tool viewer*, or simply *tool*. Tool viewers typically collect related functions together in a set of *commands*. We will return to commands in a moment. A third type of text document is shown in the user track. This document contains prose text that describes the Oberon system. Although we distinguish between different types of text documents according to content, there is no intrinsic distinctions between them – they are all texts that can be freely edited.

In principle, new viewers are allocated their position automatically using heuristics. For example, tool viewers are opened in the system track, and document viewers in the user track. However, you can override any automatic allocation by first placing the *marker* (sometimes called *pointer*) at the location where you desire the top of the new viewer to be placed. The marker is star–shaped (⁎), and it is placed by moving the mouse focus to the desired position and then hitting the F1 key.

2.2 Concept of Commands and Tools

Among the classes of possible objects to be handled by a computer system, the class of *texts* plays a key role. Not only are input and output data frequently represented as text, but also objects and commands are often identified by their name. *Text* is therefore a predefined class of object in Oberon.

A tool viewer contains a list of command names (*commands* in short), some of them followed by parameters. Commands in Oberon are of the written form `M.P` where `M` designates a module (package) and `P` a procedure (operation) that is provided by the module. A user *activates* a command simply by pointing at its name with the mouse and clicking the MM key. For example, activating the command `System.Time` will result in the current time to be written to the log.

Care should be taken not to confuse commands with file names as the latter are written in a similar manner and also appear in tool texts. Commands are written in such a way to specify an action, for example, `System.Open`, whereas file names can often be recognized by extensions like `.Text`, `.Panel`, `.Mod` and so forth.

More often than not, the execution of a command is parameterized. For example, the opening of a document needs the specification of its name, as in `Desktops.OpenDoc Gadgets.Panel`. Although typical, this is not by far the most general case of a parameter specification. Some commands accept an entire list of names following the command name and execute repeatedly for each member of the list. The list must be terminated by a symbol other than a name, preferably a special character that draws the attention. By convention, Oberon uses for that purpose the tilde character "~" which will be referred to as *list terminator*. From now on, we shall use the terms *parameter* and *parameter list* in the restricted sense of "item following the command name" and "list of items following the command name" respectively. In principle, a text adhering to an arbitrary syntax (understood by the command) could be passed over equally well. Commands may even expect as parameters objects of any kind currently existing in the system such as viewers, text selections, caret, and the star–shaped marker.

We shall call a location or an object "marked" if it is visibly or invisibly marked by the marker (⁎). The visibility of the marker is irrelevant in most cases. As an exception, we mention the explicit allocation (or overriding of the automatic allocation) of a viewer which requests the marker to be visible. The marker is initially invisible and placed in the lower left corner of the display.

Some commands even allow different ways of parameter specification. For example, if `Desktops.OpenDoc` is called with a "↑" symbol instead of a file name following the command name, then the file name is taken from the most recent text selection. In general, a "↑" symbol following a command name always refers to the current text selection.

It is noteworthy that tools are ordinary texts distinguishing themselves from more usual texts only by their structure and contents. Oberon System 3 is delivered with a set of standard tools which are text documents stored in files which have been given the file extension `.Tool` by convention. In particular, tools are amenable to editing operations. Looking at this differently, we recognize that commands like `Desktops.OpenDoc Explanations.Text` may well slip into a prose text and be activated directly in place. Obviously, no limits are set to fantasy exploiting this universal scheme of command interpretation.

One rather moderate application of the universal scheme discussed above is the construction of interconnected texts. As a matter of fact, the set of standard tools is structured as a tree with the `System3.Tool` as ancestor and the tools listed in the `System3.Tool` as its descendants. We recall that the hierarchical tool system may easily be customized on the fly by adjusting command lists (including parameters) to personal requirements, reconfiguring the tool hierarchy, installing new tools, or even providing on–line documentation.

2.3 Text Documents

We have stated earlier that extensibility was a key objective in the design of Oberon. It was therefore enticing to realize also system–oriented commands as extensions of the system core on a highest possible level in the modular hierarchy, thereby achieving maximal flexibility. Such a strategy is particularly appropriate for text editing. It manifests itself in the existence of an editing package providing an extensible set of powerful editing commands. As a future programmer of the Oberon system, you will be able to extend the existing text editing facilities with your own special–purpose commands. Nevertheless, several built–in commands are interpreted directly by text objects. They include positioning the text within its viewer, placing the caret, inserting a typed character, selecting a part of text, deleting a selected part of text, copying a selected part of text, copying text attributes and, most importantly, executing an arbitrary command which is specified by its name.

2.3.1 Mouse commands

Text positioning. In order to reposition the visible part of a longer text within a viewer, move the mouse into the viewer's scrolling zone first. This is a vertical bar along the left borderline about 5 mm in width. Now, you can scroll forward by pressing the ML key, moving the mouse, and releasing the key when the text line that you want to become the top line is underlined. Notice that every text viewer shows a small crossbeam indicating the current position of the displayed section within the entire text. You can position a text directly by clicking the MM key at the location where you want the crossbeam to be. Scrolling backwards is accomplished in a similar manner with the MR key. The MM key behavior is modified by interclicking the other two keys to scroll to the beginning or end of the text. The MM key combined with a ML key interclick scrolls to the end of the text. The MM key combined with a MR key interclick scrolls to the beginning of the text.

Placing the caret. If you want to place the caret, move the mouse to the desired text, press the ML key and, while keeping it down, move the caret to the desired position. Any subsequent characters typed on the keyboard are then inserted at this position. The font used for typed characters depends on the font that the character just *before* the caret has. On a PC, the special characters ä, ö, ü and ß can be typed directly by pressing the CTRL key (or the ALT key for PC Native Oberon and Windows Oberon) and a, o, u and s respectively and, the uppercase Ä, Ö, Ü are obtained by pressing SHIFT at the same time. The four arrow keys (left, right, up, down) are used to move the caret to the previous or next character or text line. Once the caret is set, the Page Up and Page Down keys are used to scroll one page up or down respectively. By default, pressing the ENTER key results in auto–indentation. The same number of TAB or space characters found on the previous text line is inserted on the newly created empty line, a convenient feature when writing Oberon modules. Unfortunately not all computer keyboards have the same keys, so some of the keystrokes mentioned above might be mapped to other keys on your keyboard, or might be missing completely. A platform–specific guide included with your Oberon release provides additional details.

Selecting text. You can select a stretch of text by moving the mouse to the desired beginning, pressing the MR key and, while holding it down, extending or reducing the selection by moving the mouse. If you click twice at the beginning, the selection is automatically extended to the origin of that text line. A separate selection may be active for each displayed text section: the selection is not unique. If several selections exist simultaneously on the display, commands normally refer to the most recent one, or to the most recent ones. If a piece of text is too large to be selectable within a single viewer, use `[Copy]` in the menu bar to open an adjacent second viewer. Then, select the *beginning of the text entirely* in one viewer and the *entire end of the text* in the other viewer. The selection will then extend across viewers. Placing the caret and pressing the right or the left cursor keys on the keyboard, will move the selection in that viewer to the right or to the left. TAB characters are automatically inserted at the beginning of the line or respectively removed.

There are interesting interclick variants of caret placing and text selection that combine these marking operations effectively with text editing. But keep this general rule in mind: any mouse–controlled operation that is currently under execution can be nullified by interclicking all remaining mouse keys.

Copying text. If you interclick the MM key while you are placing the caret, the most recent selection is automatically copied to the caret's position as soon as you release the ML key. This feature is particularly convenient for copying a specific template to several different places. Alternatively, if the caret is already set and you click the MM key while you are selecting a piece of text, the selected text is copied to the caret's position when you release the select key. This option is most conveniently used in order to copy a given string to various places.

Copying text attributes. If you interclick the MR key while you are placing the caret, the *character attributes* (font, color, vertical offset) of the character just after the caret are automatically applied to the most recent selection as soon as you release the ML key.

Deleting text. If you click the ML key while selecting a text, the selected text is deleted. Notice that the copy variant and the delete variant of the select command apply also in the case of large selections involving a viewer with multiple copies.

Activating a command. Activating a named command from within a text viewer is generic and therefore the most general built–in operation. In order to do it, simply point to the command's name and click the MM key or *activate key*. We shall however speak of "activate key" only when the MM key is used alone inside a gadget. Sometimes, like in a module development and testing phase, it is important that the newest version of the module providing the desired command is loaded before the command is actually executed. In order to force this, interclick the ML key while pressing the MM key and pointing to the command's name.

Opening a document. As Oberon is an example of a document–based system, you can open documents of all types directly without knowing their associated "applications". The conventional way of doing this is with the `Desktops.OpenDoc` command. As opening a document appears so often, an interclick combination has also been reserved for the task. Simply MM click on the document name you want to open, and interclick with the MR key.

Nullifying a mouse command. Perhaps the easiest and most important rule to remember is that the current action is *nullified*, if all remaining mouse keys are interclicked, though not necessarily simultaneously, during the action.

Neutralize key. To remove all marks on the display, including the caret, the marker and text selections, press the Neutralize key. The F2 key is defined as Neutralize key for all the Oberon system implementations. In addition, a system can be customized to recognize the ESC key as Neutralize key *or* as key generating the ESC character CHR(27). By default, the ESC key is a Neutralize key.

Command execution may lead to an error condition, which when detected by the associated module, is reported directly to the user in the system log. If command execution fails altogether, the system falls into a *trap*. A trap handler viewer is automatically opened whenever a trap has occurred. It displays the state of the interrupted process, including the stack of procedure activations. If a trap appears often, and you suspect that it is related to a mistake in the Oberon system, you can report it to the ETH developers by mailing the trap contents. It often contains enough information to correct the fault. You may continue your work after a trap occurs, although in some rare cases you will have to exit the Oberon system with the command `System.Quit` and to restart it again.

The following summarizes the *basic* meaning of the three mouse keys. The ML key is the *point key*: it is used to focus a certain location; that is, to place the caret. The MM key is the *activate key*: clicking it causes the appropriate command interpreter to be called. The MR key is the *select key*: it is used to select text and other objects within a viewer.

Remark: The editing operations presented are not applicable to text only, but are often applicable to most other visual objects too. Activating a command by pointing at its name and clicking the activate key is a more universal operation which applies equally to gadgets with associated commands: placing the mouse focus on such a gadget causes the activation of the command when the activate key is clicked.

2.3.2 Editing commands

According to Oberon's basic scheme, additional functionality is provided by the text editing package. It contains the following three commands generally applicable to documents and a collection of commands applicable to text documents. The latter are described in Chapter 4 under the heading "TextDoc" and are also listed in the `TextDocs.Tool` delivered with the Oberon distribution.

`Desktops.OpenDoc DocName`
`Desktops.OpenDoc ^`
opens a document. The document name is alternatively specified by a parameter on the command line or, if a "↑" symbol follows the command name, by the most recent selection of a name. In order to override automatic viewer allocation, place the marker anywhere on the screen. When a document is opened in the user or system track, it remembers its location, a hint to where it will be opened in future. Notice that the document menu bars change their content according to the width of the track where they are opened, and might vary accordingly each time the document is opened. A fresh text document can be opened by specifying any name not matching the name of an existing file. By convention, a text document is given a name ending with a `.Text` extension, but that is not compulsory. Remember that the `.Tool` extension also denotes a text document with a special meaning for the user. A document can also be opened by interclicking the MM key and the MR key on the document name alone.

`Desktops.PrintDoc Default *`
`Desktops.PrintDoc Default namelist`
prints either the marked document or all the documents named in the list on the printer specified by the first name. Depending on your platform, the printer name may vary, but in most cases the printer name is simply ignored. Please check your implementation guide for more details.

`Desktops.Recall`
recalls the document closed most recently. The document re–appears in the

same attitude as it had when it was closed and it includes the last changes although the document had not been saved. The exclamation point in [Store] will however not re–appear (See section 2.3.3).

2.3.3 Menu commands

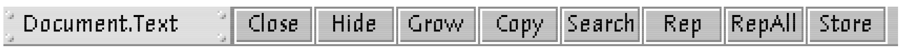


Figure 2.2 A typical menu bar

A text document has a menu bar indicating its name and a sequence of command buttons applicable to that text viewer. The name is normally the name of the file in which the document is stored or the empty string for a new document. In a few cases, it is the name of the command that opened the viewer. The name appears in what is called a NamePlate. The NamePlate and the buttons are examples of visual objects called gadgets. Although a NamePlate can contain only a simple character string, most editing operations work in the same way as those in a main editable text as is explained in Chapter 4 under the heading "NamePlate". Thus, the name can be changed at will and a new name can be assigned to a document before storing it using the [Store] button. We shall be referring to a button by placing its caption text between square brackets as in the previous example. Menu commands related to text documents are the following:

`[Close]`
removes the viewer. A viewer closed by mistake can be recalled with the command `Desktops.Recall`

`[Hide]`
minimizes the document, whereby the meaning of minimizing depends on the environment. When the document is placed on a desktop, the document is removed from the desktop and its name is placed in the Finder. Otherwise, the menu bar of the document is pulled down to the very bottom of the track, leaving the rest of the document invisible.

`[Grow]`
lets the viewer grow to the size of a whole track or, if applied to a viewer already filling a track, to the size of the whole display. The original constellation will be re–established when the grown viewer is later closed.

`[Copy]`
opens a copy of the original viewer displaying the same instance of content. This means that editing in one viewer will cause changes to be shown in both. If you really want two different documents, you must open the document twice.

`[Search]`
searches for a pattern in the text. The pattern is defined by the most recent text selection. If none exists, the previous pattern is used. Searching is started at the position of the caret. If none exists in the marked text, searching starts at the beginning.

`[Rep]`
replaces the last pattern found (located at the caret) with the latest selection. Afterwards the following pattern is located in the text. Clicking [Rep] again will repeat this process. In this manner, all occurrences of a pattern can be replaced by another one under the user's control.

`[RepAll]`
replaces all occurrences of a pattern by another one in a single action.

`[Store]`

writes the document contents to the file with the name of the document specified in the NamePlate. You may edit the name of the document directly in its NamePlate. An exclamation point appears in the caption when the text is modified, when a gadget is inserted or removed, but not when a gadget is manipulated.

Not all document types feature the same constellation of menu commands and some document types feature the following menu commands (the sytem log for example):

`[Locate]`
positions the text in the marked viewer according to the position number indicated by the most recent text selection. Leading non–numerical items in the text selection are ignored. The position number indicates whereabout in a source module an error was detected by the compiler. During compilation, the compiler writes the error position numbers in the system log.

`[Clear]`
clears the contents of a document.

2.3.4 Fonts

Text may be written using several font families delivered with the system. Font names are written in the form (where [ ] means optional):

For screen fonts: Family Size [Style] ".Scn.Fnt"
For printer fonts: Family Size [Style] ".Pr3.Fnt" (300 dpi)

The most commonly used families are the proportional fonts Oberon and Syntax, both available in the sizes 8, 10, 12, 14, 16, 20 and 24 points (1/72 inch), and in the styles i (italic), m (medium bold) and b (bold). Oberon is a family of typefaces that was specifically designed for the Oberon system by Hans Meier [Caf96]. This typeface combines in a unique manner typical elements of antiqua and modern typefaces. The Syntax typeface family was also designed by H. Meier. The non–proportional Courier font family is available in the sizes 8, 10 and 12 points. PC Native Oberon uses the `Oberon10.Scn.Fnt` font by default. The other implementations use the `Syntax10.Scn.Fnt` font by default. Tools are often written in `Syntax10.Scn.Fnt` and titles in `Syntax12i.Scn.Fnt`. In most cases you will need to work only with screen font names when creating text documents, and the printer fonts can be ignored. On–the–fly translation is automatically done when you print a text. If your Oberon host operating system supports other fonts like TrueType, you will be able to use these fonts in a similar manner.

2.3.5 Using Styles

In addition to adjusting the font, color and vertical offset of text, the text system also supports formatting styles like left, right, center and block adjust. Formatting is controlled by TextStyle gadgets floating inside the text. A style influences the format of the text immediately following the style up to the next style. When editing a text, the styles are visible and can be directly manipulated with the mouse. When an existing text is first read and presented in a viewer, the styles are blended out. Styles are never printed. The following commands control the styles:

`TextGadgets.NewStyle`
inserts a style at the caret. The CTRL–ENTER combination is a shortcut for inserting a style during typing. In that case all the styles are made visible. On the Macintosh, use the num–lock key instead.

`TextDocs.Controls`

`TextDocs.Controls`
toggles the visibility of the styles in the marked (*) or in the selected document.

A TextStyle has the shape of a long thin dotted line, the width of which specifies the width of the text block. It is divided into two sections: the top part, above the dotted line, controls the formatting whilst the bottom part controls the setting of the tab stops.

The formatting section may show black rectangles at the left and right end of the style as an indication of the current formatting style. The rectangles on each side, called weights, are toggled on and off by pressing the MM key to the left or the right of the center point of the style. An activated weight "pulls" the text in that direction with the following effect:

No weights. Center adjust.
A left weight. Left adjust mode with word wrapping.
A right weight. Right adjust mode.
A weight to the left and the right. Block adjust mode.

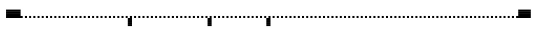


Figure 2.3 A TextStyle gadget set to block adjust and three tabulators

The weights can be grabbed and moved with the MM key to adjust the left and right margin for text formatting.

Mouse commands *below* the dotted line control the setting of the tab stops. They show up as small black rectangles. Tab stop positions are adjusted with the MM key. A new tab stop is inserted at a specific position with a MM and ML key interclick and a tab stop is removed by dragging it completely out to the left or right of the style gadget. As tabbing does not make much sense in center or right adjust mode, in such cases the tab stops are ignored.

Styles have an attribute to switch on page breaking or not. The Columbus gadget inspection tool, introduced in the next chapter, explains some more about attributes. A style with a page break attribute set shows up as a solid instead of a dotted line.

2.3.6 The EditTools tool

Oberon's extensibility makes it easy to add functionality to existing applications. A good example of extension is given by the EditTools tool for influencing the text look. The EditTools commands are listed here, in addition to being listed in the `EditTools.Tool` document:

`EditTools.ChangeFamily {old=>new}~`
changes the text selection in family `old` to family `new`. A question mark in place of `old` indicates that the operation should be applied to the selection regardless of the font family. More than one conversion can be specified in one command. Examples are:

```
EditTools.ChangeFamily Syntax=>Courier Arial=>Times~
EditTools.ChangeFamily ?=>Arial
```

`EditTools.ChangeSize {old=>new}~`
changes the text selection in size `old` to size `new`. A question mark in place of `old` indicates that the operation should be applied to the selection regardless of the font size. More than one conversion can be specified in one command. Examples are:

```
EditTools.ChangeSize ?=>12~
EditTools.ChangeSize 12=>16~
```

`EditTools.IncSize 1`

EditTools.IncSize

increases the font size of the selected text by a positive or negative number of points.

EditTools.ChangeStyle {old=>new} ~

changes the text selection in style `old` to style `new`. Old and new can be a period "`.`" (for the normal typeface), `i` (italic), `m` (medium bold), `b` (bold). A question mark in place of `old` indicates that the operation should be applied to the selection regardless of the font style. More than one conversion can be specified in one command. Examples are:

```
 EditTools.ChangeStyle ?=>.
 EditTools.ChangeStyle m=>b
```

EditTools.ChangeFont {old=>new} ~

changes the text selection in font `old` to font `new`. A question mark in place of `old` indicates that the operation should be applied to the selection regardless of the font. More than one conversion can be specified in one command. An example is:

```
 EditTools.ChangeFont Syntax10.Scn.Fnt=>Syntax12.Scn.Fnt
```

EditTools.ChangeVoff {old=>new} ~

changes the text selection with vertical offset `old` to vertical offset `new`. A question mark in place of `old` indicates that the operation should be applied to the selection regardless of the vertical offset. More than one conversion can be specified in one command. Examples are:

```
 EditTools.ChangeVoff ?=>12~
 EditTools.ChangeVoff 12=>16~
```

EditTools.IncVoff n
EditTools.IncVoff -n

increases the vertical offset of the text selection by a positive or negative number of points.

EditTools.ChangeColor {old=>new} ~

changes the text selection with color `old` to color `new`. A question mark in place of `old` indicates that the operation should be applied to the selection regardless of the color. More than one conversion can be specified in one command.

EditTools.ShowAttrs

shows the attributes of the selection. If text is selected, the position of the first characters where font transitions occur are displayed in the system log together with the text attributes: font, color and vertical offset. If gadgets are selected, their position together with their generator are displayed. Select such an information line in the system log and click the menu button `[Locate]` to set the caret at the corresponding position in the selected text.

Among the files that you might want to convert are MS–DOS ASCII text files which use CR (Carriage Return), LF (Line Feed) where Oberon uses CR. The conversion involves changing the CR/LF pairs to single CRs. If you open a text and find that it is shown with small rectangular boxes (representing the LFs) at the beginning of each line, you can be sure that an MS–DOS ASCII text is involved.

EditTools.OpenAscii filename
EditTools.OpenAscii ^

opens a document viewer displaying the named MS–DOS ASCII file converted to Oberon System 3 text.

EditTools.StoreAscii [*]

stores the marked text document as an MS–DOS ASCII file. Conversion of Oberon text to ASCII (CR/LF) is made. Objects floating in the text are discarded. The file name is taken from the document's NamePlate. If the command is executed from within a document, the document itself is implied: it need not necessarily be marked and the "*" is not required.

Among the files that you might encounter on some FTP sites are Unix ASCII text files which use LF where Oberon uses CR. The conversion involves changing these LFs to CRs. If you open a text and find that it is shown interspersed with small rectangular boxes (representing the LFs), you can be sure that a Unix ASCII text is involved.

EditTools.OpenUnix filename
EditTools.OpenUnix ^
opens a document viewer displaying the named Unix ASCII file converted to Oberon System 3 text.

EditTools.StoreUnix [*]
stores the marked text document as a Unix ASCII file. Conversion of Oberon text to Unix (LF) is made. Objects floating in the text are discarded. The file name is taken from the document's NamePlate. If the command is executed from within a document, the document itself is implied: it need not necessarily be marked and the "*" is not required.

EditTools.StoreMac [*]
stores the marked text document as a Macintosh ASCII file. Conversion of Oberon text to Macintosh ASCII is made. The file name is taken from the document's NamePlate. If the command is executed from within a document, the document itself is implied: it need not necessarily be marked and the "*" is not required.

EditTools.RemoveObjects *
removes all objects including styles from the marked document.

EditTools.Words *
EditTools.Words @
counts the number of carriage returns, words, characters and objects in the marked document or starting at the beginning of the most recent selection in a document. The result is presented in the system log.

2.4 The System tool

The System module manages system–related tasks like file copying, file deleting, module inspection, module freeing, etc. In addition to the commands listed here, you will also find a System.Tool document in your Oberon distribution with the same commands. Before we start, we must review the structure of Oberon file names. In the simplest case, an Oberon file name consists of the letters A to Z (upper or lower case), the digits 0 to 9, and the period ".". Oberon supports long file names up to 32 characters in length containing more than one period. It is very important to note that Oberon is case–sensitive!

PC Native Oberon, like the original Oberon, uses a flat file directory and does not support subdirectories. Linux, Mac and Windows Oberon allow you to access any file on your host file system. Consequently, further characters are valid in file names, typically those that are used for specifying directories. For example, on UNIX platforms, the forward slash "/" is used as a directory separator, while on DOS platforms, the backslash "\" is used instead (in additon to ":" as a drive specifier). The Macintosh platform uses ":" instead.

Oberon uses "/" as a *directory separator* and "\" as an *option character* for introducing command options.

System.Open filename
System.Open ^
opens a viewer displaying the content of the named file.

System.OpenLog
opens the system log viewer. This text document shows the results of

commands and lists compiler detected errors for example. The log content is shared between all Oberon modules.

`System.CloseTrack`
closes the marked track; that is, removes all viewers in this track.

`System.Time`
`System.Time dd mm yy hh mm ss`
displays the current date and time in the form dd.mm.yy hh:mm:ss. If date and time parameters (leading zeroes may be omitted in each component) immediately follow the command name, the command sets the date and the time accordingly.

`System.Watch`
displays the amount of currently used memory resources. Memory is allocated in a system–wide heap shared by all modules. Parts of the heap are allocated (i.e. in use) and other parts are free.

`System.Collect`
initiates a subsequent garbage collection. Garbage collection is the process with which unused memory is returned to the free part of the Oberon heap. A garbage collector is an essential part of an extensible system; without it we would not be able to determine when all extensions have released shared system resources.

`System.ShowModules`
displays a map of all currently loaded modules. A module `M` contains code to implement certain functions and is activated by executing commands in the form `M.P` For example, `System.ShowModules` calls the procedure `ShowModules` in the module `System`, which has the task of listing all loaded modules. A module is loaded from an object file only when it is required; that is, the first time you execute a command in that module. For example, the compiled module code of `System` is located in the file `System.Obj` (which was generated from the source `System.Mod` by the Oberon compiler). Once loaded, a module remains in memory until explicitly freed.

`System.ShowCommands Modname`
displays a list of all commands (in other words, parameterless procedures) exported by the named module.

`System.ShowTasks`
displays a list of all active background tasks.

`System.State Modname`
displays the global (exported) data of the named module in a viewer "System.state".

`System.Free Mod~list`
unloads every module named in the parameter list. The module names must appear in an order which guarantees that client modules are freed first –– a module having a client cannot be unloaded. If a module name is immediately followed by * (an option *not* available in PC Native Oberon), imported modules are also unloaded. Freeing a module is very dangerous when parts of its code are still required, often resulting in a trap or a completely dead system. The * option is even more dangerous and should be used with extreme care.

`System.ShowLibraries`
shows a list of the currently loaded libraries. Libraries are shared resources like fonts and reusable objects.

`System.FreeLibraries Lib~list`
`System.FreeLibraries ^`
frees shared libraries from memory. This is mostly a harmless operation as the library will simply be loaded again when required. The garbage collector frees libraries automatically when they are not required anymore.

System.CopyFiles {A=>B}~
System.CopyFiles ^
processes a parameter list of pairs A => B, copying each file A to B. In the case of a "^" following the command name, do not forget a list terminator.

System.RenameFiles {A=>B}~
System.RenameFiles ^
processes a parameter list of pairs A => B, renaming each file A to B. In the case of a "^" following the command name, do not forget a list terminator.

System.DeleteFiles namelist
System.DeleteFiles ^
deletes all files named in the list.

System.Directory template[\d]
System.Directory ^
displays the selection of all files whose names match the template specified by the parameter. The template is a string which may contain the symbol "*" as a wildcard. If the option \d s specified, additional information about file creation dates and file sizes is displayed. If your platform supports multiple directories, you can affix a directory path to the template.

System.Clear
clears the viewer in which the command is executed.

System.Quit
terminates the current Oberon session. Better to save all your files before doing this! This is the normal way to exit Oberon. With PC Native Oberon the system is powered off if the system can perform power management functions. If not, the screen will become blank: power off the system – do not attempt to continue. For other implementations the Oberon session terminates. Alternatively, with Oberon for Windows the application can be terminated by choosing "Close" in the application window's system menu or by using the keyboard accelerator Alt–F4.

The following command is available for PC Native Oberon only:

System.Reboot
terminates the current Oberon session and reboots the system. Alternatively, the usual Alt–Ctrl–Delete combination or also Ctrl–F10 may be used to reboot the system.

The following commands are available for Linux, Mac and Windows Oberon only:

System.CurrentDirectory
displays full path of the current sub–directory in the system log.

System.CreateDirectory name
creates a new sub–directory in the current directory.

System.DeleteDirectory name
deletes a sub–directory from the current directory.

System.ChangeDirectory path
System.ChangeDirectory .
sets the current working directory to that specified by the path or to the parent directory. By default, except when path names are specified, all stored and generated files land up in the current working directory. The System.Tool contains a gadget that displays the current working directory. You may use the gadget to change the current working directory.

System.Get tick [key]
System.Get ^

displays the value of the named key in the named section of the registry. If no `key` is specified, all keys contained in the section are listed together with their associated values. `section` and `key` may be either names or strings.

```
System.Set section.key := [keyvalue]
System.Set section ^
```

sets the key in the named section of the registry. If necessary a new entry is added to the section. If no `keyvalue` is specified, the entry for the given section and key is removed from the registry. `section`, `key` and `keyvalue` may be either names or strings.

Oberon is completely non–modal and often will not ask you to confirm a dangerous action. If you are afraid of executing a dangerous command by mistake, you can prefix it with a "!", as in `!System.Quit`. This will force you to add a space between the "!" and the command before it can be executed. This device is used in several `.Tool` documents.

The `System` module exports several other commands too. For example, the commands `Close`, `Copy` and `Grow` are executed from the menu bars. As these commands are hidden behind the menu buttons, we will ignore them.

User tips:

You can load a module `M` with am *MM* key click on `M.P` in a text, where `P` is a procedure name of that module. If you do not know a procedure name, you can simply use a random name, even if it is not defined in `M`. At the limit, executing the command `M.` also produces the desired effect. This will produce an error message "`Call error: nameXYZ not found`" (in the system log), but the module will have been loaded anyway.

During development, it sometimes happens that a document cannot be closed due to the fact that it is always generating a trap. In this case, use the command `System.Close*` to close the offending document.

2.5 Program Development Tools

Originally developed for teaching programming and operating systems, the Oberon system naturally has a large selection of programming tools. Most of these tools still use the textual user interface of Oberon, and thus fit perfectly into this chapter. A knowledge of the programming tools is required to use the material presented in the chapter about Oberon programming.

2.5.1 The Compiler tool

One of the most crucial parts of an extensible system is the compiler; without it you would not be able to extend the system. The Oberon system, which is implemented in the Oberon language, uses a fast compiler based on a portable compiler front–end called OP2 developed at the ETH, and provides compiler back–ends for most popular hardware architectures. This guide is not intended to give an introduction to the Oberon language; for this we recommend [Wir88], [RW92], [Mös93], [Mös96] and [ML97].

In principle, a compiler takes a syntactically correct source module (conventionally with a `.Mod` extension) and compiles it to an object file (with a `.Obj` extension) and a symbol file (with a `.Sym` extension). This holds true for PC Native Oberon. The other implementations do not generate a symbol file (see below) but the same compiler command is used, though with different options. The object file contains the machine code, and the symbol file contains the definition of the module. The definition tells the world what features clients of that module can use. The `.Sym` files allow us to compile modules separately, using exported features from imported modules without recompiling them. The resulting run–time structure is a hierarchy of Oberon modules, one importing (using) the other, and in reverse, one module having other modules as clients. Oberon modules are linked together by the run–time system when needed and they must be freed explicitely when they are not required anymore or when they are replaced by new versions during a software development session.

The Oberon compiler exports the `Compiler.Compile` command documented in the `Compiler.Tool`:

```
Compiler.Compile [\options] {filename[\options]} ~
Compiler.Compile ^
Compiler.Compile @[\options]
Compiler.Compile *[\options]
```

compiles the indicated module(s), reporting success or failure to the system log. In case of success, *object files* are generated. Compiling a sequence of modules requires you to specify their file names in the sequence of the import hierarchy from the bottom to the top; that is, clients of a module have to be compiled after the module itself. This burden placed on the user can be circumvented using the Builder tool described in the next section. The @ parameter indicates that the compiler should start the compilation process on the current selection. This allows you to compile a module text embedded in a text. Only the beginning of the module's text must be selected, the compiler will search for the final "." by itself. It is allowed to include any number of valid Oberon comments at the beginning of the selection. In a more typical situation, you will be compiling modules directly from the text editor, and thus use the * parameter. There is no need to store the text first unless you want to keep it for further reference. The command accepts a list of options following a "\" for modifying the obvious default values implied. A list of options may appear as first parameter *before* the list of file names. These options apply globally to all the compilations. Also, each file name in the parameter list may be followed by a list of options which apply locally.

Should you change the definition of a module, i.e. the exported features, the Oberon compiler will report an error 155, preventing you from overwriting an existing module definition, if the compiler option `\s` is not used. This option allows the compiler to change the definition of a module, a potentially dangerous operation that can invalidate clients of the compiled module.

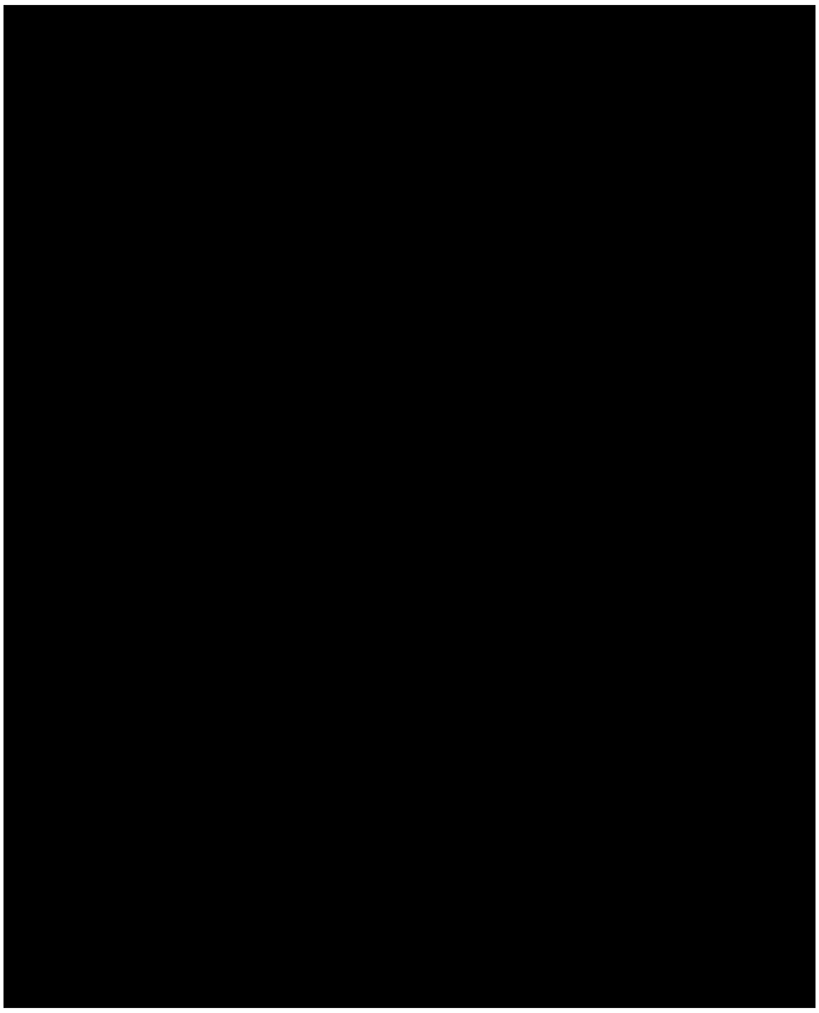


Figure 2.4 Oberon compiler options

The compiler delivered with Linux, Mac and Windows Oberon System 3 can generate *two* types of object files: classical *native object files* containing *target machine code* or *slim binaries* [FK96] by default. Slim binaries are a new form of object file that contain no object code at all, but *portable descriptions of module contents* that makes these files completely independent of the eventual target machine (platform independent). To drive the point home, let us stress that if no option or only the options marked "O" are used, the modules generated by the compiler are portable to all those platforms. In this case the compiler does not generate separate symbol files: equivalent information is stored in the sole object file. Object code generation is carried out on–the–fly [Fra94] by the module loader (depending on the underlying hardware) and takes no longer than loading traditional object files. In this system with dynamic loading, the Object Model Interface (OMI) [Cre94] has been implemented and a fine–grained interface consistency checking is built–in. If a module interface modification may invalidate clients, the compiler reports an error unless the option \s is specified. However, if a pure extension of the interface is made, such as the insertion of a new procedure, the option \e allows the generation of an extended symbol file. Thus, the object model allows a module to be extended without requiring a recompilation of client modules. Clearly, if the module interface is changed or if something is deleted from it the option \s is required to compile successfully. When generating slim binaries, only the options \p \s \u and \w can be used. Since code generation is taking place at load time, it can be influenced by commands as documented in the following table (the default states appear in bold face):

Figure 2.5 Code generation control

The option `\N` instructs the compiler to generate native object files which are *not portable* across platform boundaries. Therefore, the option `\N` should be used only when writing an extension that uses the *non–portable* module SYSTEM or the *non–portable* built–in SIZE function. If garbage collection is suppressed with the option `\g` compilation runs faster but might not be completed if too much memory or too many files are used. In this case, Oberon will trap.

The file names appearing in the parameter list of the compile command may differ from the module names. Under Linux, Mac and Windows some source modules are prefixed. The prefixes "Win." and "Win32." for example are used by Oberon for Windows.

The compiler module provided for the Linux, Mac and Windows Oberon exports an additional command:

`Compiler.SetDestPath pathName`
directs the compiler to store new object files in the specified sub–directory. Remember that those ports have a directory structure.

2.5.1.1 Compiler error handling

The system log plays an important role during program development. For example, successfully compiling a module `Hello.Mod` with the command:

```
Compiler.Compile Hello.Mod
```

results in the log:

```
compiling Hello 33
```

The number following the module name is an indication of the resulting size of the object file. Unsuccessfully compiling a module results in log output with approximately the following form:

```
compiling Hello
  pos   67 err   0   undeclared identifier
```

A log line starting with *pos* indicates an error at that character position in the source text followed by an error number and a diagnostic text. The system log menu button `[Locate]` will show the error position in the marked text when the error position (67 in this case) is selected. The complete list of error numbers with their meaning is stored in `Oberon.Text` included with the PC Native Oberon distribution or in `OberonErrors.Text` for the other implementations.

Afterwards, activate a procedure in the module by executing `M.P` where `M` is the module name and `P` is the name of an exported parameterless procedure `P`.

The module will be loaded and linked automatically into the system. If you make changes to a module, you will need to unload the previous version with the `System.Free` command or with the shortcut MM + ML key interclick on `M.P`. This works only if `M` has no client. The compiler will display "`(in use)`" after the system log message `Compiler.Compile xyz.Mod` when the module just compiled is currently loaded in memory. This is a reminder to unload the module first.

Finally, the Builder tool, described in section 2.5.2, offers a convenient error marking and error interpretation command: `Builder.MarkErrors` and two associated commands `Builder.NextError` and `Builder.ClearErrors`.

2.5.1.2 Run–time error handling

When a run–time error occurs, the system falls into a *trap*. There is no interactive debugger currently available under Oberon. However, a trap handler is automatically called. A "System.Trap" viewer is opened, displaying the state of the interrupted process, including the entire *procedure activation stack* from the initial command call to the the last procedure (`M.P`) in which an error condition was detected. For example:

```
TRAP xy    index out of range
M.P PC=12
```

The program counter (PC) value displayed can be used to locate the error in the source text by recompiling, using the option `\f`. Under the assumption that the module's source text appears in a viewer and that it is marked, proceed as follows in PC Native Oberon:

1 – select the program counter value
  (selecting the entire line with a double MR key click may be used as a shortcut, the alpha string preceding the counter value is ignored)
2 – re–compile the program with `Compiler.Compile *\f`

The error can then easily be located at the position of the caret. The interpretation is left to the programmer. With the other implementations, the trap viewer text is slightly more verbose and five steps are needed to position the caret at the error, under the same assumptions as before:

```
Oberon.Loop – 21 (index out of range)
PC =  00600051H ( 00000009H)
```

1 – select the program counter value between the round brackets
2 – re–compile the program with `Compiler.Compile *\f`
3 – the position in the source text wich corresponds to the PC value
  appears in the system log
4 – select the value after *pos*
5 – use the `[Locate]` button to position the caret at *pos* in the source text.

How to interprete the TRAP information

Each procedure call (`M.P`) is followed by an enumeration, in alphabetical order, of the procedure parameters and of the local variables with their values. Scalars, strings (ARRAY OF CHAR) and pointers appear in clear. No information appears for structured variables.

Example extracted from an Oberon Windows trap:

```
TextGadgets0.Call+ 00003D2FH
  F =  00BA3F00H
  ch = CHR(0)
  chl = 749
  cmd = ""
  cw = 1024
  cx = 0
  cy = 0
  dlink =  00B9FBA0H
  i = 1
  j = 3
  keysum = {1}
  obj =  00BA66C0H
  oldcontext =  00000000H
  par =  00C22AA0H
  pos = 218
  res = 0
```

corresponding to the following source text (extract):

```
MODULE TextGadgets0;
  PROCEDURE Call* (F: Frame; pos: LONGINT; keysum: SET; dlink: Objects.Object);
    VAR S: Texts.Scanner; res, i, j: INTEGER; oldcontext, obj: Objects.Object;
    cx, cy, cw, chl: INTEGER; par: Oberon.ParList; A: Objects.AttrMsg;
    R: Texts.Reader; ch: CHAR;
    cmd: ARRAY 256 OF CHAR;
  BEGIN
```

The practical lesson from this is that some debugging is possible by inspecting additional local variables introduced into the procedures in development.

2.5.1.3 Trap from the keyboard

Under PC Native Oberon, Ctrl–Break hit once terminates the execution of the current command at the next Input/Output operation. If no such operation is encountered and the system is in a loop, hitting Ctrl–Break a second time terminates the execution of the current command and opens a trap viewer. If by accident, all viewers have been closed, force a trap with Ctrl–Break: the trap viewer is opened and work can be resumed with `System.Open System.Tool`.

Under Windows 95 or NT, Pause terminates the execution of the current command. If by accident, all viewers have been closed, hit the Pause key to force a trap: the trap viewer is opened and work can be resumed with `System.Open System.Tool`.

Under Windows 3.1, Oberon programs cannot be interrupted with Ctrl–Break. The only way to stop Oberon is with Ctrl–Alt–Del or with the task manager. Windows 3.1 might be clever enough to kill only Oberon; sometimes it is not. Some applications, like the WWW browser, allow you to interrupt them by pressing ESC. If the system beeps at you, a recursive trap has occured.

2.5.1.4 Practical hints on how to develop programs

While developing and writing software, it is possible to at least approach error–free programming. Of course, it takes a lot of discipline, because it means reading the program text at least two or three times before proceeding to its compilation or execution. It is a fact of life that programmers do not read their texts well enough after completion, however far from it. Oberon is certainly more readable than many other programming languages, but the programmer himself has the greater responsibility of structuring his text to make it readily understandable to himself and to anybody interested. All in all, this sound attitude can save a lot of time and frustration.

Though the built–in traps normally provide enough information for debugging run–time errors, there are further ways to combat one's mistakes. In the first place, develop by stepwise refinement. Large chunks of new code are cumbersome to handle and cost a lot of time in debugging. Frequent compilation in the early phase of development helps finding syntax errors caused by inattention. Further, `Out` commands inserted at strategic places can sometimes help a lot more than a TRAP. Such commands are not just decorating a piece of code until the software is running as expected, but may become integral part of the software when they appear in well–formatted conditional sections of the code. The condition can be determined by a

BOOLEAN constant as in this module stub:

```
MODULE Stub;
IMPORT Out;
CONST Debug=TRUE;

PROCEDURE Any*;
VAR QueriedVarA: ARRAY 32 OF CHAR; QueriedVarB: BOOLEAN;

BEGIN
   ....
   IF Debug THEN
      Out.String("Current values: ");
      Out.String(QueriedVarA);
      IF QueriedVarB THEN Out.String("TRUE") ELSE Out.String("FALSE") END;
      ...
   END;
   ...
END Any;

BEGIN
   ....
END Stub.
```

in which `QueriedVarA` and `QueriedVarB` are the variables to be inspected. Instead of writing to the system log, one could choose to write to a Writer instead, with the added advantage of allowing a customized layout of the inspected data and with the advantage of faster execution which will quite noticeable with a high volume data trace. Obviously, the final program version must be compiled with `debug=FALSE` but this a very little cost compared to the benefit reaped when the module must be modified later on and tested again. The compiler will optimize the object code anyway by removing the dead code parts. Truly enough, the inclusion of permanent `Out` or `Write` statements might be regarded as cluttering what is in the eye of the writer "a well–readable source text".

In modules containing commands, it is good practice to append a list of commands to the end of the program text. Later on, these commands can then be exercised again and again for testing their correctness at least to some degree.

Example:

```
(* Text of the module TextPopups *)
...
END TextPopups.

System.Free TextPopups ~
TextPopups.Install
TextPopups.Remove
```

2.5.1.5 Using HALT to debug

On encountering a HALT(e) statement, the enclosing program is brought to an abnormal halt with a TRAP. The argument e is an integer constant ($30 <= e < 256$) whose value identifies the termination. In case of a stubborn abnormal program termination, one may attempt to insert a HALT statement at a strategic point thus forcing the program to reveal the status of some of its variables before the crash. Of course, this brute force approach is not recommendable but it may be used as an expedient for disentangling a difficult TRAP situation.

2.5.1.6 Using assertions as a debugging tool

A citation from Niklaus Wirth [Wir73] will set the stage:

Experimental testing of programs can be used to show the presence of errors but never to prove their absence.

Consequently, it is necessary to abstract from individual processes and to postulate certain generally valid conditions that can be derived from the pattern of behavior. This analytic method of testing is called program verification. In contrast to program testing, where the individual values of variables are inspected, program verification is concerned with the properties of the program by postulating generally valid ranges of values and relationships among variables.

Assertions are a means to integrate program specification aspects in program code, thereby increasing the confidence in the quality of the software. Assertions are supported by the compiler which can recognize two forms of assertions:

`ASSERT(boolean-expression);`
causes the program to terminate if the expression is FALSE.

`ASSERT(boolean-expression, integer-constant);`
causes the program to HALT if the expression is FALSE and is similar to the conditional statement:

```
   IF ~boolean–expression THEN HALT(integer–constant) END;
```

The first form uses a system defined termination code is used instead of the user defined `integer-constant`

Oberon System 3 uses the following *conventions* for this constant:

Precondition (100..109)

A precondition tests for the legal input to procedures, for example a parameter must be in a certain range.

Invariant (110..119)

One may categorize invariants into loop invariants and type invariants. A loop invariant is a condition which must be satisfied at each iteration in a loop.

Postcondition (120..129)

A postcondition verifies the outcome of the execution of a procedure.

Example

Suppose it is required to write a function `Sqrt` yielding the square root of `x` within the tolerance `tol`. An implicit specification would state that the absolute value of the difference between the square of the result `res` and `x` must be less than the tolerance. Using this, we then write:

```
      ASSERT(ABS(res*res – x) <= tol, 120);
```

This specification is, so far, incomplete in that the valid values for `x` and `tol` have not been defined. REAL will be used to denote the set of all real numbers. But this is still not enough. If `Sqrt` is to yield a REAL as result, then it can only find the square root of non–negative numbers. Similarly, the tolerance must also be non–negative. We end with:

```
      PROCEDURE Sqrt*(x, tol: REAL): REAL;
      VAR res: REAL;
      BEGIN
         ASSERT((x>=0) & (tol>=0), 100);
         calculate the square root of x into res (* Explicit specification *)
         ASSERT(ABS(res*res – x) <= tol, 120);
         RETURN res
      END Sqrt;
```

How ASSERT statements are treated by the compiler

The compiler evaluates the boolean expression of ASSERT statements with the following outcome:
 – if an expression is TRUE, the ASSERT is treated as dead code
 – if an expression is FALSE, an error message (99) is logged
 – if an expression cannot be evaluated at compile time, the ASSERT statement is included as a run–time check

How ASSERT statements are controlled in OMI modules

At load time, ASSERT statements can be treated differently depending on the prior execution of two commands:
 – after the execution of `OMI.AssertOff`, ASSERT calls are ignored
 – after the execution of `OMI.AssertOn`, the required code is generated
In contrast, HALT statements are always treated in the same fashion.

Remember that code generation is performed at load time, therefore a module with ASSERT statements must be unloaded after the execution of an `OMI.AssertOff`/`OMI.AssertOn` command. By default, no code is generated.

Information provided by Watson

ASSERT procedures appearing in *exported* procedures and commands are transformed by `Watson.ShowDef` comments of the form (at the right side):

ASSERT(param >= 10, 101);	*(* precondition (101):* param >=10 **)*
ASSERT(param >= 10, 111);	*(* invariant (111):* param >=10 **)*
ASSERT(param >= 10, 121);	*(* postcondition (121):* param >=10 **)*

2.5.1.7 Console debugging (Windows Oberon)

The `Console` module provides a number of procedures (access the module definition with `Watson.ShowDef Console`) which can be quite useful for *low–level* debugging, specially of Oberon core components. Console debugging is controlled by the Console keyword in the [System] section of the registry in the following manner (only the first character is meaningful):

Console=

None	no information is collected
Console	the information is directed to a MS Windows window having a title bar text "Oberon System 3 – Console" which is automatically opened when Oberon is started and closed when it is quitted.
Debug	the information is directed to: – the debugger tool under Windows 95 or NT – the AUX port under Windows 3.1
File	the information is directed to a file "Oberon.Log" in the root directory. Use the command `EditTools.OpenUnix` to view it.

2.5.2 The Builder tool

The Builder provides a convenient front–end to the Oberon compiler described above. It makes sure that module texts are presented to the compiler in a correct order, whatever the order of the file names in the parameter list. The `Builder` module commands are documented in the `Builder.Tool`

```
Builder.Compile [\options] namelist
Builder.Compile [\options] ^
Builder.Compile [\options] *
```
compiles the files named in the list, automatically determining the correct compilation order of the modules. Only the modules specified are compiled. The options are the same as those of the `Compiler.Compile` command but they must appear as *first* command parameter.

```
Builder.Find namelist
```

`Builder.Free ^`
unloads every module named in the list in the correct order. Since file names must appear in the parameter list, this command can only free modules for which the source text is available. To unload other modules use `System.Free`.

`Builder.InsertHierarchy namelist`
inserts an Icon for each source text module named in the list at the caret. The Icons are inserted in a correct order of compilation. Each Icon is captioned with the corresponding file name and its *Cmd* attribute value is `Desktops.OpenDoc '#Caption'`. An example of such Icons is found in the `Compiler.Panel`

`Builder.MarkErrors [^]`
when the selection contains an error message written by the compiler in the system log, this command inserts an error marker in the marked text for that message and for all of the following messages. An *error marker* is a special gadget displaying the number of the error code discovered at that location in the program text. The error message `pos 111 err 4` would cause `4` to be placed at the position 111. An MM key click on the error marker replaces the error number by a short error description. Another click toggles it back to the error number.

`Builder.NextError`
advances the caret to the next error marker. When the end of text is reached, searching wraps around to the beginning.

`Builder.ClearErrors *`
removes all error markers in the marked text. Automatically performed by a `Builder.Compile *` command. When the source text document is stored, the error markers are automatically removed –– they must not be removed first.

2.6 Backup tool

Oberon provides tools for making backups of files on diskette or for some implementations on other external storage, making it possible to exchange Oberon files with most other computer systems as well as with other systems on the same platform.

Backup tool for PC Native Oberon and Windows NT

The Backup tool writes and reads files in a proprietary ETH Oberon format compatible with Ceres Oberon or in MS–DOS format on diskette.

The following commands are defined:

`Backup.Directory [\d]`
lists the files on the currently inserted diskette. The listing shows the Oberon files with their long file names. If the option `\d` is present, additional information about file sizes and creation dates is supplied.

`Backup.WriteFiles namelist`
`Backup.WriteFiles ^`
writes all files named in the list to the currently inserted diskette.

`Backup.ReadFiles namelist`
`Backup.ReadFiles ^`
reads all files named in the list from the currently inserted diskette.

`Backup.ReadAll`
reads all files from the currently inserted diskette.

`Backup.DeleteFiles namelist`
`Backup.DeleteFiles ^`

deletes all files named in the list from the currently inserted diskette.

`Backup.SetDriveA`
`Backup.SetDriveB`
selects the appropriate diskette drive A or B to be used by the various Backup commands. Default is drive A.

Any diskette formatted with a standard DOS formatter may be used. Alternatively, the following commands are provided:

`Backup.Format \D`
`Backup.Format [\H]`
formats, in the Oberon format, a 2–sided double density (`\D`) or a high density (`\H` is optional) diskette in the selected drive. This command is not available for Windows NT.

`Backup.InitOberon [volName]`
initializes a formatted diskette; that is, creates an Oberon directory and writes a volume label. `volName` may be either a name or a string. During this process, all existing data is erased from the diskette. This is much faster than `Backup.Format`. A diskette without volume label can be read on Ceres machines.

`Backup.InitDOS [volName]`
initializes a formatted diskette; that is, it creates an MS–DOS directory, and writes a volume label. `volName` must be a valid MS–DOS name (fname.ftype). During this process, all existing data is erased from the diskette.

DOSBackup tool for Windows and Macintosh

The DOSBackup tool reads and writes Oberon files on MS–DOS diskettes. This makes it possible to exchange Oberon files with most other computer systems. As the MS–DOS diskette file system does not support the long Oberon file names, DOSBackup maintains a translation table called TRANS.TBL as a file on the diskette. Files copied onto a diskette under MS–DOS are automatically inserted into the translation table when you use DOSBackup again. On a Macintosh, you can use this tool only if MacOS 7.5 or a tool that extends the Mac with the capability of reading MS–DOS diskettes, such as PC Exchange for instance, is installed.

The five first Backup commands described earlier are also valid for this implementation, though their prefix is DOSBackup. However, the `DOSBackup.Directory` command does not accept an option `\d` for requesting additional information about file sizes and dates. The `DOSBackup.WriteFiles` command accepts an option character "%" for compressing files:

`DOSBackup.WriteFiles %FileList`
`DOSBackup.WriteFiles %^`
writes all files specified by the parameter list to the currently inserted diskette using the LZW compression algorithm. The compressed files have a special header which is automatically detected by the `DOSBackup.ReadFiles` command for decompressing the files.

The following commands are also defined:

`DOSBackup.Init`
erases all the Oberon files and the translation table file on the currently inserted diskette. All the other DOS files stored on the diskette are not affected.

`DOSBackup.SetPath path`
sets the path for subsequent `DOSBackup` commands. It can thus be used to change the current directory on the currently inserted diskette, but this command is specially useful to direct the backup operation to a hard disk subdirectory as an alternative to using diskettes. Keep this in mind if you have installed more than one Oberon System 3 on your PC.

DOSBackup does not provide a diskette formatting function. Formatting a new diskette must be done under the underlying operating system.

The following command is available in Mac Oberon only:

```
DOSBackup.SetDrive
```
selects the appropriate diskette in the diskette drive.

Backup tool for Macintosh

The Backup tool reads and writes Oberon files from and to a diskette that has been formatted in any of the normal Macintosh floppy diskette formats.

The five first `Backup` commands described in "Backup tool for PC Native Oberon" are also valid for this implementation.

The following command is also defined:

```
Backup.Eject
```
ejects the currently inserted diskette.

Mac Oberon offers an alternative CBackup tool writing and reading files in the original Ceres format. The commands are the same.

Backup tool for Linux

The Backup tool reads and writes Oberon files from and to a MS–DOS formatted diskette. Only the double density format (720KB) is currently supported.

The five first `Backup` commands described in "Backup tool for PC Native Oberon" are also valid for this implementation.

The following commands are also defined:

```
Backup.SetFd0
Backup.SetFd1
```
selects the appropriate diskette drive 0 or 1 to be used by the various Backup commands. Default is drive 0.

```
Backup.Init
```
initializes a formatted diskette; that is, it creates an Oberon directory. During this process, all existing data is erased from the diskette.

2.7 Archiving tools

Oberon also provides tools for compressing multiple files into an archive, and for exchanging files with other Oberon users via electronic mail.

Compress tool

The Compress tool allows you to compress multiple files into the same archive using the LZSS compression technique (Note: the file format is not compatible with the UNIX utility of the same name). The archive is portable among all Oberon platforms. Compressed files with an `.Arc` extension can be opened directly with the command `Desktops.OpenDoc`

```
Compress.Directory Archive.Arc
Compress.Directory \d Archive.Arc
Compress.Directory ^
Compress.Directory \d ^
```

opens a text viewer listing the names of the files contained in the archive. If the archive does not yet exist, it is automatically created. If the option `\d` is used, additional information is given for each file: date and time when added to the archive, compressed size in bytes, and size in percent of the original uncompressed file.

`Compress.Add Archive.Arc namelist ~`
`Compress.Add Archive.Arc ^`
adds all the files named in the list to an archive. If the archive does not already exist, a new one is created.

`Compress.Extract Archive.Arc namelist ~`
`Compress.Extract Archive.Arc ^`
unpacks all the files named in the list from an archive.

`Compress.ExtractAll Archive.Arc`
unpacks all the files from an archive.

`Compress.Delete Archive.Arc namelist ~`
`Compress.Delete Archive.Arc ^`
deletes all the files named in the list from an archive.

`Compress.Open Archive.Arc filename`
`Compress.Open Archive.Arc ^`
unpacks a file from an archive and opens it. None of the archived files is stored on disk.

`Compress.Rename Archive.Arc {A=>B}`
processes a list of pairs following the archive name, renaming each file A to B in the archive.

AsciiCoder tool

AsciiCoder can be used to encode and decode any text (including gadgets) visible on screen as well as to encode and decode arbitrary files. The data stream generated by encoding is fully ASCII and is apt to be sent by electronic mail. It is always presented in a text document where it can be edited to become a part of an e-mail message for instance. Once encoded, the resulting text will contain the appropriate command to decode the contents again.

`AsciiCoder.CodeText *`
`AsciiCoder.CodeText @`
encodes the text of the marked viewer or the text starting at the beginning of the most recent selection. A viewer named "AsciiCoder.CodeText" is automatically opened. It presents the encoded data preceded by an appropriate `AsciiCoder.DecodeText` command.

`AsciiCoder.CodeFiles namelist ~`
`AsciiCoder.CodeFiles ^`
encodes all the files named in the list. A viewer named "AsciiCoder.CodeFiles" is automatically opened. It presents the encoded data preceded by an appropriate `AsciiCoder.DecodeFiles` command.

`AsciiCoder.DecodeText @`
decodes the ASCII encoded text, starting at the beginning of the most recent selection.

`AsciiCoder.DecodeFiles @`
decodes the ASCII encoded text, starting at the beginning of the most recent selection, and writes the data to files. Existing files are not overwritten since they are first renamed to `.Bak` files. The file names must appear at the beginning of the selection, before the coded data.

Inserting an (optional) `%` between the command name and the first parameter, will cause the data to be compressed or expanded, in addition:

```
AsciiCoder.CodeText %*
AsciiCoder.CodeText %@
AsciiCoder.CodeFiles %^
AsciiCoder.CodeFiles % ~

AsciiCoder.DecodeText %@
AsciiCoder.DecodeFiles %@
```

Base64 tool

Base64 can be used both to decode a Base64 encoded data stream appearing in a viewer and to encode files. The ASCII data stream generated by encoding can be forwarded by electronic mail. It is always presented in a text document where it can be edited to become a part of an e–mail message.

`Base64.Decode filename ^`
decodes the data, starting at the beginning of the most recent selection, and writes the data to the named file.

`Base64.Decode filename ~ encodedData`
decodes data which follows the "~", and writes the data to the named file.

`Base64.Encode filename ~`
encodes the named file. A document viewer named "Encode" is automatically opened. It presents the encoded data, which may then be sent by e–mail.

UUDecoder tool

UUDecoder can be used to decode Unix uuencoded data.

`UUDecoder.Decode name-list ~`
`UUDecoder.Decode ^`
decodes all the files named in the list, and writes the decoded data to a file whose name appears in the uuencoded files themselves.

`UUDecoder.Decode @`
decodes the data, starting at the beginning of the most recent selection, and writes the decoded data to a file whose name appears in the uuencoded data.

`UUDecoder.Decode "begin" encodedData ~`
decodes the data which follows the command and writes the decoded data to a file whose name appears in the uuencoded data.

BinHex tool

BinHex can be used to decode BinHex encoded data.

`BinHex.Decode filename ^`
decodes the data, starting at the beginning of the most recent selection, and writes the decoded data to the named file.

`BinHex.Decode filename encodedData`
decodes data which follows the `filename`, and writes the decoded data to the named file.

`BinHex.Decode filename binHexName`
decodes the `binHexName` file, and writes the decoded data to the named file.

UnZip tool

UnZip can be used to unpack zip archive files. Files with a `.zip` file name extension can be opened directly with `Desktops.OpenDoc`.

`UnZip.Directory Archive.zip`
opens a text viewer listing the names of the files in an archive.

`UnZip.Open Archive.zip filename`
unpacks a document and opens it. No file is written to disk.

`UnZip.OpenAscii Archive.zip filename`
unpacks an Ascii text and opens it.

`UnZip.Extract Archive.zip namelist`
unpacks all the files named in the list from an archive.

`UnZip.ExtractAll Archive.zip`
unpacks all the files from an archive.

OldFiles tool

OldFiles can be used to read files from older Oberon installations (DOS and Win32s) using MS–DOS 8.3 file names. This tool is not available for PC Native Oberon.

`OldFiles.SetPath path`
sets the path for subsequent OldFiles commands.

`OldFiles.Directory`
`OldFiles.Directory \d`
displays the selection of all disk files found in the current path, previously set by an `OldFiles.SetPath` command. If the option `\d` is specified, additional information about file date and time and file sizes is displayed.

`OldFiles.ReadFile {A => B}`
processes a parameter list of pairs A => B reading each file A in the current path, previously set by an `OldFiles.SetPath` command, and copying it to the current directory under the name B.

`OldFiles.ReadFiles namelist`
reads the named files in the current path, previously set by an `OldFiles.SetPath` command, and copies them to the current directory under the same name.

`OldFiles.DeleteFiles namelist`
deletes the named files in the current path, previously set by an `OldFiles.SetPath` command.

FileUtils tool

FileUtils facilitates the backup of a load of files without naming them each individually, as must be done with the other backup tools. This tool is not available for PC Native Oberon.

`FileUtils.Backup {\pattern} srcDir destDir`
copies the files found in `srcDir` to `destDir`, ignoring those files with a name matching the patterns appearing in an option list. Each pattern must be a string preceded by a "\".

`FileUtils.FindSame [\S] {directory}`
searches all the duplicate file names in the listed directories. When the option `\S` is specified, the search is extended to the Oberon search path specified in the [System] section of the registry.

Chapter Three

The Gadgets User Interface

3.1 Introduction

The previous chapter presented the textual user interface of the Oberon system. Although the text interface is an important part of the Oberon system, it is by no means its only component. In Oberon, text is an example of one *component* in a large collection of components. Components, or *objects* as they are also called, build the run–time environment of the Oberon system. The components of Oberon are part of a component framework called *Gadgets.* In this framework, the components themselves are called *gadgets* too. The Oberon user uses gadgets to write documents and to create applications. The Oberon programmer manipulates already built gadgets under program control, or creates new gadgets by writing Oberon modules.

Gadgets cover a large spectrum. Most of them are of a visual nature and are seen on the display. Examples are Buttons, Scrollbars, Panels, TextGadgets, menu bars etc. When a *visual gadget* is visible on the display, we say that it is located in the *display space.* The display space is the data structure that forms your Oberon display. When a visual gadget is not located in the display space, we say that it is *off screen*. In contrast to the visual gadgets, the *non–visual* gadgets operate behind the scenes to manipulate and store information. The latter type of gadget we call *model gadget* or *model*, for short. It is possible to make the hidden information of a model gadget visible by attaching it to a visual gadget, and by placing the latter in the display space. In such a setup, the visual gadget has to visualize the model gadget attached to it. More examples of gadgets and how to configure them will be given in the following section.

Gadgets are examples of *persistent objects*. We can store them in files and transfer copies of them across the network to other machines. In addition, gadgets are *universal.* This means that they can be used wherever required. For example, a gadget belonging logically to a drawing application (like a circle), can be inserted into a text, a page layout program, a Panel, another graphical editor, and can float anywhere on the display. In this sense, a gadget does not belong to a certain application but to the system as a whole; it is a first class citizen. In fact, the whole concept of an application is a little fuzzy in Oberon. As gadgets are shared between everybody, it is nothing magic to take an application belonging to somebody else, extract some gadgets that you require from it, and use them in your own application.

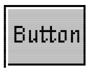
To complete the picture, Oberon does not distinguish between applications and document files. A file, like a text document, consists of persistent objects. An application consists of objects too (perhaps ones that really do something). In Oberon, the distinction between application and document vanishes. Opening a document is the same as starting an application, and starting an application is the same as opening a document. Mixing application aspects and document aspects is allowed as well; imagine having a working application inside an e–mail. Clearly this could be confusing if we do not clear up the terminology as early as possible. In future, we will refer loosely to an application as a collection of objects (perhaps divided between multiple documents) that does something approximately the same (for example, a bitmap or font editor application). A document is a persistent collection of gadgets (objects or components, if you want). The normal case is for a document to be stored in a file, although in some cases it is generated on the fly from information located somewhere else. For example, world–wide web (WWW) pages and file transfer servers (FTP) are also documents (active ones!)

in the Oberon world.

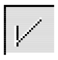
3.2 A Gadget Classification

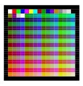
As mentioned earlier, gadgets are classified into visual and non–visual gadgets. This section will give an informal overview of the gadget classification.

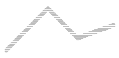
The *visual gadgets* are further divided into elementary, container, document and camera view gadgets. As the name indicates, a visual gadget is something you see on your Oberon display. The most abundant are *elementary gadgets.* They do not contain any further gadgets themselves and thus act as *leaf* or *terminal* gadgets in the display space. Examples of elementary gadgets are:

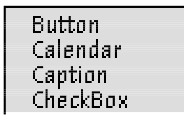


A *Button* for activating Oberon commands

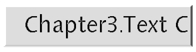

A *CheckBox* for checking an item


A *ColorPicker* for changing the color of gadgets or text

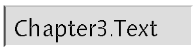

A *Line* for drawing a graphic



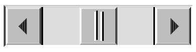
A *ListGadget* for displaying a list of items



A *NamePlate* for giving a name to a document



A *TextField* for entering short strings or numbers


A *Scrollbar* for entering a numerical value

Container gadgets, just as the name indicates, contain other visual gadgets as *direct descendants* or *children*. Containers can be nested in each other. Thus, a child in turn might have further children (making them indirect descendants). The container is called the *parent* or *direct context* of its descendants. Although each child gadget is able to edit itself independently, the container provides additional editing functionality for groups of children. Such groups are identified by *selecting* the gadgets that belong to the group. By convention, once a gadget has been selected, the container assumes all further editing responsibility for the selected group. Selected gadgets are identified by a white semi–transparent selection pattern that covers them. The selected gadgets, or more precisely, the most recently selected gadgets, are often the targets of further processing by Oberon commands. The two most important containers are the TextGadget and the Panel gadgets.

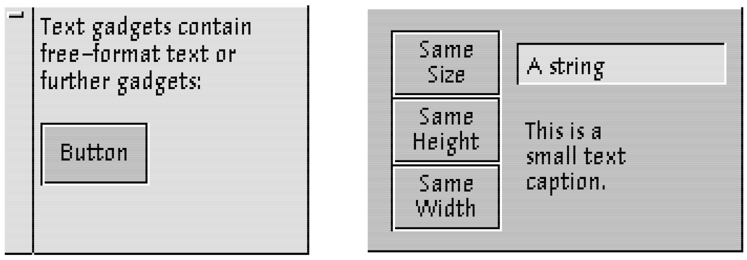

Text gadgets contain free–format text or further gadgets:
Button
Same Size
Same Height
Same Width
A string
This is a small text caption.

Figure 3.1 A TextGadget and a Panel

The prototype of a container is a *Panel*, a rectangular surface containing other visual gadgets. A Panel supports operations on groups of children plus the layout and alignment of children. The children of a Panel are organized in a priority sequence and may overlap each other. A child keeps its priority when moved around in a Panel and newly inserted gadgets are always placed in front. Oberon commands put a gadget behind or in front of other gadgets. An

Organizer is an extension of Panel which imposes constraints on the placement of its components.

A TextGadget visualizes a text and implements a text editor. Also a TextGadget is a container and visual gadgets may float inside the text stream.

Model gadgets represent another class of gadgets. Model gadgets contain data values useful to Oberon applications. The Gadgets system provides a set of model gadgets that can store the basic types like INTEGER, REAL, BOOLEAN, SET and *string* of the Oberon language. They are simple in structure and behavior and cannot display themselves on the display. To visualize them, *visual* or *view* gadgets from the elementary and from the container classes introduced before are used. This way of arranging things is called the Smalltalk *Model–Viewer–Controller* framework (MVC). The Oberon system also uses the MVC concept, although in a slightly modified form: the viewer and the controller are united in a single class *Frames*. In the MVC framework, the model gadget contains the data displayed by the view gadget. Many view gadgets can show the value of the same model gadget. Each view ensures that it remains consistent with the model it represents. We say that the view is *linked* to a specific model gadget. An example is a gadget constellation where a TextField and a Scrollbar, acting as view gadgets, are both linked to the same Integer model gadget. Behind the scenes, things are organized as in Figure 3.2.

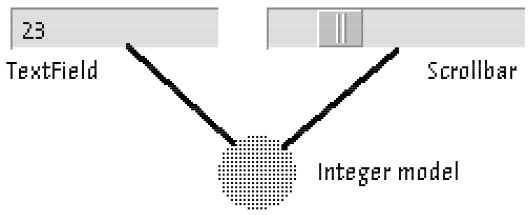


Figure 3.2 Two visual gadgets linked to a model gadget

Changing the position of the Scrollbar causes the Integer gadget to be updated accordingly, which in turn causes the TextField to be informed that the value of its model has changed. The idea with the model gadget is that it can easily be manipulated by an Oberon program without regard to how it is visualized. The visualization of the model is updated accordingly without the application knowing what or how many visualizations exist. This allows application programmers to insulate code against changes made to a user interface. Interestingly, many visual gadgets are programmed in such a way that they can also function without a model gadget linked to them. For example, the Scrollbars and TextFields work just as well without models. This fact gives some insight on how models and views are implemented. Both the model and the view contain the data to be represented, often in a format convenient for the gadget (for example, a TextField remembers strings whereas an Integer gadget stores INTEGERs). When one value changes, a communication protocol between the model and the view ensures that the values, even though of different but compatible formats, are made consistent with one another. No updating takes place when the formats are incompatible.

The Gadgets system provides only a limited set of model gadgets; typically application programmers add their own model gadgets to the system depending on their needs. For example, it is sometimes useful to construct *compound models* that store more than one data value The Complex gadget is an example.

Another variant of the Model–View–Controller framework is possible in the Gadgets system. Often you want not only to share the same model gadget, but also to have two different *camera views* on the same displayable gadget. Imagine you are editing a large drawing and you would like to see two different parts of the same drawing at the same time. Again, the structure behind the scenes is similar to the one above, but in this case it can be seen directly on the display (Figure 3.3).

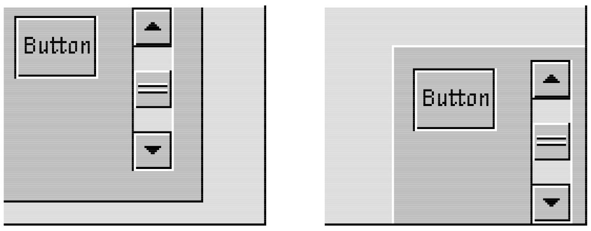


Figure 3.3 Two camera views of the same Panel

Here we have two different camera views onto the same Panel containing a single Button and a Scrollbar. Changing the Panel in one view causes it to be updated in the other as well. This is useful when you want to share the same gadget between different applications. It is also the way icons are shared between applications in the system.

There are many more gadgets available in the Gadgets component framework. The interested reader is invited to browse through their descriptions in Chapter 4.

3.3 Composing gadgets interactively with the mouse

One of the interesting and novel aspects of gadgets is that they can be changed in size and position, in addition to being used, from creation onwards until they are eventually explicitly locked by the user. This is in contrast to other systems where the user receives his user interface locked, and must "take it or leave it". A significant part of the time spent by a user of the Gadgets system is in organizing and building new user interfaces. This can be compared to the user adjusting his tool texts under the textual Oberon user interface. In this section we explain how interactive composition of gadgets is done at run–time.

All visual gadgets have a certain size or extent that they occupy on the display. When the mouse is located inside this area, the gadget can do whatever it pleases depending on how it was programmed by its creator. Fortunately, gadget mouse commands often stay close to the Oberon conventions (in the Table 3.4, mouse events are indicated by the first (mouse) key pressed, followed by a "+" sign to indicate that an interclick follows).

Table 3.4 The Oberon mouse conventions

Key		**Associated action**
ML	Point	Set the caret to mark the insertion point
ML + MM	Copy to	Set the caret and copy an existing selection to the caret
MM	Activate	Activate command in text or in *Cmd* attribute of gadget. Also manipulate gadget.
MR	Select	Select text/gadget or group of gadgets
MR + ML	Delete	Select text/gadget and delete
MR + MM	Copy over	Select text/gadget and copy over to caret
ML + MM + MR	Nullify	Nullify current mouse action

By "selection" we mean either a text selection (the selected text stretch) or a gadget selection (the selected gadget or group of gadgets). The system keeps track of the gadget and text selection separately and the mouse keys interclick uses one or the other depending on the context. For example, while working with Panels, the gadget selection is used, and while working with texts or strings, the text selection is used. However, in some cases the most recent gadget selection or text selection is used. Placing the caret in a gadget is called *focusing* a gadget. There is only one focus active at any one time.

Some gadgets support only a subset of mouse commands. For example, often a gadget does not support a caret, in which case mouse commands starting on the ML key are ignored.

Most gadgets have inner areas reserved for control. These areas can be used to resize the gadget or move it from one container to another on the display. The control areas may be absent if the gadget cannot be moved or resized. Most visual gadgets are *rectangular* in shape, and for those gadgets the control area is a border a few pixels wide *inside the edge* of the gadget. Inside the gadget, but not in the control area, the gadget does whatever it is supposed to do (Buttons can be pushed, CheckBoxes checked, etc). In the control area, the gadget responds to the control mouse combinations listed in Table 3.5.

Table 3.5 Mouse commands in the control area of a gadget

Key	**Associated action**
ML	Uninterpreted
MM (drag in a corner)	Resize gadget
MM (drag on a side)	Move gadget around in current container
MM + ML (drag on a side)	Move gadget and insert in other container (consume shallow copy)
MM + MR (drag on a side) mouse	Move gadget and insert a copy at position (shallow copy)
MR	Select gadget
MR + ML	Select gadget and delete
MR + MM	Select gadget and copy over to caret

Gadgets can be picked up on a side with the MM key and moved from one position to another on the display. Movement with the MM key alone is restricted to the current container, and may be rejected when attempting to move the gadget completely out of its current context. Movement to another context can be done only with an explicit *consume interclick* or a *copy interclick*. A consume interclick initiates a *consume* operation, often called *drag–and–drop* in popular terminology. The new container *consumes* the gadget and removes it from its old container. Not only is a consume operation used to move a child object from one container to another, but also to provide a more general drag–and–drop facility. The consumer (called the *receiver* or *recipient*) may interpret the consume event in different ways: it may either absorb the consumee (which is in turn called the *sender* or *initiator*) as a descendant (like most containers do) or it may initiate some other event. For example, a compiler Icon may compile source text Icons that it consumes, or a trash can Icon may delete the file gadget Icons that it consumes. Some receivers may execute a user–defined Oberon command on a consume event. The Icon gadget is such a receiver. The consume interclick can also be used to bring a gadget to the front in a Panel. Whereas a consume interclick moves a gadget to a new context, a copy inserts a copy in the new context, leaving the original at its place. In both cases we may say that a gadget is dropped into another location.

The MR combinations are the same as those listed in the Table 3.4.

By default, most gadgets are freely editable. Mechanisms exist that allow the user to *lock* gadgets and thus prevent the user from inadvertently changing a gadget.

Though most visual gadgets are rectangular in shape, that is fill their bounding box completely, a few others have an irregular shape. These are called *transparent gadgets* because sub–sections of their bounding box are transparent and show what lies below them. This is particularly useful for drawing geometric figures. The gadgets Circle, Line, Rectangle, Rectangle3D and Spline fall in this category.

3.4 Creating new gadgets

To help users compose user interfaces interactively, a `Gadgets.Panel` is provided. The command `Desktops.OpenDoc Gadgets.Panel` opens a `Gadgets.Panel` (Figure 3.6).

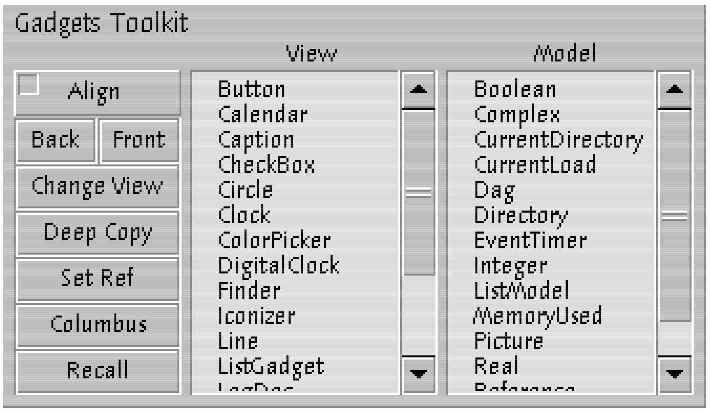


Figure 3.6 The Gadgets.Panel

The two ListGadgets contain the set of visual (View) and model gadgets (Model) delivered with your Oberon system. An MM click on one of the entries in the *View* list, inserts a visual gadget of that type at the caret. The caret can be located either in a text or inside a Panel (where it shows up as a small cross). Clicking on an entry in the *Model* list, links a new model gadget of that type to the current gadget selection.

The *Align* Iconizer of the `Gadgets.Panel` allows you to control the layout and the alignment of the selected gadgets in containers. It is usually used as a pop–up menu but it can also be flipped with an MM key click on the switch pin located in the top left corner.

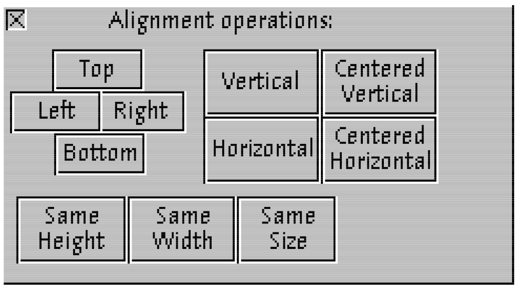


Figure 3.7 The Align menu of the Gadgets.Panel

Alignment normally takes place relative to some imaginary line. For example, *Left* alignment means that all selected gadgets must be lined up on their left edges, the reference edge being that of the left most gadget. When gadgets are being sized, the height or the width (or both) of the largest gadget is applied to all gadgets. Gadgets floating in the text of a TextDoc, a TextGadget, a TextNote or a LogDoc cannot be aligned, but sizing is possible. The commands controlling the alignment and sizing operations are documented in the description of the Panel gadget in Chapter 4.

The Buttons `[Back]` and `[Front]` change the *display priority* of the selected gadgets. By display priority is meant which gadgets overlap others in a Panel or in an Organizer. New gadgets are always inserted in front of other gadgets in a container, and they keep their priority until it is explicitly changed. One exception to this is the document gadgets presented later in this chapter; these pop to the front when they are focused (with an ML click). To bring a gadget to the front of the container, the consume interclick can be used as a shortcut. The mouse focus must however remain in the same container, otherwise the gadget may change containers unexpectedly.

Activating the `[ChangeView]` Button changes (or transforms) the selected gadgets into a new type selected from the View list. For example, Buttons may be changed to Checkboxes, Circles to Lines, though not every transformation is meaningful.

Copying a gadget can be done in two different ways. A *deep copy* is used when both the view and the linked model are copied. A *shallow copy* means copy only the view gadget and thus have the copy display the *same* model as the original. Shallow copies are always made directly with either a copy interclick or a consume interclick introduced in the previous section.

A deep copy has to be made explicitly with the `[DeepCopy]` Button located in the `Gadgets.Panel`. The Deep Copy command takes the selection, makes a deep copy of it, and inserts it at the caret. Deep copies are structure–preserving, which means that an exact duplicate of the gadget data structure is made.

The `[SetRef]` Button instantiates a gadget of a type appearing in the selection and places a Reference to this gadget in the marked (or in the selected) RefFrame. The practical advantage of a RefFrame is that it contains a Reference to an object irrespective of its type (visual gadget or non–visual) and of its size. The Reference can be dragged–and–dropped or copied over to another context just as easily as the object it represents.

Activating the `[Columbus]` Button opens a Columbus inspection tool. It is discussed in section 3.6.

Finally, the `[Recall]` Button recalls the gadgets deleted most recently from a Panel or an Organizer and inserts them at the caret. This operation can be assimilated to a paste operation after a cut operation, and it may be repeated any number of times.

3.5 Attributes

Gadgets have attributes that customize their behavior. Each attribute consists of an *attribute name* and an *attribute value* pair. Attributes are typically used to specify colors, captions, and commands that gadgets should execute. For example, Buttons have an attribute called *Caption* that contains the caption string that is displayed inside it. A tool called *Columbus* is used to inspect and edit the attributes of a gadget. Attributes are *typed*; that is, the attribute type can be enforced by the gadget to be a *boolean*, *integer*, *real* or *string* type. Each gadget class defines its own set of attributes. In the remainder of this section, we introduce the common attributes of gadgets.

Although most gadgets have different attributes, all of them have a common attribute: the *Name* attribute, which is one of their most important attributes. The name is used to identify or to find a gadget. Gadgets often use names to refer to one another's attributes. In general, we refer to the attributes of gadgets with the following syntax:

```
ObjectName.AttributeName
```

`ObjectName` is the name of a gadget and `AttributeName` the name of an attribute of that gadget. The Gadgets system employs a search strategy for locating an object with a certain name. The scope in which the system searches for gadgets is determined by the hierarchy of container gadgets. The current scope is determined when a gadget executes a command (more precisely a command attribute), and is exactly the container (or parent) in which the gadget is located. Behind the scenes, a special *message* is sent to the parent to search for the named object. This message propagates in a breadth–first fashion from the container to all children until the named object is found.

A few visual gadgets which can function with a model gadget, can indicate which component or *field* of the model gadget they are interested in. These gadgets have a *Field* attribute in which the name of the model gadget's attribute must be stored. It is thus quite possible to build a model gadget of a compound nature (the Complex gadget for example) which can be visualized by several different visual gadgets, each of which is displaying a different attribute of the model it is linked to. If the *Field* attribute contains the empty string, the name "Value" is assumed and the *Value* attribute of the model is visualized.

Other commonly appearing attributes are the *command attributes.* They play a special role when combining gadgets together in a user interface. The command attributes specify what action should be taken when the gadget is activated. The action is specified in the form of a command attribute string containing an Oberon command to be executed. The command specifies a module and procedure name to be invoked and may optionally pass parameters to the command. Command attributes have the same syntax as the familiar Oberon commands. Two different command attributes are particularly often used. The first one, named *Cmd* attribute, contains the command to be executed when the gadget is activated (with an MM key click). For example, the command

```
Desktops.OpenDoc Test.Text
```

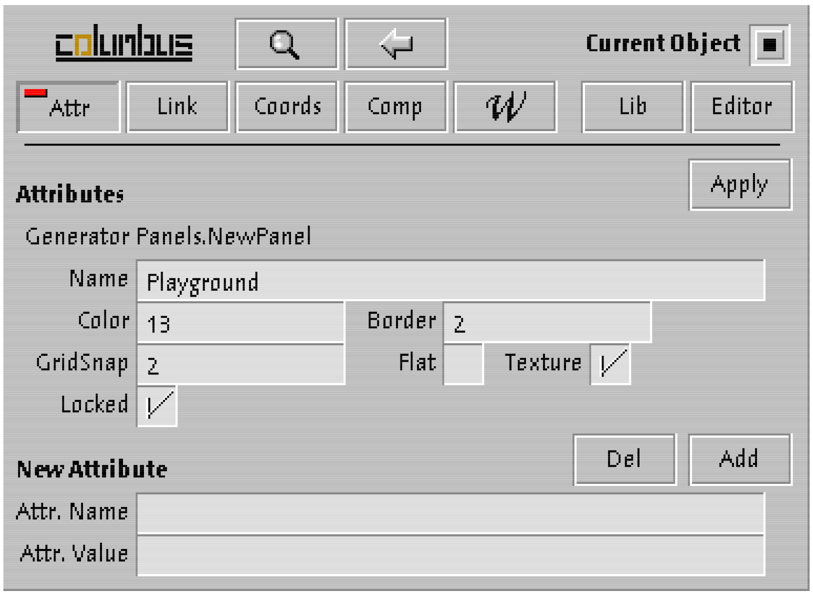
assigned to the *Cmd* attribute of a Button, will open the file `Test.Text` when the Button is activated. The second one, named *ConsumeCmd*, contains the command to be executed when another gadget is dropped into a gadget which has such an attribute.

The string assigned to a command attribute may contain *macro characters* that allow other attributes to be accessed, selections to be processed, parameters to be edited, and drag–and–drop operations to be controlled. These macros are used to combine different gadgets together and have enough power to build graphical interfaces for text–based Oberon applications. Macro characters are presented in section 3.8.

Another attribute that is frequently used is the *Locked* attribute, which allows you to freeze user interfaces. Often you need to restrict the editability of the documents you create. By editing is meant changing the position or size of the gadgets in a Panel or other container. The editability of a gadget is determined by two things. First, the gadget itself might be programmed in such a way that it cannot be moved or resized. This is then a deliberate and inherent restriction determined by the programmer of the gadget. Second, the context or container of a gadget can determine if that gadget is locked or not. This implies that when a gadget moves from one context to another, its editing behavior changes depending on its container. A parent can lock or unlock *all* its *direct* descendants at once; it cannot selectively lock only some of them. The locked flag affects only the *direct* children of a container; you have to lock nested containers yourself. Panel, TextGadget, Iconizer, NoteBook, Organizer, etc. are examples of lockable gadgets.

3.6 Columbus

Columbus is a very versatile instrument that can be applied to any gadget (visual or model) for an inspection of its identity (generator) and where applicable, of its attributes, links, components and other properties such as the size and relative position in a container. When applied to a specific gadget, the tool adjusts its shape, to represent an attribute form for this gadget. Note that this form is a document created by a program. The tool can also be used to manage public libaries and their content.



Attributes

Generator Panels.NewPanel

Name	Playground				
Color	13	Border	2		
GridSnap	2	Flat		Texture	✓
Locked	✓				

New Attribute

Attr. Name	
Attr. Value	

Figure 3.8 A Columbus Attributes view

Columbus presents itself in *two* different "variable geometry" panel documents (they always fit in the system track). One of them, the Columbus Panel, is used to inspect and configure the state of objects (Figure 3.8). The other one, called "Libraries.Panel", is used for manipulating public libraries (see section 3.11). The Columbus Panel is opened with the command:

```
Columbus.Inspect
```

or by activating the `[Columbus]` Button in the `Gadgets.Panel`.

What you see in the Figure 3.8 is the *Attributes view* of the Panel being inspected; this is evidenced by the little horizontal bar (representing a red LED) in the `[Attr]` Button. Several other object views can be accessed via the other Buttons which, with the exception of the `[W]` Button, function as radio buttons.

The value of a boolean attribute is visualized by a CheckBox, whereas the value of an integer is visualized by a short TextField and that of a string by a long TextField. An attribute name may have up to 32 characters.

3.6.1 Object views

`[Attr]` *Attributes view*
This view displays the attributes of the inspected object with their values in the **Attributes** section. It is the *preferred view* which is always presented first when a new *object* is inspected. However, for a library (`*.Lib`), the library view is presented first. You can change the attribute values and apply them to the object by clicking the `[Apply]` Button on the right.

The **New Attribute** section is used for adding an attribute to the current object. It features two TextFields: in the first one, enter an attribute's name, in the second one, its value. To add the attribute activate the `[Add]` Button. An added attribute can be removed with the `[Del]` Button.

This view is the starting point for an organized or an impromptu inspection tour of the current object with excursions to other related objects or libraries. Every step is recorded in a history stack, making it possible to regress step by step by activating the Button. On returning to the starting point, the Button disappears. An MM click on the **Current Object** reference at the top right, will project this view again. This can be used as a shortcut to return immediately to the Attributes view.

`[Link]` *Links view*
This view shows the links of the inspected object in the **Links** section. You can change the object links and apply them to the object with the `[Apply]` Button on the right side.

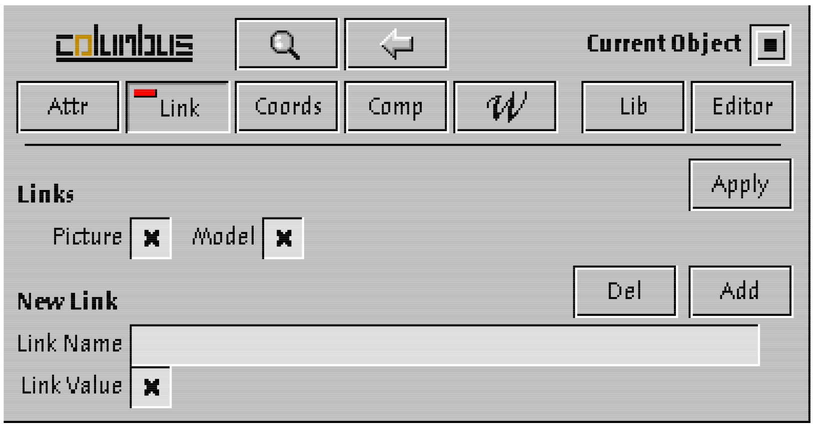
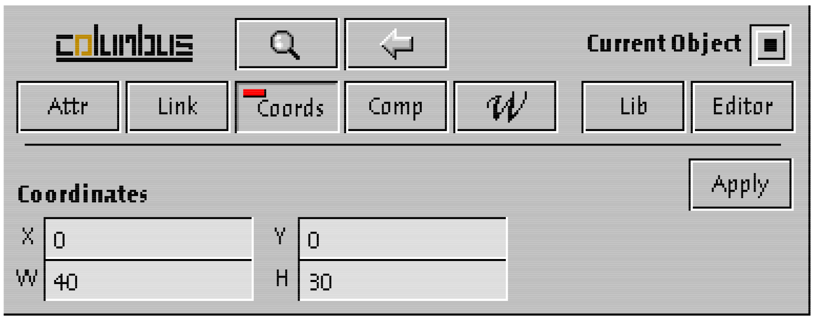


Figure 3.9   A Columbus Links view

The **New Link** section is used for adding a link to the current object. It features a TextField for specifying a link's name, and an empty visual reference where you drop the object to be linked. To add the link activate the `[Add]` Button. An added link can be removed with the `[Del]` Button.

`[Coords]` *Coordinates view*
This Button is present *only* when a *visual* gadget is being inspected. This view shows the X and Y coordinates of a visual gadget with respect to the upper left corner of its container, and its width W and height H, all measured in pixels.



X 0   Y 0
W 40   H 30

Figure 3.10   A Columbus Coordinates view

`[Comp]` *Components view*
This Button is present *only* when a *container* such as NoteBook, Organizer or Panel, is being inspected and when such a container effectively contains a component. This view lists the components of the container gadget with their generator procedure and a visual reference. You may inspect each reference –– with an MM key click. You may drag a reference away to insert the referenced object in other documents, but you can neither change nor delete such a visual reference.

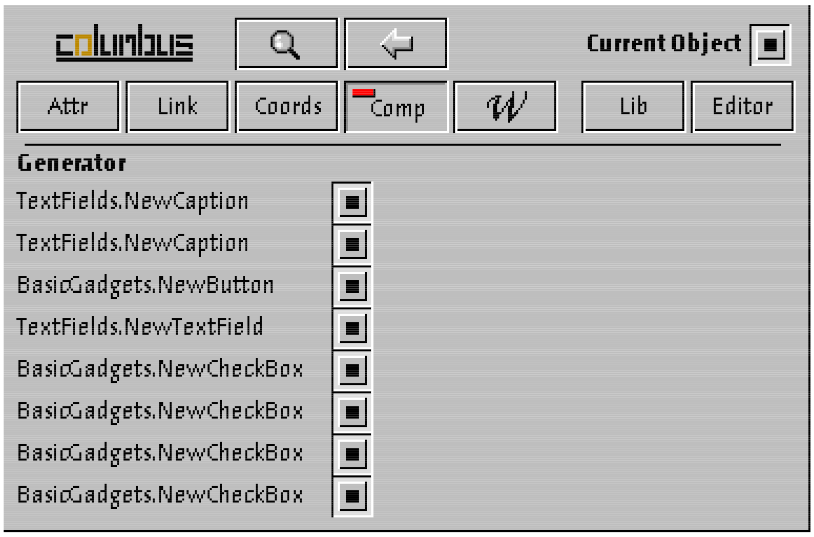


Figure 3.11 A Columbus Components view

W – *call Watson*

Columbus gives you the opportunity to see additional descriptions of the inspected object. These are definition files or module files. Activating this Button calls `Watson.ShowObj` with the object's generator name as parameter.

`[Lib]` – *library view*

This Button is present *only* when the object inspected is bound to a *public* or a *private* library (see section 3.11). Obviously enough, when a library is inspected, the library view is the preferred view which is always presented first. The view lists the objects contained in the library. The generator procedure, the reference number, and in the case of a public library, the name in the index are displayed for each object, together with a visual Reference. Each object can be inspected by MM clicking its visual Reference: an *Attributes view* of the object is presented. The name of the inspected public library is displayed in the top right corner. In the case of a private library, the text is simply "(Private)". The object which was being inspected when the library view was called is easily spotted: look for the blue text line. If there is no current object, no line is highlighted.

`[Editor]` – *library editor*

This Button is present *only* when the object inspected is bound to a *public* library. With the library editor you can unload the library from memory, cleanup freed objects from the library, store the library and perform many other maintenance tasks in the same way as with the `LibrariesPanel` described in section 3.11.

3.7 Inspecting Module Definitions with Watson

During software development, you will often need to refer to the definitions of modules delivered with the Oberon system. A *definition module* describes a module interface and is a summary of the complete module. It contains the declaration of the exported names which may be used by other modules. It is also known as the *public view* of the module and its advantage is its textual compactness. With this clearly defined module interface, a module can be used without knowledge of how it is implemented.

In Oberon, it is not necessary to write down the definition of a module: Watson takes care of extracting the best information available about a specified module. For example, should the source text of a module be available, Watson can scan it for so–called *exported comments* which it presents to the user together with conventional module interface. An exported comment is a program comment starting with a double "*": (** .... *). The comment typically contains more information about how to use specific module features. As the source code of a module is sometimes not available to the user, Watson effectively goes in search of a previously generated definition file (with a `.Def` extension) first. Should both the definition file and the module source be missing, Watson searches for the module symbol file. If the latter is missing too, Watson tries to extract information about the module from its object file. In each step of the Watson search strategy, the amount of information Watson finds is diminished. Watson also has some further tricks up its sleeve. To reduce the clutter of many definition files, several definition files may be compressed and packed into a single archive file (with a `.Arc` extension). Watson can automatically extract and decompress definitions from such an archive file.

Watson is controlled with a graphical interface accessed by executing the command `Desktops.OpenDoc Watson.Panel`. It features a number of Buttons and is controlled by the Setup Iconizer and by CheckBoxes appearing in the lower part. It is however just as easy to use the few commands that are associated with its control Buttons.

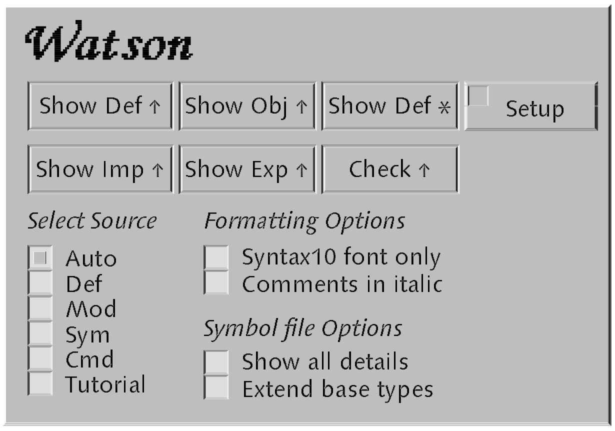


Figure 3.12 Watson Panel

Watson presents the information it finds as hypertext. This means that hyperlinks allow you to explore type structures and the imported modules directly by MM clicking on the text marked in blue.

3.7.1 Watson settings

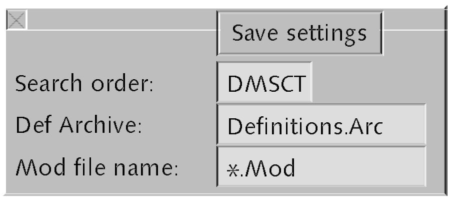

Save settings
Search order: DMSCT
Def Archive: Definitions.Arc
Mod file name: *.Mod

Figure 3.13 Watson Setup

The settings are defined on the reverse side of the Setup Iconizer. Experience has shown that the default settings provided with the system are quite adequate in most situations. If not, the settings can be modified and saved in `Watson.Lib` with the `[Save settings]` Button. The three TextFields have the following meaning:

Search order: contains a string of up to five capital letters which determines the order, from left to right, in which the various information sources are consulted for creating a definition text. The sources are:

D	an existing file (`.Def`) or a file contained in the archive specified in the TextField "Def Archive".
M	the text of a module in a module file or in a marked viewer.

S the symbol file (`.Sym`)
C the object file (`.Obj`). Only the module commands are shown. Equivalent to what is obtained with `System.ShowCommands`
T a tutorial text associated with the module.

The default order "DMSCT" may be changed and information sources not explicitely specified are ignored. The set order can also be overridden with the *Select Source* radio–buttons.

Def Archive: specifies the name of the archive (file) containing the definition texts. The module definitions of the entire Oberon system (but not of the archived applications) are delivered in the archive `Definitions.Arc`. This file may be extended with the definitions of custom developed modules. Alternatively, new separate archives may be created, but only one such archive can be searched.

Mod file name: specifies a module name filter of the form `[prefix.][*[.postfix]]` in which Watson will replace the *asterisk* by the name specified in the Watson command. If the system cannot find a file with the name `prefix.*.postfix` (or `prefix.*` if the postfix is omitted), it will drop the prefix and attempt to find a file named `name.postfix` (or respectively `name`). Without asterisk, Watson assumes a postfix ".Mod" and the module name filter becomes `prefix.name.Mod`. When the field is empty, the filter is `name.Mod`.

A number of Oberon modules have a prefix such as IDE in PC Native Oberon or Win, Win32 in Oberon for Windows. Watson can produce the definition module of almost any module on a Windows platform if this field contains "Win." or "Win.*.Mod".

Assuming that "Sample" is the parameter of a Watson command, these examples show which file names are searched:

Win.*.Mod2	=>	Win.Sample.Mod2	then	Sample.Mod2
Win.*	=>	Win.Sample	then	Sample
Win.	=>	Win.Sample.Mod	then	Sample.Mod
*.Mod3	=>	Sample.Mod3		
*	=>	Sample		
empty string	=>	Sample.Mod		

3.7.2 Select Source

When the radio–button "Auto" is checked (this is the default setting), the information sources search order applies, but it can be temporarily overridden with one of the remaining radio–buttons. The information source corresponding to the checked radio–button is then searched first, while the remaining sources specified in the search order follow in order.

3.7.3 Formatting Options

When a module text in a marked viewer or in a .Mod file is used to create a definition text, the created text may be formatted in three different forms:

– without alteration to the source module text,

– with all text appearing in Oberon10 font: *Oberon10 font only* is checked (command option `\p`). This option overrides the *Comments in italic* one. The Syntax10 font is used in the Windows, Linux and Mac implementations.

– with comments in italic (command option `\i`).

These two options are ignored, if a `.Def` file is found first.

3.7.4 Symbol file Options

When symbol information extracted from a symbol file or an object file is used to create a definition text, the created text may offer three different information

content:
- without addition,
- with all details (command option `\d`)
- with extended base types information (command option `\x`)

These two options are ignored, if a `.Def` or a `.Mod` file is found first.

3.7.5 Watson commands

The three Buttons in the first row in Figure 3.12 appear in all the different Oberon system implementations. Each of them can activate one of the commands described below:

`[ShowDef^]` activates `Watson.ShowDef ^`
`[ShowObj^]` activates `Watson.ShowObj ^`
`[ShowDef*]` activates `Watson.ShowDef *`

`Watson.ShowDef ([\options] module | ^)`
`Watson.ShowDef [\options] *`
opens a document viewer named "module.Def" displaying the definition of the named module. The name must be an Oberon name, of which only the first part is used. Watson attempts to open the information sources in a predefined search order and reports on the outcome of the search: if the requested information is found, the name of the source is listed in the system log, otherwise an error message "no information about ... available" appears. If the information source is the marked viewer (*), this marked text must be a valid Oberon program text. If it is not, the location of the error is listed in the log. The options belong to three different categories:

1 – the information sources identified by the capital letters D, M, S, C, T in the desired order. This order takes precedence over the search order defined in the Watson setup. Not all sources must be explicitely named – the remaining positions will be taken over from the setup.
Example:
　　Search order in Setup: DMSCT
　　Command parameter: \MS
　　Resulting search order: MSDCT

2 – the formatting options: d and x
3 – the symbol file options: i and p

`Watson.ShowObj ([\options] module.object | ^)`
opens a document viewer named "module.Def" displaying the definition of the selected object in the selected module. The parameter must contain a two–part qualified name, where `object` is the name of an object exported by `module`. If that object is not found, the entire module definition is displayed. The options are the same as those of `Watson.ShowDef`.

`Watson.MakeDefs ([\c] {modName} ~ | ^)`
`Watson.MakeDefs [\c] *`
creates a definition module (.Def) for each module file (.Mod) in the list. If the option `\c` is used, HTML document files (.Def.html) are created instead. If a matching definition or HTML file already exists, it is overridden. Such a definition module may be added to an archive of definitions (.Arc) at any time, if it is of some importance to your installation.

`Watson.ConvertDefs ({DefName} ~ | ^)`
converts each definition module (.Def) named in the list into a HTML document file (.Def.html). The definition modules to convert must be found in the system directory or in the archive file specified in the Watson panel settings. If a matching HTML file already exists, it is overridden.

features three additional Buttons (second row in Figure 3.12) which can each activate further commands:

```
[ShowImp^] activates Watson.ShowImports ^
[ShowExp^] activates Watson.ShowExports ^
[Check^]   activates Watson.Check ^
```

`Watson.ShowImports ([options] module | ^)`
opens a viewer named "module.Imp" displaying a map of all the modules imported by the named module, and of all the exported objects in those modules which are effectively imported. This "uses what?" request may take some time to complete. Only the first part of a qualified module name is used.

`Watson.ShowExports ([options] module | ^)`
opens a viewer named "module.Def" displaying the definition of the named module, in the same fashion as `[ShowDef^]`, but the objects exported by the selected module which are effectively used in other modules are shown in green. When the MM key is pressed on such a green spot, a list of modules using the object is presented. Selecting one of the module with the mouse cursor and releasing the MM key opens a viewer displaying the module source text. This "where used?" request may take some time to complete. Only the first part of the qualified module name is used.

`Watson.Check ({module[*]} ~ | ^)`
`Watson.Check all`
opens a viewer named "Check.Out" containing an enumeration of the module names which can be loaded. Each module name appears in red and an MM key click on it shows its import map (`Watson.ShowImports`). For each module that cannot be loaded, a message "module not found" is displayed in the system log. The objective is to check the consistency of the named modules or of all modules. Only the first part of qualified module names is used.

For these implementations which all use a directory structure, the module files and the archive file to use must be found in the the current directory or in the system search path. The meaning of "S" in the search order is then for these implementations:

S	the object file (.Obj) is searched

and not the symbol file anymore.
The formatting options in those environments refer to Syntax10 which is the default font at installation time.

3.8 The Command Macros

Command macros act as glue between gadgets in a user interface. Typically, when activated, a gadget executes an Oberon command with parameters obtained from the selection, from the attributes of a gadget itself, or from the attributes of a gadget located in the same context. A few macro symbols in a command attribute direct the parameter gathering task. The macros are first expanded completely and the resulting text is passed to the executed procedure in the normal Oberon fashion. This makes it possible to add user interfaces to existing text–based Oberon applications. However, in the realm of the Gadgets interface, some of this overhead can be avoided by using the special "%" character right in front of a command. This character signals that the command is parameterless (parameter scanning is not required), and the called procedure is responsible for collecting additional information from the environment. A short description of the predefined macros of the Gadgets system follows:

Activator macro #

The activator macro returns an attribute value of the gadget that is executing the command attribute. It is used in the form `#AttrName`, where AttrName is the name of the attribute. For example, the ListGadget has an attribute *Point* that contains the item that is clicked upon with the *MM* key (provided the gadget is linked to a valid list model). Setting the *Cmd* attribute of the ListGadget to:

```
    Desktops.OpenDoc #Point
```

will open the document whose name is pointed at in the ListGadget.

Lookup macro &

The lookup macro is useful when you want to pass a parameter to a command, where the value of the parameter is the attribute value of a certain gadget. The macro has the syntax `&ObjName.AttrName`. On expansion it searches for the object named `ObjName` inside the *current context*, extracts its `AttrName` attribute value, and inserts this value into the command string. Take the case of a simple application interface Panel with two components: a Button and a TextField, which thus appear in the same context. Assume that you wish to open a text file, whose name will appear in the TextField, by activating the Button. To implement this, you first have to name the TextField, say *TF*, using Columbus. Then, the *Cmd* attribute of the Button is set to:

```
  Desktops.OpenDoc &TF.Value
```

A double && instructs the substitution mechanism to look for the object in the context of the context and each additional & gives access to the next higher level context.

Substitute macro '

The substitute macro expands to a double quote character ". It must be used where a double quote would inhibit the substitution mechanism. An example is given below.

Initiator macro !

When objects are consumed by another object, those being consumed are called initiators or senders, and the consuming object is called the recipient or receiver. This macro is used in the *ConsumeCmd* attribute of a visual gadget, and gives access to the attribute values of the initiators. The consume command is executed by the recipient. The macro takes the form `!AttrName` and expands to a list of attribute values of each object in the group of initiators. Suppose that a source text file is represented by an Icon, and that a *Filename* attribute is added to the Icon. Then, the name of the file is assigned to this new attribute. Finally, the following string is assigned to the *ConsumeCmd* attribute of another Icon representing the compiler:

```
 Compiler.Compile !Filename
```

Now, by dropping one or several text file Icons into the *compiler* Icon, the compiler will compile each file named in the list resulting from the expansion of the parameter.

Selection macro ↑

The selection macro expands to the current selection. The macro `^AttrName` expands into a list of attribute values of all the selected objects. When no gadget is selected, the text selection is used. The selection returned always contains *at least one whole word*; if the beginning of a word is selected, the whole word is returned as the selection. If no selection is active, the selection macro expands to itself, the character "↑ ".

Most macros behave in a way that may seem strange at first. Macros are identified by a special first symbol (!, &, etc.) followed by a parameter. While expanding macros, the system needs to know when to terminate parsing the parameter: this is either when no characters are left to be scanned (end of string) or when a space is reached. However, when a space is reached first, the space is *not* included in the expanded macro. If you want a space to be included in the expanded text, you have to explicitly insert another space after the first. This feature allows you to concatenate macros. For example, if O1.Value is "Hello" and O2.Value is "World", then:

`'&O1.Value &O2.Value'` (one space)

will expand to "`HelloWorld`", while:

`'&O1.Value  &O2.Value'` (two spaces)

will expand to "`Hello World`".

User defined macros

Programmers can add their own macros by identifying special symbols for macros and by writing handlers for these new macro symbols. The mechanism is implemented in the `Attributes` module.

```
    MODULE Attributes;

        MacroHandler =
        PROCEDURE (ch: CHAR; VAR T: Reader; VAR res: Texts.Text; VAR beg: 
LONGINT);

        PROCEDURE AddMacro(ch: CHAR; handler: MacroHandler);
```

A new macro symbol is registered by calling the procedure `AddMacro`. The handler is called when character `ch` is read using the reader/scanner, and it must return:

`res`	the substitution text. If NIL, no substitution was made.
`beg`	the position in the text where reading/scanning must continue.

The macro might take parameters, that is, letters that follow immediately after the macro symbol and which must be read and interpreted by the macro handler using the passed Reader.

3.9 Composition Commands

So far, we have discussed how to create new gadgets and how to change their attributes using prepared user interfaces like the `Gadgets.Panel` and `Columbus`. There is also a textual interface with commands to perform the same tasks. These commands are provided for the sake of completeness and need not necessarily be used. The commands do however come in handy when creating your own user interfaces. We will discuss these commands in this section, but before that, we investigate how gadgets are instantiated.

A new instance of a gadget is created with an Oberon command. Let us assume that the module `M` contains a procedure `P` whose task it is to create a new instance of a certain object type. Here `M.P` is called the object's *new procedure* or *generator*. Executing the procedure `M.P` causes a new instance of that object type to be created and initialized to a default state so that it is ready to accept messages (i.e. it is completely functional). As the object often does not know what to do with itself after creation, another command is needed to display the new object (that is, if it is a displayable gadget). The module `Gadgets` provides a standard interface for instantiating and inserting objects at the caret. It has a command *Insert* with the following BNF syntax:

```
Gadgets.Insert ViewGenerator [ModelGenerator] ~
```

`ViewGenerator` denotes the generator of the visual (view) gadget to be instantiated. The `ModelGenerator` denotes the model gadget to be linked to the newly created gadget. The `Gadgets.Insert` command performs all that is needed to create the view and model behind the scenes. As an example, the following command creates a *Model–View* pair (consisting of a CheckBox linked to a Boolean) and inserts the CheckBox at the caret:

```
Gadgets.Insert BasicGadgets.NewCheckBox BasicGadgets.NewBoolean ~
```

We see from the generator procedure names what types of gadget are involved. A generator is typically named `M.NewType` where `NewType` is the type of the gadget. As mentioned before, many gadgets function both with and without models, which explains why the `ModelGenerator` parameter is optional.

If a visual gadget has been instantiated without model, it can be linked to a model with the command:

```
Gadgets.Link ModelGenerator
```

The gadgets contained in the gadget selection are linked to the model gadget of the type specified in the parameter. All the visual gadgets share then the same model gadget. The same command may be used to change the linked model.

There are many generator procedures (one for each gadget type). Remembering all of them can be difficult. The Gadgets system supports an aliasing feature which allows you to use shorter names than those of the generator procedures. It is for instance possible to rewrite the previous example as:

```
Gadgets.Insert CheckBox Boolean ~
```

The registry contains a section [Aliases] which determines these aliases. Refer to the Appendix A for more details on how to configure your Oberon system. Each alias must appear once in a text line having the following format:

```
AliasName=M.P
```

where `AliasName` identifies an alias for the *new generator procedure* `M.P`. The commands `Gadgets.Insert` and `Gadgets.Link` are the principal clients of aliasing.

The `Gadgets` module has a few more useful commands:

`Gadgets.ChangeAttr attrName attrValue ~`
sets the selected gadget's attribute. `attrValue` can take several forms, depending on the attribute type:

names	for string attributes
Yes/No	for boolean attributes
1234	for number attributes
"strings"	for string attributes

`Gadgets.Change ViewGenerator`
changes (or transforms) the selected gadgets into a new type identified by the `ViewGenerator` parameter. An alias may be used. This command is used by the `[Change View]` button in the `Gadgets.Panel`.

`Gadgets.Copy`
makes a structure–preserving copy of the selection and inserts it at the caret. This command is used by the `[Deep Copy]` button in the `Gadgets.Panel`.

`Gadgets.Set Obj.Attr value ~`

changes the value of the attribute (`Attr`) of the indicated gadget (`Obj`). The gadget must exist in the same context.

3.10 Documents and Desktops

A document is a named collection of objects. The name identifies the document and must be known to retrieve the document content (the object collection it contains). Most often, the document name is the name of the file in which the document content is stored. By specifying the document name as parameter to the `Desktops.OpenDoc` command, the document is located, prepared and displayed in a viewer. The `OpenDoc` command is applicable to documents of all classes. Typically, in the Oberon system, we use PanelDocs, TextDocs and bitmap documents – to mention a few. The fact that the `OpenDoc` command is applicable to all document classes hints to us that a document must be an object of some type. Indeed, we can request the document object to load itself, store itself, or print itself on a printer.

As documents are first class objects, we can imagine that they can do more complicated things than just loading their contents from a disk when they are opened. In fact, many documents "generate" their contents when required and may even require to contact a remote server to retrieve the information. Documents are sometimes also active; while a document is being used it might decide to update its contents. For example, we can imagine having an electronic mailbox on a remote server presenting itself as a document. In fact, Oberon regards the whole world as a source of documents. The network software included in Oberon allows you to open any document identified by the uniform resource locator (URL). As a result, file servers, discusssion groups, and World–Wide Web (WWW) pages all appear as documents in the Oberon system [Zel97]. This is why it is such a pleasure to browse between different information sources with Oberon.

In fact, a document is nothing more than an advanced visual container. We can communicate with this container and request it to load, store, print and so on. Being a container, a document can have any number of other gadgets as descendants. Currently, documents in the Oberon system support only a single child (although this is not a restriction). For example, a PanelDoc contains a single Panel as a child, and a TextDoc contains a single TextGadget as child. Note the difference between the Panel and its container, the PanelDoc. The Panel itself is not a document. If it were, nested Panels in the same user interface would end up in different files on disk.

In a way, a document can be regarded as a special type of gadget wrapper. It "wraps" additional functionality around its content. A document might also provide useful operations on the document content; the TextDocs provide a Search–and–Replace facility for text, for example. A document also provides for other functions such as constructing a menu bar or generating an Icon for the document.

Interestingly, being a true visual object itself, a document can be inserted into other containers. It is possible to insert a PanelDoc inside a TextDoc (for documentation purposes, for example). Opening the TextDoc will automatically load the contents of the PanelDoc it contains. In this way, the TextDoc will always contain the latest version of the PanelDoc, even if it is located across the network. This embedding feature of documents brings up the question how a document container should present itself on the display: if it is embedded in text, we would at least want it to look presentable when printed.

Therefore, it was decided to make documents "invisible". A document is always as large as its child and there is no way to distinguish a PanelDoc from the contained Panel. Most of the time, the user is however implicitly aware of the presence of a document. Its presence can also be verified by trying to select the document in question. Documents use a two–phase selection protocol. On the first try, the document content is selected in the normal white selection pattern. Selecting again selects the document in a blue selection pattern. A third selection attempt removes the selection completely. This allows us to select a document for adaptation by Columbus.

But where do the menu bar and the viewer of an open document come from if the document itself is invisible? This is the task of the `Desktops` module. After loading a document, the `Desktops` module creates a viewer for the document. The viewer consists of a menu bar (generated by the document itself for the `Desktops` module), and of the document content itself. This gives the desktop display system more freedom for creating appropriate viewers for a document. The Oberon system namely supports two different document viewing models concurrently. This is commonly referred to as the windowing model in other systems.

When Oberon was originally developed, it supported only the tiling organization of viewers. In that model, viewers share the screen area without obscuring each other partially. The tiling viewer system is the default viewer allocation strategy when starting Oberon. Later, with the development of the Gadgets framework, it became possible for visual objects to partially overlap each other. This resulted in the development of *desktops*. A desktop is a large surface embedded in a single viewer. Typically, different desktops are used for different tasks (they are containers). A desktop contains viewers overlapping each other together with other visual gadgets. It also ensures that a viewer pops to the front for use when it is focused with an ML key click inside it. Interestingly, desktops can be stored on disk, usually in a file with a `.Desk` name extension. This allows the user to remember useful configurations. The default desktop is called `Oberon.Desk`. In the current Oberon implementation, the tiling viewer system does not support this saving feature. A desktop does not have a menu bar. Instead, commands like Store, Grow, Copy etc. are provided as freely positionable Buttons inside the desktop itself. Like any other viewers, the desktop location is changed by grabbing it with the ML key in an unmarked area a few pixels high *along the top edge* of the desktop.

Providing two viewer systems concurrently creates the problem of deciding where to show freshly opened viewers. A simple heuristic solves the problem: when a viewer is opened, it is opened in the same viewing mode (tiled or in desktop) from which the command to open the viewer was executed. This ensures that once you start working in one model, viewers will be opened only in that very model. As usual, you can override this placement strategy by placing the star marker at the location where you want a viewer to appear.

After this introduction to the concept of documents and desktops of Oberon, we discuss in more detail the commands applicable to them.

3.10.1 Desktops Commands

The `Desktops` module is the controlling instance for all document and desktop–related tasks. Currently the desktop system supports both overlapping and tiled organization of viewers.

Before listing the `Desktops` commands, we first have to discuss how new documents are created. Being objects themselves, documents also have generator procedures. Calling a generator procedure causes an empty document of that type to be created. To distinguish between opening an existing document and creating a new one, the `Desktops.OpenDoc` command uses a special parameter syntax. Opening an existing document involves specifying its name. Creating a new document involves specifying its generator procedure (or its alias) in parentheses:

`Desktops.OpenDoc docname`	Open document called *docname*
`Desktops.OpenDoc (Generator)`	Open new document with *generator* as type

Instead of specifying a generator, it is also possible to open a new document by specifying a document name which does not already exist on the system (the search path is used). In that case the type of the new document is determined

by the name extension (see Table 3.14). By default, a TextDoc is opened.
Thus we can open a new PanelDoc with either of these two commands (the second uses an alias):

```
Desktops.OpenDoc (PanelDocs.NewDoc)
Desktops.OpenDoc (PanelDoc)
```

Sometimes, it is useful to change a document into a different class. This assumes that the new document class can understand the original document format. In such a case, specify both the document name followed by the wanted document type between parentheses:

```
Desktops.OpenDoc Main.html(TextDocs.NewDoc)
Desktops.OpenDoc System.Tool(PanelDocs.NewDoc)
```

In many cases a conversion will fail, as it would in the second example.

The Oberon system is delivered with a few document classes, summarized in the following table. Additional classes are supported by add–on packages.

Table 3.14 The Standard document classes

Document Class	Generator Procedure	Alias	File name ext.
Panel	`PanelDocs.NewDoc`	`PanelDoc`	`.Panel`
Picture	`RembrandtDocs.NewDoc`	`RembrandtDoc`	`.Pict`
Text	`TextDocs.NewDoc`	`TextDoc`	`.Text, .Mod, .Tool`
System Log	`TextDocs.NewLog`	`LogDoc`	

The following class was created for Columbus only:

Columbus	`Columbus.NewDoc`	`Columbus`	

As a conclusion to this section, we summarize the `Desktops` commands:

`Desktops.Open filename`
`Desktops.Open ^`
opens the desktop stored in the named file or opens a new default one if a file with that name does not exist.

`Desktops.Grow`
grows the current desktop.

`Desktops.Copy`
copies the current desktop.

`Desktops.Close`
closes the current desktop.

`Desktops.Store`
stores the current desktop.

`Desktops.OpenDoc filename`
opens a document stored in the named file. If no file with that name exists, a new empty document is opened. The type of the new document is determined by the name suffix used.

`Desktops.OpenDoc (DocumentGenerator)`
creates an empty document of the specified type.

`Desktops.OpenDoc filename(DocumentGenerator)`
casts the named document to a new type. The document may already exist or it may be a new document.

`Desktops.InsertDoc filename`
inserts the document stored in the named file at the caret, without a menu bar.

`Desktops.ReplaceDoc fileName`
replaces the current document from which the command was executed with another stored in the named file. This allows to switch in–place from one document to another.

`Desktops.CloseDoc`
closes the current document.

`Desktops.StoreDoc`
`Desktops.StoreDoc *`
stores the current document or the marked document in a file. The file name is taken from the document's NamePlate.

`Desktops.PrintDoc printer name list`
`Desktops.PrintDoc printer name *`
prints all the documents named in the list or the marked document. On some platforms the printer name is ignored and the attached printer is assumed as print destination.

`Desktops.ChangeBackdrop picture-fileName`
changes the backdrop of the marked desktop. To clear the backdrop, use a non–existing Picture file name (`*.Pict`) or else use Columbus.

3.11 Libraries

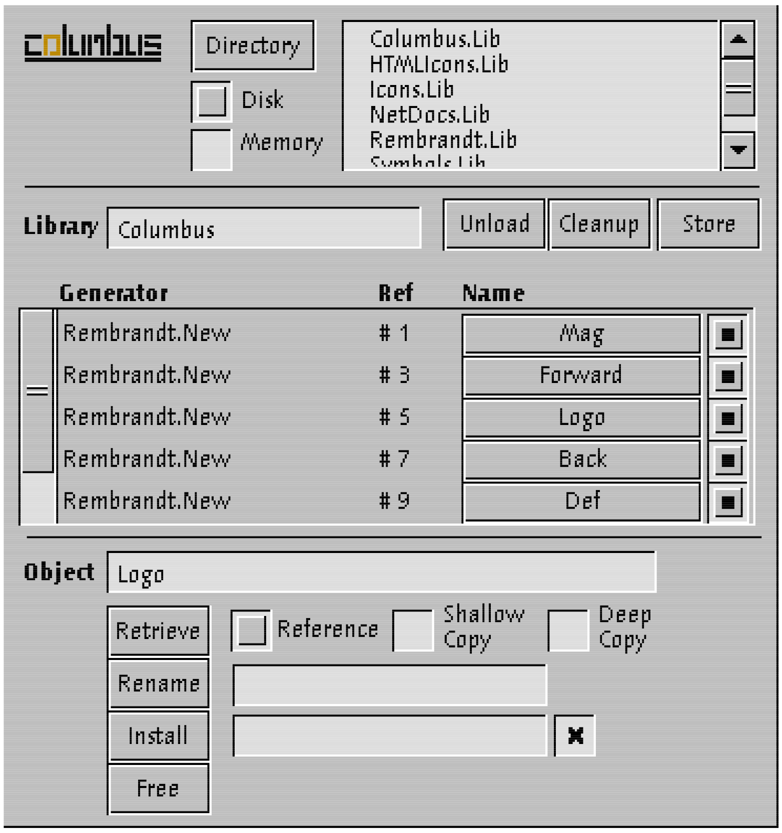
The Oberon system uses a technique called *libraries* for making objects persistent. Documents typically use libraries to store their contents in files. Although libraries are primarily of interest to Oberon programmers, a little knowledge about their use is required by the Oberon user too. As long as a gadget is not associated with any library it is *free*. Once a gadget belongs to a certain library it is *bound* and it is made permanent by storing the whole library to disk. The libraries are divided into two classes: *private* and *public*.

Public libraries, uniquely identifiable by name, are shared between applications, loaded once from disk and cached in memory until they are not needed anymore. The fonts and the Icon library are examples of public libraries. The list of currently loaded public libraries can be shown with the Oberon command `System.ShowLibraries`. Columbus also shows to which library an inspected object belongs. Libraries can be uncached explicitly from memory by the user, although this should be done with care; it may be that an application depends on an object in the public library.

Private libraries, in contrast, are loaded from disk each time they are required, are never shared or cached, and do not have a name. Often these nameless libraries are called anonymous. They are useful to protect the contents of a document from outside influences and are of interest only to the programmer.

Each library has an indexing mechanism associated with it. The index stores the names of the objects that are to be *exported* from the library. We shall often refer to a public object `L.O` assuming that the public library `L` contains an object `O`. It is possible for an object to have an intrinsic name, that is, the value of its *Name* attribute differs from the name it is allocated under in a library. Exported names are used only in public libraries.

Public libraries are quite useful repositories of objects. The `LibrariesPanel` is a convenient tool for managing public and private libraries.



Columbus.Lib
HTMLIcons.Lib
Icons.Lib
NetDocs.Lib
Rembrandt.Lib

Library: Columbus

Generator	Ref	Name
Rembrandt.New	# 1	Mag
Rembrandt.New	# 3	Forward
Rembrandt.New	# 5	Logo
Rembrandt.New	# 7	Back
Rembrandt.New	# 9	Def

Object: Logo

Figure 3.15 The Libraries.Panel

Executing the command `Desktops.OpenDoc Libraries.Panel` opens a library Panel. It is divided into three parts. The top part contains the Button `[Directory]`, two radio buttons and a ListGadget. The ListGadget will contain a list of library names. You can choose the kind of libraries to be listed: libraries stored on disk or libraries currently loaded in memory. Activate the `[Directory]` Button to see the list of your choice.

Pick a library in the list with an MM click. The TextField **Library** is updated: it shows the library which will become the target of the library management operations controlled by the Buttons in the middle part.

`[Unload]` Removes the library from memory.
`[Cleanup]` — Collects the unused objects and stores the library on disk.
`[Store]` — Stores the library on disk (the library remains cached in memory).

At the same time, the list of objects contained in the library is displayed. Each object can be inspected by MM clicking its visual Reference at the right side. A Columbus Panel is directly opened presenting the *Attributes view* of the object.

Pick an object in the list with its **Name** Button. The TextField **Object** is updated: it shows the object which will become the target of the operations controlled by the Buttons in the bottom part.

`[Retrieve]` Inserts the object at the caret or opens it as a document (if it is a document gadget). The three CheckBoxes at the right indicate if retrieval from the library should involve retrieving a Reference to the object, retrieving a Shallow Copy of it or retrieving a Deep Copy of it. The latter is selected by default.

`[Rename]` Renames the object with the specified name. All changes made to the object will be reflected immediately to the clients of the object. The TextField on the right is used to specify the new name.

`[Install]` — Inserts the object under the specified name in the library. An object with this name is overwritten. The TextField on the right is used to specify the name. A Reference to the object to install must be placed in the empty RefFrame gadget first.

`[Free]` Frees the selected object from the library.

Public libraries are used for a few tasks in the system. They often contain objects needed by applications, like Icons and pre–fabricated menu bars, or a state that needs to be made global. In the next section, we will discuss one of these topics of particular interest to the end–user.

3.11.1 User–Customized Menu Bars

Often, typically when working on a low–resolution display, the standard menu bars used in the viewer system and on the desktop are too big to fit in a track, or do not contain the menu options you want. When opening a TextDoc, the `Desktops` module requests a document to appear with a different menu bar in the system track, in the user track or in the desktop, depending on where the document viewer should be placed. The system searches for an optional public library named `TextDocs.Lib` which may contain different prefabricated menu bars as objects. The following tables list the names that must be given to these objects according to the situation:

Table 3.16 Customized menu bars in `TextDocs.Lib`

Menu for	**TextDoc**	**LogDoc**
desktop	DeskMenu	LogDeskMenu
system track	SystemMenu	LogSystemMenu
user track	UserMenu	LogUserMenu

A similar strategy is used for constructing the menu bar for a PanelDoc:

Table 3.17 Customized menu bars in `PanelDocs.Lib`

Menu	**PanelDoc**
desktop	DeskMenu
system track	SystemMenu
user track	UserMenu

If the indicated public library object is missing, a default menu bar is constructed. That is the case for a freshly installed Oberon system, because it is delivered without these public libraries. The default menu bars are detailed in the Chapter 2 and in the descriptions of LogDoc, PanelDoc and TextDoc in Chapter 4.

All the menu objects listed (DeskMenu, SystemMenu etc.) are for use with a display width (`Display.Width`) exceeding 1000 pixels. Another set of menu objects may be constructed for use with small display units. The names of these additional menu objects must be "DeskMenuS", "SystemMenuS", "UserMenuS", "LogDeskMenuS", etc.

All these menu objects may easily and accurately be constructed and stored in public libraries using the LayLa [Der96] tool described in Chapter 6. You may of course create your own menu bars from all available gadgets. The default ones are just simple Panel gadgets with a NamePlate and a few (command) Button gadgets. In accordance with these defaults, it is good practice to implement the standard `Close`, `Hide` and `Grow` commands in customized menus. If the opened document should become a persistent object, also include a `Desktops.StoreDoc` command. To give the Panels a good look, you should at least set their border width to 0 pixels. You may also add Iconizers to the menu bars if you want pull–down (or drop–down) menus. You can use the layout tools in the Align iconizer of the Gadgets.Panel to improve the layout of your own menus. Also, using the Libraries.Panel, you can change the contents of the menu bars stored in these libraries. An example of customized menu bar is given in Chapter 6.

Chapter Four

The Standard Gadgets Reference

4.1 Introduction

In an extensible system, new components are continually being added to the system. It is thus possible to give a snapshot only of the components supplied with the Oberon System 3 Release 2.2 described here. This chapter lists the most often used gadgets included in the Oberon distribution. A few of them are superseded by better ones.

The gadget descriptions are ordered alphabetically by name. The generator and alias for each gadget are listed, followed by a description of the gadget's function and attributes. The attributes, with the exception of the *Name* attribute (every gadget has one), are presented in the order in which they appear, when inspected with Columbus (refer to Chapter 3). The initial (or default) value assigned to each attribute after instantiation with a `Gadgets.Insert ViewGenerator [ModelGenerator]` or `Gadgets.Link ModelGenerator` command is specified after the attribute type between square brackets [ ]. "[ ]" represents the empty string and "[?]" means that the value is determined by the context. When a gadget is instantiated its *Name* attribute contains the empty string. Remember also, that every *visual* gadget can be linked to a model gadget, though this may not be meaningful for several of them. The existence of such a link can be revealed with Columbus and by determining whether the *Model* link contains a Reference or not. The *Model* link is however not documented since it is omnipresent. In addition, a few visual gadgets may be linked to additional gadgets in the same fashion. Such links are explicitely documented. Where relevant, commands related to the gadget are listed at the end.

With a few exceptions, all these visual and model gadgets can easily be instantiated using the `Gadgets.Panel`.

Whenever the registry is mentioned, remember to refer to the Appendix A to find out how to tailor your system profile to your personal requirements.

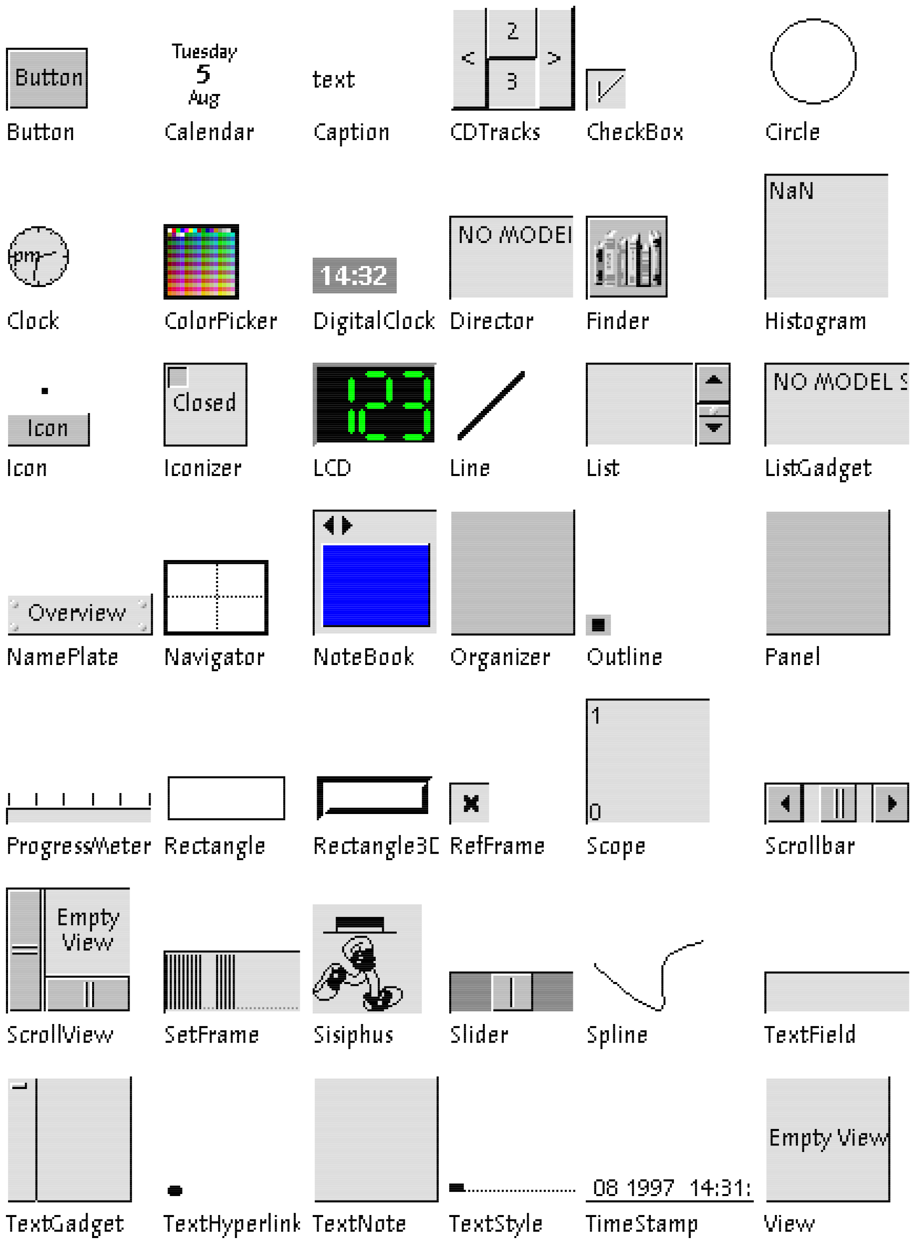


Figure 4.1 Visual gadgets overview

Boolean

Classification	Model gadget
Generator	BasicGadgets.NewBoolean
Alias	Boolean

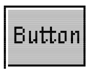
Function

A Boolean is a model gadget that stores a BOOLEAN value. Booleans function as models of Button and CheckBox gadgets.

Attributes

Value
Boolean [FALSE] value.

Button



Classification	Visual elementary gadget
Generator	BasicGadgets.NewButton
Alias	Button

Function

A push–button with a 3D–effect and a user–definable caption. Buttons are pushed with the MM key. This toggles them from *off* (pushed out) to *on* (pushed in) and vice–versa. Additionally, the *Popout* attribute indicates if the Button should pop out immediately after being pressed. This attribute is typically set when the Button activates a command. A Button may be linked to a Boolean or to an Integer model gadget. Several Buttons and CheckBoxes can be linked to the *same* Integer model gadget. They then act as radio buttons: only Buttons with a *SetVal* attribute value matching that of the Integer model are switched on. Normally, one and only one radio button in a set is "on".

Attributes

Caption
String [Button] with the text to appear on the Button. When this attribute is set to the empty string, a visual gadget may be dropped *once* inside the Button. The consumed gadget then becomes the Button's linked *Look* gadget. Normally, it will be a RembrandtFrame with a Picture model gadget. Selecting a Button so captioned and issuing the command `BasicGadgets.Break` reverses the operation (see below). The *Caption* string can still be changed after the Button has consumed a gadget: the string takes precedence and masks the gadget.

Value
Boolean [FALSE] indicating the state of the Button (off or on).

Popout
Boolean [TRUE] – TRUE indicates that the Button must immediately return to the off state after being pressed.

Led
Boolean [TRUE] – TRUE indicates that an LED must appear when the Button is pressed.

LedColor
Integer [1] specifying the color of the LED.

Color
Integer [13] specifying the color of the Button.

SetVal
Integer [0] used in conjunction with an Integer model linked to the Button. It indicates for which value of the Integer model, the Button should switch itself on.

YesVal
String [ ] that is returned by a Button only when switched on. When off, an empty string is returned, even though the attribute value is remembered. This attribute is useful in conjunction with the lookup macro "&" for specifying a command option depending on the state of the Button.

Field

String [ ] indicating which attribute of the linked model gadget should be visualized. When empty, the *Value* attribute of the linked model gadget is used by default.

Cmd
String [ ] executed as a command when the gadget is toggled or pushed.

Links

Look
The visual gadget which appears as caption. The visual gadget can be installed only if the *Caption* attribute contains the empty string by drag and drop. Using the services of Columbus, the *Look* gadget may be installed, changed and removed.

Commands

`BasicGadgets.SetValues`
assigns a unique *SetVal* number to each of the selected Buttons and CheckBoxes. Numbering is from zero onwards in their order of selection. This command is used when creating radio buttons. The common model must be an Integer.

`BasicGadgets.Break`
takes the selected Button apart, removing the linked *Look* gadget and inserting it at the caret.

Calendar

Classification	Visual elementary gadget, transparent
Generator	Clocks.NewCalendar
Alias	Calendar

Function

A calendar showing the current day of the week and the date.

Attributes

Color
Integer [15] specifying the color of the text. "Sunday" is always displayed in red, regardless of this value.

TimeDiff
Integer [0] that sets a time difference in hours relative to the system time. May be used to display the day of the week and the date in a different time zone.

Commands

`Clocks.InsertDate`
inserts the current date at the caret. The date is formatted according to the specifications contained in the DateFormat key in the [System] section of the registry. If the caret is positioned inside a TextDoc, a TextField, a TextGadget or a TextNote, it is inserted as a string. If the caret is positioned inside a container such as a Panel, a PanelDoc or a desktop, it appears as the *Value* attribute of a Caption.

Remark

When the `Clocks` module is loaded, a background task is installed in the Oberon loop. To remove the task, remove all the visual gadgets it controls from the display space and execute a `System.Free Clocks` command.

Caption

Sample caption
on two lines

Classification Visual elementary gadget, transparent
Generator TextFields.NewCaption
Alias Caption

Function

A Caption is a text entity used for titles or comments in Panels and other container gadgets. A Caption may consist of multiple lines of text, but only limited editing capabilities are provided. The keyboard focus is set with an ML key click, the gadget is then framed in a thin rectangle but no caret is visible. When focused, characters typed in with the keyboard are *appended* to the caption text. The backspace key deletes the last character. A text stretch, including mixed fonts and colors, may be appended to the focused Caption with a copy interclick (copy to caret or copy over). The font and the color can be changed using the commands listed below. This gadget is always linked to a Text model gadget which may be replaced by another, using Columbus for instance, but which may never be deleted.

Useful hint: A Caption may be typed directly in a Panel or in a desktop at the caret position. It is not necessary to insert the Caption first with `Gadgets.Insert Caption`

Attributes

Value
String [ ] appearing in the Caption. Only the first 64 characters are returned. The text may contain mixed fonts and colors. The text color is reset to black when this attribute value is manipulated with Columbus.

Font
String [Syntax10.Scn.Fnt] specifying the font of the text. The string is assigned the value "mixed" after a mixed font or mixed color text stretch was copied to the Caption.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key.

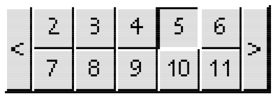
Underlined
Boolean [FALSE] – TRUE indicates that the text must be underlined.

Commands

`TextDocs.ChangeColor col-no`
applies the color specified by the parameter to the most recently selected Caption. The text color can also be changed with the ColorPicker. The color is reset to black when the gadget attributes are manipulated with Columbus.

`TextDocs.ChangeFont font-name`
applies the font specified by the parameter to the most recently selected Caption. The font name must be specified in full, e.g. `Courier10.Scn.Fnt`

CDTracks



Classification	Visual elementary gadget
Generator	AudioGadgets.NewTrack
Alias	none

Function

An auto–adaptive graphic with a 3D–effect for a number of numbered cells contained between a left and a right scroll button. The cells and the scroll buttons adjust themselves to fit on the available surface when the gadget is resized. If the specified number of cells cannot all fit, the scroll buttons become active (indicated by the < or > captions). Scrolling through the cells is controlled by MM key clicks on the scroll buttons. One of the cells, identified by the *Value* attribute, appears pressed. A cell also appears pressed when the MM key is pressed or dragged on it. This gadget is used in the `CDAudio.Panel` for which it was originally designed, but can be used in any other environment too.

Attributes

Value
Integer [0] specifying the number of the cell that must appear pressed. This value is stored in the linked *Model* gadget. The value "0" means "no cell pressed".

Tracks
Integer [20] specifying the number of cells to distribute on the available surface. Cells are numbered from 1 onward. This value is stored in the linked *Tracks* gadget.

Point
Integer [0] that remembers the last clicked cell (*read–only*).

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key. In the context of the `CDAudio.Panel` the attribute is assigned the value "`CDAudioPlayer.Play #Point`

Links

Tracks
Model gadget which stores the value assigned to the *Tracks* attribute. In the context of the `CDAudio.Panel`: Reference to the Integer model named "CDTracks" containing the number of tracks of the CD loaded in the CD–ROM device.

Commands

`AudioGadgets.InsertTrack [Model-generator [Tracks-generator]]`
inserts a CDTracks gadget at the caret, without linked model gadget, linked to a *Model* gadget, or linked to a *Model* gadget and to a *Tracks* model gadget.

Remark

The Audio application must be installed.

CheckBox

Classification	Visual elementary gadget
Generator	BasicGadgets.NewCheckBox
Alias	CheckBox

Function

CheckBoxes function pretty much like Buttons, except that they show a check mark when switched on. A CheckBox may be linked to a Boolean or to an Integer model gadget. Several CheckBoxes and Buttons can be linked to the *same* Integer model gadget. They then act as radio buttons: only CheckBoxes with a *SetVal* attribute value matching that of the Integer model are switched on. Usually, one and only one radio button in a set is "on". When used as radio buttons, Checkboxes show rectangular check marks instead.

Attributes

Value
Boolean [FALSE] indicating the state of the CheckBox (not checked or checked).

YesVal
String [ ] that is returned by a CheckBox only when it is switched on. When off, Columbus does not show its value, even though it is set behind the scenes. This attribute is useful in conjunction with the lookup macro "&" for specifying command options depending on the state of the CheckBox.

SetVal
Integer [0] used in conjunction with an Integer model linked to the CheckBox. It indicates for which value of the Integer model, the CheckBox should check itself.

Field
String [ ] indicating which attribute of the linked model gadget should be visualized. When empty, the *Value* attribute of the model gadget is used by default.

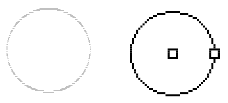
Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key.

Color
Integer [14] specifying the color of the CheckBox.

Commands

`BasicGadgets.SetValues`
assigns a unique *SetVal* number to each of the selected Buttons and CheckBoxes. Numbering is from zero onwards in their order of selection. This command is used when creating radio buttons. The common model must be an Integer.

Circle



Classification Visual elementary gadget, transparent
Generator BasicFigures.NewCircle
Alias Circle

Function

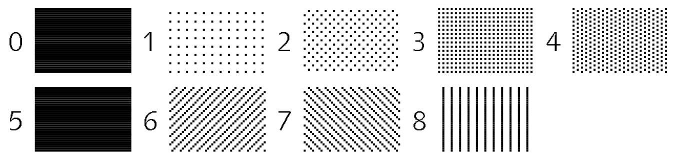
A circle with an adjustable radius. Selecting a Circle causes two control points to appear, one in the center and the other on the circumference of the Circle. Pressing the MM key on one of the control points and dragging changes the position or the radius of the Circle which takes its final shape when the key is released.

Attributes

Color
Integer [15] specifying the color of the Circle. If the Circle is filled, the color also applies to the interior. The color can also be changed with the ColorPicker.

Width
Integer [1] specifying the width in pixels of the Circle.

Pattern
Integer [0] specifying in which pattern the Circle is to be drawn. If the Circle is filled, the pattern also applies to the interior. Patterns, which are exported by the module `Printer3`, are numbered from 0 to 8:



The value 0 (no pattern) and invalid values are mapped to the value 5 corresponding to a pattern named "solid".

Closed
Boolean [FALSE] – not interpreted.

Filled
Boolean [FALSE] – TRUE indicates that the Circle must be filled. If so, the filled part has the color specified by the *Color* attribute and the pattern specified by the *Pattern* attribute.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key.

Clock

Classification	Visual elementary gadget, transparent
Generator	Clocks.NewClock
Alias	Clock

Function

An analogue clock with hour and minute hands, showing the current time.

Attributes

Color
Integer [15] specifying the color of the Clock.

TimeDiff
Integer [0] that sets a time difference in hours relative to the system time. May be used to display the time in a different time zone.

Commands

`Clocks.InsertTime`
inserts the current time at the caret. The time is formatted according to the specifications contained in the TimeFormat key in the [System] section of the registry. If the caret is positioned inside a TextDoc, a TextField, a TextGadget or a TextNote, it is inserted as a string. If the caret is positioned inside a container such as a Panel, a PanelDoc or a desktop, it appears as the *Value* attribute of a Caption. Refer to the TimeStamp gadget for details on the time format.

Remark

See Calendar.

ColorPicker

Classification Visual elementary gadget
Generator ColorTools.NewColorPicker
Alias ColorPicker

Function

A ColorPicker shows the current color palette entries of the system. The colors are listed from left to right, top to bottom, from color index 0 to the number of colors available. Pressing the MM key on the ColorPicker pops up a menu from which a color can be selected. When the key is released, this color is then applied to the most recent selection, either text, gadget or group of gadgets which have a *Color* attribute, by virtue of the value assigned by default to the *Cmd* attribute (see below). The color attribute of a selected text piece can also be changed by copying the text attributes of another character, with a ML + MR key interclick. However, the typeface and the vertical offset attributes of the character chosen are also copied. The ColorPicker does a finer grain change.

Attributes

Colors
String [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15] that specifies the first 16 indices shown by the ColorPicker. It defaults to the first colors 0 to 15 in the color palette.

Col
Integer [15] that remembers the last picked color.

Cmd
String [ColorTools.ChangeColor #Col ~] executed as a command when a color is picked.

Commands

`ColorTools.ChangeColor col-no`
changes the color of the selection to the specified color. This command is both applicable to the text and gadget selections. In the case of a gadget, the *Color* attribute is assigned a new value.

ColorWell

Classification	Visual elementary gadget
Generator	ColorWells.NewColorWell
Alias	ColorWell

Function

A ColorWell stores a current color value. The color value runs from 0 to the number of colors available. This gadget is typically linked to an Integer model gadget, though it may quite well be used without a model. A drag–and–drop operation allows dropping the current color onto another visual gadget having a *Color* attribute. In case the color of a gadget is controlled by an attribute with another name, the *TargetAttr* can be used.

Pressing the ML key on a ColorWell activates a ColorPicker with which a color can be picked directly. While dragging on the key, the color positioned under the mouse focus is highlighted. This color becomes the current color when the key is released.

Attributes

Color
Integer [0] that stores the current color.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key. In order to change the color attribute of a selected text piece, one can assign the value "ColorTools.ChangeColor #Color ~" to this attribute.

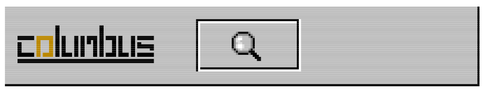
Field
String [ ] indicating which attribute of the linked model gadget should be visualized. When empty, the *Value* attribute of the linked model gadget is used by default.

TargetAttr
String [ ] indicating which attribute must be changed when a drag–and–drop operation is performed. The value of this gadget's *Color* attribute is assigned to the named attribute of the consuming gadget. For example, to change the background color of a Scrollbar, the value "BackColor" must be specified. When empty, the default name is "Color". The named attribute does not even have to control a color.

Remark

The Leonardo application must be installed.

Columbus



Classification Visual document gadget
Generator Columbus.NewDoc
Alias Columbus

Function

Columbus is an extension of a PanelDoc containing a single Columbus Panel. It is opened with the commands `Desktops.OpenDoc (Columbus)` or `Columbus.Inspect`, or with the `[Columbus]` button in the `Gadgets.Panel`. It appears with a menu bar containing buttons captioned `[Close]`, `[Hide]` and `[Grow]`. The function of the `[Hide]` button is explained in the Finder commands. Clicking the [🔍] button in an existing Columbus Panel for inspecting another object or library re–uses the Panel and returns it to an attribute view or library view as described below, that is, no new document is opened. Refer to the detailed description in Chapter 3.

Implementation restriction: Columbus cannot have attributes and it cannot be named, though the NamePlate contains "Objects.Panel". Also, it can neither be saved nor inserted at the caret with `Gadgets.Insert Columbus`.

Commands

```
Columbus.Inspect *
Columbus.Inspect ^
Columbus.Inspect L.O
```

opens a Columbus Panel in the *Attributes view* of the target object to inspect. The latter may be the marked frame, the frame selected most recently, the name `L.O` of an object in a public library `L` or a name contained in the most recent selection. If no target is found, an empty Columbus is opened. The first version of this command using the marker (*) is of great utility to inspect a marked component in a locked container (in which the components cannot be selected). When the parameter value specifies the name of a library, a Columbus Panel in the *Library view* is opened.

Complex

Classification	Model gadget
Generator	Complex.New
Alias	Complex

Function

Complex is a compound model gadget representing a complex number.

Attributes

Real
real [0] value representing the real part of the complex number.

Imag
real [0] value representing the imaginary part of the complex number.

Rho
real [0] value representing the polar vector length of the complex number.

Phi
real [0] value representing the polar angle in radians of the complex number.

CurrentDirectory

Classification	Model gadget
Generator	Directories.NewDrv
Alias	CurrentDirectory

Function

A CurrentDirectory gadget stores the current directory path the system is using. This model gadget is typically visualized by a TextField gadget.

Attributes

Value
String [?] value representing the current directory path. An empty string is returned in PC Native Oberon.

Commands

`System.ChangeDirectory path`
`System.ChangeDirectory ..`
sets the current directory either to that specified by `path` or to the parent (..) directory.

Remark

When the `Directories` module is loaded, a background task is installed in the Oberon loop. To remove the task, remove all the gadgets it controls from the display space and execute a `System.Free Directories ~` command.

Example

Insert a directory indicator at the caret with the command:

```
Gadgets.Insert TextField CurrentDirectory ~
```

By assigning the string `System.ChangeDirectory #Value` to the *Cmd* attribute of the TextField gadget, the TextField can also be used to change the current directory. An example can be found in the `System.Tool`

CurrentLoad

Classification	Model gadget
Generator	Gages.NewLoad
Alias	CurrentLoad

Function

A CurrentLoad model gadget calculates the current workload of the Oberon system. The workload is estimated from the frequency at which the system obtains control in the Oberon loop. This model gadget is typically visualized by a Scope gadget.

Attributes

Value
Integer [?] value representing the current workload. It has no scale.

Remark

When the `Gages` module is loaded, a background task is installed in the Oberon loop. To remove the task, remove all the gadgets it controls from the display space and execute a `System.Free Gages ~` command.

Example

Insert a load indicator at the caret with the command:

```
Gadgets.Insert Scope CurrentLoad ~
```

Dag

Classification	Model gadget
Generator	ListDags.New
Alias	Dag

Function

A Dag stores strings organized as a directed acyclic graph. The list may contain duplicates or not. This model gadget is typically used as a model by a ListGadget.

Attributes

Unique
Boolean [FALSE] – TRUE indicates that no duplicate items are allowed on a level.

Commands

See ListGadget.

DigitalClock

Classification	Visual elementary gadget, transparent
Generator	Clocks.NewDigiClock
Alias	DigitalClock

Function

A digital clock showing the current time in hours and minutes.

Attributes

Color
Integer [0] specifying the color of the DigitalClock's text.

TimeDiff
Integer [0] that sets a time difference in hours relative to the system time. May be used to display the time in a different time zone.

Commands

`Clocks.InsertTime`
inserts the current time at the caret. The time is formatted according to the specifications contained in the TimeFormat key in the [System] section of the registry. If the caret is positioned inside a TextDoc, a TextField, a TextGadget or a TextNote, it is inserted as a string. If the caret is positioned inside a container such as a Panel, a PanelDoc or a desktop, it appears as the *Value* attribute of a Caption.

Remark

See Calendar.

Directory

Classification Model gadget
Generator Directories.New
Alias Directory

Function

A Directory stores strings organized as a tree representing the structure of the computer's external storage. It is typically visualized by a DirList or a ListGadget gadget.

Attributes

Mask
String [*] used as a mask for selecting the list items. The only wildcard character allowed in the mask is the "*".

RootDir
String [ ] determining the path for selecting the list items.

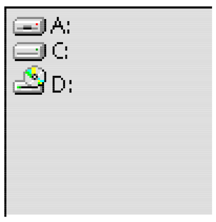
Commands

See ListGadget.

Remark

See CurrentDirectory.

DirectoryView



Classification	Visual elementary gadget
Generator	Directories.NewDirList
Alias	DirectoryView

Function

A DirList is designed to visualize a Directory. It features the same properties and the same attributes as a ListGadget, except that it displays fancy icons representing the external storage devices and folders. These icons are created from Picture gadgets provided in the public library `Symbols.Lib`

Attributes, Links and Commands

See ListGadget.

Remark

See CurrentDirectory.

EventTimer

Classification Model gadget
Generator Gages.NewLap
Alias EventTimer

Function

This model gadget calculates how long the execution of commands takes, by measuring the time during which the system obtains control in the Oberon loop. Only events longer than half a second are measured.

Attributes

Value
Integer [?] value measuring the elapsed time in milliseconds.

Remark

See CurrentLoad.

Example

Insert an EventTimer at the caret with the command:

```
  Gadgets.Insert EventTimer
```

Finder

Classification Visual elementary gadget
Generator Finder.NewFrame
Alias Finder

Function

A Finder is used to quickly find a document among the many documents piled in a desktop and to place it on top of all others, open and ready. Moving the mouse focus to the Finder and pressing the MM key opens the Finder showing a list of document names from which one can be selected. When the MM key is released, the selected document is brought to the front of the desktop and the Finder is closed. An open document that is partially visible can also be brought to the front by a simple ML click.

The Finder has three parts: the top part lists the names of the open documents (in blue), the middle part lists the minimized documents (in green) and the bottom part lists the documents contained in the [FinderTemplates] section of the registry. The names of open and minimized documents are taken from their NamePlates. The names include the full path specification and they are preceded by an ideogram representing the document type. A document having an empty NamePlate appears as "Untitled document".

The names in the three parts have an order. In the list of open documents, the document placed on back of the desktop appears first, while the document placed in front appears last. The name order in the second list is determined by the system. In the last list, the names appear in the order of their appearance in the [FinderTemplates] section, which may be customized at will. Each entry in the section must appear in a text line with the following format:

```
Name=documentName|documentType
```

e.g. `Gadgets=Gadgets.Panel` or `newText=(TextDocs.NewDoc)` for an untitled TextDoc.

Finally, if several desktops share the display, their respective Finders, if any, will also share the unique [FinderTemplates] section, but will display different open and minimized document name lists.

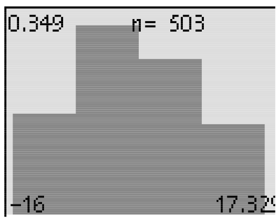
In the Oberon for Windows, Linux and Mac versions, one may drop any open document from the desktop on the Finder: this document is then added to the [FinderTemplates] section and to the last part of the Finder.

Commands

`Finder.Minimize`
minimizes the document in which the command is executed, whereby the meaning of minimizing depends on the environment. When the document is placed on a desktop, the document is removed from the desktop and its name is placed in the Finder. If no Finder can be found on the desktop, nothing happens. In the tiled viewer system, the menu bar of the document is pulled down to the very bottom of the track, leaving the rest of the document invisible. This command is used in the `[Hide]` button found in most menu bars.

`Finder.UpdateTemplates`
reads the [FinderTemplates] section of the registry again. This command is used after an update of the registry in the current session.

Histogram



Classification	Visual elementary gadget
Generator	Histogram.NewFrame
Alias	none

Function

A Histogram displays a chart of the previous values of the model gadget linked to it. The exact behavior depends on the setting of the Histogram attributes. By default, the Histogram uses the first values to determine an appropriate histogram geometry, and adapts to new values as the model gadget changes.

Attributes

adaptive
Boolean [TRUE] – TRUE indicates that the Histogram operates in learning mode. FALSE indicates that the Histogram parameters are not adapted.

moving
Boolean [FALSE] – TRUE indicates that the Histogram shows a moving plot of the recent values. FALSE indicates that the Histogram will accumulate all previous observations.

suspended
Boolean – TRUE indicates that the Histogram does not accumulate new values.

DoResetCounts
Boolean [FALSE] – Used to control the Histogram. TRUE indicates that the value remains unchanged, but all Histogram counts are reset to zero.

DoResetAll
Boolean [FALSE] – Used to control the Histogram. TRUE indicates that the value remains unchanged, but all Histogram geometry parameters and counts are reset to zero.

NrObs, TooSmall, TooLarge, NaNs
Integers showing the global Histogram counts. These attributes should not be set directly. They may however be reset using *DoResetCounts.*

Min, Max, CellWidth, nrCells
Reals resp. integers defining the global Histogram geometry. These attributes should only be set directly when Histogram action is suspended. They may however be reset using *DoResetAll. DoResetAll* resets the Histogram counts.

LogTransform
Boolean [FALSE] – TRUE forces a logarithm transformation of the model values. Changing this value forces a *DoResetAll*.

Diff
Boolean [FALSE] – TRUE forcing to take differences of the (possibly transformed) model values. Changing this value forces a *DoResetAll*.

Example

Insert a Histogram at the caret with the command:

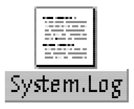
```
Gadgets.Insert Histogram.NewFrame ModelGenerator
```

Insert a Histogram at the caret showing differences between subsequent values of the linked model gadget with the command:

```
Gadgets.Insert Histogram.NewDiffFrame ModelGenerator
```

To insert a Histogram in suspended state, use the generators `Histogram.NewSuspendedFrame` or `Histogram.NewSuspendedDiffFrame`.

Icon



Classification	Visual elementary gadget, transparent (superseded by Finder)
Generator	Icons.NewIcon
Alias	Icon

Function

An Icon provides a caption for the linked *Model* gadget it visualizes. When it is instantiated, an Icon has no model. At this stage, a visual gadget may be dropped inside it, but once only. Normally, the dropped gadget is a RembrandtFrame with a Picture model gadget. Instead of dropping a gadget into an empty Icon, the *Icon* attribute may be used to set the linked *Model* gadget to visualize. Selecting such an Icon and issuing the command `Icons.Break` reverses the operation (see below). An Icon created with the `Icons.InsertIcon` command is a "minimized" version of the original stored document. It occupies little space on a desktop and the stored document can be re–opened with a simple MM key click, by virtue of the value assigned to its *Cmd* attribute. Icons can consume other gadgets and they are otherwise very flexible. An example is given in section 3.8.

Using the `[Hide]` button to minimize a document and a Finder to insert it back on the desktop, represents a different approach: an already opened document and not a stored document re–appears on the desktop.

Attributes

Caption
String [Icon] containing the Icon caption text. Executing the command `Icons.InsertIcon` assigns a new value to this attribute.

Icon
String [ ] used to set the linked *Model* gadget to visualize. The string is the name of a visual object `O` located in a public library `L` (notation `L.O`). A library `Icons.Lib` of RembrandtFrames is provided for this purpose. The `Libraries.Panel` is used to investigate the content of this library.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key. The string is typically a command of the form `Desktops.OpenDoc '#Caption'` for opening the document named in the *Caption* attribute. Executing the command `Icons.InsertIcon` assigns that value to this attribute.

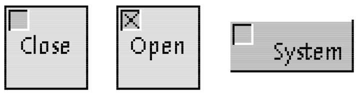
ConsumeCmd
String [ ] executed as a command when a gadget or a group of gadgets is dropped inside the Icon. The originals of the dropped gadgets (or initiators) remain in place. This command is executed only when the Icon has a linked *Model* gadget. No command is executed the first time a gadget is dropped. The command string will usually contain an initiator macro "!".

Commands

`Icons.Break`
takes the selected Icon apart, removing the linked *Model* gadget and inserting it at the caret.

`Icons.InsertIcon`
creates an Icon for the marked document and inserts it at the caret. The *Caption* attribute is set to the document name, the *Cmd* attribute is assigned the string `Desktops.OpenDoc` '`#Caption`' and the linked *Model* Gadget is set according to the document type. For example, if it is used with a document "Work.Text", the Icon `Icons.Text` is extracted from `Icons.Lib`

Iconizer



Classification	Visual container gadget
Generator	Icons.NewIconizer
Alias	Iconizer

Function

An Iconizer is used to build various types of menus. It is a flip–card, with a gadget on each side of the card. A switch pin is located in the top left corner of the Iconizer. Clicking on this pin with the MM key, flips the Iconizer between its two sides. Initially, the Iconizer has no links and the two "empty" sides can be distinguished by their captions "Closed" and "Open". On each side, a visual gadget may be dropped, but once only. The dropped gadget then becomes the side's linked gadget (*Open* or *Closed*) which is viewed by this side. The two sides can be distinguished by the cross on the switch pin of the "Open" side. By setting the attributes, Iconizers can be changed into different types of pop–up menus. When building a menu, the "Closed" side represents the menu, and the "Open" side represents the menu items themselves.

Attributes

Popup
Boolean [FALSE] – TRUE indicates that the Iconizer functions as a pop–up menu when the MM key is clicked. When the *Menu* attribute is set to FALSE, the "Open" side gadget will pop–up and will track the mouse until the key is released. As an example, in this state an item can be picked directly from a TextGadget or a TextNote contained in the "Open" side of the Iconizer. An example is given below.

Menu
Boolean [FALSE] – Together with the *Popup* attribute, the *Menu* attribute allows you to make pop–up menus. When the menu is popped up, a menu item may be selected. The menu item is any text piece located in the "Open" side of the card that contains a *Cmd* attribute. The *Cmd* of the selection will be executed when the mouse key is released. In this mode, the closed side of the Iconizer cannot be edited anymore.

FixedViews
Boolean [TRUE] – TRUE indicates that the two sides of the card are displayed at their default positions. Otherwise, the two sides may be located at different positions in the current parent context. Moving one side of the card does not affect the position of the other.

Pin
Boolean [TRUE] – TRUE indicates that the switch pin must be visible. When set to FALSE, the Iconizer can be flipped only by executing the command `Icons.Flip` from within it, or with the help of Columbus. Also, when set to FALSE, the *Popup* attribute could be set to TRUE to be able to access the "Open" side temporarily. The menu on that side is well protected, whatever the value of the *Locked* attribute.

Locked
Boolean [FALSE] – TRUE indicates that the Iconizer cannot be edited or resized.

Links

Closed and *Open*
The visual gadgets appearing on the two sides of this gadget.

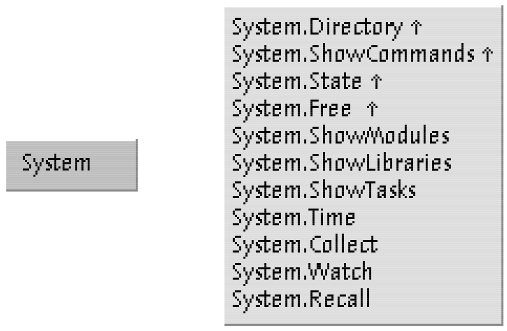
Commands

`Icons.Break`
takes the selected Iconizer apart, removing the visual gadget appearing on both sides and inserting them at the caret if it is set.

`Icons.Flip`
flips the Iconizer when executed from within it.

Example

A typical Iconizer is constructed with a Panel as its *Closed* link and a TextNote (or a TextGadget, a Panel, etc.) as its *Open* link. The Panel has a Caption in it for the title of the pop–up menu, and the TextNote contains a list of commands or file names. The Iconizer attributes are set to: *Popup* = TRUE, *Menu* = FALSE, *FixedViews* = TRUE, *Pin* = FALSE. In the case of file names, the *Cmd* attribute of the TextNote must contain a command, such as `Desktops.OpenDoc#Point` for example, as dictated by the application.



```
System.Directory ↑
System.ShowCommands ↑
System.State ↑
System.Free ↑
System.ShowModules
System.ShowLibraries
System.ShowTasks
System.Time
System.Collect
System.Watch
System.Recall
```

Closed side   Open side

This kind of pop–up menu is very useful in tool texts and is even more compact than an alternative implementation based on TextHyperlinks. The `Programming.Tool` contains a few examples of such menus in Iconizers, with a visible switch pin though.

The following module comes in handy to maintain menu texts in such Iconizers:

```
MODULE IconizerEdit;

IMPORT Attributes, Display, Gadgets, Links, Oberon, Objects, Out, Texts, TextGadgets0;

VAR w: Texts.Writer;

(* Copy selected Iconizer menu for editing: IconizerEdit.EditMenu.
   The "Closed" side may contain several components, one of them may be a Caption. *)
PROCEDURE EditMenu*;
VAR obj1, obj2: Objects.Object;  time: LONGINT;  name, cmd: ARRAY 32 OF CHAR;
      t: Texts.Text;  f: Display.Frame;
BEGIN
   Gadgets.GetSelection(obj1, time);
   IF (time >= 0) & (obj1 # NIL) THEN
      Attributes.GetString(obj1, "Gen", name);
      IF name = "Icons.NewIconizer" THEN
         Links.GetLink(obj1, "Closed", obj2);
         IF (obj2 # NIL) & (obj2 IS Display.Frame) THEN
            name := "";
            f := obj2(Display.Frame).dsc;     (* first child *)
            WHILE f # NIL DO
               Attributes.GetString(f, "Gen", name);
               IF name = "TextFields.NewCaption" THEN
                  Attributes.GetString(f, "Value", name);  f := NIL
               ELSE
```

```
                    name := "";  f := f.next
                END
            END
        END;
        Links.GetLink(obj1, "Open", obj2);
        IF (obj2 # NIL) & (obj2 IS TextGadgets0.Frame) THEN
            obj2 := Gadgets.Clone(obj2, TRUE);
            WITH obj2: TextGadgets0.Frame DO
                obj2.state0 := obj2.state0 +
                    {TextGadgets0.sizeadjust, TextGadgets0.grow, TextGadgets0.shrink}
            END;
            NEW(t);  Texts.Open(t, "");
            Texts.WriteString(w, "IconizerEdit.MakeMenu ");
            Texts.Write(w, 22X);  Texts.WriteString(w, name);  Texts.Write(w, 22X);
            Texts.Write(w, " ");
            Attributes.GetString(obj2, "Cmd", cmd);
            Texts.Write(w, 22X);  Texts.WriteString(w, cmd);  Texts.Write(w, 22X);
            Texts.WriteLn(w);
            Texts.WriteObj(w, obj2);  Texts.WriteLn(w);
            Texts.Append(t, w.buf);
            Oberon.OpenText("", t, 200, 200)
        ELSE Out.String("no text");  Out.Ln
        END
    ELSE Out.String("not an Iconizer");  Out.Ln
    END
  ELSE Out.String("no selection");  Out.Ln
  END
END EditMenu;

(* Make a new Iconizer with a Caption "name" and a TextNote obj as third parameter.
  The string "cmd" is to be assigned to the TextNote.
  IconizerEdit.MakeMenu "caption" "cmd" obj *)
PROCEDURE MakeMenu*;
CONST W = 64;  H = 25;
VAR name, cmd: ARRAY 64 OF CHAR;  s: Texts.Scanner;  obj1, obj2, obj3: Objects.Object;
BEGIN
  Texts.OpenScanner(s, Oberon.Par.text, Oberon.Par.pos);  Texts.Scan(s);
  IF s.class = Texts.String THEN
    COPY(s.s, name);  Texts.Scan(s);
    IF s.class = Texts.String THEN
        COPY(s.s, cmd);  Texts.Scan(s);
        IF s.class = Texts.Object THEN
            obj1 := Gadgets.CreateObject("Iconizer");
            Attributes.SetBool(obj1, "Popup", TRUE);
            Attributes.SetBool(obj1, "Pin", FALSE);
            Gadgets.ModifySize(obj1(Display.Frame), W, H);
            obj2 := Gadgets.Clone(s.obj, TRUE);     (* open contents *)
            Attributes.SetString(obj2, "Cmd", cmd);
            Links.SetLink(obj1, "Open", obj2);
            obj2 := Gadgets.CreateObject("Panel");     (* closed contents *)
            Gadgets.ModifySize(obj2(Display.Frame), W, H);
            obj3 := Gadgets.CreateObject("Caption");
            Attributes.SetString(obj3, "Value", name);
            Gadgets.Consume(obj2(Gadgets.Frame), obj3(Gadgets.Frame), 8, -18);
            Attributes.SetBool(obj2, "Locked", TRUE);
            Links.SetLink(obj1, "Closed", obj2);
            Gadgets.Integrate(obj1)
        ELSE Out.String("no gadget");  Out.Ln
        END
    END
  END
END MakeMenu;

BEGIN
  Texts.OpenWriter(w)
END IconizerEdit.
```

Integer

Classification Model gadget
Generator BasicGadgets.NewInteger
Alias Integer

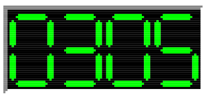
Function

An Integer is a model gadget that stores a LONGINT value. It may be visualized by TextFields, Scrollbars, Sliders, Buttons or ProgressMeters. When linked to a set of Buttons or CheckBoxes (or a mixture of Buttons and CheckBoxes), these gadgets function as radio buttons. In this case, each Button and CheckBox should be assigned a unique integer value stored in their respective *SetVal* attributes. This can be conveniently done with the command `BasicGadgets.SetValues` A Button or CheckBox is "on" when its assigned integer value corresponds to the Integer gadget value.

Attributes

Value
Integer [0] value.

LCD



Classification	Visual elementary
Generator	AudioGadgets.NewLCD
Alias	none

Function

An auto–adaptive LCD display with a 3D–effect. The LCD segments are green and the background is black. The LCD segments adjust themselves to fit on the available surface when the gadget is resized. This gadget is used in the `CDAudio.Panel` for which it was originally designed, but can be used in any other environment too.

Attributes

Value
Positive integer [0] value displayed.

Digits
Integer [2] specifying the number of digits to display. The displayed value is right–adjusted and left–padded with zeros or left–truncated depending on the current *Value*.

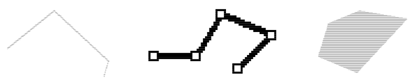
Width
Integer [3] specifying the width in pixels of the LCD segments.

Border
Integer [2] specifying the width of the 3D border surrounding the gadget.

Remark

The Audio application must be installed.

Line



Classification	Visual elementary gadget, transparent
Generator	BasicFigures.NewLine
Alias	Line

Function

A line or polygonal line. Selecting a Line causes control points to appear at the line joints. The control points can be adjusted by dragging on the MM key. The Line takes its final shape when the MM key is released. While dragging a control point, an MR key interclick inserts an additional control point, whereas an ML key interclick deletes that control point.

Attributes

Color
Integer [15] specifying the color of the Line. If the Line is filled, the color also applies to the interior. The color can also be changed with a ColorPicker.

Width
Integer [1] specifying the width in pixels of the Line.

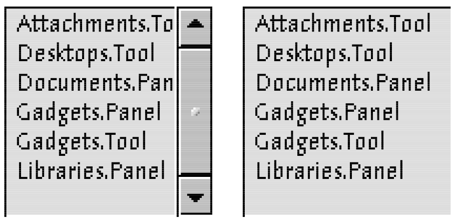
Pattern
Integer [0] specifying in which pattern the Line is to be drawn. If the Line is filled, the pattern also applies to the interior. Patterns are numbered from 0 to 8. Refer to the *Pattern* attribute of Circle.

Closed
Boolean [FALSE] – TRUE indicates that the two end points of the Line should be joined.

Filled
Boolean [FALSE] – TRUE indicates that the Line must be filled. If so, an open polygonal line appears closed, and the filled part has the color specified by the *Color* attribute and the pattern specified by the *Pattern* attribute.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key.

List



Classification	Visual elementary gadget (superseded by ListGadget)
Generator	Lists.NewList
Alias	List

Function

A List presents strings in list–like fashion. The list may be sorted or unsorted, and it may have a scrollbar or not. Copying strings into the List is done by focusing the List with an ML key click, and then copying text into it using a right + middle keys interclick. The MR key is used to select list entries. The normal interclick combinations to delete and copy over text from text gadgets apply. Lists support a quick search facility: first focus the list, then type the first few characters of the string entry to jump directly to that entry. Press RETURN to restart the search process. The MM key is used to pick items from the list. While the MM key is dragged, the list will scroll up and down if the mouse focus is moved above and below the extent of the gadget. This gadget is always linked to a Text model gadget which may be replaced by another, using Columbus for instance, but which may never be deleted. The List gadget is an earlier development, now replaced by a ListGadget / ListModel pair. It should no longer be used.

Attributes

Scrollbar
Boolean [TRUE] – TRUE indicates that a scrollbar must be included.

Sorted
Boolean [TRUE] – TRUE indicates that the list must be alphabetically sorted. No duplicate entries are allowed in a sorted list.

Sel
The currently selected items (*read–only*).

Point
String [ ] pointed at, just before the *Cmd* is executed. It is often used by the *Cmd* attribute itself (*read–only*).

Cmd
String [ ] executed as a command when a List item is clicked on with the MM key.

Commands

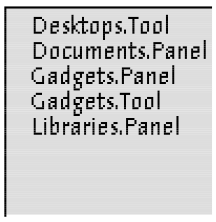
`Lists.Directory "pattern" ObjName`
searches the current directory and inserts a list of all the file names that match the pattern, into the List named `ObjName` The only wildcard character allowed in the pattern is "*".

`Lists.Diskette "pattern" ObjName`
searches the currently selected diskette and inserts a list of all the file names that match the pattern, into the List named `ObjName` The only wildcard

character allowed in the pattern is "*". This command is not implemented on all ports of the system.

`Lists.Library 'lib' ObjName`
searches the library `lib` and inserts a list of all the object names into the list named `ObjName`

ListGadget



Classification	Visual elementary gadget
Generator	ListGadgets.NewFrame
Alias	ListGadget

Function

A ListGadget is designed to visualize list models. At the present time, four list model gadgets are offered: ListModel, Tree, Dag and Directory. Depending on the model chosen, this visual gadget displays different properties. The combination of a ListGadget and a ListModel represents the simplest case, which is very much the same as a List gadget.

Whatever the model is, a list is focused with an ML key click, which places the caret (denoted here by a horizontal line) immediately under the closest list item or at the top of an empty list. Drag on the ML key up or down, above and below the extent of the list window, to scroll through the list. The scrolling speed can be controlled by changing the vertical distance between the ListGadget and the mouse focus: the greater the distance, the faster the scrolling. This is handy for long lists. Alternatively, a long focused list can be explored quickly with the help of the keyboard keys:

– Home	go to the top of the list
– End	go to the end
– PageDown	the bottom item appears at the top
– PageUp	the top item appears at the bottom
– Cursor up	move up one line
– Cursor down	move down one line.

Copying string(s) into the list is done in one of the two classical Oberon ways:
- by focusing the ListGadget first, and then copying a selected text to it (MR + MM keys interclick), or
- by selecting a text stretch first and copying the selection at the caret in the list (ML + MM keys interclick).

If the attributes *Sorted* and *Unique* of the model list are such that the list is unsorted and contains duplicates, the copied items are inserted after the caret.

The MR key is used to select list items. Here also, dragging on the MR key scrolls through the list. Multiple adjacent items are selected in that way, if allowed by the *MultiSel* attribute. Finally, the items to select must not be adjacent, when the *ExtendSel* attribute is set to TRUE. The normal interclick combinations to delete and copy over text from or to text gadgets apply.

The MM key is used to pick an item from the list. Here again, dragging on the MM key scrolls through the list.

When the model linked to a ListGadget is a Tree or a Dag, the displayed model features a few specialities denoted by an arrow pointing to the right or an arrow pointing down. The right arrow indicates that the item contains a collection of hidden sub–items. An MM click on the arrow causes the hidden sub–items to be inserted in the list after the arrowed line. The sub–list is indented to the right by the amount of pixels specified in the *TabSize* attribute.

At the same time, the arrow changes its shape to a down arrow. Another click on the arrow returns the display to its former look. The mouse click sensitive area can be extended to the entire item line when the *ExpOnPoint* attribute is set to TRUE. A sub–list is constructed in the following manner: select any number of adjacent items as a group, or even several such groups in a *focused* list and hit the keyboard cursor key –>. The selected item group(s) become sub–items of the item which precedes them. That item, respectively items, is/are thus arrowed. To remove a sub–list, select any number of items in a *focused* list and hit the keyboard cursor key <–. The selected items are placed at the same level as the item which precedes them, i.e. their root items. That item thus loses its arrow. Finally, the command specified in the *Cmd* attribute is not executed when clicking on an arrowed item, when the *ExpOnPoint* attribute is set to TRUE.

For visualizing a Directory model gadget, one may also use a DirList which displays fancy icons for representing the external storage devices and folders.

Attributes

TabSize
Integer [10] specifying the indenting space of sub–items in pixels. This attribute has no meaning when a ListModel gadget is linked to this gadget.

BackCol
Integer [14] specifying the color of the list background.

TextCol
Integer [15] specifying the color of the list items.

PointCol
Integer [15] specifying the color of the list items which have been pointed at with an MM key click.

InclPath
Boolean [FALSE] – TRUE indicates that the full tree path is created when an item is copied to a text. The path components are separated by "/", the path data is enclosed in double quotes and ends with a carriage return.

MultiSel
Boolean [TRUE] – TRUE indicates that multiple list items may be selected by dragging on the MR key.

ExtendSel
Boolean [FALSE] – TRUE indicates that the selection can be extended to non–adjacent list items by pressing the ALT–key while clicking with the MR key.

ExpOnPoint
Boolean [TRUE] – FALSE indicates that only the right arrow, or down arrow as the case might be, is mouse clicks sensitive. TRUE indicates that the entire item line is sensitive. In that case, the string in the *Cmd* attribute is not executed as a command when the item is arrowed.

Locked
Boolean [FALSE] – TRUE indicates that no new item can be added to the list.

Font
String [Syntax10.Scn.Fnt] specifying the font of the list items.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key. The four *read–only* attributes may be used in conjunction with an activator macro "#" in this string.

The following attributes are *read–only* attributes:

Point
String pointed at with the MM key. It may be used with the activator macro "#" in the *Cmd* attribute.

Sel
String of the first list item selected. It may be used with the activator macro "#" in the *Cmd* attribute.

PointKey
Unique key (integer) of the model item corresponding to the item pointed at with the MM key. It may be used with the activator macro "#" in the *Cmd* attribute.

SelKey
Unique key (integer) of the model item corresponding to the first list item selected. It may be used with the activator macro "#" in the *Cmd* attribute.

Links

VPos
Reference to the model gadget whose *Value* attribute represents the first item shown in the list.

VRange
Reference to the model gadget whose *Value* attribute represents the number of lines in a list (read only).

HPos
Reference to the model gadget whose *Value* attribute represents the first indent shown in the list.

HRange
Reference to the model gadget whose *Value* attribute represents the number of indents (read only).

For all these links, the model value is interpreted as an integer value, though the model may be an Integer, a Real or a String.

Commands

`ListGadgets.InsertVScrollList`
inserts a ListGadget, with a vertical Scrollbar on its right side, at the caret. The inserted gadget is in reality a composite construction based on an Organizer with two components: a ListGadget and a Scrollbar. Each of these components is decorated with a *Constraints* attribute.

`ListGadgets.InsertHVScrollList`
inserts a ListGadget, with a vertical Scrollbar on its right side and a horizontal Scrollbar on the lower side, at the caret. The inserted gadget is in reality a composite construction based on an Organizer with three components: a ListGadget and two Scrollbars. Each of these components is decorated with a *Constraints* attribute.

`Directories.Directory [\p] ObjName`
finds all the file names in the current directory that match a specified mask and inserts them in a list model gadget (ListModel, Tree or Dag) named ObjName in the current context. If the option \p is specified, the file names are prefixed with their relative path in the current directory. When the Directories module is loaded, a background task is installed in the Oberon loop: refer to the remark under CurrentDirectory.

Example

The module `TextPopups` uses this gadget, linked to a ListModel. See description in section 6.2.

ListModel

Classification	Model gadget
Generator	ListModels.NewList
Alias	ListModel

Function

A ListModel stores items in a linear fashion. The list items may be sorted or unsorted and the list may contain duplicates or not. This model gadget is typically used as a model by a ListGadget. Such a combination has the same properties as a List.

Attributes

Sorted
Boolean [FALSE] – TRUE indicates that the list items are sorted.

Unique
Boolean [FALSE] – TRUE indicates that no duplicate list items are allowed.

Commands

See ListGadget.

LogDoc

Classification Visual document gadget
Generator TextDocs.NewLog
Alias LogDoc

Function

A text document (see TextDoc) displaying the system log. If it is opened with the command `Desktops.OpenDoc(LogDoc)`, it appears with a menu bar containing buttons `[Close]`, `[Hide]`, `[Grow]`, `[Copy]`, `[Locate]` and `[Clear]`. The button `[Clear]` empties the log. The button `[Locate]` is used for locating selected syntax errors reported by the compiler in the system log inside the marked module source text. The function of the `[Hide]` button is explained in the Finder commands. This gadget is always linked to a TextGadget model gadget.

Attributes

DocumentName
String [System.Log] specifying the name of the document. This string appears in the NamePlate of the document.

Commands

See TextDoc.

MemoryUsed

Classification	Model gadget
Generator	Gages.NewMem
Alias	MemoryUsed

Function

A MemoryUsed model gadget contains an up–to–date indication of how much of the Oberon heap is currently being used.

Attributes

Value
Integer [?] value representing the memory usage in bytes.

Remark

See CurrentLoad.

Example

Insert a memory monitor at the caret with:

```
  Gadgets.Insert TextField MemoryUsed ~
```

NamePlate

Classification	Visual elementary gadget
Generator	NamePlates.NewNamePlate
Alias	NamePlate

Function

A NamePlate shows the name of the document in the document menu bar or the name of the desktop when located inside one. The name is the string value found in the *DocumentName* attribute of a document gadget. Most editing operations work in the same way as those in a main editable text: the name may be changed and a text stretch may be selected, deleted or copied. Placing the caret is however a little different depending on the environment: if the document is placed on a desktop, the caret is placed with an ML key click anywhere in the NamePlate, whereas if the document is placed in a track, the mouse has to point at the very bottom of it.

PC Native Oberon does not have a directory system. Consequently, the document name appears alone.

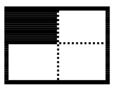
In Linux, Mac and Windows Oberon, the document name is followed by the full path name of the document, which is however often not completely visible. To view the remaining text, place the caret in the NamePlate and scroll right or left with the corresponding keyboard cursor keys. An MM key click on the gadget causes the full name to appear in the system log, by virtue of the default value assigned to the *Cmd* attribute (see below).

Attributes

Value
String [?] indicating the name of a document.

Cmd
String [Out.Echo '#Value'] executed as a command when the gadget is clicked on with the MM key.

Navigator



Classification Visual elementary gadget
Generator Navigators.NewNavigator
Alias Navigator

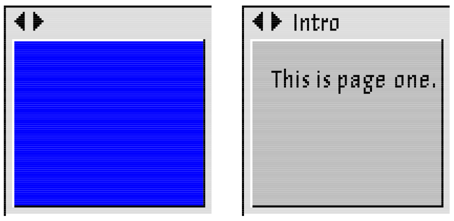
Function

An Oberon desktop, for example `Oberon.Desk` is 2 x 2 times the size of the display screen: normally only the top left quadrant is visible. A Navigator is a miniature representation of the four quadrants: the currently visible one is always shown in black whereas the three remaining ones show the silhouettes of the objects dropped on the desktop. A Navigator allows accessing all four quadrants of a desktop: clicking with the MM key on any quadrant, makes it visible. Gadgets can be moved (or copied) to the neighbouring quadrants as usual by dragging (and clicking) them across the fictitious boundary line. All quadrants display the navigator in the same absolute position on the display screen. The top left corner of the display is a very convenient location for it. Navigators can be used only in desktops; outside desktops they are inactive.

Whereas a Finder can be used to quickly find a document among the many documents piled in the visible part of a desktop, the navigator can be advantageously used to segregate those many documents into up to four application oriented quadrants: this is an entirely arbitrary subdivision left to the user's appreciation and taste. In the realm of program development it seems fairly obvious to elect to use one quadrant for managing source text documents and another for compiling and running programs. Remember that a system log may be opened in each quadrant.

An alternative approach is to use different desktops instead of using several quadrants in a single desktop.

NoteBook



Classification Visual container gadget
Generator NoteBooks.New
Alias NoteBook

Function

A NoteBook organizes a heterogenous collection of visual gadgets as pages of a notebook. When it is instantiated, a NoteBook contains no page. Any visual gadget, except a transparent one, can be inserted into the NoteBook by dropping it inside the top part. Each additional gadget is inserted on a new page at the end. The top part of the NoteBook shows the name of the current page: it is the *Name* attribute of the visual gadget appearing on the page. Two arrowed buttons allow turning the pages forward or backward. All the pages have the same size and resizing a single page resizes all the other pages accordingly. A page can be removed by moving it to a different location out of the NoteBook or with a delete interclick.

Attributes

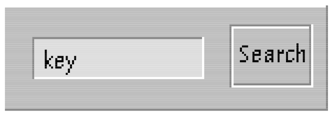
Locked
Boolean [FALSE] – TRUE indicates that pages cannot be resized or removed, and no new page can be added.

Commands

`NoteBooks.Show {"First" | "Last" | "Previous" | "Next" | "page-name"}`
pages to the indicated page of the book in the current context. This command must be executed somewhere inside the NoteBook itself.

`NoteBooks.Show notebook-name {"First" | "Last" | "Previous" | "Next" | "page-name"}`
pages to the indicated page of the named book, that is, the book in the current context identified by `notebook-name`.

Organizer



Classification	Visual container gadget
Generator	Organizers.NewPanel
Alias	Organizer

Function

An Organizer is a Panel extended with a simple constraint solver to reorder the children automatically when the Organizer is changed in size. The constraint system [Car86] is based on virtual wires: four wires are strung between the four sides of a descendant and the edges of the Organizer. The horizontal sides of the child gadget are attached to the top or bottom edges, and the vertical sides to the left or right edges. The lengths of the four wires are given as four numbers in the *Constraints* string attribute of the child. Positive numbers string the wire "outward" and negative numbers "over" the gadget. That is:

```
Gadgets.ChangeAttr Constraints "10 20 30 40" ~
```

sets constraints on the selected child, where the left side of the gadget is 10 pixels from the left edge of the Organizer, the top side 20 pixels from the top edge, the right side 30 pixels from the right edge, and the bottom side 40 pixels from the bottom edge. Thus the sequence of numbers are left, top, right, and bottom wire distances. Changing the 20 to –20 attaches the top side of the gadget 20 pixels from the bottom edge. `TestOrganizer.Panel` is an example including an Organizer.

Attributes and Links

See Panel.

Commands

```
Organizers.Exchange
```
flips the marked Organizer between an Organizer and a Panel. *Constraints* attributes assigned to contained gadgets remain in existence.

Also refer to the commands of PanelDoc.

Outline

■ ▶This is the **folded** text.◀

Classification	Visual elementary gadget
Generator	Outlines.New
Alias	Outline

Function

An Outline, which functions inside a text *only*, implements a way to fold text and gadgets enclosed between two arrows away into the gadget. When folded, the Outline is a black rectangle. An MM click on a folded Outline "unfolds" its content between the arrows. The two arrowed parts of an unfolded Outline are themselves Outlines. The left part has the same attributes as the folded Outline. Text and gadgets in an unfolded Outline may be freely edited and Outlines may recursively contain other Outlines to any depth. An MM click on one of the two arrows folds the text back into the Outline: an interesting device for presenting detail or explanatory information without overloading a main text. Be cautious however, deleting an arrowed Outline may prevent an Outline from being folded again. Note that the text editing commands "search", "replace", "replace all" , etc. do not operate on folded text, and that the compiler cannot compile the text contained in *folded* Outlines. You first need to unfold them all (refer to the commands below).

Attributes

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key. A different command string may be assigned to the two parts of an unfolded Outline.

Commands

`Outlines.Insert`
embeds the most recent selection in a folded Outline, or if no selection exists, inserts an empty Outline at the caret in a text.

`Outlines.Collapse *`
`Outlines.Collapse`
collapses all Outlines, including the nested ones, in the marked text document (*) or in the text document in which it is executed.

`Outlines.Expand *`
`Outlines.Expand`
expands all Outlines, including the nested ones, in the marked text document (*) or in the text document in which it is executed.

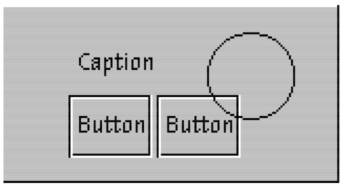
`Outlines.Replace`
replaces the text stretch previously searched with an `Outlines.Search` command with the selection in the document. If the searched text pattern is found in an Outline, it is expanded first.

`Outlines.ReplaceAll`
replaces all instances of the text stretch previously searched with an `Outlines.Search` command with the selection in the document. If the searched text patterns are found in Outlines, they are expanded.

`Outlines.Search *`
`Outlines.Search`
searches for the selected text stretch, starting at the caret in a document or if the caret is not set, from the beginning of the marked text document (*) or

document in which it is executed. If the text pattern is found in an Outline, it is expanded.

Panel



Classification Visual container gadget
Generator Panels.NewPanel
Alias Panel

Function

A Panel is like a sheet of paper on which gadgets are organized in a two–dimensional fashion. The gadgets contained inside are called the *direct descendants* or children of that Panel, and the Panel itself the children's father, context or *container*. As Panels are gadgets too, Panels may appear inside Panels, nested to any depth. They support operations on groups of gadgets plus the layout and alignment of children. The children of a Panel are organized in a priority sequence and may overlap each other. A child keeps its priority when moved around in a Panel and newly inserted gadgets are always placed in front. Oberon commands put a gadget behind or in front of other gadgets. The simplest way to bring a gadget to the front is to pick it up and consume it with an MM + ML key interclick. A snapping grid allows easy positioning of children. The grid is not visible and the grid spacing is adjusted with the *GridSnap* attribute. Children can be aligned to the same base line, height, size, or width, or organized vertically and horizontally. Operations on Panels can be performed with the tools provided in the `Gadgets.Panel`. Behind the scene, these tools use the command `Panels.Align` and other commands described below.

Panels use the normal mouse conventions:

An ML key click inside a Panel sets the caret. Combined with an MM key interclick, a shallow copy of the selection is placed at the caret.

An MM key click on a child gadget activates it. Use the MM key inside the border to pick up a Panel and move it around dragging on the key. The border is not visible but its width is adjusted with the *Border* attribute.

An MR key click on a child selects it. Dragging on the MR key in a free area causes a spanning rectangle to appear for selecting several gadgets at the same time. All the gadgets *entirely* contained in the rubber–banding rectangle are selected when the key is released, and they can then be edited as a group. An MR key drag followed by an ML key interclick deletes all the selected gadgets. An MR key drag followed by an MM key interclick makes a shallow copy of the selection and copies it over to the caret position. An MR key click in a free area of the Panel clears the selection. An MR key click inside the border of a Panel selects it. Another click deselects it.

Key		**Associated action**
ML	Point	Set the caret to mark the insertion point.
ML + MM	Copy to	Set the caret and copy an existing selection to the caret.
MM	Activate	Activate or manipulate a child: depends on the child type and on the mouse focus.
MM + ML		Bring the focused child to the front.

MR	Select	Select child or group of children.
MR + ML	Delete	Select child(ren) and delete.
MR + MM	Copy over	Select child(ren) and copy over to caret.
ML + MM + MR	Nullify	Nullify current mouse action.

A Panel can be converted to an Organizer, an extension of a Panel, with the `Organizers.Exchange` command. This works on a PanelDoc too.

Attributes

Color
Integer [13] specifying the background color of the Panel. The color of a Panel can also be changed with the ColorPicker. If the gadget is linked to a backdrop picture, it takes precedence over the color.

Border
Integer [2] specifying the width in pixels of the clipping area around the edge of the panel. Children are clipped when placed overlapping this area. This value does not influence the look of the Panel border. Use the attribute *Flat* for that purpose.

GridSnap
Integer [2] specifying the snapping grid spacing in pixels. The grid is not visible.

Flat
Boolean [FALSE] – FALSE indicates that the gadget must be surrounded by a 3D border.

Texture
Boolean [FALSE] – TRUE indicates that, if a backdrop picture has been installed with a `Panels.ChangeBackdrop` command, it should fill the whole panel background as a texture. When the value FALSE is assigned, the picture is placed in the upper left corner of the Panel.

Locked
Boolean [FALSE] – TRUE indicates that the Panel's content cannot be edited.

Links

Picture
Reference to the backdrop picture which is installed, changed or removed with the `Panels.ChangeBackdrop` command.

Commands

`Panels.Recall`
recalls the gadgets deleted most recently and inserts them at the caret.

`Panels.ToFront`
brings the selected gadgets to the front of the Panel.

`Panels.ToBack`
puts the selected gadgets to the back of the Panel.

`Panels.ChangeBackdrop picture-fileName`
changes the backdrop of the selected Panels. To clear the backdrop, use a non–existing Picture file name or else use Columbus. See also Picture.

`Panels.Align type`
aligns the selected gadgets along an imaginary reference line or resizes them. For example, in a *Left* alignment all selected gadgets must be lined up on their left edges, the reference edge being that of the left most gadget. The following

alignment types are available:

`top`	to common top border
`bottom`	to common bottom border
`left`	to common left border
`right`	to common right border
`vertical`	in a vertical fashion
`horizontal`	in a horizontal fashion
`width`	to the same width
`height`	to the same height
`size`	to the same size
`verticalcenter`	centered on a common vertical line
`horizontalcenter`	centered on a common horizontal line

When the parameter is `width`, `height` or `size`, the largest dimension of the selected gadgets is applied to all of them.

`Organizers.Exchange`
flips the marked Panel between an Organizer and a Panel. *Constraints* attributes assigned to contained gadgets remain in existence.

Also refer to the commands of PanelDoc.

PanelDoc

Classification	Visual document gadget
Generator	PanelDocs.NewDoc
Alias	PanelDoc

Function

A PanelDoc is a document containing a single Panel or Organizer. If it is opened with the command `Desktops.OpenDoc` `(PanelDoc)`, it appears with a menu bar containing buttons captioned `[Close]`, `[Hide]`, `[Grow]` and `[Store]`. The function of the `[Hide]` button is explained in the Finder commands. The document may be given a name and can be saved in a file with that name.

Attributes

DocumentName
String specifying the name of the document. This string appears in the NamePlate of the document.

Commands

`PanelDocs.AppendPanel` `L.O`
inserts a pre–fabricated component (Panel or Organizer) `L.O` at the bottom of an existing container (Panel or Organizer). The command must be executed in the context of the container. This effect is obtained by inserting a gadget which has a *Cmd* attribute, a Button for example, in the container. Then, the command is assigned as an attribute value. When the gadget is activated, the object `L.O` is appended. This operation can be undone in two ways by executing one of the two commands described below.

`PanelDocs.RemovePanel`
removes a component (Panel or Organizer) from its context (Panel or Organizer). The command must be executed in the context of the component itself. This effect is obtained by inserting a gadget which has a *Cmd* attribute in the component. Then, the command is assigned as attribute value. When the gadget is activated, the component is removed.

`PanelDocs.DetachPanel`
removes a component (Panel or Organizer) from its context (Panel or Organizer) and opens a separate PanelDoc that contains it. The command must be executed in the context of the component itself. This effect is obtained by inserting a gadget which has a *Cmd* attribute in the component. Then, the command is assigned as an attribute value. When the gadget is activated, the component is removed.

`Rembrandt.Panel` and `Leonardo.Panel` use these three commands.

`Desktops.Recall`
recalls the document closed most recently.

Picture

Classification	Model object
Generator	Pictures.NewPicture
Alias	Picture

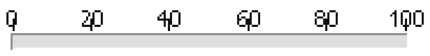
Function

A Picture is a model object containing a colored bitmap visualized by a RembrandtFrame to which it is linked as a model. It has no attributes at all, and no further attribute can be attached to it. A picture editor called Rembrandt is included in the system to edit such bitmaps.

Attributes

Implementation restriction: Picture gadgets cannot be named or have attributes.

ProgressMeter



Classification	Visual elementary gadget, transparent
Generator	ProgressMeters.NewFrame
Alias	ProgressMeter

Function

Display a meter showing the value of a linked model gadget, e.g. an Integer or a Real. This visual gadget typically visualizes the Integer model gadget named "Progress" located in the `NetDocs.Lib` Library provided for this purpose. A ProgressMeter having that gadget as a model shows the progress of a data transfer over the network interface. An example is found in the `HyperDocs.Panel`

Attributes

Value
Integer [0] specifying the current value of the meter. This value is normally stored in a linked *Model* gadget.

Min
Integer [0] specifying the minimum value of the meter.

Max
Integer [100] specifying the maximum value of the meter.

Color
Integer [1] specifying the color index of the measured *Value*.

Step
Integer [20] specifying the interval at which graduations must appear.

Marks
Boolean [TRUE] – TRUE indicates that graduation marks must appear at the interval specified in *Step*.

Labels
Boolean [TRUE] – TRUE indicates that graduation values must appear at the interval specified in *Step*.

Field
String [ ] indicating which attribute of the linked model gadget should be visualized. When empty, the *Value* attribute of the linked model gadget is used by default.

Real

Classification	Model gadget
Generator	BasicGadgets.NewReal
Alias	Real

Function

A Real is a model gadget that stores a LONGREAL value.

Attributes

Value
Real [0] value.

Rectangle

Classification	Visual elementary gadget, transparent
Generator	BasicFigures.NewRect
Alias	Rectangle

Function

A rectangle, possibly filled. Selecting a Rectangle causes two diagonally opposite control points to appear. The control points can be adjusted by dragging on the MM key. The Rectangle takes its final shape when the MM key is released.

Attributes

Color
Integer [15] specifying the color of the Rectangle. If the Rectangle is filled, the color also applies to the interior. The color can also be changed with the ColorPicker.

Width
Integer [1] specifying the width in pixels of the Rectangle.

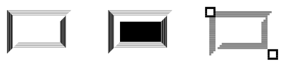
Pattern
Integer [0] specifying in which pattern the perimeter line is to be drawn. If the Rectangle is filled, the pattern also applies to the interior. Patterns are numbered from 0 to 8. Refer to the *Pattern* attribute of Circle.

Closed
Boolean [FALSE] – not interpreted.

Filled
Boolean [FALSE] – TRUE indicates that the Rectangle must be filled. If so, the filled part has the color specified by the *Color* attribute and the pattern specified by the *Pattern* attribute.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key.

Rectangle3D



Classification	Visual elementary gadget, transparent
Generator	BasicFigures.NewRect3D
Alias	Rectangle3D

Function

A rectangle with a 3D–effect, possibly filled. Selecting a Rectangle3D causes two diagonally opposite control points to appear. The control points can be adjusted by dragging on the MM key. The rectangle takes its final shape when the key is released.

Attributes

See Rectangle.

Reference

Classification	Model gadget
Generator	RefGadgets.NewReference
Alias	Reference

Function

A reference to any object which is an extension of Objects.Object. This model gadget is visualized by a RefFrame gadget.

RefFrame

☒

Classification Visual elementary gadget
Generator RefGadgets.NewFrame
Alias RefFrame

Function

A RefFrame (also called *visual reference gadget)* provides a frame for visualizing a Reference to any object of type Objects.Object. This frame can have three different representations:

☒ – when it does not refer to an object,
■ – when it refers to a visual gadget,
● – when it refers to a model gadget.

The practical advantage of this gadget is that it contains a Reference to an object irrespective of its type (visual gadget or model) and of its size. The Reference can be dragged–and–dropped or copied over to another context just as easily as the object it represents.

A visual reference can be changed either by dropping a visual gadget into its frame or by copying over a selected visual gadget. Alternatively, a reference from another reference can be dropped or copied over with the same effect. When a reference is changed by such a user interaction, it will execute the command specified in the *ConsumeCmd* attribute. To remove the reference use a delete interclick on the reference. If a model reference is dropped on a visual gadget, the reference becomes a model of it. Like most other visual gadgets, a reference may execute a command when clicked on. This gadget is used in Columbus for which it was originally designed, but can be used in any other environment too.

Attributes

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key.

ConsumeCmd
String [ ] executed as a command when a gadget is dropped inside the RefFrame and when the Reference is deleted with a delete interclick.

Locked
Boolean [FALSE] – TRUE indicates that the RefFrame refuses 'drop', 'copy over' and 'delete' actions.

Drag
Boolean [TRUE] – TRUE indicates that the reference may be dragged away.

Links

Value
Reference to the object represented.

Commands

`RefGadgets.Creator`
creates a new object and sets the *Value* link of an existing destination RefGadget which is either the selected or the marked or the executor gadget.

RembrandtDoc

Classification	Visual document gadget
Generator	RembrandtDocs.NewDoc
Alias	RembrandtDoc

Function

A picture document containing a single RembrandtFrame gadget. When the document is opened with the command `Desktops.OpenDoc(RembrandtDoc)`, it appears with a menu bar containing buttons captioned `[Close]`, `[Hide]`, `[Grow]`, `[Copy]`, `[+]`, `[-]` and `[Store]`. The button `[+]` magnifies the selection, or the entire picture if there is no selection. A magnification of 4, 7, 10, 13 and 16 is obtained by clicking the button repeatedly, a maximum of 5 times. This makes it very easy to retouch pictures pixel–wise. The button `[-]` reduces the entire picture. It may be clicked a maximum of 5 times in succession until the original size is restored, and *not* beyond. The function of the `[Hide]` button is explained in the Finder commands. The contained Picture can be edited with the tools provided in the `Rembrandt.Panel`. RembrandtDocs are usually stored in files having a file name extension ".Pict". The Oberon system can also read other graphical files such as GIF or JPEG files. The [PictureConverters] section of the registry contains a list of acceptable file formats.

Attributes

DocumentName
String specifying the name of the document. This string appears in the NamePlate of the document.

Commands

`Desktops.Recall`
recalls the document closed most recently.

RembrandtFrame

Classification	Visual elementary gadget
Generator	Rembrandt.New
Alias	none

Function

A RembrandtFrame provides a frame for visualizing a single Picture model gadget. The Picture model gadget may be replaced by another one, using Columbus for instance, but it may never be deleted.

Attributes

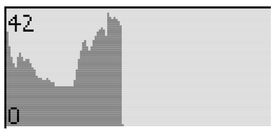
Color
Integer [14] specifying the color index of the background.

Cmd
String [ ] executed as a command when the gadget is *locked* and clicked on with the MM key.

Locked
Boolean [FALSE] – TRUE indicates that the contained Picture cannot be edited, but it can be scaled by rezising the frame. When the gadget is unlocked, the Picture returns to its original size.

Border
Boolean [TRUE] – TRUE indicates that the gadget must be surrounded by a 3D–border.

Scope



Classification Visual elementary gadget
Generator Gages.NewFrame
Alias Scope

Function

A Scope is a bar chart of the previous values of the model gadget linked to it. The model must be an Integer. A new vertical bar is added to the right when the model is updated. When the chart fills the Scope, the bar chart scrolls to the left as new vertical bars are added. The minimum and the maximum value observed in the visible portion of the chart are displayed on the left. The Scope adjusts its scale automatically.

Remark

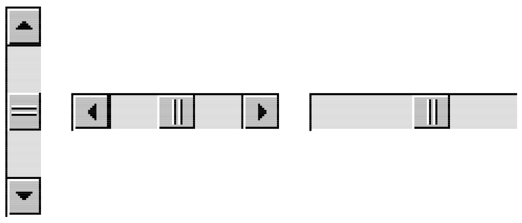
See CurrentLoad.

Example

Insert a load monitor at the caret with the command:

```
Gadgets.Insert Scope CurrentLoad ~
```

Scrollbar



Classification Visual elementary gadget
Generator Scrollbars.New
Alias Scrollbar

Function

A Scrollbar visualizes a continuous range of values from which the user can select by adjusting the Scrollbar's knob position with the MM key. If arrow boxes are used, the knob can also be adjusted by MM key clicks on these arrows. Keeping this key pressed, also adjusts it repeatedly until the key is released. It can be linked to String, Integer or Real gadgets. The String must represent a numerical value. It is typically used in combination with a ListGadget.

Attributes

Min
Integer [0] specifying the minimum value of the attribute *Value*. Alternatively, this value can be stored in a linked *Min* gadget.

Max
Integer [100] specifying the maximum value of the attribute *Value*. Alternatively, this value can be stored in a linked *Max* gadget.

Value
Integer [50] specifying the current value of the Scrollbar. This value is normally stored in a linked *Model* gadget.

StepSize
Integer [1] specifying by how many units the current value must be incremented or decremented by MM key clicks on the arrow boxes, if present, or by dragging the knob with the MM key. Alternatively, this value can be stored in a linked *Step* gadget.

BackColor
Integer [14] specifying the color of the background.

Field
String [ ] indicating which attribute of the linked model gadget should be visualized. When empty, the *Value* attribute of the linked *Model* gadget is used.

Cmd
String [ ] executed as a command after the knob's position was changed.

Vertical
Boolean [TRUE] – TRUE indicates that the Scrollbar is upright.

ArrowBoxes
Boolean [TRUE] – TRUE indicates that the Scrollbar must be decorated with MM key sensitive arrow boxes at its extremities.

HeavyDrag
Boolean [FALSE] – TRUE indicates that the model gadget linked to the Scrollbar is updated at the same time as the knob is moved. FALSE indicates that the model gadget is updated only when the mouse key is released. This is relevant in the case where multiple views of the model exist in the display space.

Links

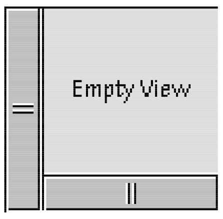
Min
Reference to the model gadget whose *Value* attribute is used as the *Min* value.

Max
Reference to the model gadget whose *Value* attribute is used as the *Max* value.

Step
Reference to the model gadget whose *Value* attribute is used as theStep value.

For all these links the model value is interpreted as an integer value, though the model may be an Integer, a Real or a String.

ScrollView



Classification	Visual elementary gadget
Generator	ScrollViews.NewView
Alias	ScrollView

Function

A ScrollView is a View which can be fitted with a horizontal or a vertical Scrollbar or both according to its attribute values.

Attributes

HScrollBar
Boolean [TRUE] – TRUE indicates that a horizontal Scrollbar is used.

VScrollBar
Boolean [TRUE] – TRUE indicates that a vertical Scrollbar is used.

Set

Classification	Model gadget
Generator	SetGadgets.NewSet
Alias	Set

Function

A Set is a model gadget that stores a 32–bit set. It is normally visualized by a SetFrame gadget, but also a TextField, a Slider, a Button, a CheckBox or a group of them may be linked to it.

Attributes

String
String [{}] representing a 32–bit set (Oberon language notation).

Value
Integer [0] representation of the 32–bit set in the *String* attribute.

SetFrame

Classification	Visual elementary gadget
Generator	SetGadgets.NewFrame
Alias	SetFrame

Function

A SetFrame provides a frame for visualizing a set. It is most frequently linked to a Set model gadget, but an Integer or a String (representing a set in the Oberon language notation) may also be used. Any bit can be included (indicated by a vertical bar) or excluded by MM key clicking on its placeholder. The least significant bit is at the left in position 0.

Attributes

Value
Integer [0] representation of the 32–bit set visualized. This value is normally stored in a linked *Model* gadget.

String
String [{}] representing a 32–bit set (Oberon language notation).

Field
String [ ] indicating which attribute of the linked model gadget should be visualized. When empty, the *Value* attribute of the linked model gadget is used by default.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key.

Sisiphus

Classification	Visual elementary gadget
Generator	Sisiphus.New
Alias	Sisiphus

Function

An animated cartoon representing a walking man called Sisiphus. An MM key click on a Sisiphus activates a screen saver which can be interrupted by pressing the Neutralize key.

Attributes

Color
Integer [14] specifying the background color of Sisiphus. The color can also be changed with the ColorPicker.

Age
Integer [0] specifying the age of the gadget. It counts how many times the gadget has been updated by the background task.

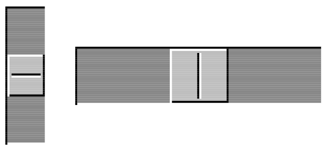
Commands

`Sisiphus.Sync`
synchronizes the selected Sisiphus gadgets and resets their age to 0.

Remark

When the `Sisiphus` module is loaded, a background task is installed in the Oberon loop. To remove the task, remove all Sisiphus gadgets it controls from the display space and execute a `System.Free Sisiphus` command.

Slider



Classification	Visual elementary gadget (superseded by Scrollbar)
Generator	BasicGadgets.NewSlider
Alias	Slider

Function

A Slider visualizes a continuous range of values from which the user can make a selection by adjusting the Slider's knob position with the MM key. It can be linked to a String, Integer or Real model gadget. The Slider is an earlier development, now replaced by the Scrollbar. It should no longer be used.

Attributes

Min
Integer [0] specifying the minimum value of the attribute *Value*.

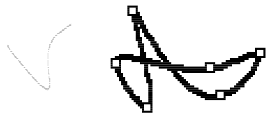
Max
Integer [100] specifying the maximum value of the attribute *Value*.

Value
Integer [50] specifying the current value of the Slider. This value is normally stored in a linked *Model* gadget.

Field
String [ ] indicating which attribute of the linked model gadget should be visualized. When empty, the *Value* attribute of the linked model gadget is used by default.

Cmd
String [ ] executed as a command after the knob's position was changed.

Spline



Classification	Visual elementary gadget, transparent
Generator	BasicFigures.NewSpline
Alias	Spline

Function

A spline, possibly filled. Selecting a Spline causes control points to appear at the line joints. The control points can be adjusted by dragging on the MM key. The Spline takes its final shape when the MM key is released. While dragging a control point, an MR key interclick inserts an additional control point, whereas an ML key interclick deletes that control point.

Attributes

Color
Integer [15] specifying the color of the Spline. If the Spline is filled, the color also applies to the interior. The color can also be changed with the ColorPicker.

Width
Integer [1] specifying the width in pixels of the Spline.

Pattern
Integer [0] specifying in which pattern the Spline is to be drawn. If the Spline is filled, the pattern also applies to the interior. Patterns are numbered from 0 to 8. Refer to the *Pattern* attribute of Circle.

Closed
Boolean [FALSE] – TRUE indicates that the two end points of the Spline should be joined.

Filled
Boolean [FALSE] – TRUE indicates that the Spline must be filled. If so, an open Spline appears closed, and the filled part has the color specified by the *Color* attribute and the pattern specified by the *Pattern* attribute.

Cmd
String [ ] executed as a command when the gadget is clicked on with the MM key.

String

Classification	Model gadget
Generator	BasicGadgets.NewString
Alias	String

Function

A String is a model gadget that stores a string value of up to 64 characters. It is usually visualized by a TextField. If the string represents a set in the Oberon language notation, it may be visualized by a SetFrame.

Attributes

Value
String [ ] value.

Text

Classification	Model object
Generator	Texts.New
Alias	Text

Function

Text is the abstract data type used for manipulating text in the Oberon system. Texts can be of (practically) unlimited length, and contain font, color, and vertical offset information. Gadgets can float inside a text stream. Texts are not true gadgets and can have no attributes, except the *Name* attribute. More about text can be found in Chapter 2 describing the textual user interface of Oberon.

TextDoc

Classification Visual document gadget
Generator TextDocs.NewDoc
Alias TextDoc

Function

A TextDoc is a document containing a single TextGadget. If it is opened with the command `Desktops.OpenDoc (TextDoc)`, it appears with a menu bar containing buttons captioned `[Close]`, `[Hide]`, `[Grow]`, `[Copy]`, `[Search]`, `[Rep]`, `[RepAll]` and `[Store]`. The `[Search]` button searches for the selected text stretch in the text. Each time the `[Search]` button is pressed, the caret advances to the next location in the text where the pattern appears. When the end of text is reached, the searching wraps around to the beginning of the text. The `[Rep]` button replaces the last searched pattern with the current text selection and the caret then advances to the next occurrence of the searched pattern. Repeatedly pressing the `[Rep]` button replaces all the following occurrences of the search pattern. A replacement can be skipped by pressing the `[Search]` button as many times as required. At any time the `[Rep]` button can be pressed to start replacing occurrences of the search pattern. The `[RepAll]` button is not present when the document is opened in the system track. The function of the `[Hide]` button is explained in the Finder commands.

Attributes

DocumentName
String specifying the name of the document. This string appears in the NamePlate of the document.

Commands

`TextDocs.ChangeColor colno`
applies the color specified by the parameter to the most recent text selection. Colors are numbered 0 to 255. For example, white is 0, red is 1, green is 2, blue is 3 and black is 15. The color can also be changed with the ColorPicker.

`TextDocs.ChangeFont fontname`
applies the font specified by the parameter to the most recent text selection. The font name has to be specified in full, e.g. `Courier10.Scn.Fnt`. The default font is Oberon10.Scn.Fnt for PC Native Oberon. For other implementations, the default font is defined in the [System] section of the registry. If it is not defined there, Syntax10.Scn.Fnt is used.

`TextDocs.ChangeOffset pixels`
applies the vertical offset specified by the parameter to the most recent text selection. The offset is specified in number of pixels. Positive numbers make text super–scripts, and negative numbers make sub–scripts.

`TextDocs.Clear`
clears the content of a document (often used in menu bars).

`TextDocs.Controls *`
`TextDocs.Controls`
toggles the visibility of the text control gadgets, that is, TextHyperlinks and TextStyles, in the marked (*) or in the selected document. TextStyles which have their *Pagebreak* attribute set on are always visible.

`TextDocs.Locate`
locates the compiler error in the marked text document. The compiler error position must be selected.

`TextDocs.PrintSetup [\Hon|\Hoff] [\Pon|\Poff]`
controls the printing of headers and page numbers, displaying the new setup in the system log. If any of the parameters is missing, the current setup is displayed.

`TextDocs.Recall`
inserts the most recently deleted piece of text at the position of the caret. The same text can be recalled many times, until another piece of text is deleted, the Backspace key is pressed or the Delete key is pressed.

`TextDocs.Replace`
replaces the previously searched text stretch with the selection (used in the `[Rep]` Button of menu bars).

`TextDocs.ReplaceAll`
replaces all instances of the previously searched text stretch with the selection in the document (used in the `[RepAll]` Button of menu bars).

`TextDocs.Search`
searches the selected text stretch in the document in which the command is executed (used in the `[Search]` Button of menu bars). Differences in the text attributes font, color and vertical offset are ignored.

`TextDocs.Search "string"`
searches the specified string in the document in which the command is executed. Differences in the text attributes font, color and vertical offset are ignored.

`TextDocs.SearchColor`
searches a specified color in a text document. Select a text stretch first. The color to search is the color of the first character in the stretch. Then, set the caret in the text and execute the command. The text stretch with the specified color appears selected and the caret is set at the end of it.

`TextDocs.SearchDiff`
`TextDocs.SearchDiff \w`
compares two open documents and searches the first difference between them, starting at the selection in the last two selected texts. The first position where a difference is found will be selected. The `\w` option regards sequences of white spaces (blanks, TAB characters or CR characters) as a single space.

`TextDocs.Show M.word`
`TextDocs.Show ^`
opens a TextDoc displaying the text of module `M`, with the caret positioned at the first occurrence of the specified `word`, provided the source text file M.Mod is available. Otherwise, an empty document is opened. If `word` is not found, the text is positioned at the beginning and the caret is not set.

`Desktops.PrintDoc Default *`
`Desktops.PrintDoc Default file~list`
prints a document on the printer specified by the first name. Either the marked document or a list of document names are accepted as parameters. Depending on your platform, the printer name might vary, but in most cases the printer name is simply ignored. Please check your implementation guide for more details.

`Desktops.Recall`
recalls the document closed most recently.

In addition, a large palette of text editing commands is provided by the EditTools tool described in Chapter 2.

TextField

Classification	Visual elementary gadget
Generator	TextFields.NewTextField
Alias	TextField

Function

A TextField allows the editing of a single line of text. A TextField can be linked to Integer, Real, Set, String, or compatible model gadgets. It can be used to visualize the *String* attribute of a Set in Oberon set notation. Full Oberon–like text editing capabilities are available for TextFields. The caret or focus point is set with an ML key click and selections are processed with the MR key. A TextField enters a temporary, local editing mode during editing. During this time the frame of the TextField seems to pop out from the screen. If part of the text is not visible, scrolling to the left or to the right is possible with the corresponding keyboard cursor keys. While in editing mode, changes made to the contained text string are not immediately reflected in the model gadget linked to the TextField. As soon as the cursor is removed or when the CR key is pressed, the local editing mode is left and the model and view are made consistent. The previous value (if still available from the model gadget) is restored in the TextField when the Neutralize key is pressed. When TextFields are linked to model gadgets other than the string model gadget, a value conversion will take place to the same format as the model gadget. In some cases, a conversion is not possible, resulting in a default value being shown depending on the exact nature of the model gadget. When multiple TextFields are located in the same container, the TAB key advances the caret from one TextField to the next. Setting the caret and pressing the left and right arrow keys scrolls the contained text horizontally when the content is wider than the width of the TextField.

Attributes

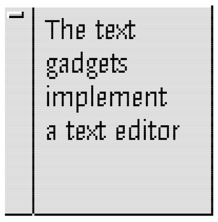
Value
String [ ] value of the TextField. This value is norammly stored in a linked *Model* gadget.

Color
Integer [14] specifying the color of the TextField. The color can also be changed with the ColorPicker.

Field
String [ ] indicating which attribute of the linked model gadget should be visualized. When empty, the *Value* attribute of the linked model gadget is used by default.

Cmd
String [ ] executed as a command when the gadget is focused and the RETURN key is hit.

TextGadget


The text
gadgets
implement
a text editor

Classification	Visual container gadget
Generator	TextGadgets.New
Alias	TextGadget

Function

A TextGadget visualizes a Text model and implements a text editor. Texts may contain visual gadgets floating along in the text. A scrollbar with a small crossbeam on the left allows you to scroll forward or backward in the text. The ML key scrolls the underlined text line to the top of the gadget (scroll forward) and the MR key scrolls the underlined text line to the bottom of the gadget (scroll backward). The MM key is used to position absolutely in a text. An MM + ML key interclick scrolls to the end of the text, while an MM + MR key interclick scrolls to the beginning of the text. When users are typing text, the text scrolls up to show what is being typed. A more detailed description of using and editing text can be found in the chapter on the textual user interface. The Text model gadget may be replaced by another one, using Columbus for instance, but it may never be deleted. Selected gadgets can be resized using the `Gadgets.Panel`

Attributes

Color
Integer [14] specifying the color of the background. The color can also be changed with the ColorPicker.

Flat
Boolean [FALSE] – FALSE indicates that the gadget must be surrounded by a 3D border.

Point
String [ ] containing the last word clicked on with the MM key. The value of this attribute can be used by the *Cmd* attribute (*read–only*).

Locked
Boolean [FALSE] – TRUE indicates that the gadget's *direct* descendant cannot be edited. The components themselves remain unlocked and can be manipulated.

Cmd
String [ ] executed as a command when clicking on a word with the MM key. Since the *Point* attribute contains the Oberon name or string that was clicked on, a command string such as `Desktops.OpenDoc #Point` will open the document whose name is clicked on.

Commands

See TextDoc.

TextHyperlink

Classification Visual elementary gadget, transparent
Generator TextGadgets.NewControl
Alias TextHyperlink

Function

A TextHyperlink is a text control gadget. The visibility of the TextHyperlink gadgets is toggled with the `TextDocs.Controls` command. A TextHyperlink contains an Oberon command to be executed when the *colored* piece of text located immediately in front and delimited by a blank at its left is activated with an MM key click. That piece of text can be activated independently of the visibility of the TextHyperlink but the color of the text must differ from black (15). The *Cmd* attribute value is typically `Desktops.OpenDoc docuName` or `Desktops.ReplaceDoc docuName`, but any other suitable command string may be used. To edit this field, make the gadget visible first and then click on it with the MM key: a Columbus Panel is opened automatically, allowing an easy editing of the command.

Attributes

Cmd
String [ ] executed as a command when the colored text in front of the TextHyperlink is clicked on with the MM key. The TextHyperlink must not necessarily be visible.

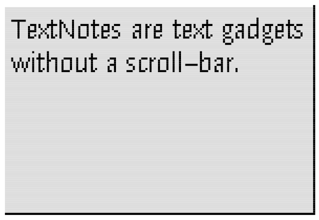
Commands

```
TextDocs.Controls*
TextDocs.Controls
```
toggles the visibility of the text control gadgets, that is, TextHyperlinks and TextStyles, in the marked (*) or in the selected document.

Example

TextHyperlinks are used extensively by the many `.Tool` documents supplied.

TextNote


TextNotes are text gadgets without a scroll–bar.

Classification	Visual container gadget
Generator	TextGadgets.NewNote
Alias	TextNote

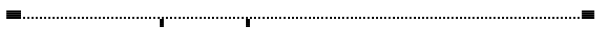
Function

A TextNote is a TextGadget without a scrollbar. A TextNote will adjust itself in size depending on the size of the text it contains. More details about text editing can be found in the chapter on the textual user interface. This gadget is often used as a memo pad or post–it note. Sometimes they are used as the pages of a NoteBook. Alternatively, they are used in pop–up menus as an alternative to a list. TextNotes may be colored to enhance their presentation.

Attributes and Commands

See TextGadget.

TextStyle



Classification	Visual elementary gadget, transparent
Generator	TextGadgets.NewStyleProc
Alias	TextStyle

Function

A TextStyle is a text control gadget that influences the formatting of text. The visibility of style gadgets is toggled with the `TextDocs.Controls` command. They have the shape of a thin horizontal dotted line with black weights at the end. A style can be inserted at the caret with the `TextGadgets.NewStyle` command, or by pressing CTRL–ENTER. In that case all the styles are made visible. On the Macintosh, use the num–lock key instead. The style is divided into two sections: the top part, above the dotted line, controls the formatting whilst the bottom part controls the setting of tab stops. MM key clicks *above* the dotted line are as follows: when pressing and dragging on the area the black weights occupy, the left border and the formatting width can be specified; clicking next to the weight (but not on top of it), switches the weight on and off. The weights pull the left and right text ends of a text line to them. Left, right, block and center adjust can be selected in this way:

A left weight. Left adjust mode with word wrapping.
A right weight. Right adjust mode.
A weight to the left and the right. Block adjust mode.
No weights. Center adjust.

A new tab stop is inserted with an MM + ML key interclick when the mouse focus is positioned *below* the dotted line. The tab stop shows up as small black rectangle and can be moved by dragging on the MM key. Tab stops are removed by dragging them completely out of the TextStyle to the left or to the right. All these adjustments can be made more comfortably than with the mouse, by assigning values to the attributes *Tabs*, *Left* and *Width* with the help of Columbus. As tabbing does not make much sense in center or right adjust mode, in such cases the tab stops are not visible. Copying a stretch of text always makes a copy of the gadgets contained in that stretch (TextStyles are gadgets too). The *Pagebreak* attribute is used to specify if a page break should be inserted before the style when printing. Styles that cause a page break are shown with a solid instead of a dotted line. When a TextStyle is named and placed in a public library, it can be re–used many times in the same document or even in different documents, which is very handy for creating a uniformly formatted document. The `Libraries.Panel` offers three ways to retrieve a TextStyle (or any other gadget) from a public library: a reference, a shallow copy or a deep copy. The most general and most flexible approach is to retrieve it by reference. In that manner, the format of the document(s) in which it is used can be modified by simply changing the attributes of the (master) TextStyle stored in the public library. These TextStyles are easily recognizable by the label at the right end: `libraryName.TextStyleName`

Attributes

Pagebreak
Boolean [FALSE] – TRUE indicates that a page break must be inserted at this text position during printing. Styles with this flag set are shown as a solid line and are always visible in the text.

WYSIWYG
Boolean [FALSE] – TRUE indicates that when the document is printed the

printout corresponds to the display representation. Otherwise, only printer metrics are used.

Adjust
Boolean [TRUE] – TRUE indicates that the printout must be adjusted to the printer frame.

Tabs
String [ ] of integers delimited by commas, specifying the position of the tab stops in pixels. A maximum of 32 tab stops can be set. The tabs are not visible in center or right adjust mode.

Left
Integer [0] specifying the position of the left end in pixels.

Width
Integer specifying the width of the TextStyle in pixels.

Commands

```
TextDocs.Controls *
TextDocs.Controls
```
toggles the visibility of the text control gadgets, that is, TextHyperlinks and TextStyles in the marked (*) or in the selected document. TextStyles which have their *Pagebreak* attribute set on are always visible.

```
TextGadgets.NewStyle
```
inserts a style at the caret.

Example

TextStyles are used extensively by the many `.Tool` documents supplied.

TimeStamp

25.08.97 12:03:47

Classification	Visual elementary gadget, transparent
Generator	TimeStamps.New
Alias	TimeStamp

Function

When a TimeStamp is inserted at the caret, it shows the current date and time. If it appears in a document, it is updated when the document is stored to disk. Consequently, when the document is opened the TimeStamp shows the date and time the document was last stored. The date and the time are formatted according to the specifications contained in the DateFormat and the TimeFormat keys in the [System] section of the registry.

Attributes

Color
Integer [15] specifying the color of the time stamp. The color can also be changed with the ColorPicker.

Font
String [Syntax10.Scn.Fnt] specifying the font of the text.

Tree

Classification	Model gadget
Generator	ListModels.NewTree
Alias	Tree

Function

A Tree stores strings in a tree fashion (lists of lists). The list items may be sorted or unsorted and the list may contain duplicates or not. This model gadget is typically used as a model by a ListGadget.

Attributes

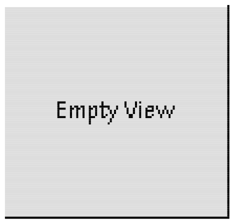
Sorted
Boolean [FALSE] – TRUE indicates that the list items are sorted on a level.

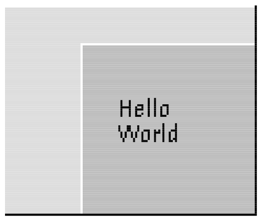
Unique
Boolean [FALSE] – TRUE indicates that no duplicate list items are allowed on a level.

Commands

See ListGadget.

View





Classification	Visual camera view gadget
Generator	Views.NewView
Alias	View

Function

A View acts as a camera displaying a single visual gadget or document. The visual gadget is linked to the View as a model gadget and it can not be un–linked with Columbus. A visual gadget may be dropped in an empty View, but once only. The viewed gadget can be removed by moving it to a different location out of the View or with a delete interclick. Each View has a certain viewpoint of the thing it displays (camera angle). This feature is useful when the viewed gadget has a rigid size exceeding the size of the View. The viewpoint can be changed by picking up the viewed gadget by pressing the MM key on its border and moving it to a new location. Should the border be invisible because the gadget is too big, an invisible area in the top left corner of the view can be used to grab and move the viewed gadget. The gadget's bounding box will appear as a rectangle showing the size and relative position of the gadget. Refer also to the ScrollView which is a View with optional horizontal and vertical Scrollbars.

Chapter Five

The Programmer's Guide

5.1 Introduction

Programming with the Oberon system involves extending the Oberon run–time environment. The reader familiar with programming will equate programming with writing a program. Once completed, the program is *run*. The program requests input data, does some calculation, and then outputs the result. This may happen many times until the program terminates and releases the resources it used.

In Oberon, there are no programs. Programs are relics of the days when computers had little memory and other resources. Programs wait in line until the user decides to execute them. After termination, the program removes itself from memory to make space for the next program. Communication between different programs running at different times takes place by storing a *message* in non–volatile storage (a file for example). Today however, computers have more memory and programs routinely run concurrently with each other. This is called *multi–tasking*. The computer resources are shared between all running programs. Unfortunately, communication between different programs has not progressed much further than in the earlier days of batch processing, which makes the cooperation and integration of different programs a difficult task.

The reason for this state of affairs can be traced back to the technology used for writing programs. If a program were *unsafe*, that is, doing the wrong thing, it could damage the integrity of the system, and thus negatively influence the other programs running on the computer. Most of today's programming languages allow the programmer to write (knowingly or unknowingly) programs that crash the system when run. Rather than solving the problem at its root (i.e. the bad programs), software systems started using the concept of memory protection. Using memory protection, a program is encapsulated in a "shield", preventing other programs from damaging it. Sadly enough, a memory protection shield also prevents easy communication with a program, thus hindering integration and cooperation between different programs.

In contrast, the Oberon system is an example of an open and extensible system. Open means that a high level of cooperation, integration and re–use of code between applications is practised. Extensible means that anybody can add a new part to the Oberon system. This new part might be using a part somebody else added, or might be used itself by another part added later. To achieve this flexibility, Oberon does away with programs completely. Instead, Oberon provides two concepts: *modules* and *type safety*.

Modules. An Oberon module contains (part of) the executable program code of an Oberon application. Modules are typically much smaller than programs, and an application often consists of more than one module. The modules of all activated Oberon applications share the memory of the Oberon system. There are no barriers between modules of different applications. Once loaded into memory, a module normally remains there until the computer is switched off. A module X may use (or re–use) the code contained in other modules A, B, C, etc. We say that module X *imports* modules A, B, C, and that module X is a client of modules A, B and C. Because a module is always visible to other modules (when in an import relationship), a large level of code or module re–use is possible. That means that applications can share useful modules between each other. For example, the Oberon system provides modules for managing text, bitmaps, data compression, network communication like file transfer and e–mail. These and other modules are often shared between different applications. The set of modules loaded into memory form the module hierarchy. The Oberon module hierarchy is a *directed acyclic graph* (DAG), in other words, no recursive imports are allowed.

But how do modules get into the computer memory in the first place? The Oberon system contains a module loader that can dynamically load and link a module into the running system. All imported modules are loaded automatically if they are not loaded into memory already. Instead of running a program, Oberon allows you to execute a command located in a module. A command is nothing more than a procedure located in a specific module, that is, a command `M.P`results in procedure `P` of module `M` being called. Thus, executing a command will result in a module being loaded into the system (from disk) by the module loader. The module typically remains in memory until the computer is shut down or until it is freed explicitly by the user.

Type safety. To prevent a module from corrupting the system, a danger in such an open arrangement, the Oberon programming language provides type safety. Type safety guarantees that a module cannot do bad things (by mistake or on purpose) to the system and to other modules. This is accomplished by a strong typing system in the Oberon language, and by checking the correct use of modules by the Oberon compiler. Each module provides additional functionality to the Oberon system, the use of which is determined by the module's *interface* or *definition*. The interface of a module tells us what components of a module are visible to the outside world (i.e. to the other modules in the system).

We say that these components are *exported* from the module. Type safety ensures that only these components and nothing more can be accessed by client modules. This allows the Oberon programmer to hide implementation details behind module interfaces and so ensure that private data structures can not be altered from outside the module. The value of type safety should not be underestimated. It protects the system and the programmer in a world of hundreds of cooperating but also at times menacing modules.

How do modules communicate with each other? As expected, one module can call the exported procedures of other modules directly. Another powerful technique is to share data structures between modules. In Oberon, all dynamically allocated memory is shared by modules in the so–called *heap*. Type safety ensures that only valid references to memory allocated on the heap are passed as pointers from one module to another. In fact, this is the basis of much of the run–time behavior of the Oberon system. We can imagine the heap to be a large database of collective data. Activating a command causes a module to transform data in the database, the result of which is again inserted into the database. This is a powerful way for applications to communicate directly without barriers, and even for applications to influence each other. As an example we can write a module containing a command that colors all the occurrences of the word "and" in a text document in red – simply by directly accessing the abstract text data type of a text document. In a similar way, we can add new functionality to all Oberon applications.

In conclusion, it should now be clear that Oberon has an advantage over other systems when it comes to integrating different applications with each other, protecting the system from the user (or the user against himself), supporting the ordered re–use of code, and constructing applications rapidly from prefabricated building blocks. The remainder of this chapter will enable you to do so yourself.

5.2 The Module Hierarchy

Learning to program the Oberon system involves studying the Oberon module hierarchy. From the modules themselves, you will learn the run–time structure of the system (i.e. the contents of the heap). There are lots of modules in the Oberon system and getting to know them all at once is not recommended. It is best to start with the easier modules and then work yourself up to the more difficult ones. Depending on your needs, you might not even need to learn how to build more complicated modules with the system – most of them are already available and can just be used "as is". If you fall into this category, the bulk of this chapter can be ignored, and you can concentrate on the programming of commands that manipulate existing components of the system. For this purpose, the examples provided later in this chapter are invaluable.

First, we discuss the larger structure of the system. The Oberon modules can be conceptually divided into the *inner core*, the *outer core*, the *object core*, the *text system*, the *gadget core*, the *gadget catalog* and the *document catalog*. The inner core is responsible for memory management, file management, and module loading. The outer core additionally provides device drivers for display, printer, keyboard, mouse, network etc. Other parts of the outer core provide viewer management and task and event dispatching. The distinction between inner core and outer core is rather artificial. The inner core is the minimum part of the system that needs to be present to boot Oberon. It is consequently always present and statically linked into a boot image. The outer core contains other important parts of the system.

The object core contains the Oberon persistent object manager, whilst the text system contains the font and text machinery. The gadget core provides the basis of the Gadgets component framework on top of which the remainder of the system is built. The gadgets catalog refers to the set of modules implementing the different gadgets. In the same manner, the document catalog refers to the modules of the different document classes. The remainder of the modules are typically *command modules*, that is, modules that have no clients and provide the commands the user sees. A command module mostly consists of parameterless procedures (commands). Each command module typically has a tool text, summarizing the usage of its commands, associated with it; for example, the `System`module has an associated text `System.Tool`. More graphically oriented modules have associated user interface panels.

The Figure 5.1 shows an extract of the Oberon module hierarchy. This *import diagram* shows modules as rectangular boxes with lines showing the import relationship. An import diagram is read from the bottom to the top, the upper modules being the clients of the modules below. We will be discussing these modules in the remainder of the section, but first we summarize the functionality of some important modules. Note that this list is not exhaustive. Depending on the underlying hardware and software platform, some additional modules might be included in the inner core. Often a set of modules to interface with the underlying operating system are included. Their source text modules are typically named according to the operating system in use (Windows, Macintosh, Linux, etc).

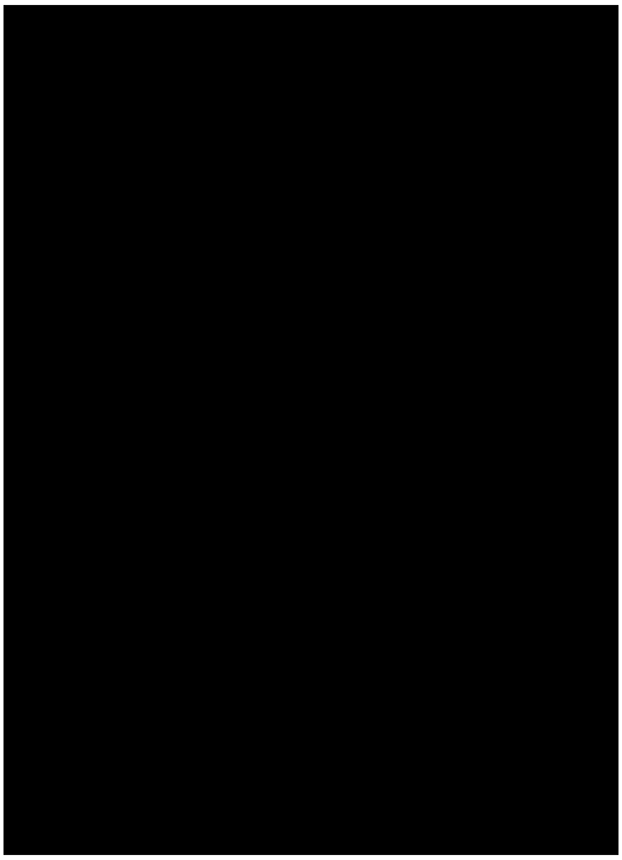


Figure 5.1 Extract of the Oberon Module Hierarchy

Furthermore, a single large module is sometimes split into two smaller modules for the sake of simpler management. In such a case, a "0" is appended to the name of the lower of the two modules in the import hierarchy. For example, `TextGadgets` is split in modules `TextGadgets0` and `TextGadgets`. To be more explicit, we might add that `TextGadgets0` implements the base functionality of an editor whereas `TextGadgets` is an extension of such an editor. In this case the two modules are tightly coupled and should logically be regarded as a single module.

The remainder of this chapter gives examples how to use the Oberon modules. These examples should be studied in conjunction with the module definitions.

Summary of the Oberon Module Hierarchy

Module Name	*Purpose*
Inner Core	
Kernel	Memory management and garbage collection
Disk	Disk driver (PC Native only)
FileDir	File directory support
Files	File handling
Modules	Oberon module loader and command execution
Outer Core	
Display	Display driver
Printer	Printer driver
Input	Timer, keyboard and mouse driver
V24	RS–232 serial communication port driver
Pictures	Bit mapped graphics handling
Reals	Floating point number support
Math	Mathematical functions for REALs
MathL	Mathematical functions for LONGREALs

Viewers	Implementation of Viewers of the display system
Oberon	Event handling and task dispatching
System	Command module with system related functions
Configuration	Configuration control at system startup

Object Core

Objects	Persistent object and library manager

Text System

Fonts	Font loader
Texts	Abstract data type implementation for texts
Out	Standard text output routines
In	Standard text input routines

Gadget Core

Display3	Clipped display routines
Printer3	Clipped printing routines
Effects	Special effects like rubber banding, cursors, menus.
Attributes	Attribute handling routines for gadgets
Links	Link handling routines for gadgets
Gadgets	Toolbox routines for implementing gadgets

Gadget Catalog

BasicGadgets	Implementation of gadgets like Boolean, Integer, Button, Slider, etc.
TextFields	Implementation of Captions and TextFields
Icons	Implementation of Icons and Iconizers
ListGadgets	Implementation of list gadgets
Panels	Implementation of Panels
TextGadgets	Implementation of a text editor
Rembrandt	The bitmap editor gadget
...	and several more modules implementing other gadgets

Document Catalog

Documents	Toolbox for implementing documents
Desktops	Manager of the overlapping display system
TextDocs	Implementation of text documents
PanelDocs	Implementation of panel documents
RembrandtDocs	Bit map editor toolkit
...	and so on for other document classes

5.3 Procedure Calls, Input and Output

Before examining some modules of the Oberon system, we have to understand how modules are typically used. Each module adds some functionality to the Oberon system and can be used either by client modules that import the module or by users who execute commands of that module.

Static Procedure Calls. Writing an Oberon module in the Oberon language involves importing other modules using the `IMPORT` statement (and thus making the module a client of the imported module). Importing makes the exported features of the imported module accessible to the programmer. This allows the programmer to access the exported global variables, types and procedures of the module. Typically, communication between the importing and imported module is realized by calling procedures of the imported module and passing parameters. This link is statically defined when the importing module is compiled and is correct according to the rules of type safety. For example, the called procedures may manipulate abstract data types like bitmaps, texts and the like.

From the user's standpoint, using a module involves calling exported commands of a module. A command is an exported procedure with no formal parameters. Suppose we have the following module, which writes a message to the system log text using module `Out`

```
MODULE Hello;

   IMPORT Out;

   PROCEDURE World*;
   BEGIN
      Out.String("Hello, world!"); Out.Ln()
   END World;

END Hello.
```

```
Hello.World
```

The asterisk marking ("*") of procedure `World` indicates that it is an exported procedure. In addition, the lack of formal parameters indicates that it is a command. This implies that the procedure `World` of module `Hello` can be called directly with an MM key click on the string `Hello.World` written somewhere on the Oberon display. This can be likened to you the user "importing" the module `Hello` and calling `World`

```
MODULE User;

   IMPORT Hello;

   PROCEDURE MyAction*;
   BEGIN Hello.World
   END MyAction;

END User.
```

Of course, the latter module does not exist, not even in the mind of the user, as users do not think about making modules when they execute commands! It does however illustrate quite well that executing a command is nothing more than calling a procedure. Modules that export lots of commands are called *command modules*, and are seldom imported by other modules.

Dynamic Procedure Calls. Suppose that we have the strings "Hello" and "World", and want to execute the corresponding command. This is the situation a typical text editor is faced with when a user clicks with the MM key on a string displayed on the screen, as the text editor knows only about texts (and thus strings). The solution is to request the module loader in module `Modules` to locate the correct procedure, which can then be called:

```
MODULE DynamicCall;

  IMPORT **Modules**;

  PROCEDURE DoIt*;
  VAR mod: Modules.Module; P: Modules.Command;
  BEGIN
    mod := Modules.ThisMod("Hello");    (* load module, if not already loaded. *)
    P := Modules.ThisCommand(mod, "World");    (* locate command. *)
    P    (* and execute command. *)
  END DoIt;

END DynamicCall.
```

A few things should be observed. First, we are using module `Modules` to do some work for us. This is done by statically calling procedures and using the types of that module. To find out exactly what the module `Modules` (and also `Out`, which we met earlier) can do for us, we look up its definition. There we see that type `Modules.Command` is a PROCEDURE, and thus `P` can be called directly. In the remainder of this chapter, we will assume that you look up the definition of modules we use in our examples without prompting you as we are doing now. If you are working in front of the computer, the definition is as far away as activating Watson (Chapter 2).

A second observation is that module `DynamicCall` does not import the module `Hello`; the command has been activated dynamically. As the called module lies typically higher up in the module hierarchy than the calling module, an *up–call* is involved. There are other examples of "calling up" in the module hierarchy.

A few more comments on the module Hello. The system log is a model object of type `Texts.Text`. The content of the log text is automatically shown in the log viewer which is opened when the system is started. Module `Out` offers a comprehensive set of procedures for writing (in reality "appending") text stretches to the log, without a need to know how text is maintained and managed by the system. Writing a string (`Out.String`) and a carriage return (`Out.Ln`) are only two examples. Putting the other procedures to work is straightforward.

Parameter Passing. Typically, when statically calling an imported procedure, parameters are passed. But how are text parameters passed to commands? We know already that most Oberon commands have parameters that are written following the command in the form of a text stream. By convention, the text parameters are passed from the caller (or client) to the callee by temporarily assigning them to a global variable (which will be discuss later). Thus the caller (the text editor, for example), puts a pointer to the text parameter in this global variable and uses module `Modules` to call the command. The called command picks up the parameters from the global variable and processes them; simple but effective.

To get beginners started as soon as possible with Oberon, the scanning of command parameters is however hidden or covered by a module called `In`. That is, the whole process of parameter passing that happens behind the scenes, is hidden from us (until the next section, at least). This is illustrated by `Example1` which sums up a series of numbers:

```
MODULE Example1;
  IMPORT **In, Out**;

  PROCEDURE Sum*;
  VAR x, total: REAL;

  BEGIN
    total := 0;
    **In.Open;**    (* Initialize parameter scanning. *)
    **In.Real(x);**    (* Scan a REAL parameter into x. *)
    WHILE In.Done DO    (* Was a REAL really encountered? *)
      total := total + x;
      **In.Real(x)**    (* Try scanning the next number. *)
```

```
        END;
        Out.String("The total is "); Out.Real(total, 10);
        Out.Ln
    END Sum;

END Example1.
```

This example expects the parameters to be summed to be written as follows:

```
Example1.Sum 4 8 1AFH 4.0E2~
```

Note the "~' which ends the parameter list. Not being a numeric value, `In.Done` is false after scanning "~", and thus the summing loop is terminated. (Did you already take a look at the definition of `In`?)

`In.Real` delivers the next REAL value, resulting from the conversion of a textual stream of digits. All other procedures in module `In` have similar properties.

5.4 Texts

As most readers probably noticed while studying module `In`, this technique of scanning parameters is not very robust since the programmer has to assume that the correct parameters are passed to the command. A more reliable way of scanning parameters is implemented behind the scenes of module `In`, the investigation of which also reveals the "parameter scratchpad" introduced before. Also, module `Out` hides the complexity of writing text to the log, which is implemented behind the scenes with lower level modules.

To understand what the modules `In` and `Out` do, we need to introduce to two further modules, namely `Texts` and `Oberon`. Module `Texts` defines an abstract data type that manages text streams. Module `Oberon` has a few system–wide tasks, one of which involves managing the previously mentioned global variable which is called `Oberon.Par`. `Oberon.Par` is a RECORD containing a reference to the parameter text (`Oberon.Par.text`), the starting position in that text (`Oberon.Par.pos`), and diverse other fields describing the environment in which the command was executed. The parameter text starts at the position `Oberon.Par.pos` in this text, with the first character in a text having position zero. Further study of the component `Scanner` in module `Texts` shows that it is useful for scanning parameters. It parses the associated text for tokens. While parsing, white spaces (i.e. blanks, TAB characters and RETURN characters) are skipped. The same example can now be rewritten without the use of module `In` by replacing it with a *Scanner*:

```
MODULE Example2;
    IMPORT Oberon, Out, Texts;

    PROCEDURE Sum*;
    VAR total: REAL;
            S: Texts.Scanner;
    BEGIN
        total := 0;

        (* Open a Scanner at the starting position of the parameter. *)
        Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos);
        Texts.Scan(S);     (* Scan first parameter. *)
        WHILE (S.class = Texts.Int) OR (S.class = Texts.Real) DO
            IF S.class = Texts.Int THEN
                total := total + S.i
            ELSE
                total := total + S.x
            END;
            Texts.Scan(S)     (* Scan next parameter. *)
        END;

        Out.String("The total is "); Out.Real(total, 10); Out.Ln
    END Sum;

END Example2.
```

`Example2.Sum 4 81AFH 4.0E2~`

The field `class` of the scanner tells us what was scanned; that is, it identifies the type of the scanned symbol. Depending on its type, the scanned value must be retrieved from a different scanner field.

In fact, module `Texts` provides much more than Scanners. In `Example2`, we still used module `Out` for writing output to the log, but `Out` is easily replaced with a *Writer*. Writers are used for *efficiently* creating large amounts of text. Module `Out` uses a Writer "behind the scenes" to write its output to the system log. A typical example of using a Writer is the following:

```
MODULE Example3;
   IMPORT Oberon, Texts;
   VAR W: Texts.Writer;

   PROCEDURE Time*;
      VAR time, date: LONGINT;
   BEGIN
      Oberon.GetClock(time, date);
      Texts.WriteString(W, "The date and time are ");
      Texts.WriteDate(W, time, date);
      Texts.WriteLn(W);
      Texts.Append(Oberon.Log, W.buf)  (* Append W's text buffer to the log *)
   END Time;

BEGIN Texts.OpenWriter(W)   (* Initialize Writer at module load time. *)
END Example3.
```

`Example3.Time`

Usually, only one shared writer is declared per module. The text written to a Writer is temporarily kept in the writer's buffer `W.buf`. Then, the writer's buffer is appended (`Texts.Append`) to a text and subsequently cleared in the operation (a side effect). Further writing to the Writer starts filling up the buffer again until it is cleared again. There is practically no limit to the size of a buffer or a text. Finally, notice how the system log is represented by a global variable called `Oberon.Log` of type `Texts.Text`.

The Text selection. The *text selection* is an integral part of the Oberon system. The current text selection is the text stretch last selected by the user. More than one selection may be visible, but only the *most recently* selected stretch (according to system time) is important to us. The procedure `Oberon.GetSelection` returns the required information on the latest selection: the text itself, the starting and the ending position of the selection, and the time (according to `Oberon.Time`) when the selection was made. To obtain the latest selection, `Oberon.GetSelection` broadcasts a special message to all the text editors active in the system to return their selection. At the moment though, the exact behavior can be ignored.

What can we do with the selection? We already know about the `Texts.Scanner`, so we can use it to scan for tokens in the selection. We can also use a `Texts.Reader` to read through the selection character by character. Readers are new in our discussion, so we present an example of a Reader calculating some statistics about the selection:

```
MODULE Example4;
IMPORT Oberon, Texts;
VAR W: Texts.Writer;

PROCEDURE Count*;
VAR T: Texts.Text;
        beg, end, time: LONGINT;
        letters, digits, others: INTEGER;
        ch: CHAR;
        R: Texts.Reader;
BEGIN
    Oberon.GetSelection(T, beg, end, time);
    IF time >= 0 THEN    (* is a selection present? *)
        letters := 0; digits := 0; others := 0;
        Texts.OpenReader(R, T, beg);    (* start reading at position beg in text T. *)
        Texts.Read(R, ch);
        WHILE beg < end DO
            CASE ch OF
                "A" .. "Z", "a" .. "z": INC(letters);
                | "0" .. "9": INC(digits)
            ELSE INC(others)
            END;
            INC(beg);
            Texts.Read(R, ch)
        END;

        Texts.WriteString(W, "# digits = "); Texts.WriteInt(W, digits, 10);
Texts.WriteLn(W);
        Texts.WriteString(W, "# letters = "); Texts.WriteInt(W, letters, 10);
Texts.WriteLn(W);
        Texts.WriteString(W, "# others = "); Texts.WriteInt(W, others, 10);
Texts.WriteLn(W);
        Texts.Append(Oberon.Log, W.buf)
    END
END Count;

BEGIN    Texts.OpenWriter(W)
END Example4.
```

Since the selection is always a valid text stretch (starting at the character position `beg` and ending just before character position `end`), we do not have to be concerned about reading past the end of the text. If `Oberon.GetSelection` fails to find a selected text stretch, the `time` field contains a negative value. Each text `T` has `T.len` characters, and when the reader `R` reads the `T.len`th character, the boolean flag `R.eot` indicates that the end of the text has been reached. The condition `beg < end` can be replaced by `~R.eot` to read past the end of the selection right up to the end of the text containing the selection.

Opening a text viewer. How do texts get on the display anyway? Usually, an end–user opens a text file by executing a `Desktops.OpenDoc` command. We can open a text file also under program control with the following module:

```
MODULE Example5;
IMPORT Oberon, Texts;

PROCEDURE Open*;
VAR
   S: Texts.Scanner;
   T: Texts.Text;
   beg, end, time: LONGINT;

BEGIN
   Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos);
   Texts.Scan(S);
   IF (S.class = Texts.Char) & (S.c = "↑") THEN
      Oberon.GetSelection(T, beg, end, time);
      IF time >= 0 THEN    (* is a selection present? *)
         Texts.OpenScanner(S, T, beg);
         Texts.Scan(S)
      END
   END;
   IF S.class = Texts.Name THEN    (* was a valid file name scanned? *)
      NEW(T); Texts.Open(T, S.s);
      Oberon.OpenText(S.s, T, 250, 200)
   END
END Open;

END Example5.
```

This module is invoked with

```
Example5.Open Example4.Mod~
```

or

```
Example5.Open ^ [the name of a text file is selected]
```

to open the text file whose name is passed as a parameter (and scanned with the scanner above). During parameter scanning, we explicitly test for the case when a "↑" is used as a parameter. In this case, we have to re–open the scanner on the current text selection, as Oberon command conventions indicate. The statement `NEW(T)` creates an empty text descriptor on the heap, which is subsequently filled with text by `Texts.Open(T, S.s)`. `S.s` contains the name of the file from where the text is retrieved. Should we require an empty text instead (i.e. one containing no characters), we should replace the latter with `Texts.Open(T, "")`. The procedure `Oberon.OpenText` presents the text `T` in a text document ready for editing. The first argument to `Oberon.OpenText` is the title of the document, followed by the text to be displayed and the width and height in pixels of the viewer. The latter is regarded as a hint to the system where to open the text document viewer. For example, should the width be wider than the system track, the system will open the viewer in the user track. Hint: the width ratio of the user and the system tracks is 5 to 3.

Multiple calls to `Oberon.OpenText` with the same text `T` as parameter will create a fresh viewer in each case, and that each will be displaying the same text. Thus, changes made in one text document will be reflected in the others automatically. We say that the viewers are the view components, and the text `T` is the model component. On the other hand, multiple executions of `Example5.Open` create fresh viewers displaying separate texts.

Retrieving the marked text. Once a text is displayed by a text document, how can we get back to the text itself again? By convention, Oberon requires the user to mark the intended text with the star marker ("*") which is set by hitting the F1 key on the keyboard when the mouse focus is located inside the document's viewer. A helper procedure `Oberon.MarkedText` returns the marked text.

```
MODULE Example6;
IMPORT Fonts, Oberon, Texts;

PROCEDURE Highlight*;
VAR T: Texts.Text; fnt: Fonts.Font;
BEGIN
   T := Oberon.MarkedText();    (* get marked text *)
   IF T # NIL THEN    (* was a text found? *)
      fnt := Fonts.This("Syntax10m.Scn.Fnt");
      Texts.ChangeLooks(T, 0, T.len, {0}, fnt, 0, 0)    (* change font of entire text *)
   END
END Highlight;

END Example6.
```

```
Example6.Highlight
```

In this example we introduce fonts. Font management is done under control of the `Fonts` module. In the present case, `Fonts.This` loads a 10 point Syntax screen font, which is then applied to the whole text `T` (characters `0` to `T.len`) with `Texts.ChangeLooks`. `ChangeLooks` can also be used to change the color and the vertical offset of a text stretch. More details can be found in the definition of `Texts`

Before continuing our examples for manipulating texts, we need to cover some necessary theory first. The following two sections introduce the display space, messages and broadcasting.

5.5 The Display Space

In earlier chapters, the *display space* was introduced. In essence, the display space is a data structure containing *visual* and *non–visual* objects. The visual objects, called *frames*, are the gadgets such as documents, Buttons, Scrollbars, etc. you see on the display. The non–visual objects are the model gadgets linked to these gadgets. We should emphasize here the abstract nature of the display space we are referring to. The display space should not be confused with the geometrical space of the display screen. To understand the display space, we have to discuss the nature of the single elements of the display space, and the nature of the connections between them.

The type hierarchy. Before going into detail about the structure of the display space, we first investigate the type hierarchy of the visual and non–visual objects. All visual objects are extensions of the type `Display.Frame`. In turn, each frame is an extension of a more basic type called `Objects.Object`. The non–visual objects are also extensions of `Objects.Object`, but not of `Display.Frame`

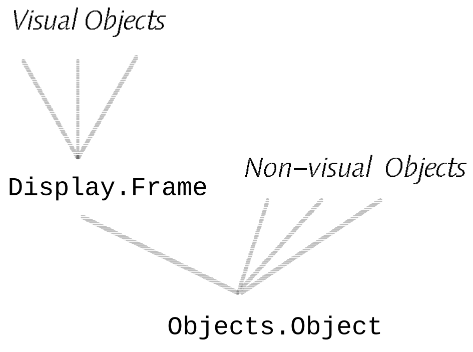


Figure 5.2 The Object type hierarchy

The lines of Figure 5.2 show the type extension relationship between objects. To simplify the programming of gadgets, two further base types form the basis of the visual and model gadgets. These two types, `Gadgets.Frame` and

Gadgets.Object are extensions of Display.Frame and Objects.Object respectively. Thus when programming gadgets, we are faced with the type hierarchy depicted in Figure 5.3.

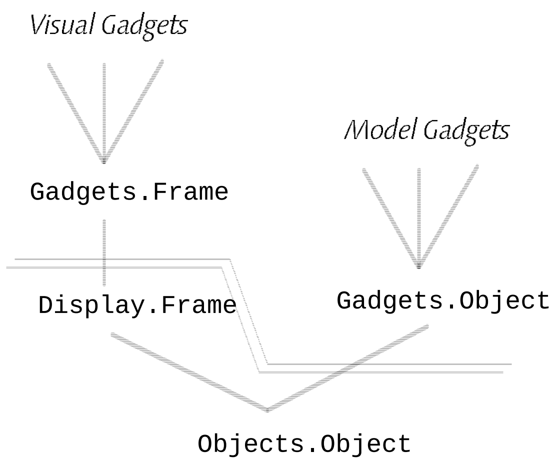


Figure 5.3   Base system / Gadgets system relationship

The double horizontal line of Figure 5.3 signifies a split between the base system and the Gadgets system. There is a good reason for emphasizing this split. The base system provides the mininum for implementing a framework of components. The Gadgets system "fills in" part of the functionality in this framework, that is, it adds rules and user interface conventions. Theoretically, we can imagine yet another type of user interface system built on the same base but implementing a different type of user interface framework. In fact, the textual user interface of the Oberon system is such a framework which existed before the development of the Gadgets system. This also illustrates that such a dichotomy turns out useful for guaranteeing compatibility with older applications, as both gadgets–enabled and older applications are different aspects of the same underlying object model.

Before continuing with the display space structure, it is informative to show the type definitions of Objects.Object and Display.Frame, although the meaning of several of the RECORD fields is not obvious yet, and thus partially hidden:

```
Handler = PROCEDURE (obj: Object; VAR M: ObjMsg);

Object = POINTER TO ObjDesc;
ObjDesc = RECORD
    ... some more fields ...
    handle: Handler    (* Message handler. *)
END;

Frame = POINTER TO FrameDesc;
FrameDesc = RECORD (Objects.ObjDesc )
    next, dsc: Frame;
    X, Y, W, H: INTEGER
END;
```

The definition of Object reveals that *each* object has a *message handler*, which is responsible for interpreting the messages that are sent to it. Furthermore, we see that frames are type extensions of objects with the additional fields next, dsc, X, Y, W and H, which are described in the next paragraph.

The Display space structure. Frames have a location and a size: the record fields F.X and F.Y contain the position of the frame F, and the record fields F.W and F.H its width and height. When a frame is located in the display space, it is always nested in another frame. The situation can be clarified in the following way, but for the sake of simplicity we shall confine the description to the tiled display space model used by the textual interface. The display itself is a large (imaginary) frame that covers the whole display surface. It has the location (0,

0) and size (`Display.Width` and `Display.Height`). (By the way, the origin of the Oberon display is in the bottom left corner, in contrast to many other systems.) This "root" frame is divided into two tracks, the user and the system tracks. Each track is a frame nested or contained in the root frame. The tracks are further divided into viewers, which are frames too (nested in their respective tracks). Each viewer is divided into two further frames, the menu bar frame and the main frame. The main frame might for instance contain a text editor frame. Inside this text editor frame we have the text itself, and possibly further frames signifying the visual objects that "float" in the text. These frames might contain further frames, and so on to any nesting depth.

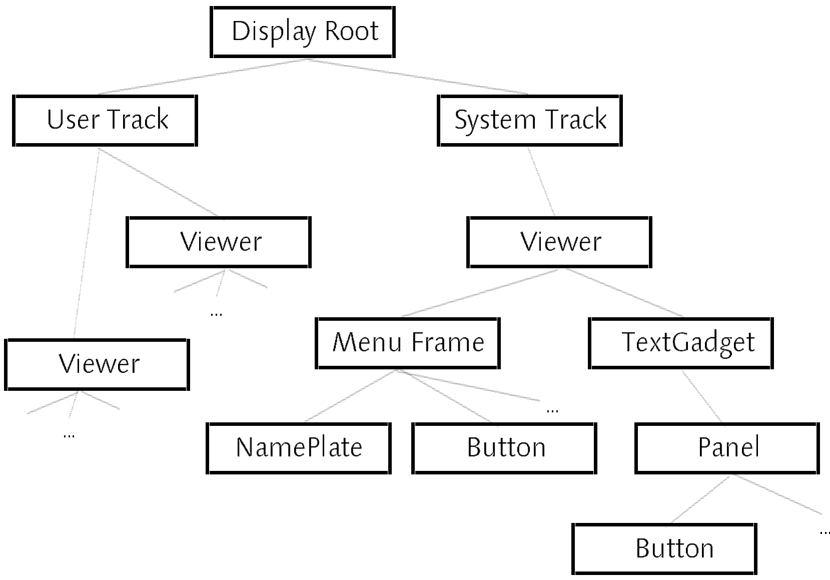


Figure 5.4 Example structure of the display space

The nested nature of the display space is extremely important in the system and should be well understood. Figure 5.4 gives a diagrammatic view of the situation. Frames containing other frames as children are called *container* gadgets. Frames not containing any further frames are *elementary* gadgets. Each parent frame is completely responsible for its direct descendant frames and thus indirectly responsible for its indirect descendants. *By convention*, the "nested" structure of the display space is defined by the two fields `dsc` and `next` of `Display.Frame`. The `dsc` field of a container points to the *first* child of that container, and is the start of a list of direct descendants linked by the `next` field. The following code fragment shows how the children of a container `F` are traversed:

```
PROCEDURE Traverse (F: Display.Frame);
VAR f: Display.Frame;
BEGIN
   f := F.dsc;
   WHILE f # NIL DO
      ... do something with f ...
      f := f.next
   END
END Traverse;
```

Remark. The property of *parental control* dictates which of these fields a frame may modify when receiving a message. A frame `F` is allowed to change its own `F.dsc` record field and the `next` fields of its direct descendants. But a frame should *never* change its own `next` field. This property also requires that you should *never* interfere with the `dsc`-`next` list of a container; although visible to the outside world, it is under complete control of the container itself. Later, we explain how external influences can be applied to containers.

A further complication of the display space is the location of model gadgets.

We already know that frames can possibly have a model linked to them. The visual gadgets, for example, reserve a field called obj to refer to a model gadget. As many frames can refer to the same model object, the use of these fields "close" the structure of the display space at the bottom (Figure 5.5). The general structure of the display space is that of a directed acyclic graph (DAG). In this Figure we have a Panel floating in two TextGadgets: the Panel containing a Button and a CheckBox both linked to a Boolean model.

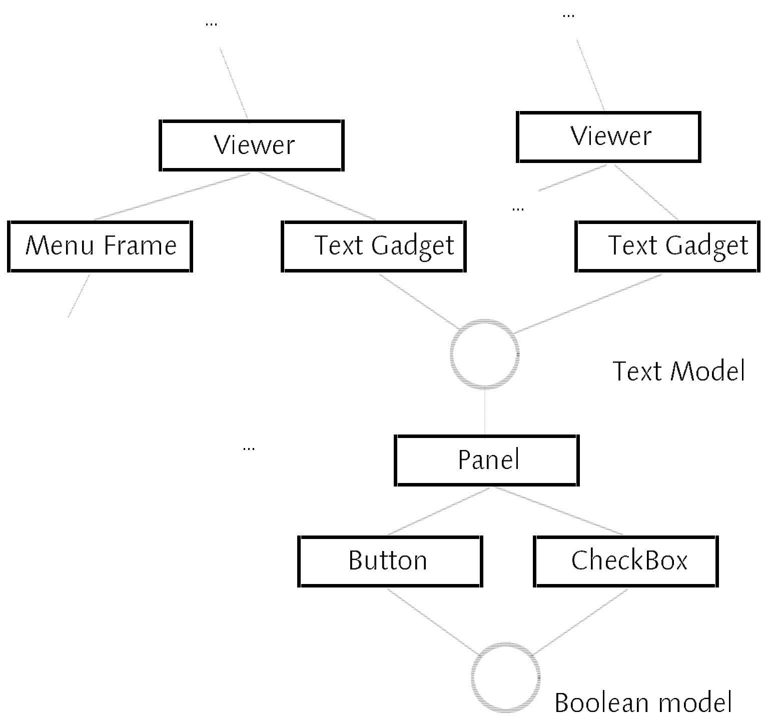


Figure 5.5 Example structure of the display space with model objects

In our previous explanation, we mentioned that the location of a frame F in the display space is determined by the record fields F.X and F.Y. We have to specify exactly what this means in the context of the display space. By convention, F.X and F.Y are the *relative* coordinates of a frame in its parent (i.e. container) frame. As the Figure 5.6 illustrates, the X and Y fields of a frame measure the offset of its bottom–left corner relative to the top–left corner of its parent frame. Since the Oberon display origin is in the left bottom corner of the display, the Y offset of a gadget is always a negative number. This setup provides the best efficiency when calculating the coordinates of a gadget.

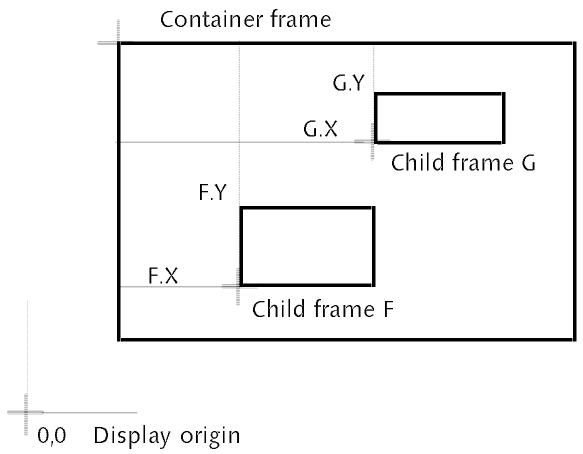


Figure 5.6 Frame coordinate system

We should mention that a parent frame clips away those parts of its children which lie outside the rectangle of the parent. The relative coordinates of a frame in a container allow us to move that container and all its descendants directly by modifying only the relative coordinates of the parent in its own container.

5.6 Messages and Broadcasting

Now that the structure of the display space has been presented in detail, we can discuss how objects (everything extended from `Objects.Object`), communicate with each other in the display space. Recall that every object in the Oberon system has a *message handler*, responsible for "handling" the messages sent to that object. There exists a large number of message types. For example, there are messages to request an object to make a copy of itself, to store itself to a file, to display or print itself, and so on. However, not every message is applicable to all objects. For example, model gadgets, that is, non–visual objects, do not understand a "display yourself" message.

Messages are divided into classes. For the moment, we mention two important ones: *object messages* and *frame messages*. Object messages are those messages that *all* objects must respond to. This class consists of a very small collection of messages like "copy yourself", "store yourself" and so on. Frame messages are those messages that frames respond to. That class includes ones like "display yourself, "print yourself" and "move your position".

Objects might not understand some of the messages sent to them. Why would we send a message to an object if it does not understand it anyway? The answer lies in the display space. Let us take the example of multiple text documents showing the same text (evoked in Example5). We already know that when we change the contents of a text (using module `Texts`), each text document displaying it is magically updated. Behind the scenes this is solved by a special message protocol between the text document (a view) and the text object (a model). This protocol is "special" in that only text documents and text objects are aware of it; they are "insiders", so to say.

Model–view consistency. Let us trace what happens when a character is inserted into a text or more correctly from a programming standpoint, when a text buffer is inserted into a text. The user of module `Texts` does this with a call to `Texts.Insert`. Behind the scenes, `Texts.Insert` has to inform all the text documents displaying the text that a change has taken place. Now, by convention, texts do not know what text document(s) they are attached to. The only possibility left open to the text is to *broadcast* a "text `T` has changed" message to all objects in the display space. Since text documents are listening to messages of the type "text `T` has changed" (where `T` is the text they are linked to), they can update themselves when this broadcast message is received. The broadcast functions in the following manner. First, the message is sent to the display root. As the root does not understand the message (because it knows nothing about texts), the only logical thing it can do is to *forward* the message to the tracks it contains, in the hope that these can have something to do with the message. However, the tracks are just as clueless, and are forced to forward the message to the viewers they contain. The same thing happens again, so each viewer forwards the message to its (two) sub–frames. The message thus travels through the entire display space in a depth–first way. Luckily, the text documents located in the display space understand the message, and can update the display with the character typed (which is specified in the message by indicating the part of the text that has changed).

This is the general scheme Oberon uses to inform views that a model has changed. If no view is interested in the model that has changed, the message travels through the whole display space without effect.

Types of broadcast. In reality, there are two types of broadcasts: *true broadcast* and *directed broadcast*. True broadcasts reach all frames in the display space. Directed broadcasts are addressed to a certain destination frame in the display space. This might sound a little paradoxical. Why do we need to broadcast a message into the display space if we know the frame that should receive it? Why not simply send the message directly to the intended frame (without going through the display root, tracks, viewers etc)? Although several reasons exist (one of them related to the Oberon display model), only one reason will be mentioned here. As messages travel in the display space from the root frame to track, to viewer, and so on, at any point a container frame can make a decision if it wants to handle, ignore, change or forward a message. Thus a frame can influence the messages its descendants "see". This is again the all important property of parental control. Should a parent frame not control the message a child frame sees, the child might "misbehave". Parental control might force a child frame to move itself only in a restricted way in a container; or prevent a child frame from deleting itself.

Terminating a broadcast. Under certain circumstances, it is necessary to terminate a broadcast early. This is typically the case when it is known that no further frame could have an interest in the message. Thus, Oberon provides a way to *invalidate* a message. An invalidated message is not forwarded by containers, thus terminating the broadcast abruptly.

5.7 A Message Protocol

After this excursion into the display space, messages and broadcast, we continue with our examples for manipulating texts. These examples are generally applicable to other objects too. To communicate with a text editor frame located somewhere in the display space, we have to send messages to it. These messages are related to the visual aspects of the text, such as the caret or the selection. In general, we can say that each type of visual gadget responds to messages. Often, a frame responds not only to a single message protocol but also to a set of *message protocols*. For example, the text editor–like frames (in module `TextFrames`) respond to the *text message protocol*, in addition to the protocol for objects (object messages) and for frames (frame messages). We need to know only that text editor frames respond to the text message protocol, but we do not have to import the text editor modules. This allows somebody to create a new text editor frame at a later point in time *without invalidating existing modules*. In short, the examples below function with all text editors, future and past.

Messages as first class citizens. In contrast to nearly all other systems, messages in Oberon usually are statically allocated RECORDs that is, RECORDs that are temporarily allocated on the activation stack. A message is *sent* to an object by passing it as a VAR parameter to the message handler of that object. A speciality of Oberon is that type extension is applied also to message RECORDs. The base message type is defined in module `Objects`

```
ObjMsg = RECORD
   stamp: LONGINT;
   dlink: Object
END;
```

`stamp` and `dlink` are administrative fields which we shall ignore for the moment. Frame messages are needed to communicate with (text editor) frames and additional information must be supplied to these frames. The frame message type extension is defined in module `Display`

```
FrameMsg = RECORD ( Objects.ObjMsg )
   F: Frame;
   x, y: INTEGER;
   res: INTEGER
END;
```

Except for the `F` field of the `FrameMsg`, let us further ignore the other message fields. The F field of a message specifies the target frame of a directed broadcast. When it is set to NIL, no target is specified and a true broadcast is involved.

Finally, the messages of the text frame protocol are, in turn, extensions of FrameMsg. Let us take a look at one of them, the CaretMsg, which is defined in module Oberon and which is used to manipulate the text caret remotely:

```
CaretMsg = RECORD ( Display.FrameMsg )
    id: INTEGER;    (* get, set, reset *)
    car: Display.Frame;    (* Destination frame, returned text editor frame. *)
    text: Texts.Text;    (* Text represented by car. *)
    pos: LONGINT    (* Caret position. *)
END;
```

Abstractly, we see from the discussion above that Oberon uses a type extension hierarchy for messages too. This allows us to extend the message protocols of objects without changing or recompiling the whole system. Before proceeding with the text message protocol, we illustrate the message hierarchy in Figure 5.7.

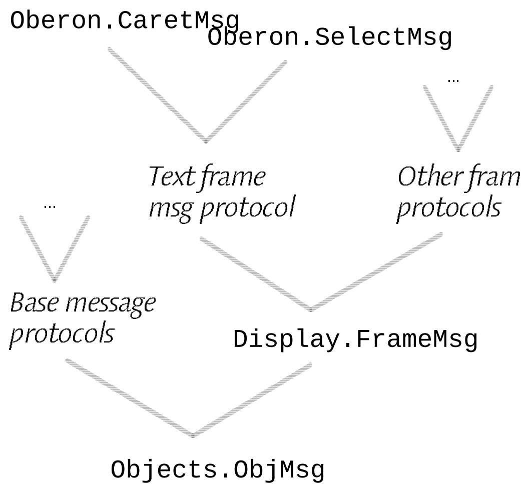


Figure 5.7 Extract of the Message type hierarchy

Reminder. The introduction of the message type hierarchy illustrates the importance of *hierarchies* in the Oberon system. At the start of this chapter, we investigated the module hierarchy, which forms the basis of code re–use in the Oberon system. In the introduction to the display space, we got to know the object type extension hierarchy (for reusing and extending object classes), and the display hierarchy of container/child and model/view relationships between objects (the run–time system organization). Now, we have just become acquainted with the message hierarchy that defines the protocols used to communicate with objects. Later, we will describe another hierarchy that organizes "templates" of pre–configured objects.

This hierarchical system organization is important since it forms the basis of an extensible system. Let us summarize *what* we would like to extend and *how* we do it in Oberon. To extend the code of the system, we write a new module that uses (i.e. imports) existing modules. To extend an object with new functionality, we make a sub–type (or class) of that object. To extend the run–time system with functionality, we insert an object in the display space. To extend the way we communicate with objects, we add new message protocols to the message type hierarchy.

To communicate with a text editor and thus manipulate its caret, we need to define a CaretMsg, fill out the RECORD fields of the CaretMsg message correctly, and then broadcast the message into the display space. The general scheme of declaring, filling out and broadcasting a message is illustrated by the following code fragment:

```
PROCEDURE DoIt;
VAR M: Oberon.CaretMsg;    (* define message *)
BEGIN
    (* fill out the message fields *)
    M.F := ... ;
    M.id := ... ;

    (* Broadcast the message *)
    Display.Broadcast(M)

    (* ... process return values ... *)
END DoIt;
```

The `Example7` in the next section will offer a practical application of this scheme.

To use messages in the Oberon system a programmer requires a knowledge of how the message fields are interpreted by their receivers. The fields of the object and display messages have a fixed meaning, and their semantics will be explained together with the following example. We now review the meaning of the individual fields of the `CaretMsg` (the first six fields belong to the base message types `Objects.ObjMsg` and `Display.FrameMsg`).

`stamp`	A time stamp set by the `Display.Broadcast` procedure to indicate the time the message was broadcast. Time, in this sense, is simply a counter incremented each time a message is broadcast.
`dlink`	A pointer to the object that forwarded the message to the sender.
`F`	The destination frame in the display space for which this message is intended. A NIL value indicates that the message is addressed to all frames in the display space.
`x`,`y`	The absolute screen coordinates of the container frame of the receiver.
`res`	A number that indicates if the message is valid or not. Invalid messages are not forwarded further in the display space (terminated broadcast). An invalid message has a `res` value of zero or more. `Display.Broadcast` automatically sets `res` to a negative value before the message is broadcast.
`id`	A message selector specifying which sub-operation the destination frame must complete. In this case we can either set the caret, retrieve the current caret position, or reset, i.e. remove, the caret.
`car`	When `id` is set to `get`, the `car` field returns the frame that contains the caret (after the message broadcast).
`text`	Either the text model of the frame that contains the caret when `id`=`get` (`text` and `car` are consistent), or the text of the destination frame otherwise.
`pos`	The caret position in the text of the destination frame or of the returned frame.

After this introduction, a short discussion of the message fields is in order.

First, as user/sender of messages, fields like `stamp`, `dlink`, `x` and `y` are of secondary importance as they are updated automatically while the message travels through the display space. These fields need not be filled in before broadcasting a message. The only fields that are of interest to us at the moment are `F` (the destination frame) and the four fields of the `CaretMsg` itself.

Second, the semantics of fields change depending on the message selector `id`. In some cases, the fields of the `CaretMsg` are *out* parameters, and sometimes they are *in* parameters. *Out* parameters deliver results from the destination frame(s) to the message broadcaster or sender. The parameter values are available in the message RECORD itself *after* the message broadcast. *In* parameters are used to pass parameters from the sender to the destination frame.

`id`	**Meaning of the fields**
`get`	Message broadcast to all frames, thus *F := NIL*.

	`car`, out parameter with the frame that has the caret (if any).
	`text`, out parameter with the text model of `car` that has the caret.
	`pos`, out parameter with the position of the caret in `text`.
`set`	Message broadcast to a specific frame, thus *F := someFrame*.
	`car`, in parameter with the frame *F* where to set the caret.
	`text`, in parameter with the text model of `car`.
	`pos`, in parameter with the position of the caret in `text`.
`reset`	Message broadcast to a specific frame, thus *F := someFrame*.
	`car`, in parameter with the frame *F* where to reset the caret.
	`text`, in parameter with the text model of `car`.
	`pos`, ignored.

The description also shows that a frame might be displaying more than one text at the same time, and we should specify which of these texts are meant. This generality is however seldom used in the Oberon system (but might be in the future). Consequently, in the set and reset cases, the `text` field is mostly redundant but it is checked for correctness *anyway*.

Finally, observe that the meaning of some fields of the base message types have not been explained completely yet (`dlink` and `stamp`, in particular); this explanation will be deferred till later.

The SelectMsg. This message of the text frame protocol controls the text selection. Our knowledge of messages in general can now be easily applied to the SelectMsg, which is also defined in module `Oberon`

```
SelectMsg = RECORD ( Display.FrameMsg )
    id: INTEGER;    (* get, set, reset *)
    time: LONGINT;    (* Time of the selection. *)
    sel: Display.Frame;    (* Destination frame, returned frame. *)
    text: Texts.Text;    (* Text represented by sel. *)
    beg, end: LONGINT    (* Text stretch of the selection. *)
END;
```

Most of the fields should look familiar from comparison with the definition of the `CaretMsg`. Instead of `car`, `sel` indicates the frame containing the selection. The `time` field specifies the (true) time of the selection (not to be confused with the message time stamp), while `beg` and `end` specify the extent of the text selection. Otherwise, most of the principles of the `CaretMsg` apply to the `SelectMsg`. In particular, the `SelectMsg` is also always broadcast.

The ConsumeMsg. This text frame protocol message controls the insertion of a text stretch at the caret and is also defined in module `Oberon`

```
ConsumeMsg = RECORD ( Display.FrameMsg )
    text: Texts.Text;    (* Text to be inserted. *)
    beg, end: LONGINT    (* Text stretch to be inserted. *)
END;
```

The RecallMsg. This text frame protocol message controls the insertion of a last deleted text stretch at the caret and is also defined in module `Oberon`

```
RecallMsg = RECORD ( Display.FrameMsg )
END;
```

5.8 Text Protocol Examples

Font Search Tool. Our next objective is to write a command which searches a text for a specified font, starting at the current position of the caret, and to re–position the caret after the first character in that font. The search starts at the caret, if set. The font name is the sole command parameter. To aid the understanding, we should mention that the `lib` field of a Reader `R` is a pointer to the font of the last character read.

```
MODULE Example7;

IMPORT Display, Oberon, Out, Texts;

PROCEDURE SearchFont*;
VAR ch: CHAR; C: Oberon.CaretMsg;
    R: Texts.Reader; S: Texts.Scanner;
BEGIN
    Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos);
    Texts.Scan(S);
    IF S.class = Texts.Name THEN
        C.id := Oberon.get; C.F := NIL; C.car := NIL; C.text := NIL;
        Display.Broadcast(C);     (* get caret position *)
        IF C.text # NIL THEN
            Texts.OpenReader(R, C.text, C.pos);
            Texts.Read(R, ch);
            WHILE ~R.eot & (R.lib.name # S.s) DO Texts.Read(R, ch) END;
            IF ~R.eot THEN
                C.id := Oberon.set; C.F := C.car; C.pos := Texts.Pos(R);
                Display.Broadcast(C)
            ELSE
                Out.String("Font not found"); Out.Ln
            END
        ELSE
            Out.String("No caret set"); Out.Ln
        END
    END
END SearchFont;

END Example7.
```

The module can be activated with:

```
Example7.SearchFont Syntax10.Scn.Fnt
```

Finding at which position in a text the caret is set, is done by broadcasting a `CaretMsg`, with the `car` field set to `NIL`, to all the documents in the display space. If a caret is found, we read through the text with a Reader until the font is found. The font descriptor has a name field (`R.lib.name`) containing the font name. If the WHILE loop terminates before reaching the end of the text, we know that we were successful, and we can set the caret again. Setting the caret outside the visible part of the text on the display will scroll the text automatically, so that the caret position becomes visible. After broadcasting the first caret message, the fields `car` and `text` are already initialized for the second caret message broadcast.

Insert text Tool. This example makes use of the messages described earlier to insert a text stretch at the caret. The first command copies the selection to the caret via a buffer and terminates by selecting the copied text. The second one copies the selection to the caret using a `ConsumeMsg`. The third one inserts the last deleted text stretch using a `RecallMsg`.

```
MODULE Example8;

IMPORT Display, Oberon, Out, Texts;

PROCEDURE CopySelection*;
VAR T: Texts.Text; B: Texts.Buffer; beg, end, time: LONGINT;
        C: Oberon.CaretMsg; S: Oberon.SelectMsg;
BEGIN
    Oberon.GetSelection(T, beg, end, time);
    (* This procedure, which is provided with the system, does indeed
        broadcast a SelectMsg. Its implementation is equivalent to:

        S.id := Oberon.get; S.F := NIL; S.time := -1; S.text := NIL;
        Display.Broadcast(S);
        T := S.text; beg := S.beg; end := S.end; time := S.time
    *)
    IF time >= 0 THEN
        NEW(B);
        Texts.OpenBuf(B);
```

```
      Texts.Save(T, beg, end, B);   (* make a copy of the selection in buffer B *)
      C.id := Oberon.get; C.F := NIL; C.car := NIL; C.text := NIL;
      Display.Broadcast(C);   (* get caret position *)
      IF C.text # NIL THEN
        Texts.Insert(C.text, C.pos, B);   (* insert the selection at the caret *)
        S.id := Oberon.set;
        S.F := C.car; S.sel := C.car; S.text := C.text; S.time := -1;
        S.beg := C.pos; S.end := C.pos + (end - beg);
        Display.Broadcast(S);
      ELSE Out.String("No caret set"); Out.Ln
      END
    ELSE Out.String("No selection."); Out.Ln
    END
  END CopySelection;

  PROCEDURE CopyToCaret*;
  VAR T: Texts.Text; beg, end, time: LONGINT;
      C: Oberon.ConsumeMsg;
  BEGIN
    Oberon.GetSelection(T, beg, end, time);
    IF time >= 0 THEN
      C.F := NIL; C.text := T; C.beg := beg; C.end := end;
      Display.Broadcast(C)
    END
  END CopyToCaret;

  PROCEDURE Recall*;
  (* The same functionality is provided by the command TextDocs.Recall *)

  VAR R: Oberon.RecallMsg;
  BEGIN
    R.F := NIL; Display.Broadcast(R)
  END Recall;

  END Example8.
```

`Example8.CopySelection`

`Example8.CopyToCaret`

`Example8.Recall` You have to delete a text stretch before using this command otherwise nothing happens.

In all cases, the caret is automatically repositioned after the last character inserted.

5.9 Introducing Gadgets

In our discussions so far, we have discussed not only how to use texts but also how the system is structured (typing and run–time organization) and how to communicate with frames in the display space (message broadcasting). This is already enough background knowledge to explore gadget manipulation as the next topic. By gadget manipulation we mean creating, destroying, inserting, changing etc. gadgets that already exist. In the user guide you learned how to manipulate gadgets interactively with the mouse and Columbus. This section explains how gadgets are manipulated under program control.

The Gadget types. Earlier it was mentioned that gadgets are extensions of `Objects.Object` and `Display.Frame`. The gadget extension of `Display.Frame` is called a *visual gadget*, and the gadget extension of `Objects.Object` is called a *model gadget*. We now take a look at the corresponding types in module `Gadgets`:

```
  (* Base type of the model gadgets *)
  Object = POINTER TO ObjDesc;
  ObjDesc = RECORD (Objects.ObjDesc)
    attr: Attributes.Attr;   (* Attribute list. Private variable. *)
    link: Links.Link;   (* Link list. Private variable. *)
  END;
```

```
    (* Base type of the visual gadgets *)
    Frame = POINTER TO FrameDesc;
    FrameDesc = RECORD (Display.FrameDesc)
        attr: Attributes.Attr;    (* Attribute list. Private variable. *)
        link: Links.Link;    (* Link list. Private variable. *)
        state: SET;
        mask: Display3.Mask;
            (* Cached display mask. Can be NIL to indicate no/invalid mask. *)
        obj: Objects.Object;    (* Model object, if any. *)
    END;
```

Both types export a field `attr` and a field `link` which will be described later on (cf. Decorations). Furthermore, the visual gadgets have a `state` field containing four boolean *flags* with information about the gadget state, a `mask` field that remembers the visible area of the gadget (for display clipping purposes), and a pointer called `obj` to the model gadget attached to the visual gadget (if any). These fields are only of interest when programming new gadgets. The *selected* flag indicates if the gadget is selected, the *lockedsize* flag indicates if the width and the height of the gadget is constant, the *lockedcontents* flag locks all the children of the gadget against further editing and the *transparent* flag indicates if the gadget has a transparent background.

We now focus on what gadgets are available and how to manipulate them. We pick a gadget we would like to use from the Standard Gadgets Reference, find out what its *generator procedure* is, and use this information to find the module that implements this gadget. Then we use Watson to retrieve the module definition with the types and operations of that gadget. After doing this, we have enough information to use that gadget.

We select two standard gadgets, Integer and Slider, for further study. These gadgets have generators `BasicGadgets.NewInteger` and `BasicGadgets.NewSlider` respectively, implying that both of them are implemented in the same module `BasicGadgets`. Using Watson, we can extract the definitions of these two gadgets from their module:

```
        Integer = POINTER TO IntegerDesc;
        IntegerDesc = RECORD (Gadgets.ObjDesc)
            val: LONGINT;    (* current value *)
        END;

        Slider = POINTER TO SliderDesc;
        SliderDesc = RECORD (Gadgets.FrameDesc)
            min, max, val: LONGINT;    (* Minimum, maximum and current value *)
        END;
```

We immediately infer from their base types that a model and a visual gadget are involved. Each gadget has some local fields storing LONGINT values. Before doing any manipulation, we have either to create new instances of these gadgets or to locate already existing gadgets of these types, for example in a user interface Panel.

Attributes. Each gadget class typically defines several instance variables, some of which, called *attributes*, are distinguished from other instance variables by being visible and modifiable at run–time by the end–user using Columbus. Attributes are properties of gadget instances that define their state, representation and behavior. Each attribute has a name and a typed value. Allowed types are string (ARRAY OF CHAR), LONGINT, REAL, LONGREAL CHAR, BOOLEAN and SET, a subset of the Oberon language elementary types. The accessibility of attributes by the end–user places a responsability on the programmer of the class to "export" only those internal details of a gadget that can easily be understood and that can be of possible use to the Oberon user.

Creating new instances of gadgets. There are quite a few ways of creating an instance of a gadget under module control. As the name generator indicates, simply calling the generator procedure creates a new gadget instance of that type. This is also the way the `Gadgets.Panel` creates a new gadget (with the help of module Modules). The trick to know is that a generator always assigns the newly created gadget to the global variable called Objects.NewObj, from where the caller is expected to pick it up:

```
IMPORT BasicGadgets, Objects;

PROCEDURE CreateSlider;
VAR S: BasicGadgets.Slider;
BEGIN
    BasicGadgets.NewSlider;
    S := Objects.NewObj(BasicGadgets.Slider);

    (* and set fields directly *)
    S.val := 0; S.min := 0; S.max := 1000
END CreateSlider;
```

A type guard is required to make sure that an object of the right type is assigned to the local variable S. However, a gadget is not located in the display space just after creating it – it is said to be *off–screen*. It must be inserted into the display space explicitly, as indicated later. While in this *off–screen* state, we may manipulate the fields of a gadget directly as is done in the example, although this is not a recommended practice.

For the curious, we now illustrate how NewSlider works. This is also the way to create a Slider without calling the generator procedure:

```
IMPORT BasicGadgets;

PROCEDURE CreateSlider;
VAR S: BasicGadgets.Slider;
BEGIN
    NEW(S); BasicGadgets.InitSlider(S);    (* allocate and initialize *)

    (* and set fields directly *)
    S.val := 0; S.min := 0; S.max := 1000
END CreateSlider;
```

In this case, we have to allocate an own Slider with NEW and then initialize it. By convention, for each gadget we have a corresponding InitX procedure, where X is the type of the gadget. The task of the InitX procedure is to install the message handler of the gadget and to initialize its RECORD fields to a consistent state. Afterwards, it is ready for use.

There is however a problem with the approach above: we are using the module BasicGadgets by importing it directly. This makes an extremely strong coupling between our new code and module BasicGadgets. Should module BasicGadgets change in future, we might have to recompile or even adjust our module too. In Oberon, however, we can use gadgets *without* importing them. This idea plays an important role in an extensible system and will be observed many times in the remainder of the book. The trick is to use a special procedure called Gadgets.CreateObject to create a gadget from the generator string:

```
IMPORT Gadgets;

PROCEDURE CreateSlider;
VAR obj: Objects.Object;
BEGIN
    obj := Gadgets.CreateObject("BasicGadgets.NewSlider")
END CreateSlider;
```

Behind the scenes, Gadgets.CreateObject uses the module loader to load and execute the generator procedure of a gadget, and thus avoids importing a gadget implementation directly.
At this point, the attentive reader will be wondering how we can access the private RECORD fields of the Slider without importing module BasicGadgets and without defining a variable of type BasicGadgets.Slider. The answer is found in the special attribute message. This message is used by Columbus to inspect

and change the state of gadgets.

Attribute Handling. The attribute mechanism, which was introduced in Chapter 3, has two aspects. First, a message mechanism is used to set, retrieve and enumerate the attributes of a gadget. This is the interface or the "outside" of a gadget. Second, a gadget is responsible for storing attributes internally. This is the implementation or the "inside" of a gadget. From the outside, the actual implementation or storage of the gadget attributes is not visible. As users of the gadget, we are only interested in the attribute interface to a gadget and not in how it is implemented (more about the implementation is found in the sections about programming a gadget).

In a similar fashion to `Texts.Scanner`, the attribute mechanism uses a single RECORD message type to pass attributes of a selection of basic Oberon types:

```
    AttrMsg = RECORD ( Objects.ObjMsg )
        id: INTEGER;    (* get, set or enum. *)
        Enum: PROCEDURE (name: ARRAY OF CHAR);
            (* Called by object to enumerate attribute names. *)
        name: Name;    (* Name of the attribute to be set or retrieved (ARRAY OF CHAR).
*)
        res: INTEGER;    (* Return result: < 0 no response, >= 0 action completed. *)
        class: INTEGER;    (* Attribute class (Inval, Int, Real, LongReal, Char, Bool or
String). *)
        i: LONGINT;
        x: REAL;
        y: LONGREAL;
        c: CHAR;
        b: BOOLEAN;
        s: ARRAY 64 OF CHAR
    END;
```

Again we observe a message selector `id` that determines whether we want to read, to write or to enumerate attributes. After sending an attribute message, a zero or positive `res` value indicates that the message was handled successfully by the receiving object. The `AttrMsg` and all messages of the base object protocol (defined in module `Objects`) are sent *directly* to an object, and is not broadcast into the display space. This involves calling the message handler of an object directly. Here is how to get or retrieve the *Value* attribute of an Integer gadget:

```
    PROCEDURE Example*;
    VAR obj: Objects.Object; M: Objects.AttrMsg;
    BEGIN
        obj := Gadgets.CreateObject("BasicGadgets.NewInteger");

        M.id := Objects.get;
        M.name := "Value";    (* attribute to be retrieved *)
        M.res := -1;    (* init result code *)
        obj.handle(obj, M);    (* direct message send *)

        IF M.res >= 0 THEN    (* success ? *)
            Out.String("The Value attribute of the Integer is ");
            IF M.class = Objects.String THEN Out.String(M.s)
            ELSIF M.class = Objects.Int THEN Out.Int(M.i, 0)
            ELSE
                Out.String("(Unknown type)")
            END
        ELSE
            Out.String("Object did not understand the message")
        END;
        Out.Ln
    END Example;
```

This example illustrates a few things. First, a direct send to a gadget involves calling the handler of an object directly and passing the object itself as the first parameter and the message as a second parameter. In general, a typical message send has the structure:

```
x.handle(x, M)
```

where `M` is the message and `x` is the destination object. Verify the type of `Objects.Handler` with Watson to assure yourself of this fact. If we did not pass the receiving object as first parameter, there would be no way for a message handler to find out which object it is bound to.

After the message is sent, the `res` field indicates if the gadget was able to return the requested attribute. If it did return succesfully, the `class` field indicates the type of the attribute and *in which field* corresponding to that type the attribute value is returned. That is, an attribute of type string is returned in field `M.s` whereas an attribute of type integer is returned in `M.i`, and so forth.

Setting or writing an attribute involves doing the operation above in reverse:

```
PROCEDURE Example*;
VAR obj: Objects.Object; M: Objects.AttrMsg;
BEGIN
    obj := Gadgets.CreateObject("BasicGadgets.NewInteger");

    M.id := Objects.set;
    M.name := "Value";    (* attribute to be written *)
    M.class := Objects.Int;    (* type of the attribute *)
    M.i := 42;
    M.res := -1;    (* init result code *)
    obj.handle(obj, M);    (* direct message send *)

    IF M.res >= 0 THEN    (* success ? *)
        Out.String("Attribute was set")
    ELSE
        Out.String("Attribute could not be set")
    END;
    Out.Ln
END Example;
```

The situation where a user tries to set an attribute with the wrong type, for example when assigning an existing string attribute to an integer value, is worth some consideration. According to the ground rules of object orientation, the exact behavior depends on the object handling the message. It could either try to convert the integer into a string representation or simply refuse the operation, but we would never know for sure without inspecting the description of the gadget. A related point is what happens when an attribute is set that does not exist at the time the message is sent. In the current implementation this is often regarded as a combined create and set operation. This behavior hints at the fact that the attributes of a gadget can be a dynamic and growing set of attributes. The alert reader will suspect that the `attr` field of gadgets must provide the solution.

The only other selector value of the attribute message is `enum` used for enumerating the gadget attributes. This is done by passing a call–back procedure to the gadget, which then dutifully calls the call–back for each attribute it has:

```
PROCEDURE MyCallback (name: ARRAY OF CHAR);
BEGIN
    Out.String("Gadget has an attribute called ");
    Out.String(name);
    Out.Ln
END MyCallBack;

PROCEDURE Example*;
VAR obj: Objects.Object; M: Objects.AttrMsg;
BEGIN
    obj := Gadgets.CreateObject("BasicGadgets.NewInteger");

    M.id := Objects.enum;
    M.Enum := MyCallback;
    M.res := -1;    (* init result code *)
    obj.handle(obj, M)    (* direct message send *)
END Example;
```

As suspected, this is the mechanism Columbus uses to figure out which attributes a gadget has.

At this point it is instructional to illustrate how to manipulate the Integer gadget (or for that matter any other gadget) directly without the attribute message. This is of course a valid option for the programmer, except that a strong dependency on an Integer object is then created. This implies that a change to a different type of gadget is not possible without rewriting some code. As will be shown later in the manipulation of gadgets in a user interface, this would result in an unneeded type dependency between program code and a user interface.

```
IMPORT BasicGadgets, Gadgets, Objects;

PROCEDURE Example*;
VAR obj: Objects.Object;
BEGIN
   obj := Gadgets.CreateObject("BasicGadgets.NewInteger");
   WITH obj: BasicGadgets.Integer DO
      obj.val := obj.val + 1
   END
END BasicGadgets;
```

Locating a gadget in the display space. Manipulating existing gadgets in the display space is the basis of all applications created with the Gadgets system. Once a gadget is located in the display space, we can manipulate it either directly by reading and setting its record fields (a strong dependency) or indirectly by using the attribute mechanism (a weak dependency). How do we locate a gadget?

There are essentially three ways of locating gadgets: by *position*, by *state* (the current gadget selection), or by *name*.

Given a certain `X,Y` *position* on the display, we can ask exactly what gadget is located at that position. By convention, such a position is picked with the star marker. The definition of the module `Oberon` reveals that the marker's coordinates are `Oberon.Pointer.X`, `Oberon.Pointer.Y` and that the procedure `Oberon.MarkedFrame` returns the marked frame. This knowledge leads to the following program fragment:

```
MODULE Example9;
IMPORT Display, Oberon, Objects, Out;

PROCEDURE Info* (obj: Objects.Object);
VAR A: Objects.AttrMsg;
BEGIN
    A.id := Objects.get; A.name := "Gen"; A.res := -1;
    obj.handle(obj, A);    (* Retrieve the generator procedure name *)
    Out.String(A.s); Out.Ln
END Info;

PROCEDURE Locate*;
VAR F: Display.Frame;
BEGIN
    F := Oberon.MarkedFrame();
    Info(F)
END Locate;

END Example9.
```

As we already know that communication with the display space is through message broadcasting, the procedure `Oberon.MarkedFrame` makes us suspect that a message broadcast must be hidden behind it. In fact, the `Display.LocateMsg` does the job for us.

```
    LocateMsg = RECORD ( FrameMsg )
        loc: Frame;    (* Result. *)
        X, Y: INTEGER;    (* Absolute location. *)
        u, v: INTEGER    (* Relative coordinates in loc. *)
    END;
```

This message is broadcast to locate the frame positioned at the absolute coordinates X, Y on the display. Here is the implementation copied from the module `Oberon`

```
(** Returns the star (F1) marked frame. *)
PROCEDURE MarkedFrame*(): Display.Frame;
    VAR L: Display.LocateMsg;
BEGIN
    L.loc := NIL; L.X := Pointer.X; L.Y := Pointer.Y; L.F := NIL; L.res := -1;
    Display.Broadcast(L);
    RETURN L.loc
END MarkedFrame;
```

The `L.loc` field returns the frame located at the position `L.X`, `L.Y` and the `L.u`, `L.v` fields return the *relative* position of `L.X`, `L.Y` inside the located frame.

Now that the gadget is located by its frame, all that remains is to identify it. This is best done by the name of its generator procedure found in the *Gen* attribute as is done in the `Info` procedure. This procedure is exported because it will be used again in further examples. Knowingly, the attribute message returns a collection of basic Oberon types from which to select. Thus a cautious programmer would surely include some test of the result field `res` and of the `class` field before proceeding with writing the attribute value to the log. In fact, the necessary test and type conversion are supplied by a collection of procedures found in module `Attributes`. The following modified `Example9` makes use of one of them:

```
MODULE Example9bis;
IMPORT Attributes, Display, Oberon, Out;

PROCEDURE Locate*;
VAR F: Display.Frame; generator: ARRAY 32 OF CHAR;
BEGIN
    F := Oberon.MarkedFrame();
    Attributes.GetString(F, "Gen", generator);
    Out.String(generator)
END Locate;

END Example9bis.
```

The procedures which take care of preparing and issuing an attibute message follow the pattern:

```
GetType(obj: Objects.Object; name: ARRAY OF CHAR; VAR x: T);
```

for retrieving the value `x` of an attribute `name` from an object `obj`, and converting it to type T. The following conversions are performed:

Type	T	Attribute classes converted
Int	LONGINT	Int, String, Real, LongReal
Real	REAL	Real, String, LongReal, Int
LongReal	LONGREAL	LongReal, String, Real, Int
Bool	BOOLEAN	Bool, String, Char
String	ARRAY OF CHAR	String, Int, Bool, Real, LongReal, Bool

Assigning a new value to an attribute is possible with the corresponding "Set" procedures:

```
SetType(obj: Objects.Object; name: ARRAY OF CHAR; x: T);
```

for setting the value of an attribute `name` in an object `obj`, and converting the type T to the attribute class understood by the object. The same conversions are performed.

We shall soon see that the name of a gadget, that is, the value of its *Name* attribute, is of central importance. This remark comes right in time to mention two procedures of the `Gadgets` module that serve the purpose of setting and retrieving the *Name* attribute under program control:

```
GetObjName(obj: Objects.Object; VAR name: ARRAY OF CHAR);
NameObj(obj: Objects.Object; name: ARRAY OF CHAR);
```

The second way of locating a gadget in the display space is *by state*, finding out whether a gadget is selected or not. Just as the module `Texts` provides a procedure for obtaining the current text selection, the module `Gadgets` provides a way to find the gadget selection, or more correctly the frame selection. Whereas the text selection just consisted of a text with a starting and ending position, the gadget selection might consist of a set of gadgets. How is such a set returned? So far, we have not mentioned some fields defined in the base type `Objects.Object`

```
Object = POINTER TO ObjDesc;
ObjDesc = RECORD
    stamp: LONGINT;    (* Time stamp of last message processed by object. *)
    dlink,    (* Next object in the message thread. *)
    slink: Object;    (* Next object in a list of objects. *)
    lib: Library; ref: INTEGER;    (* Library and reference number of object. *)
    handle: Handler    (* Message handler. *)
END;
```

One of them, called `slink`, strings objects together in a list, allowing us to visit them by following the chain. There is no restriction on the type of objects included in such a list. Accordingly, when requesting the current gadget selection, a list of frames is returned. By convention, the selection can consist only of children of one and the same container. This means that two different "threads" are maintained for each container. The `dsc-next` thread contains all the children, whereas the `slink` thread contains only the selected children of the container.

Here is a procedure to display the generator procedure names of the selected gadget(s):

```
MODULE Example10;

IMPORT Gadgets, Objects, Example9;

PROCEDURE Locate*;
```

```
VAR obj: Objects.Object; time: LONGINT;
BEGIN
    Gadgets.GetSelection(obj, time);
    IF time >= 0 THEN
        WHILE obj # NIL DO
            Example9.Info(obj);
            obj := obj.slink
        END
    END
END Locate;

END Example10.
```

There is one caveat though. The `slink` is such a useful mechanism for stringing objects together that it is used for more than one purpose. Although the `s` in `slink` suggests "selection link" at first glance, it really stands for *static link*. This relates to the fact that the `slink` chain is valid only for the periods between message broadcasts that use the chain. It would be even better to say "temporary" instead of static, to be quite correct: a programmer should thus never assume that the `slink` chain remains intact for an indefinite period of time. A container typically builds the `slink` chain when requested, by traversing its children and checking whether they are selected or not, and inserting them into the chain accordingly. A gadget is selected when the constant `Gadgets.selected` is a member of the `state` field of a gadget.

As expected, `Gadgets.GetSelection` is implemented by a message broadcast:

```
PROCEDURE GetSelection* (VAR objs: Objects.Object; VAR time: LONGINT);
VAR SM: Display.SelectMsg;
BEGIN time := -1; objs := NIL;
    SM.id := Display.get; SM.F := NIL;
    SM.sel := NIL; SM.obj := NIL; SM.time := -1;
    Display.Broadcast(SM);
    IF (SM.time >= 0) & (SM.obj # NIL) THEN
        time := SM.time; objs := SM.obj
    END
END GetSelection;
```

In a way similar to that in which the `Oberon.SelectMsg` controls the text selection, the `Display.SelectMsg` controls the selection of gadgets. The `obj` field points to a selected object. Further selected objects in the same container follow in a chain which extends from there. The `sel` field points to the containing frame.

```
    SelectMsg = RECORD ( FrameMsg )
        id: INTEGER;    (* get, set, reset. *)
        time: LONGINT;    (* Time of the selection. *)
        sel: Frame;    (* Parent of selection. *)
        obj: Objects.Object    (* List of objects involved. *)
    END;
```

The third way of locating a gadget in the display space is *by name*. Although we discuss this technique last, it is definitely the most popular and useful way. Locating a gadget by name involves broadcasting an `Objects.FindMsg` into the display space. As each gadget receives the `FindMsg`, it checks whether the object requested matches its name, and returns itself accordingly. Whereas most other broadcasted messages travel in a depth–first fashion through the display space, the `FindMsg` does a breadth–first traversal. This means that the "nearest" gadget from the root of the display space is located.

```
    FindMsg = RECORD ( ObjMsg )
        name: Name;
        obj: Object    (* Result object, if found. *)
    END;
```

Unfortunately, the find technique as sketched here is flawed. The reason is that a user might decide to open the same document twice, thus the named gadget in the document appears twice in the display space. Although separate gadget instances are involved, they have the same name, and are thus difficult to separate. To eliminate the name ambiguity we have to determine where in the display space the search must start, in other words, at what "virtual" root. Typically, such a root is the document instance from where a command is executed. The situation can be clarified as follows.

Suppose a document D contains a gadget named X. Opening D twice results in two viewers (D1 and D2), each with an instance of gadget X. We immediately have the problem of separating the two X's (X1 and X2) from each other. Which of these is meant?

In principle, the user of one of the documents D decides which X is meant. For example, clicking on a button in D1 means the X of document D1 (thus selecting D1 as search context), or clicking on a button in D2 means the X of document D2 (thus selecting D2 as search context). In fact, the scheme used is slightly more general than this. First, the search context or virtual root is determined not only by clicking on a button in a document, but also by executing any command from a user interface element in a document. More concretely, executing a command from a user interface involves executing the *Cmd* or *ConsumeCmd* attribute of a gadget manipulated. Second, the direct container of the gadget executing the command is used as search context, not the document. This allows a fine–grained searching capability and allows a document to contain multiple containers with descendants having the same name. Before a gadget executes a command, it deposits a reference to its direct container in the global variable `Gadgets.context`. This variable specifies the search context and is used in the following manner to locate a gadget in that context:

```
MODULE Example11;
IMPORT BasicGadgets, Gadgets, Objects, Out, Example9;

PROCEDURE Locate*;
VAR obj: Objects.Object;
BEGIN
    obj := Gadgets.FindObj(Gadgets.context, "Test");
    IF (obj # NIL) & (obj IS BasicGadgets.CheckBox) THEN
        WITH obj: BasicGadgets.CheckBox DO
            IF obj.val THEN Out.String("The CheckBox is checked.")
            ELSE Out.String("The CheckBox is not checked.")
            END
        END
    ELSE Out.String("Test not found.")
    END;
    Out.Ln; Out.String("Executor: ");
    Example9.Info(Gadgets.executorObj);
    Out.Ln
END Locate;

END Example11.
```

This example assumes that a gadget named "Test" exists in the context of the command executed. This situation is easily built interactively by inserting a CheckBox and a Button into a Panel, naming the CheckBox "Test" and adding the command `Example11.Locate` to the Button.

Additional global variables of the `Gadgets` module contain references to objects that are of interest when executing a command from a user interface. Among those variables, we may cite `Gadgets.executorObj` which is a reference to the gadget that executed the command.

As suspected, `Gadgets.FindObj` is just a convenient front–end for a message send:

```
PROCEDURE FindObj (context: Objects.Object; name: ARRAY OF CHAR):
Objects.Object;
VAR obj: Objects.Object; M: Objects.FindMsg;
BEGIN
    obj := NIL;
    IF context # NIL THEN (* search by find message *)
        M.obj := NIL; COPY(name, M.name); context.handle(context, M);
        obj := M.obj
    END;
    RETURN obj
END FindObj;
```

In our example, a strong and unneeded dependency was created on a CheckBox gadget. There are two solutions in this case. First, we can program a cascade of IF statements to check the type of the found object. In an extensible system you will however have to keep updating the IF statements when new gadgets become part of the system. A better solution, used in the next example, would be to retrieve, update and then set the *Value* attribute using the `AttrMsg` mechanism instead of importing a specific gadget. Completion of this exercise will quickly illustrate that programming without dependencies can result in blown up code sequences. In this case we would have to take into account that attributes are typed too. This is an unfortunate side effect of statically typed programming languages. In the pragmatic world, most Gadgets programmers make some tradeoff between complete generality and tight user interface coupling.

Display Updates. Only one ingredient is missing before you can link an application module to its user interface. We already know how to locate a gadget in the display space (using one of three techniques), and how to read and change its attributes and RECORD fields (by `AttrMsg` or directly). What is missing now is a way to redisplay a gadget should the value of one of its attributes have changed. More specifically, we have to redisplay a gadget should a changed attribute have an influence on the visual representation of a gadget. This does not happen automatically when we change a setting in the gadget and for a good reason: should we change several attributes at the same time, an immediate update would cause an unneeded display flickering as the gadget would be redisplayed each time. Thus, by convention, the instance responsible for changing a gadget has also the responsibility of updating the display when all changes have been completed. A display update is trivially accomplished by a call to `Gadgets.Update`

```
MODULE Example12;
IMPORT Attributes, Gadgets, Objects;

PROCEDURE Update*;
VAR obj: Objects.Object; val: BOOLEAN;
BEGIN
    obj := Gadgets.FindObj(Gadgets.context, "Test");
    IF obj # NIL THEN
        Attributes.GetBool(obj, "Value", val);
        Attributes.SetBool(obj, "Value", ~val);
        Gadgets.Update(obj)
    END
END Update;

END Example12.
```

Behind the scenes, one of two messages is broadcast into the display space by `Gadgets.Update` In our case, we are manipulating a visual gadget directly, so it suffices to broadcast a `Display.DisplayMsg`

```
    DisplayMsg = RECORD ( FrameMsg )
        device: INTEGER;    (* screen, printer *)
        id: INTEGER;    (* full, area, contents. *)
        u, v, w, h: INTEGER    (* Area to be restored. *)
    END;
```

to redisplay our gadget:

```
PROCEDURE MyFrameUpdate (obj: Display.Frame);
VAR D: Display.DisplayMsg;
BEGIN
    D.device := Display.screen;
    D.id := Display.full;
    D.F := obj;
    Display.Broadcast(D)
    END MyFrameUpdate;
```

In this code fragment the message is broadcast in a directed way to the frame. The id field is set to a value Display.full which requests a complete redisplay.

More generally, the DisplayMsg unifies the functions of displaying and printing a gadget in a single message. The screen variant is a request to a visual gadget to display itself either completely (variant full), or to display a rectangular area of itself (variant area). In the latter case, the area u, v, w, h inside the destination frame is redrawn. As usual, the u and v coordinates are relative to the top–left corner of the destination gadget, thus v is negative. The printer variant requests a gadget to print a *display approximation* or *snapshot* of itself on the printer (variant full), or to print its data contents (variant contents). This can be a multi–page document, as for example in the case of a text document.

We might also be manipulating a model gadget in the display space, which as a non–visual object, cannot respond to a DisplayMsg. In this case, we have to introduce a new message, as explained in the following section.

```
    PROCEDURE MyModelUpdate (obj: Display.Frame);
    VAR M: Gadgets.UpdateMsg;
    BEGIN
        M.obj := obj; M.F := NIL;
        Display.Broadcast(M)
    END MyModelUpdate;
```

Here it is important to note that a true broadcast is used and that M.obj specifies the model gadget that has changed.

Manipulating model gadgets. Earlier, we used Gadgets.FindObj to find a gadget in the display space. The same Gadgets.FindObj can also be used to locate a named model gadget in the display space, and just as before, we can manipulate the state of that model gadget. A call to Gadgets.Update is then used to broadcast a Gadgets.UpdateMsg into the display space to indicate that the model has changed its value, and that each view depending on this model should update its representation accordingly:

```
    MODULE Example13;
    IMPORT Attributes, Gadgets, Objects;

    PROCEDURE Increment*;
    VAR obj: Objects.Object; val: LONGINT;
    BEGIN
        obj := Gadgets.FindObj(Gadgets.context, "Counter");
        IF obj # NIL THEN
            Attributes.GetInt(obj, "Value", val);
            Attributes.SetInt(obj, "Value", (val + 1));
            Gadgets.Update(obj)
        END
    END Increment;

    END Example13.
```

Thus, depending on the type of the gadget that has changed (model or visual), a Gadgets.UpdateMsg or a Display.DisplayMsg is broadcast into the display space (hidden in Gadgets.Update). The reader will notice that the Texts.UpdateMsg is the analogue of Gadgets.UpdateMsg in the text sub–system of Oberon.

In general, it is a better idea to manipulate model gadgets rather than visual gadgets in a user interface. The reason is that your application module is insulated against model representation changes (i.e. a new view for an existing model) in the user interface.

Manipulation of gadgets in the display space. Typically, user interfaces are constructed interactively using the Gadgets.Panel. In some cases however, we need to construct them by program. This requires mechanisms for inserting, deleting and moving gadgets in the display space.

A straightforward way to add a gadget to the display space is to request it to be inserted at the current caret position. This moves the gadget from *off–screen* to *on–screen*:

```
MODULE Example14;
IMPORT Gadgets, Objects;

PROCEDURE Insert*;
VAR obj: Objects.Object;
BEGIN
    obj := Gadgets.CreateObject("BasicGadgets.NewButton");
    Gadgets.Integrate(obj)
END Insert;

END Example14.
```

`Gadgets.Integrate` is a convenient front–end for hiding the message broadcast that happens behind the scenes:

```
PROCEDURE Integrate (obj: Objects.Object);
VAR C: Display.ConsumeMsg;
BEGIN
    IF obj # NIL THEN
        C.id := Display.integrate;
        C.obj := obj; C.F := NIL;
        Display.Broadcast(C)
    END
END Integrate;
```

The `Display.ConsumeMsg` is a general mechanism for adding gadgets to the display space.

```
    ConsumeMsg = RECORD ( FrameMsg )
        id: INTEGER;    (* Drop, integrate. *)
        u, v: INTEGER;    (* Relative coordinates in destination when drop. *)
        obj: Objects.Object    (* List of objects to be consumed *)
    END;
```

This message comes in two varieties. The *integrate* mechanism inserts a gadget at the caret (if any), and the *drop* mechanism inserts a gadget at a specific *relative* position u, v in a specific gadget. The following example shows how to insert a gadget exactly at the star marker:

```
MODULE Example15;
IMPORT Display, Gadgets, Oberon;

PROCEDURE Drop*;
VAR F: Display.Frame; u, v: INTEGER;
    C: Display.ConsumeMsg;
BEGIN
    Gadgets.ThisFrame(Oberon.Pointer.X, Oberon.Pointer.Y, F, u, v);

    C.F := F;
    C.id := Display.drop;
    C.obj := Gadgets.CreateObject("BasicGadgets.NewButton");
    C.u := u; C.v := v;    (* relative position in container *)
    Display.Broadcast(C)
END Drop;

END Example15.
```

In this example, notice how the relative coordinates of the star marker inside the located gadget are used as the position of the inserted gadget. An interesting question is what happens if a programmer tries to insert a gadget into an elementary gadget. Again the concept of parental control, or parental *eavesdropping* comes to our rescue. The container of the elementary gadget can monitor the `ConsumeMsg`, and determine if the child actually did consume the gadget (`M.res>=0`). If not, the container can take the gadget for itself, causing it to appear overlapping the elementary gadget.

The `ConsumeMsg` also allows you to insert more than one gadget into the display space. As explained before, the `slink` field is used to link all gadgets together. Here is an example of its use:

```
MODULE Example16;
IMPORT Display, Gadgets, Oberon, Objects;

PROCEDURE Drop*;
VAR F: Display.Frame; u, v: INTEGER;
    C: Display.ConsumeMsg; obj, obj1: Objects.Object;
BEGIN
    Gadgets.ThisFrame(Oberon.Pointer.X, Oberon.Pointer.Y, F, u, v);

    obj := Gadgets.CreateObject("BasicGadgets.NewCheckBox");
    obj1 := Gadgets.CreateObject("BasicGadgets.NewButton");
    obj1(Gadgets.Frame).X := obj(Gadgets.Frame).W + 10;    (* position it *)

    obj.slink := obj1;    (* link gadgets together *)
    C.F := F;
    C.id := Display.drop;
    C.obj := obj;
    C.u := u; C.v := v;
    Display.Broadcast(C)
END Drop;

END Example16.
```

The setting of the X coordinate of the second gadget prevents both gadgets from being inserted on top of each other. The list of gadgets in the `slink` chain forms a local coordinate system where the *relative* position between gadgets is maintained in their new container.

Removing a gadget from the display space uses the `remove` variant of the `Display.ControlMsg`.

```
MODULE Example17;
IMPORT Display, Oberon;

PROCEDURE Remove*;
VAR F: Display.Frame; R: Display.ControlMsg;
BEGIN
    F := Oberon.MarkedFrame();
    R.id := Display.remove; R.F := F;
    Display.Broadcast(R)
  END Remove;

  END Example17.
```

The `Display.ControlMsg` unifies two unrelated functions in a single message. The `remove` variant removes the destination gadget from the display space. The `restore` and `suspend` variants are of informational nature, as sketched below.

```
    ControlMsg = RECORD ( FrameMsg )
      id: INTEGER    (* Remove, suspend, restore. *)
    END;
```

To remove *many* gadgets from the display space, the destination frame is interpreted as the head of the list of `slink` connected frames to be removed. This is a break in style as the destination frame is interpreted as a list. After removing the necessary children, the container updates its display representation. The `suspend` variant warns that all frames from the destination downwards in the display space will be temporarily removed from the display space: only those frames located in the display space receive message broadcasts. The `restore` variant informs the gadget that it is about to be reached by broadcast again, allowing it to re–synchronize with its model.

Finally, it remains for us to specify how to change the location of a gadget; there are two possibilities. First, a gadget can be moved from one container to another in a drag–and–drop fashion. This is accomplished with a remove from the old container followed by a consume into the new container, with the messages introduced before. Second, a gadget may change its position (or size) inside the same container with the `Display.ModifyMsg`.

```
ModifyMsg = RECORD (FrameMsg)
  id: INTEGER;    (* Reduce, extend, move. *)
  mode: INTEGER;    (* Modes display, state. *)
  dX, dY, dW, dH: INTEGER;    (* Change from old coordinates (delta). *)
  X, Y, W, H: INTEGER    (* New coordinates. *)
END;
```

The following example shows how to move the selected gadget a number of pixels to the right:

```
MODULE Example18;
IMPORT Display, Gadgets, Objects;

PROCEDURE Move*;
VAR F: Objects.Object; time: LONGINT; M: Display.ModifyMsg;
BEGIN
    Gadgets.GetSelection(F, time);
    IF time >= 0 THEN
        WITH F: Display.Frame DO
            M.F := F; M.id := Display.move; M.mode := Display.display;
            M.X := F.X + 10; M.Y := F.Y; M.W := F.W; M.H := F.H;

            M.dX := M.X – F.X; M.dY := M.Y – F.Y;
            M.dW := M.W – F.W; M.dH := M.H – F.H;
            Display.Broadcast(M)
        END
    END
END Move;

END Example18.
```

The RECORD fields of the `ModifyMsg` have the following meaning:

`id`	Either `Display.move`, `Display.extend` or `Display.reduce`. `Display.move` requests a translation, whereas `Display.extend` or `Display.reduce` request a translation or change in size (in the Gadgets system both options are regarded the same).
`mode`	`state` requests an update of coordinates whereas `display` requests an update of coordinates and an immediate redisplay of the gadget.
`X,Y,W,H`	New relative location and size of the gadget inside its container.
`dX,dY,dW,dH`	Change from previous coordinates (must be set).

The `mode` field allows optimization of a move operation. In the implementation above, the gadget is moved immediately after the message broadcast. If we had to move a thousand gadgets this could take unnecessary time for each update. When we set the `mode` flag to `Display.state`, however, the gadget is moved but not displayed immediately. A call to `Gadgets.Update` then updates the container of the gadget in one go:

```
MODULE Example19;
IMPORT Display, Gadgets, Objects;

PROCEDURE Move*;
VAR SM: Display.SelectMsg; M: Display.ModifyMsg; obj: Objects.Object;
BEGIN
    SM.id := Display.get; SM.F := NIL; SM.sel := NIL; SM.obj := NIL;
    SM.time := –1;
    Display.Broadcast(SM);

    IF SM.time >= 0 THEN
        obj := SM.obj;
        WHILE obj # NIL DO
            M.F := obj(Display.Frame);
            M.id := Display.move; M.mode := Display.state;
            M.X := M.F.X + 10; M.Y := M.F.Y; M.W := M.F.W; M.H := M.F.H;

            M.dX := M.X – M.F.X; M.dY := M.Y – M.F.Y;
            M.dW := M.W - M.F.W; M.dH := M.H - M.F.H;
            Display.Broadcast(M);

            obj := obj.slink
         END;
         Gadgets.Update(SM.sel)
      END
   END Move;

   END Example19.
```

This example illustrates how the `Display.SelectMsg` is used to return the parent of the selection (`SM.sel`). A call to `Gadgets.GetSelection` is not possible here as it does not return the parent.

Link Handling. Recall that gadgets have attributes that configure their state and behavior. The attribute message (`Objects.AttrMsg`) allows us to inspect and manipulate attributes. When a gadget is located in the display space though, a gadget also knows about gadgets in its vicinity, for example its brother, or its model. This knowledge is made explicit with *links*. A link is a one–way connection between a gadget and its "friends" (i.e. the other gadgets that it knows about). The "Model" link of a gadget, for example, is a connection to the gadget's model (if any). Links, like attributes, have names. Every visual visual gadget for example has a *Model* link. Some other links might be "decorations" that are added to a gadget. The analogue message to the `Objects.AttrMsg` is the `Objects.LinkMsg` with the following definition:

```
   LinkMsg = RECORD (Objects.ObjMsg)
      id: INTEGER;    (* get, set or enum. *)
      Enum: PROCEDURE (name: ARRAY OF CHAR);
         (* Called by object to enumerate link names. *)
      name: Name;    (* Link name. *)
      res: INTEGER;    (* Return result: < 0 = no response, >= 0 action completed. *)
      obj: Object    (* Value of the link to be set, or link result. *)
   END;
```

The message is used to set, retrieve and enumerate links between objects. As the `LinkMsg` has a similar behavior to the `AttrMsg`, we won't go into details about it. However, the `class` field is absent, since only objects can be set or retrieved. Although each gadget has an `obj` field in which the object to be set or retrieved as a link is passed, this field is not exclusively used as reference to the gadgets model. This is in accordance with the principle of complete control that a gadget may possess; it may manage its own internal state as it sees fit. The latter prerogative forces us to use the link message to build or inspect the structures existing between gadgets. The primary usage of the link message is to either inspect or set the model of a gadget. `Gadgets.CreateViewModel`, for instance, uses it to create a view/model pair in one go.

Module `Links` provides a more convenient procedural interface for setting and getting links which hide the messaging mechanism:

```
GetLink(obj: Objects.Object; name: ARRAY OF CHAR; VAR ob1: Objects.Object);
SetLink(obj: Objects.Object; name: ARRAY OF CHAR; ob1: Objects.Object);
```

Decorations. Experience shows that it is often useful to "attach" new attributes to objects that the objects did not originally define. The attached attributes live with the predefined attributes and are usually ignored by the object itself, as it would not know what to do with them. A similar facility of attaching links also exists, enabling the user or programmer to connect objects to others. Just as for attributes the object is not aware of links that have been attached to it. The types `Gadgets.Object` and `Gadgets.Frame`, introduced at the beginning of this section, export a field `attr` that references an abstract data structure that keeps track of these attributes. The same holds true for the common field `link` that manages the link data structure.

The Attribute Scanner. We shall now make a small digression to discuss a useful tool that simplifies the scanning of text parameters. Technically it has little to do with gadgets, except that it is delivered as a part of the Gadgets system to expand the macro characters of commands executed by gadgets.

Module `Attributes` provides a Scanner very similar to that of module `Texts`. The `Texts.Scanner` has already been introduced earlier in this chapter, so the general behavior of the `Attributes.Scanner` can be inferred directly. The attribute scanner has some useful features that distinguish it from the text scanner. First, the attribute scanner can also scan objects that float in the parameter text. For these a new scanner class called `Attributes.Obj` is introduced. The scanned object is returned in the `obj` field of the scanner. Second, the attribute scanner expands macro characters appearing in commands directly as the text is scanned. The most useful feature is the automatic expansion of the text selection macro character "↑". The attribute scanner automatically replaces ↑ with the selection. In this case, the selection is more precisely defined as the text stretch starting at the first selected character and ending at the first whitespace character past the end of the selection, that is, the last word is automatically included. The use of the attribute scanner is recommended, since it eliminates the need to test explicitly for a "↑" as a parameter. Compare the next example with `Example5` written earlier using the `Texts.Scanner`.

```
MODULE Example20;
IMPORT Attributes, Oberon, Texts;

PROCEDURE Open*;
VAR S: Attributes.Scanner;
      T: Texts.Text;
BEGIN
   Attributes.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos);
   Attributes.Scan(S);
   IF S.class = Attributes.Name THEN    (* was a valid file name scanned? *)
      NEW(T); Texts.Open(T, S.s);
      Oberon.OpenText(S.s, T, 250, 200)
   END
END Open;

END Example20.

Example20.Open Example4.Mod~
Example20.Open ↑
```

How to find objects in text. Gadgets floating in text can be detected in two different ways. First, using the attribute scanner which recognizes objects as belonging to the class `Attributes.Obj`, among other things. Second, and somewhat more efficiently, using a Finder which detects exclusively objects. It is thus quite easy to hop from object to object, ignoring the surrounding text.

```
MODULE Example21;
IMPORT Attributes, Oberon, Objects, Texts, Example9;

PROCEDURE ScanObjects*;
VAR S: Attributes.Scanner;
BEGIN
   Attributes.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos);
   Attributes.Scan(S);
   WHILE ~S.eot DO
      IF S.class = Attributes.Obj THEN
         Example9.Info(S.o)
      END;
      Attributes.Scan(S)
   END
END ScanObjects;

PROCEDURE FindObjects*;
VAR F: Texts.Finder; obj: Objects.Object;
BEGIN
   Texts.OpenFinder(F, Oberon.Par.text, Oberon.Par.pos);
   Texts.FindObj(F, obj);
   WHILE ~F.eot & (obj # NIL) DO
      Example9.Info(obj);
```

```
        Texts.FindObj(F, obj)
      END
    END FindObjects;

    END Example21.

    Example21.ScanObjects ~
    Example21.FindObjects ~
```

Gadget Construction. In general, the creation of new gadgets with `Gadgets.CreateObj`, the changing of attributes with `Objects.AttrMsg`, and the construction of links with `Objects.LinkMsg`, are enough to build interesting gadget constellations. However, many Gadgets system modules provide procedures to glue gadgets together. For example, the Iconizer gadgets provide a way of creating an Iconizer out of the two "cards" appearing on each side of it (see the description in the Standard Gadgets Reference). Now, take a look at a complete example for building little pop–up menus. It shows all the features of the Gadgets system we have described so far, including the use of exported comments (an exported comment statement has the form `(** *)`).

```
MODULE Popups; (* jm 14.11.94 *)

(**
    This is an example program to show how you can combine gadgets
    together by program control. The same effect can also be obtained
    completely interactively with the graphical user interface.

    Insert a popup at the caret with the command:

            Popups.Insert "My menu"

    The module builds a small popup menu from a Panel with a
    descriptive text on one side, and a TextNote on the other side. The
    Iconizer can be opened in–place and text entered inside the
    TextNote. You may also change the "Cmd" attribute of the TextNote
    to directly open what you click on with:

            Gadgets.ChangeAttr Cmd "Desktops.OpenDoc #Point" ~

    Note: There is one problem though: should the command be
    executed from a TRAP viewer, the popup will not close again.
*)

IMPORT
    Attributes, Display, Gadgets, Icons, Oberon, Objects, Panels,
    TextFields, TextGadgets, Texts;

PROCEDURE InsertFrame (F, f: Display.Frame; u, v: INTEGER);
VAR C: Display.ConsumeMsg;
BEGIN
    C.id := Display.drop; C.F := F; C.obj := f; C.u := u; C.v := v; C.res := –1; C.dlink := NIL;
    F.handle (F, C)
END InsertFrame;

PROCEDURE Build* (desc: ARRAY OF CHAR; T: Texts.Text; W, H: INTEGER):
Objects.Object;
VAR F: Icons.Iconizer; close: Panels.Panel; open: TextGadgets.Frame;
    caption: TextFields.Caption; t: Texts.Text;
BEGIN
    NEW (close); Panels.InitPanel (close);
    close.W := W; close.H := H;

    Attributes.StrToTxt (desc, t);
    NEW (caption); TextFields.InitCaption (caption, t);

    IF caption.W + 20 >= close.W THEN close.W := caption.W + 20 END;

    InsertFrame (close, caption, close.W – caption.W – 5, –close.H + 3);

    Attributes.SetBool (close, "Locked", TRUE);
    NEW (open); TextGadgets.Init (open, T, TRUE);
    NEW (F); Icons.MakeIconizer (F, close, open);
    Attributes.SetBool (F, "Popup", TRUE);
    RETURN F
END Build;

PROCEDURE Insert*;
VAR S: Attributes.Scanner; T: Texts.Text;
BEGIN
    Attributes.OpenScanner (S, Oberon.Par.text, Oberon.Par.pos);
    Attributes.Scan (S);
    IF (S.class = Attributes.Name) OR (S.class = Attributes.String) THEN
        NEW (T); Texts.Open (T, "");
        Gadgets.Integrate (Build (S.s, T, 60, 25))
    END
END Insert;

END Popups.
```

We suggest that you use the commented module definitions to trace exactly what happens in the example above. Of course, the program is not strictly needed; everything that this program does can be done interactively or with the help of a LayLa description as is shown in Chapter 6.

Copying Objects. Although it might seem a harmless operation at first sight, copying an object is definitely not a simple thing. This is because only the object knows how to copy itself and at the same time an object does not have complete knowledge of the data structure it finds itself in. This conflict between local and global knowledge makes it difficult to make an exact copy of a complicated data structure.

Both shallow and deep copies are initiated by sending an `Objects.CopyMsg` to the root object of the data structure to be copied:

```
    CopyMsg = RECORD (Objects.ObjMsg)
        id: INTEGER;    (* Copy style: Objects.deep or Objects.shallow. *)
        obj: Object    (* Result of the copy operation. *)
    END;
```

The receiver returns a copy of itself in the `obj` field of the message. To implement a deep copy, a receiver container forwards the `CopyMsg` to its children, which in turn return copies of themselves, which are then inserted into the newly created container.

When the data structure being copied is a tree, the technique above works well. We have however to make special provision for the case when the data structure is a DAG or a graph; that is, the recursively propagated `CopyMsg` arrives through two or more paths at the same object, which promptly makes two or more copies of itself (although only one copy should be made, of course). The solution is to distinguish between the first time and the remainder of the times an object receives the `CopyMsg` by using the message time stamp. Should an object receive the `CopyMsg` a second time, it returns the copy it made the first time. Only in this way can perfect deep copies be guaranteed. For the moment though we can ignore the exact behavior of an object on the `CopyMsg`; more important to know is that the message time stamp should be set correctly when a deep copy is to be made.

The following example makes a deep copy of the selection, inserting the result at the caret:

```
    PROCEDURE Copy*;
    VAR M: Display.SelectMsg; p, nl: Objects.Object; time: LONGINT;
            C: Objects.CopyMsg;
    BEGIN
        Gadgets.GetSelection(p, time);
        IF time > 0 THEN
            nl := NIL;
            Objects.Stamp(C);    (* set the message time stamp *)
            WHILE p # NIL DO
                C.id := Objects.deep; p.handle(p, C);
                C.obj.slink := nl; nl := C.obj;
            p := p.slink
        END;
        Gadgets.Integrate(p)
    END;
END Copy;
```

We keep in `nl` a reference to the last copied gadget to link it to the next gadget copied. Also notice how all the instances of `CopyMsg` sent have the same time stamp. This procedure is exactly what is provided by `Gadgets.Copy` which frees the programmer from setting up an `Objects.CopyMsg`

The Message Thread. We have discussed so far how to create gadgets, manipulate them and copy them. All of these operations have a direct relation to the display space. Truly, the exact structure of the display space is hidden from the gadget programmer, as each container takes the responsibility for its children, and we should never interfere from the outside directly in the matters of a container. Instead, we should broadcast messages to influence gadgets in the display space. Also not mentioned explicitly so far is that a child seldom knows in what container it is located (i.e. there is no direct back pointer). This is an important design decision in the Gadget system, and forms the basis of the complete integration property of gadgets.

The lack of knowledge a gadget has about the display space it is located in, sometimes conspires against us. In some cases, a gadget should at least know the container it is located in (to find its siblings using the FindMsg), and in less frequent cases the document it is located in. In principle, a gadget needs to know about all its ancestors right up to the display space root, but admittedly in practice it will seldom be the case. Since an object obtains control only when a message is sent to it, and most often, through a message broadcast, is suffices to "remember" the path that a message travelled to reach it. This we call the *message thread*.

The message thread is a list of objects (often visual gadgets) through which a message travelled through the display space to reach a gadget. The list is constructed in a backwards fashion from the receiver to its container, then to the container of that container and so on all the way up to the display root at the top. As a gadget knows that it received a message, the gadget itself must not be inserted into the message thread; the message thread thus starts at the container of the receiver. The start of the message thread is passed inside of the message itself; this is the function of the `dlink` field which points to the container of the message receiver. The thread continues through the `dlink` field of the `Objects.Object` base type. The following example shows how to visit all the ancestors of a gadget in turn:

```
MODULE Example22;
IMPORT Gadgets, Objects, Example9;

PROCEDURE ShowThread*;
VAR obj: Objects.Object;
BEGIN
    obj := Gadgets.context;
    WHILE obj # NIL DO
        Example9.Info(obj); obj := obj.dlink
    END
END ShowThread;

END Example22.
```

We use `Gadgets.context` as the starting point of the message thread. This global variable is set when a gadget executes a command, and is exactly the `dlink` value of the `InputMsg` that caused the command to be executed in the first place (when you clicked on the gadget). The exact behavior of the `InputMsg` is discussed in the section on programming visual gadgets.

5.10 Persistency and Libraries

Files. The prerequisite for persistent objects is a file system; the objects have to be written to a file in order not to disappear forever when the machine is switched off. Thus, before we can introduce the persistency mechanism of Oberon, we have to introduce the Oberon file system and its two unique features.

First, a distinction is made between a file (a potentially infinite collection of bytes) and the way that file is accessed. The latter is called a *Rider*. The rider access mechanism can be set to a certain location in the file (offset from the beginning of the file), from where data can be read or written, advancing the rider forward by one position for each byte read or written. More than one rider can be positioned on the same file, each rider having potentially a different position. During reading, the rider sets an end–of–file flag when the end of the file is reached. Writing beyond the end of a file with a rider enlarges the file.

The second unique feature is that files can be anonymous (nameless) in Oberon. In fact, a newly created file does not appear in the file directory under its name until it is explicitly *registered*. The separation of a file and its location (directory) provides two interesting possibilities to the Oberon programmer. It is possible to write temporary data to an anonymous file without making the file visible to any other Oberon module. The atomic nature of the register operation allows a file to appear suddenly in the directory, possibly replacing a file in a flash. When an open named file is suddenly deleted by another module, the file becomes anonymous, but it is still possible to access it without fear of an unexpected system behavior. Unregistered anonymous files are simply deleted from disk when they are not used anymore.

The Oberon `Files` module provides the interface to the Oberon file system. To ensure portability of documents between all Oberon platforms, the `Files` module provides a way to store the basic types of the Oberon programming language in a machine independent way. The types `File` and `Rider` are the only exported types of the Files module:

```
   TYPE
      File = POINTER TO Handle;
      Rider = RECORD
         eof: BOOLEAN;    (* Rider has reached the end of the file. *)
         res: LONGINT;    (* Rider operation result code. *)
      END;
```

Note that the file representation is hidden; `Files.Handle` is not exported. The following functions allow us to open or to create a file:

```
   (* Open an existing file. The same file descriptor is returned if a file is opened
multiple times. *)
   PROCEDURE Old (name: ARRAY OF CHAR): File;

(* Creates a new file with the specified name. The same file descriptor is not returned
with multiple calls to New with the same file name (this results in multiple copies of a
file with the same name. i.e. the files are not registered in the directory). *)
   PROCEDURE New (name: ARRAY OF CHAR): File;
```

From now on, the examples will be presented as "no frills" texts concentrating on the essential but nevertheless offering "bare–bone" workable modules and procedures. Usable implementations would require the construction of conditional statements testing and acting on conditions such as "is this string a valid name?", "does this file exist already?", "is the caret set?", "is there a selection?", etc. The examples are offered as workable solutions under the express condition that the rules dictated by the programs are adhered to. If not, a trap is around the corner.

The following example module shows how to set a rider and how to read and write data using the rider. This is a typical example of how to access files:

```
   MODULE Example23;
   IMPORT Files, Oberon, Texts, RandomNumbers;
   VAR F: Files.File; R: Files.Rider;
         S: Texts.Scanner; W: Texts.Writer;
         i, max: LONGINT; res: INTEGER; r: REAL;

   PROCEDURE StoreData*;
   BEGIN
      Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
      F := Files.Old(S.s);
      IF F = NIL THEN    (* the file does not exist, a new file is created *)
         F := Files.New(S.s);
         Files.Set(R, F, 0)
      ELSE    (* the file exists already, data is appended *)
         Files.Set(R, F, Files.Length(F))
      END;
      i := 0; max := ENTIER(RandomNumbers.Uniform() * 10);
      WHILE i < max DO
         Files.WriteReal(R, RandomNumbers.Uniform()); INC(i)
      END;
      Files.Register(F)
   END StoreData;

   PROCEDURE ProcessFile*;
   BEGIN
      Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
      F := Files.Old(S.s);
      Files.Set(R, F, 0);
      Files.ReadReal(R, r);
      WHILE ~R.eof DO
         (* .... *)
         Files.ReadReal(R, r)
      END
   END ProcessFile;

   PROCEDURE StoreTemp*;
   BEGIN
      F := Files.New("");
      Files.Set(R, F, 0);
      i := 0; max := ENTIER(RandomNumbers.Uniform() * 10);
```

```
        WHILE i < max DO
            Files.WriteReal(R, RandomNumbers.Uniform()); INC(i)
        END;
(*    Files.Register(F);   Do NOT register *)
        Files.Set(R, F, 0)
            (* Process the file as shown in ProcessFile ... *)
    END StoreTemp;

    PROCEDURE RenameFile*;
    VAR oldName: ARRAY 64 OF CHAR;
    BEGIN
        Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
        COPY(S.s, oldName);
        Texts.Scan(S);
        Files.Rename(oldName, S.s, res)
    END RenameFile;

    PROCEDURE DeleteFile*;
    BEGIN
        Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
        Files.Delete(S.s, res)
    END DeleteFile;

    END Example23.
```

```
    Example23.StoreData Test – Create the file and store data in it,
                               if it already exists append some data
    Example23.ProcessFile Test – Process the file
    Example23.RenameFile Test TestNew – Rename the file Test to TestNew
    Example23.DeleteFile Test – Delete the file

    Example23.StoreTemp – Create a temporary file and process it from the start
```

A temporary file is created with `Files.New` and is never registered. Its name may be the empty string "". The disk space is reclaimed at the time of the next session. When a file is deleted or renamed, we recommend the handling of the `res` (result code) be expanded to cope with possible error situations.

The description above provides enough knowledge for simple file operations. There are however several other operations; these can be studied in the `Files` definition and won't be repeated here.

How to store and load text from a file.

```
    MODULE Example24;
    IMPORT Display, Files, Oberon, Texts;
    VAR F: Files.File; R: Files.Rider; S: Texts.Scanner; W: Texts.Writer;

    PROCEDURE StoreText*;
    VAR beg, end, time, len: LONGINT; T, TS: Texts.Text; B: Texts.Buffer;
    BEGIN
        Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
        Oberon.GetSelection(TS, beg, end, time);    (* get the selection *)
        F := Files.New(S.s);
        Files.Set(R, F, 0);
        NEW(B); Texts.OpenBuf(B);
        Texts.Save(TS, beg, end, B);    (* copy the selection to the buffer B *)
        NEW(T); Texts.Open(T, "");
        Texts.Append(T, B);
        Texts.Store(T, F, Files.Pos(R), len);
        Files.Register(F);
        Files.Close(F);
    END StoreText;

    PROCEDURE LoadText*;
    VAR T: Texts.Text; C: Oberon.CaretMsg; B: Texts.Buffer;
            len: LONGINT; ch: CHAR;
    BEGIN
        Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
        F := Files.Old(S.s);
        Files.Set(R, F, 0);
        C.id := Oberon.get; C.F := NIL; C.car := NIL; C.text := NIL;
        Display.Broadcast(C);    (* where is the caret? *)
        NEW(T);
        Files.Read(R, ch);    (* Read the 1-byte block identifier. *)
        IF ch = Texts.TextBlockId THEN    (* It must be a text block *)
            Texts.Load(T, F, Files.Pos(R), len);
            NEW(B); Texts.OpenBuf(B);
            Texts.Save(T, 0, T.len, B);
            Texts.Insert(C.text, C.pos, B)
        END
    END LoadText;

    END Example24.
```

```
Example24.StoreText Test1  The selected text is stored in Test1.
Example24.LoadText Test1
```

The store and the load processes are asymmetric: at load time, the rider must be positioned after the first byte, that is the block identifier which was written by the store procedure.

Introductory technical background information on libraries. Although it is possible to store a single gadget in a file (for example a container without its children), the most often used case is to store a collection of gadgets like a document. To store and then later reload a gadget collection to and from a disk file requires two problems to be solved.

First, we have to store the data belonging to each gadget, for example the value field of a TextField or of a Slider. Preferably the data format must be organized in such a way that we can read the data into a different Oberon implementation independent of byte ordering. Since the `Files` module provides a platform-independent way to store multi-byte basic types, the solution is already present.

The second and more difficult aspect is to store the relationships (references) between gadgets in a document. We have to keep track of the model connected to a view, or of the children of a container, so that a load results in exactly the same structure as was stored (this is similar to making a deep copy of a gadget). At run-time, references between objects in the heap are by memory addresses, in other words a pointer from one object to another contains the address of the referenced object. As we cannot influence the address at which an object is allocated, we cannot store addresses to a file; the address would be invalid as soon as we try to load the objects from the file. The solution to this problem is to use a different format for storing pointers. Converting to and from POINTER notation to this format is called *pointer swizzling*.

The idea is quite simple. Before storing a data structure, each object in that data structure is given a number. Instead of storing the address of a reference, we can store the number of the referenced object. At load time we need a way to convert the number read back into a pointer. Of course, this scheme could use the address as the *reference number*, although in practice the magnitude of pointer addresses makes it impractical. During loading, a reference number can be converted into a POINTER only if the referenced object has already been created. This suggests two requirements of our storage scheme: there is a partial ordering between objects during storing and loading, and we need auxiliary data structures to keep track of POINTERs and reference numbers. These two problems are solved in Oberon by introducing *libraries*, and a two-phase store and load protocol.

Libraries. In essence, a library is a dynamically growing array of objects and the reference number of an object is nothing else than the index of that object in a library. A more formal definition of a library can thus be given: a library is an abstract class whose instances are collections of objects. Within the scope of a library, reference numbers identify objects uniquely and invariantly. The action of *binding* an object to a library gives it a reference number. Once objects are *bound* to a library, they can be made *permanent* by storing the whole library contents on disk. Both loading and storing of libraries is built in a standard way into module `Objects` Unbound objects are called *free*.

Libraries provide a useful mechanism to organize a collection of objects. We can imagine functions that pick selected gadgets straight from the library according to reference number. This idea is incorporated into the Oberon system by creating two types of libraries: *private* and *public*. Private libraries are used by applications only for persistent data structures. They are unnamed, hence are also dubbed *anonymous*, and are typically hidden somewhere in the data files of an application. Public libraries by contrast are used as general object repositories. Public libraries are named and are stored in files with the extension `.Lib` Whereas private libraries can be loaded multiple times into memory (each library having different object instances), public libraries are loaded when demanded and then cached in memory until not used anymore. That is, loading a public library with the same name twice, returns the same library each time.

Each object knows its library and reference number:

```
Object = POINTER TO ObjDesc;
ObjDesc = RECORD
    stamp: LONGINT;    (* Time stamp of last message processed by object. *)
    dlink,    (* Next object in the message thread. *)
    slink: Object;    (* Next object in a list of objects. *)
    lib: Library; ref: INTEGER;    (* Library and reference number of object. *)
    handle: Handler    (* Message handler. *)
END;
```

A *free* object has a NIL value in the `lib` field. Here follows a partial definition of a library as found in module `Objects`

```
Library = POINTER TO LibDesc;
LibDesc = RECORD
    name: Name;    (* name of the library. Private library when "", else public library. *)
    dict: Dictionary;    (* Object names. *)
    maxref: INTEGER;    (* Highest reference number used in library. *)

    (* Return a free reference number. *)
    GenRef: PROCEDURE (L: Library; VAR ref: INTEGER);

    (* Return the object with the indicated reference number. *)
    GetObj: PROCEDURE (L: Library; ref: INTEGER; VAR obj: Object);

    (* Insert an object under the indicated reference number. *)
    PutObj: PROCEDURE (L: Library; ref: INTEGER; obj: Object);

    (* Free object with indicated reference number. *)
    FreeObj: PROCEDURE (L: Library; ref: INTEGER);
END
```

To allow the extension of libraries, the four procedures listed above are defined as methods (procedure variables of `Objects.LibDesc`). At this moment it is enough to know that an object will bind itself to a library when we send the `Objects.BindMsg` to it:

```
BindMsg = RECORD (Objects.ObjMsg)
    lib: Library    (* Library where object should be bound. *)
END;
```

A container gadget forwards the `BindMsg` to its children and a visual gadget linked to a model gadget forwards the `BindMsg` to the model so that they are bound to the same library. The implementation of the `BindMsg` typically looks as follows (and is an auxiliary procedure in module `Gadgets` which makes use of the methods GenRef and PutObj defined above):

```
PROCEDURE BindObj (obj: Objects.Object; lib: Objects.Library);
VAR ref: INTEGER;
BEGIN
    IF lib # NIL THEN
        IF (obj.lib = NIL) OR (obj.lib.name[0] = 0X) & (obj.lib # lib) THEN
            lib.GenRef(lib, ref);
            IF ref >= 0 THEN
                lib.PutObj(lib, ref, obj);
            END
        END
    END
END BindObj;
```

In general, it is irrelevant in what type of library (public or private) objects are located. As a rule, each object is a member of at most one library and an object belonging to a public library cannot be rebound to any other library, whereas an object belonging to a private library can be rebound to another private or public library. The IF statement ensures the binding rules: the object to bind must be free or may be bound already to another *private* library.

At this stage, it is not essential to know how objects store references to each other when stored inside a library; we will discuss this when we describe programming new gadgets. For the moment, we show how to store a selected gadget into a library which is then stored in a file and how to load it back from there and insert it at the caret:

```
MODULE Example25;
IMPORT Files, Gadgets, Oberon, Objects, Texts;
VAR F: Files.File; R: Files.Rider; S: Texts.Scanner;

PROCEDURE StoreGadget*;
VAR B: Objects.BindMsg;
      time, len: LONGINT; obj: Objects.Object;
BEGIN
   Gadgets.GetSelection(obj, time);
   IF time >= 0 THEN
      Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
      F := Files.New(S.s);
      Files.Set(R, F, 0);
      NEW(B.lib); Objects.OpenLibrary(B.lib);
      obj.handle(obj, B);    (* bind the object to the library *)
      Files.WriteInt(R, obj.ref);    (* Note the reference number *)
      Objects.StoreLibrary(B.lib, F, Files.Pos(R), len);
      Files.Register(F)
   END
END StoreGadget;

PROCEDURE LoadGadget*;
VAR obj: Objects.Object; L: Objects.Library;
      ref: INTEGER; len: LONGINT; ch: CHAR;
BEGIN
   Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
   F := Files.Old(S.s);
   Files.Set(R, F, 0);
   NEW(L); Objects.OpenLibrary(L);
   Files.ReadInt(R, ref);    (* Obtain the object reference number. *)
   Files.Read(R, ch); (* Skip the 1-byte block identifier. *)
   Objects.LoadLibrary(L, F, Files.Pos(R), len);
   L.GetObj(L, ref, obj);
   Gadgets.Integrate(obj)
END LoadGadget;

END Example25.
```

```
Example25.StoreGadget Test1
Example25.LoadGadget Test1
```

The two procedures `LoadLibrary` and `StoreLibrary` of module Objects load and store a library from and to a file. The library is positioned at pos in file F and has a length len. The bound objects and dictionary are packaged inside the extent [pos, pos+len) of the file. The first procedure requests each of the objects to read their contents from the file and the second to write their contents to the file using a Rider. The two requests are combined in the `Objects.FileMsg` with variants `load` and `store`.

```
FileMsg = RECORD (ObjMsg)
   id: INTEGER;    (* load or store *)
   len: LONGINT;    (* Length of the object data on loading. *)
   R: Files.Rider    (* Rider with which to load or store data. *)
END;
```

The `FileMsg` which is used behind the scene, passes a `Files.Rider` to the object which should store or read data.

Dictionaries. We already know that an object can be given a name, which is used to find the object in a user interface made of a collection of gadgets. A similar (but not the same) idea is used to locate objects in a public library. Since the exact reference number of an object in a public library might change due to editing of the library (which can be done using the `Libraries.Panel`), we must be careful when noting and storing references to public objects. To guard against such unexpected changes, we can associate a name (a string) with a reference number in the library. Using that given name, we can then retrieve the reference number from the dictionary, which is finally used to locate the object itself. The collection of (name, reference number) pairs of a public library is called a *dictionary* and is realized as an abstract type connected to a library. Note that we do not say that a dictionary name is associated with an object in a public library; just the name and the corresponding reference number are remembered. Of course, from the reference number in the dictionary we can retrieve the object having that reference number. In effect we are creating another way to name an object (a public object, to be precise). It is important to realize that this name is not the same as the name of the object. The latter is an attribute of an object. Thus the name of an object can differ from the public name of the object.

The maintenance of the (reference, name) dictionary pairs is performed using the following primitives of module Objects:

```
   (* Associate a name with a reference number. *)
   PROCEDURE PutName (VAR D: Dictionary; key: INTEGER; name: ARRAY OF CHAR);

   (* Get name associated with a key/reference number. *)
   PROCEDURE GetName (VAR D: Dictionary; key: INTEGER; VAR name: ARRAY OF
CHAR);

   (* Given an object name, return the object reference number from the dictionary. *)
   PROCEDURE GetRef (VAR D: Dictionary; name: ARRAY OF CHAR; VAR ref: INTEGER);
```

Dictionaries have been designed in such a way that they can be used for another purpose too. The main idea is to use them also as repositories of often used strings; these strings are called *atoms* and are associated with negative reference numbers, which are then called *keys*. That explains why the second parameter of two of the above procedures is named "key" instead of "ref" as would be expected. This feature is used to reduce the file size of libraries, but at the same time it does complicate the dictionary interface a little bit. For the moment, it is of little interest to us. The following procedure in the module Objects is used to obtain a key for an atom. Note the analogy with the method GenRef mentioned earlier.

```
   (* Allocate a key (any integer < 0) to a name. *)
   PROCEDURE GetKey (VAR D: Dictionary; name: ARRAY OF CHAR; VAR key:
INTEGER);
```

The maintenance of the (key, atom) dictionary pairs is performed using the three primitives defined above.

Public Gadgets. As explained in the previous paragraph, the dictionary mechanism allows us to name objects in public libraries. These objects are called *public objects*. By convention, we refer to a public object in the form `L.O` where `L` is the library name and `O` is the object name (dictionary name). Public objects have the useful property of having only one instance at any time in the system. This is an effect of public libraries being loaded only once and then cached. This makes public objects predestined to be shared between Oberon applications. Perhaps the most often used library is the `Icons.Lib`; it contains numerous pictures representing useful icons. Libraries are used also to store the menu bars that documents typically use. In concept, a hierarchy of public libraries can be imagined, each object using objects in other libraries. The library hierarchy has an analogue in the module hierarchy, where one libary imports another library. In practice though, the library hierarchy is flat with only one level, although nothing prevents you from using them in the more general way.

Just as we can locate a named object in a user interface, so we can locate a public object specified in `L.O` notation using the procedure `Gadgets.FindPublicObj`. This procedure has the following implementation:

```
   PROCEDURE FindPublicObj (name: ARRAY OF CHAR): Objects.Object;
   VAR obj: Objects.Object; libname, objname: ARRAY 64 OF CHAR;
      i, j, k, ref: INTEGER; lib: Objects.Library;
   BEGIN
      obj := NIL; i := 0; j := 0;
      WHILE (name[i] # ".") & (name[i] # 0X) DO libname[j] := name[i]; INC(j); INC(i)
```

```
    END;
    IF name[i] = 0X THEN RETURN NIL END;
    libname[j] := 0X; k := j; INC(i); j := 0;
    WHILE (name[i] # " ") & (name[i] # 0X) DO objname[j] := name[i]; INC(j); INC(i)
    END;
    objname[j] := 0X;
    libname[k] := "."; libname[k+1] := "L"; libname[k+2] := "i"; libname[k+3] := "b";
    libname[k+4] := 0X;
    lib := Objects.ThisLibrary(libname);
    IF lib # NIL THEN
      Objects.GetRef(lib.dict, objname, ref);
      IF ref # MIN(INTEGER) THEN lib.GetObj(lib, ref, obj) END
    END;
    RETURN obj
  END FindPublicObj;
```

As can be seen, the biggest task is to take the name `L.O` apart to load the library `L.Lib` using `Objects.ThisLibrary`. `Objects.ThisLibrary` searches for the library in the public library cache first, and loads the public library from the disk file `L.Lib` if it is not there yet.

In conclusion, we show how to insert a newly created object into a public library "Test.Lib" and name it "Pluto", and how to find it again under the name "Test.Pluto" with `Gadgets.FindPublicObj`. If the library does not already exist, it is automatically created when `Objects.ThisLibrary` is called.

```
  MODULE Example26;
  IMPORT Gadgets, Oberon, Objects, Texts;
  VAR S: Texts.Scanner; L: Objects.Library; obj: Objects.Object;
    B: Objects.BindMsg; C: Objects.CopyMsg;

  PROCEDURE MakePublic*;
  BEGIN
    Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos);
    Texts.Scan(S);
    L := Objects.ThisLibrary(S.s);
    obj := Gadgets.CreateObject("Panels.NewPanel");
    B.lib := L;
    obj.handle(obj, B);
    Texts.Scan(S);
    Objects.PutName(L.dict, obj.ref, S.s);
    L.Store(L)    (* Optional: make the addition persistent. *)
  END MakePublic;

  PROCEDURE GetPublic*;
  BEGIN
    Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos);
    Texts.Scan(S);
    obj := Gadgets.FindPublicObj(S.s);
    C.id := Objects.deep; Objects.Stamp(C);
    obj.handle(obj, C);
    Gadgets.Integrate(C.obj)
  END GetPublic;

  END Example26.
```

```
  Example26.MakePublic Test.Lib Pluto
  Example26.GetPublic Test.Pluto
```

5.11 Programming New Gadgets

Proficient programmers will often notice that they require a special gadget implementation for a special–purpose application. It might also be the case that the new gadget required should function approximately like an existing gadget with a few exceptions. The programmer must thus choose either to program a new gadget from scratch or to extend an existing gadget. Different skill levels are required depending on the classification of the programmed component as model, elementary, container, camera–view or document gadget. For example, model gadgets are easier to program than elementary visual gadgets, which are easier to program than a visual gadget that contains other gadgets.

Extensive experience has shown that programming gadgets involves applying a set of standard *design patterns*. A design pattern in Oberon is a proposal how to solve a specific class of problems. In the most concrete form, a design pattern is a skeletal piece of source code that is copied and then modified for own purposes. This is a typical "fill in the blank" approach adopted by component frameworks. One does not start from scratch when programming a gadget, but some kind of prepared *skeleton* is already available. This allows programmers to create a partially functional gadget in no time at all. Most importantly, it allows them to develop objects incrementally by only adding features to things that already work. Surprisingly large parts of most gadgets are always the same and can thus be shared by most gadgets. They may be treated as skeletons with no blanks to fill in. Using design patterns or reusing code saves memory and simplifies programming. In more general terms, design patterns are guidelines for building software. They are intended to be general enough to be reused in many situations.

The supporting examples make use of these generally applicable design patterns in the development of new gadgets. Though the design patterns are easily recognizable, most examples contain noticeable variations suggesting that there is still ample room left for creativity by programmers with their own ideas. By combining design patterns in innovative ways, you can quickly create new patterns that are robust and easily understood.

5.11.1 Gadget structure and implementation steps

A component consists of a type definition, message definitions, procedures to copy and initialize an object, and the ubiquitous message handler. Here is an *outline* of the typical module structure required for implementing an object MyObject:

```
MODULE Example;
  IMPORT Objects;

  TYPE
    MyObject* = POINTER TO MyObjectDesc;
    MyObjectDesc* = RECORD (Objects.Object);
      (* extended fields *)
    END;

    MyMsg* = RECORD (Objects.ObjMsg)
      (* message arguments *)
    END;

  PROCEDURE Copy* (VAR F: Objects.CopyMsg; from, to: MyObject);
    (* Copy fields of from to to *)

  PROCEDURE Handler* (F: Objects.Object; VAR M: Objects.ObjMsg);
    (* Message handler *)

  PROCEDURE Init* (obj: MyObj);    (* install message handler *)
  BEGIN
    obj.handle := Handler;
    (* initialize own fields *)
  END Init;

  PROCEDURE New*;    (* Component generator *)
  VAR obj: MyObject;
  BEGIN
    NEW(obj); Init(obj); Objects.NewObj := obj
  END New;

END Example.
```

Two *internal views* of objects are possible, namely that of the implementer of the component intending to create a new object from scratch and that of a person who wants to extend an existing *base object class*. The primary and probably most arduous task of the implementer is the implementation of the message handler. Depending on the class of object, different message types are to be handled. The simplest objects respond only to the *object messages* whereas the visual objects respond to both the *object messages* and the *frame messages*.

To extend a component, you must export at least the object type, the message handler as well as the `Copy` and `Init` procedures of the base object. If this condition is not fulfilled, other programmers will not be able to extend your new components. The object extension must watch out that calling the `Init` procedure of the base object is done before initializing the extension and that the correct message handler is written over that of the *base object class*. Often the base type is `Gadgets.Object` for model gadgets, and `Gadgets.Frame` for visual gadgets. These base types (or better said classes) implement the standard functionality for model and visual gadgets respectively. This suggests the following implementation steps for a new gadget:

0. choose an existing gadget type as a base,
1. extend it with instance variables,
2. create a new message type or extend an existing one,
3. create a generator procedure,
4. create an initialization procedure,
5. create a message handler.

In many circumstances some of these steps may not be required at all, thus easing the construction exercise quite a bit.

5.11.2 Default message handling

Even though module `Objects` defines the `AttrMsg`, `BindMsg`, `CopyMsg`, `FileMsg`, `FindMsg` and `LinkMsg`, it does not provide any support for handling these messages. In a similar manner, module `Display` defines the set of frame messages but does not provide further support for handling them. Some aspects of these messages have consequently been factored out into default message handlers. Handlers for extensions of `Gadgets.Object` and `Gadgets.Frame` are provided in module `Gadgets`. The handlers are called `Gadgets.objecthandle` and `Gadgets.framehandle` respectively. A custom designed message handler must pass control to the default handlers, and messages not understood by this message handler should be passed to the default message handler for interpretation. The following paragraphs state the default handling of the object messages by `Gadgets.objecthandle` and `Gadgets.framehandle`.

AttrMsg – The default handling includes the universal *Name* attribute of a gadget and the management of the attached attributes.

BindMsg – The default handling binds the object and all the objects that the attached links reference. This is done in the `BindObject` procedure which was described in the section on libraries.

CopyMsg – The default handling creates a copy of an object and copies the fields belonging to the base types, that is, handle and attributes. This is done in the `CopyObject` procedure.

FileMsg – The default handling stores or loads the fields of the base types. An object should always read exactly as many bytes as it had written previously.

FindMsg – The default handling checks if the searched for object match self, and should this be the case, return itself.

LinkMsg – The default handling involves the handling of links that have been attached to a gadget. The type `Gadgets.Frame` contains a field `obj` that refers to the model of the gadget and is seen as a link called "Model" by clients.

The following paragraphs state the default handling of the frame messages by `Gadgets.framehandle`

Display.ConsumeMsg – The default handling consists in executing the command in the *ConsumeCmd* attribute if the gadget has one.

Display.ControlMsg – The default handling forwards the message to the gadget's model.

Display.DisplayMsg – The default handling draws a rectangular mask.

Display.LocateMsg – The default handling locates the marked gadget by its frame coordinates, width and height.

Display.ModifyMsg – The default handling includes changing the relative position or the size of a child in a container. This is done in the `Adjust` procedure.

Display.SelectMsg – The default handling consists in changing the `state` field of a gadget from selected to not selected or vice–versa. Containers typically interpret pressing the MR key as selecting/deselecting a gadget.

Oberon.InputMsg – This message is so special that it and its handling deserves a special description.

The input message Oberon.InputMsg

```
InputMsg = RECORD (Display.FrameMsg)
  id: INTEGER;    (* consume, track *)
  keys: SET;    (* Mouse buttons. *)
  X, Y: INTEGER;    (* Mouse position. *)
  ch: CHAR;    (* Character typed. *)
  fnt: Fonts.Font;    (* Font of typed character. *)
  col, voff: SHORTINT    (* Color and vertical offset of typed character. *)
END;
```

The input message delivers mouse (variant `track`) and keyboard events (variant `consume`) to the display space. It is repeatedly broadcast into the display space by the `Oberon.Loop` for each input event. Here resides a particularity of the message: the input message is interpreted by the handler which so to speak "consumes" it. All its fields are *out* parameters. Normally, a programmer does not write a piece of code sending out such a message. In the case of a mouse event, the fields `X` and `Y` indicate the absolute mouse position (cursor hotspot) and `keys` the mouse key state (which mouse buttons are pressed). The mouse keys are numbered 0, 1, 2 for right, middle, and left respectively. The display space normally forwards this message only to the frame located at position `X,Y` on the display. In the case of a keyboard event, the ASCII keycode is contained in the `ch` field (check the description of module `Input` for special keycodes). The fields `fnt`, `col` and `voff` give information about the requested font, color index and vertical offset in pixels.

The default handling of mouse events is contained in the procedure `Gadgets.TrackFrame` given below:

```
PROCEDURE TrackFrame* (F: Display.Frame; VAR M: Oberon.InputMsg);
VAR keys: SET; x, y, w, h: INTEGER; R: Display3.Mask;
BEGIN
   WITH F: Frame DO
      IF ~(selected IN F.state) & (middle IN M.keys) THEN
         (* only when not selected and middle key *)
         x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
```

```
          IF InActiveArea(F, M) THEN
            (* usable areas, corner, sides may be part, TRUE if locked *)
            IF HasCmdAttr(F, "Cmd") THEN
              MakeMask(F, x, y, M.dlink, R);
              Effects.TrackHighlight(R, keys, M.X, M.Y, x, y, w, h);
              IF InActiveArea(F, M) & (keys = {1}) THEN
                ExecuteAttr(F(Frame), "Cmd", M.dlink, NIL, NIL)
              END;
              M.res := 0
            ELSIF ~IsLocked(F, M.dlink) THEN MoveFrame(F, M)
            ELSE Oberon.DrawCursor(Oberon.Mouse, Effects.Arrow, M.X, M.Y)
            END
          ELSIF Effects.InCorner(M.X, M.Y, x, y, w, h) & ~(lockedsize IN F.state) THEN
            SizeFrame(F, M)
          ELSIF Effects.InBorder(M.X, M.Y, x, y, w, h) OR
                Effects.InCorner(M.X, M.Y, x, y, w, h) THEN
            MoveFrame(F, M)
          ELSE Oberon.DrawCursor(Oberon.Mouse, Effects.Arrow, M.X, M.Y)
          END
        ELSE Oberon.DrawCursor(Oberon.Mouse, Effects.Arrow, M.X, M.Y)
        END
      END
    END TrackFrame;
```

Only the MM key is handled. Once selected, a gadget does not respond to mouse events; the parent takes control of these events. When the mouse is located in a corner, the gadget is resized. When the mouse is located in the border around the gadget, the gadget is moved.

We suggest that you exercise this behavior with a Clock gadget inserted into a Panel. The final touch can be given by using Columbus to add a *Cmd* attribute to the Clock and by assigning the value "System.Time" to this attribute.

5.11.3 The class inheritance design pattern

Class inheritance means using an existing type by exchanging only the message handler. This technique can be used only if the new gadget does not have own instance variables, and has the advantage that the type hierarchy is "flatter". The immediate consequence is that the steps 1 and 2 in the list of activities above is not required and that step 4 may also be left out, unless the initialization provided by the base object is not suitable.

As an example, we suggest creating a new class of "seek buttons" equipped with the functionality to seek sequentially through an audio file of unknown size. Clearly, a "pop–up" Button can be envisaged as first approximation assuming that a delta displacement could be initiated by the execution of a command hidden in its *Cmd* attribute. But on second thought, progress through the file would be controlled by an unpredictably large number of Button clicks. A better approach would be to allow the execution of such a command as long as the Button is pressed. The following source text produces the desired effect:

```
MODULE SeekButtons;
IMPORT BasicGadgets, Display, Display3, Gadgets, Input, Oberon, Objects;
CONST middle = 1;

PROCEDURE Handler* (F: Objects.Object; VAR M: Objects.ObjMsg);
  VAR x, y, w, h: INTEGER; Q: Display3.Mask; keysum: SET;
BEGIN
  WITH F: BasicGadgets.Button DO
    IF M IS Oberon.InputMsg THEN
      WITH M: Oberon.InputMsg DO
        IF (M.F = NIL) OR (M.F = F) THEN
          IF (M.id = Oberon.track) & (M.keys = {middle}) &
                Gadgets.InActiveArea(F, M) THEN

            (* calculate the gadget's absolute display coordinates *)
            x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;

            Gadgets.MakeMask(F, x, y, M.dlink, Q);
```

```
                    Display3.Rect3D(Q, Display3.topC, Display3.bottomC,
                                          x+1, y+1, w–2, h–2, 1, Display.invert);
                    keysum := M.keys;
                    REPEAT
                        Input.Mouse(M.keys, M.X, M.Y);
                        Gadgets.ExecuteAttr(F, "Cmd", M.dlink, NIL, NIL);
                        keysum := keysum + M.keys
                    UNTIL M.keys = {};
                    Display3.Rect3D(Q, Display3.topC, Display3.bottomC,
                                          x+1, y+1, w–2, h–2, 1, Display.invert);
                    M.res := 0
                ELSE BasicGadgets.ButtonHandler(F, M)
                END
            END
        END
    ELSIF M IS Objects.AttrMsg THEN
        WITH M: Objects.AttrMsg DO
            IF (M.id = Objects.get) & (M.name = "Gen") THEN
                M.class := Objects.String; M.s := "SeekButtons.New"; M.res := 0
            ELSE BasicGadgets.ButtonHandler(F, M)
            END
        END
    ELSE BasicGadgets.ButtonHandler(F, M)
    END
  END
END Handler;

PROCEDURE New*;
  VAR obj: BasicGadgets.Button;
BEGIN
  NEW(obj); BasicGadgets.InitButton(obj); obj.handle := Handler; Objects.NewObj
:= obj
END New;

END SeekButtons.
```

The generator procedure `New` instantiates and initializes a standard Button and replaces the handler by a new one. The first important function of this handler is to supply the value of the *Gen* attribute on a "get" request in an `AttrMsg`. This is essential for Columbus and for loading the gadget from disk. This function will be omnipresent in the construction of new gadgets. The second and only other function of the handler is to react to an MM key sollicitation differently than a standard Button. When the MM key is pressed in the "active" area of the Button, this visual gadget must redraw itself to appear pressed with a 3D effect until the key is released. For this purpose:

1 – the gadget's absolute display coordinates are calculated. Earlier in the discussion about the display space structure, we learned that `F.X` and `F.Y` are the *relative* coordinates of a frame in its parent frame and that `F.Y` is *negative*. The *absolute* coordinates of the parent's top–left corner are returned in M.x and M.y by the system. Hence the local variables x, y, w and h are set to the absolute display coordinates of the visual gadget for later use in the message handler,
2 – the display mask `Q` is calculated (see below),
3 – the drawing primitive `Display3.Rect3D` is used.

As long as the key is pressed, the handler queries which key is pressed and executes the command contained in the *Cmd* attribute. Behind the scene, `ExecuteAttr` sends an `Objects.AttrMsg` to retrieve the attribute. Then, the Button must again redraw itself to appear as a popped up Button. Since nothing can have changed in the display space in the mean time, the same mask is used. Apart fom this, the handler dutifully forwards all other messages to the Button handler which is equipped with all the remaining functionality required.

Display masks

The procedure `Gadgets.MakeMask` performs the calculation of the display mask `Q` through which the gadget must draw itself. The mask generation is hidden from gadget programmers, but its mechanism deserves a special description. The imaging model of Gadgets is based on rectangles and drawing is performed with masks. As the calculation of a display mask can be an expensive process, the Gadgets framework adopts a demand–driven approach for generating masks. Each and every visual gadget has its own display mask cached in its mask field: it specifies which parts of the gadget are visible, but at any one instant, a gadget has either a valid display mask, or it it has no mask at all. During editing operations in the display space, the visible part to draw may change due to new gadgets overlapping the gadget. When a modification is applied to a gadget's shape, its mask is dropped by a process called *invalidating*. Only when a gadget wants to draw itself, it requests a new display mask by calling Gadgets.MakeMask to activate the mask generation process. This strategy means that a gadget can operate for long periods of time without a valid mask, at least until it wants to draw something on the display.

Mask calculation is a service a container provides for its children. A call to to this service sometimes finds a gadget without a valid mask. In that case, a Display3.UpdateMaskMsg, with the destination set to the gadget frame identified by the F field is broadcast into the display space.

```
UpdateMaskMsg = RECORD (Display.FrameMsg)
END;
```

All containers monitor the message, checking if it is addressed to one of their children. Should one of their own children be involved, its mask is calculated. Finally, a Display3.OverlapMsg is sent directly to the child, informing it that its new mask has been calculated. In the example, it is returned in Q.

```
OverlapMsg = RECORD (Display.FrameMsg)
    M: Mask;        (** Use NIL to indicate to a frame that its current mask is
invalid. *)
END;
```

To prevent numerous UpdateMaskMsg from being broadcast when many children have invalid masks, a container automatically recalculates all invalid masks of its children when the first UpdateMaskMsg arrives. This is under the assumption that if one child requires a mask, the others will do as well in the near future.

In our example, we are constructing a elementary visual gadget and therefore the existence of the two messages in question remains concealed in the Gadgets.MakeMask procedure. However, the two messages play a role in the construction of a container gadget.

5.11.4 The model gadget design pattern (interface inheritance)

Interface inheritance involves extending an existing type with new instance variables. In this section, we take a look at programming simple objects, typically model gadgets. Although the programming of model gadgets is not so spectacular as programming visual gadgets, the knowledge gained from programming them is directly applicable to programming visual objects. It is thus important to understand what follows.

Here is an example of a model gadget Reminder holding a text of up to 128 characters that may be attached to a document. When a document having such an *attachment* is opened in a viewer, the Reminder text is presented in the system log. Reminder is a direct extension of Gadgets.Object with a single instance variable msg Creating the generator and the Init procedures is straightforward, but the handler which recognizes three classical object messages and a yet unencountered message type is more elaborate than the handler described earlier.

The AttrMsg handling is reduced to its minimum, just like in the previous example. No provision is made for getting, setting or enumerating the msg field. Consequently, this field cannot be inspected or manipulated with Columbus: it is set only by the Init procedure when the gadget is instantiated. It is interesting to meet here a rare case where a gadget hides instance variables from the outside world by not "advertizing" them through the enumeration procedure. The `CopyMsg` handling creates a new object instance and copies the handler and all the instance variables to the new object. The `FileMsg` handling stores all the instance variables to a file or loads them from a file. The rest of the serialization/deserialization of the object is done by the handler of the base object. The last message processed by the handler is defined in module `Gadgets` and was created specially for supporting attachments:

```
CmdMsg = RECORD (Objects.ObjMsg)
   cmd: ARRAY 128 OF CHAR;
      (* Information to be passed, command to be executed; result returned. *)
   res: INTEGER (* result code *)
END;
```

When a document is opened, the string "PREPARE" is assigned to the `cmd` field of a `CmdMsg` which is then broadcast to all the objects linked to the document. If one of these objects happens to be a `Reminder`, the object's handler sends the content of the `msg` field to the log as specified earlier. This will be explained in the section on programming documents.

```
   MODULE Reminders; (* jm 1.11.95 *)

   (** Example of an object attached to a document. The object remembers
   a message that is displayed in the log when the document is opened. *)

   IMPORT Attributes, Documents, Files, Gadgets, Oberon, Objects, Out;

   TYPE
      Reminder* = POINTER TO ReminderDesc;
      ReminderDesc* = RECORD (Gadgets.ObjDesc)
         msg*: ARRAY 128 OF CHAR
      END;

   PROCEDURE Copy*(VAR M: Objects.CopyMsg; from, to: Reminder);
   BEGIN
      Gadgets.CopyObject(M, from, to);
      to.msg := from.msg
   END Copy;

   PROCEDURE Handler* (obj: Objects.Object; VAR M: Objects.ObjMsg);
      VAR obj0: Reminder;
   BEGIN
      WITH obj: Reminder DO
         IF M IS Objects.AttrMsg THEN
            WITH M: Objects.AttrMsg DO
               IF (M.id = Objects.get) & (M.name = "Gen") THEN
                  M.class := Objects.String; COPY("Reminders.New", M.s); M.res := 0
               ELSE Gadgets.objecthandle(obj, M)
               END
            END
         ELSIF M IS Objects.CopyMsg THEN
            WITH M: Objects.CopyMsg DO
               IF M.stamp = obj.stamp THEN M.obj := obj.dlink   (* copy msg arrives
again *)
               ELSE (* first time copy message arrives *)
                  NEW(obj0); obj.stamp := M.stamp; obj.dlink := obj0; Copy(M, obj,
obj0);
                  M.obj := obj0
               END
            END
         ELSIF M IS Objects.FileMsg THEN
            WITH M: Objects.FileMsg DO
               IF M.id = Objects.store THEN Files.WriteString(M.R, obj.msg)
               ELSIF M.id = Objects.load THEN Files.ReadString(M.R, obj.msg)
               END;
               Gadgets.objecthandle(obj, M)
            END
         ELSIF M IS Gadgets.CmdMsg THEN (* executed when the document is opened
*)
            WITH M: Gadgets.CmdMsg DO
```

```
                IF M.cmd = "PREPARE" THEN
                    Out.String("Reminder: "); Out.String(obj.msg); Out.Ln
                ELSE Gadgets.objecthandle(obj, M)
                END
            END
        ELSE Gadgets.objecthandle(obj, M)
        END
    END
END Handler;

PROCEDURE Init* (obj: Reminder; msg: ARRAY OF CHAR);
BEGIN obj.handle := Handler; COPY(msg, obj.msg)
END Init;

PROCEDURE New*;
    VAR obj: Reminder;
BEGIN NEW(obj); Init(obj, ""); Objects.NewObj := obj
END New;

(** Attach a reminder to a document. *)
PROCEDURE Attach*;
    VAR D: Documents.Document; R: Attributes.Reader; ch: CHAR; obj: Reminder;
            s: ARRAY 128 OF CHAR; i: INTEGER; M: Objects.LinkMsg;
BEGIN
    D := Documents.MarkedDoc();
    IF D # NIL THEN
        Attributes.OpenReader(R, Oberon.Par.text, Oberon.Par.pos);
        Attributes.Read(R, ch);
        i := 0;
        WHILE ~R.eot & (ch # "~") & (i < LEN(s) - 1) DO
            s[i] := ch; INC(i);
            Attributes.Read(R, ch)
        END;
        s[i] := 0X;
        NEW(obj); Init(obj, s);
        M.id := Objects.set; M.name := "Reminder"; M.obj := obj; M.res := -1;
        D.handle(D, M);
        IF M.res >= 0 THEN Out.String("  done") ELSE Out.String("  failed") END; Out.Ln
    END
END Attach;

END Reminders.
```

Here are two examples of short messages that may be attached to a document:

```
Meeting on Thursday~
The Align button does not work~
```

The command `Reminders.Attach^` reads up to 128 characters from the selection until a "~" or end of text and assigns this string to the `msg` field of a `Reminder` gadget which is then linked to the marked document.

The message handler is a simple cascade of IF statements to determine the message type. As soon as the message type is determined, the message fields are opened up for access with the WITH statement. The receiver object is always passed as a first parameter to the message handler (named `obj` here). This message passing scheme is *quite general* and is applicable to objects that are not of type `Objects.Object`. If needed, you can create a new object type and a message hierarchy independent of the Gadget system. What you gain by extending your objects from `Objects.Object` is compatibility with the Gadgets system. If you do this, you also have additional responsibilities though: your object should at least respond to the object messages.

An earlier discussion introduced the problems of making a structure preserving deep copy of a part of the display space. As each object knows best how to copy itself, we are forced to forward the `CopyMsg` to each member of a complicated data structure so that that member may copy itself. As the global structure of an object data structure is not known, we have to reckon with the case when two more objects forward the same copy message to the same object. Using the message time stamp, we can uniquely identify if a copy message arrives twice at the same object. This is the goal of the standard handling of the copy message. The first time the copy message arrives, we make a copy of the object, remember the time stamp, and cache a reference to the copy returned in the `dlink` field of the object. If the time stamp indicates that the copy message arrives a second time (or more), we simply return a reference to the copy we made earlier. This behavior is standard for all objects, so that the IF statement executed on receiving the copy message never changes. Instead, we define a copy procedure to copy the fields of the type extension we introduce ourselves. As suspected, the call to `Gadgets.CopyObject` copies the fields of the `Gadgets.Object`.

To conclude the discussion of the example, we may add that typical gadgets introduce new messages (implementation step 3) for communicating with them. These are often defined in the module where the gadget type is defined, although there is no strong reason to do so. In this example, the message is defined in module `Gadgets`.

5.11.5 The visual gadget design pattern (interface inheritance)

The next example demonstrates the stepwise implementation of a visual gadget, a direct extension of `Gadgets.Frame`. We may categorize this new construction as "atomic" gadget in opposition to a visual container gadget.

Creating a new visual gadget as extension of an existing gadget type:

```
TYPE
   Frame* = POINTER TO FrameDesc;
   FrameDesc* = RECORD (Gadgets.FrameDesc)
      col*: INTEGER    (* new instance variable *)
   END;
```

Creating a generator and an initialization procedure:

```
(* needed to make further extensions of the gadget *)
PROCEDURE Init* (F: Frame);
BEGIN
   F.W := 50; F.H := 50;    (* only need to specify the width and the height *)
   F.col := 1;        (* initialize instance variable *)
   F.handle := Handle    (* install message handler *)
END Init;

PROCEDURE New*;
   VAR F: Frame;
BEGIN
   NEW(F); Init(F);
   Objects.NewObj := F    (* global variable to return the generated gadget *)
END New;
```

Creating a message handler for frame messages and object messages:

```
PROCEDURE Handle* (F: Objects.Object; VAR M: Objects.ObjMsg);
   VAR x, y, w, h: INTEGER;
BEGIN
   WITH F: Frame DO
      IF M IS Display.FrameMsg THEN
         WITH M: Display.FrameMsg DO
            IF (M.F = NIL) OR (M.F = F) THEN (* message addressed to this frame *)
               (* calculate the gadget's absolute display coordinates *)
               x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;

               (* test and handle the frame message – Later on we shall see
                  that the message may addressed to a particular device. *)
            ELSE    (* message not for this frame, forward it. *)
            END
         END
      ELSIF M IS Objects.AttrMsg THEN    ...    (* handle object messages *)
      ELSIF M IS Objects.CopyMsg THEN    ...
      ELSIF M IS Objects.FileMsg THEN    ...
      ELSE Gadgets.framehandle(F, M)        (* delegate to default handler *)
      END
   END
```

```
    END Handle;
```

We save unnecessary work by testing the destination frame F as soon as possible for a true broadcast or a directed send to this gadget.

Adding an `Objects.FileMsg` handling:

```
        ELSIF M IS Objects.FileMsg THEN
            WITH M: Objects.FileMsg DO
                IF M.id = Objects.store THEN Files.WriteInt(M.R, F.col)
                ELSIF M.id = Objects.load THEN Files.ReadInt(M.R, F.col)
                END;
                Gadgets.framehandle(F, M)
            END
```

`Gadgets.framehandle` is called to store the fields of the base type. It is advisable to write a version code and to check it again on loading. This version code determines the data format that follows. The reason for this is quite simple. It is seldom that a gadget is finished; continual tweaking of the functionality eventually leads to a change of file format even though the gadget has been released long ago. To prevent the gadgets stored in the old format from becoming unloadable, we can always ensure that a gadget reads all the formats that it once wrote in its lifetime, but always writes the latest and actual format. The following example illustrates the principle:

```
      CONST
          VersionNo = 2; ModName = "Skeleton";

          ....
          ELSIF M IS Objects.FileMsg THEN
              WITH M: Objects.FileMsg DO
                  IF M.id = Objects.store THEN
                      Files.WriteNum(R, VersionNo);
                      Files.WriteInt(M.R, F.col)
                  ELSIF M.id = Objects.load THEN
                      Files.ReadNum(M.R, ver);
                      IF ver = VersionNo THEN
                          Files.ReadInt(M.R, F.col)
                      ELSIF ver = 1 THEN
                          ...
                      ELSE
                          Texts.WriteString(W, "Version "); Texts.WriteInt(W, VersionNo, 3);
                          Texts.WriteString(W, " of ");
                          Texts.WriteString(W, ModName);
                          Texts.WriteString(W, " cannot read version "); Texts.WriteInt(W, ver,
3);
                          Texts.WriteLn(W);
                          Texts.Append(Oberon.Log, W.buf);
                          HALT(99)
                      END
                  END;
                  Gadgets.framehandle(F, M)
              END
```

Copying a gadget:

```
    PROCEDURE Copy* (VAR M: Objects.CopyMsg; from, to: Frame);
    BEGIN
        to.col := from.col;    (* copy own instance variables *)
        Gadgets.CopyFrame(M, from, to)    (* copy base variables *)
    END Copy;
```

called from within the message handler (this code fragment is standard):

```
        VAR F1: Frame;
        ...
        ELSIF M IS Objects.CopyMsg THEN
            WITH M: Objects.CopyMsg DO
                IF M.stamp = F.stamp THEN M.obj := F.dlink (*non-first arrival*)
                ELSE (*first arrival*)
                NEW(F1); F.stamp := M.stamp; F.dlink := F1;
                Copy(M, F, F1); M.obj := F1
            END
        END
```

The gadget visualization part is also a standard code fragment:

```
    PROCEDURE Handle* (F: Objects.Object; VAR M: Objects.ObjMsg);
    VAR x, y, w, h: INTEGER; Q: Display3.Mask;
    BEGIN
        WITH F: Frame DO
            IF M IS Display.FrameMsg THEN
                WITH M: Display.FrameMsg DO
                    IF (M.F = NIL) OR (M.F = F) THEN (* message addressed to this frame *)
                        x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;

                        IF M IS Display.DisplayMsg THEN
                            WITH M: Display.DisplayMsg DO
                              IF M.device = Display.screen THEN
                                IF (M.id = Display.full) OR (M.F = NIL) THEN
                                   Gadgets.MakeMask(F, x, y, M.dlink, Q);    (* create display mask
*)
                                  Restore(F, Q, x, y, w, h)
                                ELSIF M.id = Display.area THEN
                                   Gadgets.MakeMask(F, x, y, M.dlink, Q);    (* create display mask
*)
                                   Display3.AdjustMask(Q, x+M.u, y+h-1 + M.v, M.w, M.h); (* clip
mask *)
                                  Restore(F, Q, x, y, w, h)
                                  END
                                ELSIF M.device = Display.printer THEN Print(F, M)
                              END
                            END
```

The procedure Gadgets.MakeMask does the calculation of the display mask Q through which a gadget must draw itself. MakeMask is passed the absolute coordinates and the message thread, hence the visual gadget can request this calculation on receiving any frame message. When M.id is set to full, the entire frame is restored. Otherwise, the rectangular area u, v, w, h inside the destination frame is is to be redrawn and the mask is adjusted by clipping it.

Restore is defined as:

```
    PROCEDURE Restore (F: Frame; Q: Display3.Mask; x, y, w, h: INTEGER);
        (* x and y are the absolute display coordinates of the mask *)
    BEGIN
        (* use clipped primitives in Display3 to draw the gadget *)
        Display3.ReplConst(Q, F.col, x, y, w, h, Display.replace);

        (* standard selection style *)
        IF Gadgets.selected IN F.state THEN
            Display3.FillPattern(Q, Display3.white, Display3.selectpat,
                x, y, x, y, w, h, Display.paint)
        END
    END Restore;
```

The drawing primitive Display3.ReplConst draws the gadget through the mask calculated earlier. When the selected flag is contained in the F.state field, the gadget shows itself in a white semi-translucent pattern. The attribute handling is also a standard code pattern:

```
        ...
        ELSIF M IS Objects.AttrMsg THEN Attributes(F, M(Objects.AttrMsg))
```

with Attributes defined as:

```
    PROCEDURE Attributes (F: Frame; VAR M: Objects.AttrMsg);
    BEGIN
        IF M.id = Objects.get THEN    (* retrieve an attribute *)
            IF M.name = "Gen" THEN    (* generator attribute *)
                M.class := Objects.String;
            COPY("Skeleton.New", M.s);    (* must cite the generator procedure ! *)
            M.res := 0
        ELSIF M.name = "Color" THEN
            M.class := Objects.Int; M.i := F.col; M.res := 0
        ELSIF M.name = "Cmd" THEN
            Gadgets.framehandle(F, M);
            IF M.res < 0 THEN (* no such attribute, simulate empty string attribute *)
                M.class := Objects.String; M.s := ""; M.res := 0
            END
        ELSE Gadgets.framehandle(F, M)
        END
    ELSIF M.id = Objects.set THEN    (* set an attribute *)
        IF (M.name = "Color") & (M.class = Objects.Int) THEN
            F.col := SHORT(M.i); M.res := 0
        ELSE Gadgets.framehandle(F, M)
        END
    ELSIF M.id = Objects.enum THEN    (* To advertize attributes to Columbus *)
        M.Enum("Color"); M.Enum("Cmd");
        Gadgets.framehandle(F, M)
    END
END Attributes;
```

It is mandatory to handle of the attribute classes and to set the `res`field correctly.

The code pieces above have been put together for inspection and for further experimentation in `Skeleton.Mod`This module contains some further typical code patterns for handling mouse input and printing.

```
MODULE Skeleton (*JM/ JG 26.7.94*);
IMPORT Display, Display3, Effects, Files, Gadgets, Oberon, Objects, Printer, Printer3;
CONST red = 1; middle = 1;

TYPE
    Frame* = POINTER TO FrameDesc;
    FrameDesc* = RECORD (Gadgets.FrameDesc)
        col*: INTEGER
    END;

(* To save memory we can use the framehandler to allocate the "Cmd" attribute
    only when one exists. We have however to simulate an attribute if
    none really exists (see the handling of "Cmd" in the "get" part below)
*)
PROCEDURE Attributes (F: Frame; VAR M: Objects.AttrMsg);
BEGIN
    IF M.id = Objects.get THEN
        IF M.name = "Gen" THEN
            M.class := Objects.String; COPY("Skeleton.New", M.s); M.res := 0
        ELSIF M.name = "Color" THEN
            M.class := Objects.Int; M.i := F.col; M.res := 0
        ELSIF M.name = "Cmd" THEN
            Gadgets.framehandle(F, M);
            IF M.res < 0 THEN (* no such attribute, simulate one *)
                M.class := Objects.String; M.s := ""; M.res := 0
            END
        ELSE Gadgets.framehandle(F, M)
        END
    ELSIF M.id = Objects.set THEN
        IF (M.name = "Color") & (M.class = Objects.Int) THEN
            F.col := SHORT(M.i); M.res := 0
        ELSE Gadgets.framehandle(F, M)
        END
    ELSIF M.id = Objects.enum THEN
        M.Enum("Color"); M.Enum("Cmd"); Gadgets.framehandle(F, M)
    END
END Attributes;

PROCEDURE Restore (F: Frame; Q: Display3.Mask; x, y, w, h: INTEGER);
BEGIN
    Display3.ReplConst(Q, F.col, x, y, w, h, Display.replace);
    IF Gadgets.selected IN F.state THEN
        Display3.FillPattern(Q, Display3.white, Display3.selectpat,
            x, y, x, y, w, h, Display.paint)
    END
END Restore;

PROCEDURE Print (F: Frame; VAR M: Display.DisplayMsg);
    VAR Q: Display3.Mask;

    PROCEDURE P (x: INTEGER): INTEGER;
    BEGIN RETURN SHORT(x * LONG(10000) DIV Printer.Unit)
    END P;

BEGIN
    Gadgets.MakePrinterMask(F, M.x, M.y, M.dlink, Q);
    Printer3.ReplConst(Q, F.col, M.x, M.y, P(F.W), P(F.H), Display.replace)
END Print;

PROCEDURE Copy* (VAR M: Objects.CopyMsg; from, to: Frame);
BEGIN to.col := from.col; Gadgets.CopyFrame(M, from, to)
END Copy;

PROCEDURE Handle* (F: Objects.Object; VAR M: Objects.ObjMsg);
    VAR x, y, w, h: INTEGER; F1: Frame; Q: Display3.Mask; keysum: SET;
BEGIN
    WITH F: Frame DO
        IF M IS Display.FrameMsg THEN
            WITH M: Display.FrameMsg DO
                IF (M.F = NIL) OR (M.F = F) THEN (* message addressed to this frame *)
                    x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
                    IF M IS Display.DisplayMsg THEN
                        WITH M: Display.DisplayMsg DO
                            IF M.device = Display.screen THEN
                                IF (M.id = Display.full) OR (M.F = NIL) THEN
                                    Gadgets.MakeMask(F, x, y, M.dlink, Q);
                                    Restore(F, Q, x, y, w, h)
                                ELSIF M.id = Display.area THEN
                                    Gadgets.MakeMask(F, x, y, M.dlink, Q);
                                    Display3.AdjustMask(Q, x + M.u, y + h - 1 + M.v, M.w, M.h);
                                    Restore(F, Q, x, y, w, h)
                                END
                            ELSIF M.device = Display.printer THEN Print(F, M)
                            END
                        END
                    ELSIF M IS Oberon.InputMsg THEN
                        WITH M: Oberon.InputMsg DO
                            IF (M.id = Oberon.track) & Gadgets.InActiveArea(F, M) &
                                (M.keys = {middle}) THEN
                                Gadgets.MakeMask(F, x, y, M.dlink, Q);
                                Oberon.RemoveMarks(x, y, w, h);
                                Display3.ReplConst(Q, Display3.FG, x, y, w, h, Display.invert);
                                keysum := M.keys;
                                REPEAT
                                    Effects.TrackMouse(M.keys, M.X, M.Y, Effects.PointHand);
                                    keysum := keysum + M.keys
                                UNTIL M.keys = {};
                                Oberon.RemoveMarks(x, y, w, h);
                                Display3.ReplConst(Q, Display3.FG, x, y, w, h, Display.invert);
                                IF keysum = {middle} THEN
                                    Gadgets.ExecuteAttr(F, "Cmd", M.dlink, NIL, NIL)
                                END;
                                M.res := 0
                            ELSE Gadgets.framehandle(F, M)
                            END
                        END
                    ELSE Gadgets.framehandle(F, M)
                    END
                END
            END
        ELSIF M IS Objects.AttrMsg THEN Attributes(F, M(Objects.AttrMsg))
        ELSIF M IS Objects.CopyMsg THEN
            WITH M: Objects.CopyMsg DO
                IF M.stamp = F.stamp THEN M.obj := F.dlink (*non-first arrival*)
                ELSE (*first arrival*)
                    NEW(F1); F.stamp := M.stamp; F.dlink := F1;
                    Copy(M, F, F1); M.obj := F1
                END
            END
        ELSIF M IS Objects.FileMsg THEN
            WITH M: Objects.FileMsg DO
                IF M.id = Objects.store THEN Files.WriteInt(M.R, F.col)
                ELSIF M.id = Objects.load THEN Files.ReadInt(M.R, F.col)
                END;
                Gadgets.framehandle(F, M)
            END
        ELSE Gadgets.framehandle(F, M)
        END
    END
END Handle;

PROCEDURE Init* (F: Frame);
BEGIN F.W := 50; F.H := 50; F.col := red; F.handle := Handle
END Init;

PROCEDURE New*;
    VAR F: Frame;
BEGIN NEW(F); Init(F); Objects.NewObj := F
END New;

END Skeleton.
```

The printing primitives provided by the `Printer3` module, a twin module of `Display3`, are used in the `Print` procedure. The gadget mask calculated first by `Gadgets.MakePrinterMask`. `Printer3` uses the same mask descriptors as `Display3` but expects all coordinates to be specified in printer coordinates (often 300dpi).

5.11.6 The singleton design pattern

Both the visual and model gadgets of a user interface are stored in files on disk. The system allows you to open the same file many times and thus create multiple identical instances of the same gadgets. There are however cases where only *one instance* of a gadget is required, for example a gadget to store the time of day, or the sound track being played on an attached CD player. Such *singleton* gadgets are possible only in conjunction with *model* gadgets. In essence, each time the generator of a singleton gadget is called, the *same* gadget is returned.

```
MODULE Timer;
IMPORT BasicGadgets, Objects;

VAR time: BasicGadgets.Integer;        (* can be of any type *)

PROCEDURE Handler* (obj: Objects.Object; VAR M: Objects.ObjMsg);
BEGIN
    IF M IS Objects.AttrMsg THEN
        WITH M: Objects.AttrMsg DO
            IF (M.id = Objects.get) & (M.name = "Gen") THEN
                M.class := Objects.String; COPY("Timer.NewTimer", M.s); M.res := 0
            ELSE BasicGadgets.IntegerHandler(obj, M)
            END
        END
    ELSIF M IS Objects.CopyMsg THEN
        WITH M: Objects.CopyMsg DO M.obj := time END
    ELSE BasicGadgets.IntegerHandler(obj, M)
    END
END Handler;

PROCEDURE NewTimer*;
BEGIN (* return the only instance *)
    Objects.NewObj := time
END NewTimer;

BEGIN (* allocate gadget on loading *)
    NEW(time); BasicGadgets.InitInteger(time); time.handle := Handler
END Timer.
```

Notice how much of the behavior is inherited, by class inheritance, from the `BasicGadgets.Integer` and how the same object is returned each time a `CopyMsg` is received.

5.11.7 The container gadget design pattern

Container gadgets contain other gadgets as descendants or children. Due to the `dsc-next` connection of descendants, a child can belong only to one parent. In comparison to elementary or "leaf" gadgets, containers have additional duties:

1. forwarding of unprocessed messages to descendants,
2. monitoring some of the messages sent to the descendants,
3. generating display masks for each descendant,
4. passing the correct display coordinates to descendants and
5. ensuring that the message thread from the display root to the descendants remains intact.

A container has full control over all its descendants. This means that when we ignore the messages sent directly to a gadget (normally those defined in the `Objects` module), all messages for descendants are delegated through the parent to the children, and the parent container has the right to pass through or even modify the messages. This is the consequence of filtering the message down from the display root in a hierarchical fashion to the containers and their descendants.

The source code pieces used in the description are extracted from a complete `Portraits` module which appears at the end of this section. The module implements a container Portrait with a single component. When the gadget is instantiated, a Skeleton gadget is automatically installed as the component. The Skeleton acts as a sentinel indicating an empty Portrait. Another Skeleton may be dropped inside the Portrait, any number of times, to replace the contained Skeleton. Any visual gadget, including a transparent one, can also be dropped inside the container to replace the Skeleton, but only once. The component dropped inside or the sentinel Skeleton cannot be deleted, but either of them can be removed by moving it to another context. When this happens, a fresh sentinel Skeleton immediately fills the gap. This assumption allows the construction of a bare minimum solution requiring a source code text of reasonable size.

The message handler of a typical container is structured as follows:

```
PROCEDURE Handle* (F: Objects.Object; VAR M: Objects.ObjMsg);
  VAR x, y, w, h: INTEGER;
BEGIN
    WITH F: Portrait DO
        IF M IS Display.FrameMsg THEN
            WITH M: Display.FrameMsg DO
                IF (M.F = NIL) OR (M.F = F) THEN (* message for this gadget *)
                    x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;

                    (* test and handle the frame messages *)
                ELSE    (* message perhaps for content *)

                    IF message for a child THEN
                        ... handle message for child
                    ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)
                    END
                END
            END
        ELSIF M IS Objects.anyMsg THEN
            ... handle object messages
        END
    END
END Handle;
```

where "message for a child" determines if `M.F` (the message destination) is a child of `F` by traversing the `dsc-next` chain, and `ToContent` delegates the message to each child in turn as follows:

```
  (* x, y is the absolute display coordinates of the bottom–left corner of the container
F *)
  PROCEDURE ToContent (F: Portrait; x, y: INTEGER; VAR M: Display.FrameMsg);
  VAR Mdlink, Fdlink: Objects.Object; tx, ty: INTEGER;
  BEGIN
    tx := M.x; ty := M.y;    (* store old message origin *)
    M.x := x; M.y := y + F.H – 1;    (* update message origin *)

     F.dlink := M.dlink; M.dlink := F; F.dsc.handle(F.dsc, M);    (* store message thread
data *)
    F.dlink := M.dlink; M.dlink := F;    (* update the message thread *)

    F.dsc.handle(F.dsc, M);    (* forward message *)

    F.dlink := Fdlink; M.dlink := Mdlink;    (* restore message thread data *)
    M.x := tx; M.y := ty    (* restore old message origin *)
  END ToContent;
```

The origin of the message must be changed before it is forwarded as a frame message travels through the display space. When `ToContent` is called, x and y contain the absolute coordinates of the bottom–left corner of the container but the coordinates of the top–left corner must be forwarded, hence the addition of the frame height to the current frame position.

More efficient implementations are obtained by restructuring the handler, in particular by creating a special forwarding procedure to forward messages to all descendants. If this is not sufficient, the handling of the message thread can be removed. Although many steps are required to forward a message, this very seldom forms a bottleneck. More typically, bottlenecks are caused by the multitude of type tests required to determine the message type.

A container must calculate the masks of its descendants from its own mask. The implementation for a container containing a *single* gadget is given here:

```
  (* Inform a gadget of its new mask *)
  PROCEDURE SetMask (F: Display.Frame; Q: Display3.Mask);
  VAR M: Display3.OverlapMsg;
  BEGIN
      M.M := Q; (* set mask *)
      M.x := 0; M.y := 0; M.F := F; M.dlink := NIL; M.res := –1;
         (* initialize rest of the message fields *)
      F.handle(F, M)
  END SetMask;

  PROCEDURE SetContentMask (F: Portrait);
  VAR Q: Display3.Mask;
  BEGIN
    IF F.mask = NIL THEN    (* Portrait has no mask, so invalidate mask of the child *)
       SetMask(F.dsc, NIL)
    ELSE
       Display3.Copy(F.mask, Q);    (* make a copy of the container's mask *)
       Q.x := 0; Q.y := 0;    (* reset the mask origin *)
       Display3.Intersect(Q, F.dsc.X, F.dsc.Y, F.dsc.W, F.dsc.H);
          (* intersect with the relative coordinates of the child *)

       Q.x := –F.dsc.X; Q.y := –(F.dsc.Y + F.dsc.H – 1);
          (* move the mask origin into the coordinate system of the child *)
       Display3.Shift(Q);
          (* and translate the mask by Q.x, Q.y into the coordinate system of the child
*)
       SetMask(F.dsc, Q)
    END
  END SetContentMask;
```

In the handler, we also have to check if the descendant requires a new mask:

```
  IF M IS Display3.UpdateMaskMsg THEN
     WITH M: Display3.UpdateMaskMsg DO
        IF M.F = F.dsc THEN SetContentMask(F)
        ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)    (* don't forget to forward it *)
        END
    END
```

or if the container itself obtains a new mask:

```
IF M IS Display3.OverlapMsg THEN
    WITH M: Display3.OverlapMsg DO
        F.mask := M.M; SetContentMask(F)
    END
```

Should a container be restored onto the display, the child must be displayed accordingly. Again for a single descendant, we have:

```
PROCEDURE Restore
    (F: Portrait; Q: Display3.Mask; x, y, w, h: INTEGER; VAR M: Display.DisplayMsg);
VAR N: Display.DisplayMsg;

    PROCEDURE ClipAgainst (VAR x, y, w, h: INTEGER; x1, y1, w1, h1: INTEGER);
    VAR r, t, r1, t1: INTEGER;
    BEGIN
        r := x + w – 1; r1 := x1 + w1 – 1; t := y + h – 1; t1 := y1 + h1 – 1;
        IF x < x1 THEN x := x1 END; IF y < y1 THEN y := y1 END;
        IF r > r1 THEN r := r1 END; IF t > t1 THEN t := t1 END;
        w := r – x + 1; h := t – y + 1;
    END ClipAgainst;

BEGIN
    ... restore the display areas belonging to F

    IF M.id = Display.area THEN     (* only a rectangular area update *)
        N.F := F.dsc; N.u := M.u; N.v := M.v; N.w := M.w; N.h := M.h;
            (* in coordinate system of container *)
        ClipAgainst(N.u, N.v, N.w, N.h, F.dsc.X, F.dsc.Y, F.dsc.W, F.dsc.H);
            (* clip to child location *)
        DEC(N.u, border); INC(N.v, border)
            (* and translate to child coordinate system *)
    END;

    ... restore the background before displaying a transparent gadget
    IF Gadgets.transparent IN F.dsc(Gadgets.Frame).state THEN
    Display3.ReplConst(Q, Display3.groupC, x + border, y + border,
      w – 2*border, h – 2*border, Display.replace)
    END;
    .....
  ToContent(F, x, y, N);

   ... standard selection follows
    IF Gadgets.selected IN F.state THEN ...
        Display3.FillPattern(Q, Display3.white, Display3.selectpat, x, y, x, y, w, h,
        Display.paint)
    END
END Restore;
```

In the last two examples, a conversion of the coordinate system of the parent to that of the child takes place.

When a child is removed:

```
PROCEDURE RemoveObj (obj: Display.Frame);
VAR M: Display.ControlMsg;
BEGIN M.id := Display.remove; M.F := obj; Display.Broadcast(M)
END RemoveObj;
```

the handler must process the `Display.ControlMsg`

```
ELSIF M IS Display.ControlMsg THEN
    WITH M: Display.ControlMsg DO
        IF (M.id = Display.remove) & (M.F = F.dsc) THEN
            Skeleton.New; PutObj(F, Objects.NewObj(Display.Frame))
        ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)
        END
    END
```

In this solution, removing the child can never realy succeed: a Sekeleton immediately takes its place. This is done on purpose to simplify the solution.

The handler must detect when a new child is inserted, removing the current child first. In the suggested solution, any visual gadget which is not a Skeleton will refuse to leave its place to another. This restriction can be lifted by omitting to test the condition `F.dsc IS Skeleton.Frame`. The correct coordinate handling is missing in the following code fragment:

```
ELSIF M IS Display.ConsumeMsg THEN
   WITH M: Display.ConsumeMsg DO
      IF (M.id = Display.drop) & (M.F = F.dsc) & (F.dsc IS Skeleton.Frame) THEN
         RemoveObj(M.obj(Display.Frame));
         PutObj(F, M.obj(Display.Frame));
         M.res := 0
      ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)
      END
   END
```

A container also has to handle the `Display.LocateMsg` that determines what is located at a certain position of the display:

```
ELSIF M IS Display.LocateMsg THEN
   WITH M: Display.LocateMsg DO
      IF (M.loc = NIL) & Effects.Inside(M.X, M.Y, x, y, w, h) THEN
         ToContent(F, x, y, M);
         IF M.loc = NIL THEN    (* no descendant hit *)
             M.loc := F;
             M.u := M.X - x; M.v := M.Y - (y + h - 1);
                (* calculate the relative coordinates of the hit point *)
             M.res := 0
         END
      END
   END
```

The container may exercise parental control over mouse events signaled by an `InputMsg`. As soon as the mouse enters into the area a gadget (in this case a Skeleton) occupies on the screen, it starts to receive mouse events. It is completely up to the gadget to do whatever it pleases with these events. If the component simply delegates the handling of mouse events to the default handler `Gadgets.framehandle`. Because that handler only processes MM key events, no response is given to MR and ML key events. In that case, the parent can take control of the mouse (here MR key events) or even prevent the mouse events from arriving at a child at all. A question arises out of this, namely who is responsible for handling certain events. In effect, a division of responsability is required; an example illustrates why this is necessary. When the user selects several gadgets in a container and wants to move them around as a whole, a single child does not know about the other selected gadgets and therefore the group editing operations are the responsability of the container and not of the children. A first approximation would be for the container not to let the selected child obtain mouse events and directly take control of editing. A more refined way is to have the child defer mouse operations under certain circumstances to the parent. This level of co–operation between parent and child gives the child some additional possibilities for controlling interactions.

```
ELSIF M IS Oberon.InputMsg THEN
  WITH M: Oberon.InputMsg DO
    IF (M.id = Oberon.track) & ~(Gadgets.selected IN F.state) THEN
      IF Effects.Inside(M.X, M.Y, x + border, y + border,
        w - 2*border, h - 2*border) THEN
        ToContent(F, x, y, M);
        (* Only the MM key is handled by the default handler. *)
        IF (M.res < 0) & (M.keys = {0}) THEN
          (* No response from child, container exercises parental control
          of MR key events. *)
          TrackSelectChild(F, M, F.dsc)
        END
```

Through the procedure TrackSelectChild, the container controls the selection and deselection of the component on recognizing simple MR key clicks. An MR + MM key interclick copies the component over to the caret as usual.

The other way to copy a gadget is by selecting it first and then to copy it to the caret with an ML + MM key interclick. This mouse action causes a Display.SelectMsg to be broadcast through the display space in order to get the most recent gadget selection. This message must also be processed by the handler which has the duty to inform the sender on the selection time of the component relatively to the other objects selected present in the display space. The selection of the component must be stored in a field of the container F.time in the first place. The second thing to do is to assign the current Oberon.Time() to this field when the component is selected (but not when it is selected and copied). There remains to the handler to decide whether the component was selected more recently than all other objects visited by the SelectMsg or not.

```
     ELSIF M IS Display.SelectMsg THEN
       WITH M: Display.SelectMsg DO
         IF M.id = Display.get THEN
           ToContent(F, x, y, M);
                IF (F.time > M.time) & (Gadgets.selected IN F.dsc(Gadgets.Frame).state)
THEN
             M.time := F.time; M.obj := F.dsc ; M.sel := F
           END
         ELSE Gadgets.framehandle(F, M)
         END
       END
```

When the child is selected, the user may want to clear all selections with the Esc key. In this case, an Oberon.ControlMsg with id=neutralize is broadcast. Here is how the handler sends a reset request to the selected child:

```
     ELSIF M IS Oberon.ControlMsg THEN
       WITH M: Oberon.ControlMsg DO
         ToContent(F, x, y, M);
         IF M.id = Oberon.neutralize THEN
           IF Gadgets.selected IN F.dsc(Gadgets.Frame).state THEN
           SM.id := Display.reset; SM.F := F.dsc; SM.sel := F; SM.res := -1;
           F.dsc.handle(F.dsc, SM); Gadgets.Update(F.dsc)
           END
         END
       END
```

In addition, a container has to intercept the Display.Modify message sent to its descendants, and update itself accordingly.

As the whole process above is rather complicated to realize for containers with multiple descendants and all optimizations possible, the source text fragments have been written so that they can easily be extended with specific behavior. The NoteBook gadget is a container which may have any number of components or none at all. Seldom will you have to write new containers completely from scratch. Here follows the complete Portraits module text:

```
MODULE Portraits; (*JM/ JG 26.7.94*)
IMPORT Display, Display3, Effects, Gadgets, Oberon, Objects, Skeleton;
CONST border = 4;

TYPE
  Portrait = POINTER TO PortraitDesc;
  PortraitDesc = RECORD (Gadgets.FrameDesc)
    time*: LONGINT    (* time of selection *)
  END;

PROCEDURE SetMask (F: Display.Frame; Q: Display3.Mask);
VAR M: Display3.OverlapMsg;
BEGIN M.M := Q; M.x := 0; M.y := 0; M.F := F; M.dlink := NIL; M.res := -1;
  F.handle(F, M)
END SetMask;
```

```
PROCEDURE SetContentMask (F: Portrait);
  VAR Q: Display3.Mask;
BEGIN
  IF F.mask = NIL THEN SetMask(F.dsc, NIL)
  ELSE Display3.Copy(F.mask, Q); Q.x := 0; Q.y := 0;
    Display3.Intersect(Q, F.dsc.X, F.dsc.Y, F.dsc.W, F.dsc.H);
    Q.x := -F.dsc.X; Q.y := -(F.dsc.Y + F.dsc.H - 1); Display3.Shift(Q);
    SetMask(F.dsc, Q)
  END
END SetContentMask;

PROCEDURE ToContent (F: Portrait; x, y: INTEGER; VAR M: Display.FrameMsg);
  VAR Mdlink, Fdlink: Objects.Object; tx, ty: INTEGER;
BEGIN
  tx := M.x; ty := M.y;
  M.x := x; M.y := y + F.H - 1;
  Fdlink := F.dlink; Mdlink := M.dlink;
  F.dlink := M.dlink; M.dlink := F; F.dsc.handle(F.dsc, M);
  F.dlink := Fdlink; M.dlink := Mdlink;
  M.x := tx; M.y := ty
END ToContent;

PROCEDURE Modify (F: Portrait; VAR M: Display.ModifyMsg);
  VAR N: Display.ModifyMsg;
BEGIN
  N.id := Display.extend; N.F := F.dsc; N.mode := Display.state;
  N.X := border; N.Y := -M.H + 1 + border;
  N.W := M.W - 2 * border; N.H := M.H - 2 * border;
  N.dX := N.X - F.dsc.X; N.dY := N.Y - F.dsc.Y;
  N.dW := N.W - F.dsc.W; N.dH := N.H - F.dsc.H;
  N.x := 0; N.y := 0; N.res := -1; Objects.Stamp(N);
  F.dsc.handle(F.dsc, N);
  Gadgets.framehandle(F, M)
END Modify;

PROCEDURE ModifyContent (F: Portrait; VAR M: Display.ModifyMsg);
  VAR N: Display.ModifyMsg;
BEGIN
  IF M.stamp # F.stamp THEN F.stamp := M.stamp;
    N.id := Display.extend; N.F := F; N.mode := Display.display;
    N.X := F.X + M.dX; N.Y := F.Y + M.dY;
    N.W := M.W + 2 * border; N.H := M.H + 2 * border;
    N.dX := N.X - F.X; N.dY := N.Y - F.Y;
    N.dW := N.W - F.W; N.dH := N.H - F.H;
    Display.Broadcast(N)
  END
END ModifyContent;

PROCEDURE Restore (F: Portrait; Q: Display3.Mask; x, y, w, h: INTEGER; VAR M:
Display.DisplayMsg);
  VAR N: Display.DisplayMsg;

  PROCEDURE ClipAgainst (VAR x, y, w, h: INTEGER; x1, y1, w1, h1: INTEGER);
    VAR r, t, r1, t1: INTEGER;
  BEGIN
    r := x + w - 1; r1 := x1 + w1 - 1; t := y + h - 1; t1 := y1 + h1 - 1;
    IF x < x1 THEN x := x1 END; IF y < y1 THEN y := y1 END;
    IF r > r1 THEN r := r1 END; IF t > t1 THEN t := t1 END;
    w := r - x + 1; h := t - y + 1;
  END ClipAgainst;

BEGIN
  Display3.Rect3D(Q, Display3.topC, Display3.bottomC, x, y, w, h, 1, Display.replace);
  Display3.Rect(Q, Display3.groupC, Display.solid, x + 1, y + 1, w - 2, h - 2, border - 2,
    Display.replace);
  Display3.Rect3D(Q, Display3.bottomC, Display3.topC,
    x + border - 1, y + border - 1, w - (border - 1) * 2, h - (border - 1) * 2, 1,
    Display.replace);
  IF M.id = Display.area THEN
    N.F := F.dsc; N.u := M.u; N.v := M.v; N.w := M.w; N.h := M.h;
    ClipAgainst(N.u, N.v, N.w, N.h, F.dsc.X, F.dsc.Y, F.dsc.W, F.dsc.H);
    DEC(N.u, border); INC(N.v, border)
  END;
  IF Gadgets.transparent IN F.dsc(Gadgets.Frame).state THEN
      Display3.ReplConst(Q, Display3.groupC, x + border, y + border,
        w - 2*border, h - 2*border, Display.replace)
  END;
  N.device := M.device; N.id := M.id; N.F := F.dsc; N.dlink := M.dlink; N.res := -1;
  Objects.Stamp(N); ToContent(F, x, y, N);
  IF Gadgets.selected IN F.state THEN
    Display3.FillPattern(Q, Display3.white, Display3.selectpat, x, y, x, y, w, h,
    Display.paint)
  END
END Restore;

PROCEDURE Copy* (VAR M: Objects.CopyMsg; from, to: Portrait);
  VAR N: Objects.CopyMsg;
BEGIN
  Gadgets.CopyFrame(M, from, to);
  N.id := Objects.shallow; Objects.Stamp(N);
  from.dsc.handle(from.dsc, N); to.dsc := N.obj(Gadgets.Frame)
END Copy;

PROCEDURE Attributes (F: Portrait; VAR M: Objects.AttrMsg);
BEGIN
  IF (M.id = Objects.get) & (M.name = "Gen") THEN
    M.s := "Portraits.New"; M.class := Objects.String; M.res := 0
  ELSE Gadgets.framehandle(F, M)
  END
END Attributes;

PROCEDURE RemoveObj (obj: Display.Frame);
  VAR M: Display.ControlMsg;
BEGIN M.id := Display.remove; M.F := obj; Display.Broadcast(M)
END RemoveObj;

PROCEDURE PutObj (F: Portrait; obj: Display.Frame);
  VAR M: Display.ModifyMsg;
BEGIN
  F.dsc := obj; SetMask(F.dsc, NIL);
  M.id := Display.extend; M.mode := Display.display; M.F := F;
  M.X := F.X; M.Y := F.Y;
  M.W := F.dsc.W + border * 2; M.H := F.dsc.H + border * 2;
  M.dX := M.X - F.X; M.dY := M.Y - F.Y;
  M.dW := M.W - F.W; M.dH := M.H - F.H;
  Display.Broadcast(M)
END PutObj;

PROCEDURE TrackSelectChild (F: Portrait; VAR M: Oberon.InputMsg; child:
Display.Frame);
VAR S: Display.SelectMsg; keysum: SET; C: Objects.CopyMsg;
BEGIN
    IF Gadgets.selected IN child(Gadgets.Frame).state THEN S.id := Display.reset
    ELSE S.id := Display.set
    END;
    S.F := child; S.sel := F; S.res := -1; Display.Broadcast(S);
    Gadgets.Update(child);
    keysum := {};
    REPEAT
        Effects.TrackMouse(M.keys, M.X, M.Y, Effects.Arrow); keysum := keysum + M.keys;
    UNTIL M.keys = {};
    IF (keysum = {0, 1}) & (S.id = Display.set) THEN  (* MR copy to focus *)
        Objects.Stamp(C);
        C.id := Objects.shallow; C.obj := NIL; child.handle(child, C);
        IF C.obj # NIL THEN Gadgets.Integrate(C.obj) END
    ELSE F.time := Oberon.Time()
    END;
    M.res := 0
END TrackSelectChild;

PROCEDURE Handle* (F: Objects.Object; VAR M: Objects.ObjMsg);
  VAR x, y, w, h: INTEGER; F1: Portrait; Q: Display3.Mask; obj: Objects.Object;
      SM: Display.SelectMsg;
BEGIN
  WITH F: Portrait DO
    IF M IS Display.FrameMsg THEN
      WITH M: Display.FrameMsg DO
    IF (M.F = NIL) OR (M.F = F) THEN
      x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
      IF M IS Display.DisplayMsg THEN
        WITH M: Display.DisplayMsg DO
          IF M.device = Display.screen THEN
            IF (M.id = Display.full) OR (M.F = NIL) THEN
              Gadgets.MakeMask(F, x, y, M.dlink, Q);
              Restore(F, Q, x, y, w, h, M)
            ELSIF M.id = Display.area THEN
              Gadgets.MakeMask(F, x, y, M.dlink, Q);
              Display3.AdjustMask(Q, x + M.u, y + h − 1 + M.v, M.w, M.h);
              Restore(F, Q, x, y, w, h, M)
            END
          ELSIF M.device = Display.printer THEN
          END
        END
      ELSIF M IS Oberon.InputMsg THEN
        WITH M: Oberon.InputMsg DO
          IF (M.id = Oberon.track) & ~(Gadgets.selected IN F.state) THEN
            IF Effects.Inside(M.X, M.Y, x + border, y + border,
              w − 2*border, h − 2*border) THEN
              ToContent(F, x, y, M);
              (* If child does not respond, the container may exercise parental control
                of mouse events. In this case, of MR key events.  *)
              IF (M.res < 0) & (M.keys = {0}) THEN
                TrackSelectChild(F, M, F.dsc)
              END
            ELSE Gadgets.framehandle(F, M)
            END
          ELSE Gadgets.framehandle(F, M)
          END
        END
      ELSIF M IS Oberon.ControlMsg THEN
        WITH M: Oberon.ControlMsg DO
          ToContent(F, x, y, M);
          IF M.id = Oberon.neutralize THEN
            IF Gadgets.selected IN F.dsc(Gadgets.Frame).state THEN
            SM.id := Display.reset; SM.F := F.dsc; SM.sel := F; SM.res := −1;
            F.dsc.handle(F.dsc, SM); Gadgets.Update(F.dsc)
            END
          END
        END
      ELSIF M IS Display.ModifyMsg THEN Modify(F, M(Display.ModifyMsg))
      ELSIF M IS Display.LocateMsg THEN
        WITH M: Display.LocateMsg DO
          IF (M.loc = NIL) & Effects.Inside(M.X, M.Y, x, y, w, h) THEN
            ToContent(F, x, y, M);
            IF M.loc = NIL THEN
              M.loc := F; M.u := M.X − x; M.v := M.Y − (y + h − 1); M.res := 0
            END
          END
        END
      ELSIF M IS Display3.OverlapMsg THEN
        WITH M: Display3.OverlapMsg DO
          F.mask := M.M; SetContentMask(F)
        END
      ELSIF M IS Display.SelectMsg THEN
        WITH M: Display.SelectMsg DO
          IF M.id = Display.get THEN
            ToContent(F, x, y, M);
                   IF  (F.time  >  M.time)  &  (Gadgets.selected  IN  F.dsc(Gadgets.Frame).state)
THEN
              M.time := F.time; M.obj := F.dsc ; M.sel := F
            END
          ELSE Gadgets.framehandle(F, M)
          END
        END
      ELSIF M.F # NIL THEN Gadgets.framehandle(F, M)
      ELSE ToContent(F, x, y, M)
      END
    ELSE (* message perhaps for content *)
      IF M IS Display3.UpdateMaskMsg THEN
        WITH M: Display3.UpdateMaskMsg DO
          IF M.F = F.dsc THEN SetContentMask(F)
          ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)
          END
        END
      ELSIF M IS Display.ControlMsg THEN
        WITH M: Display.ControlMsg DO
          IF (M.id = Display.remove) & (M.F = F.dsc) THEN
            Skeleton.New; PutObj(F, Objects.NewObj(Display.Frame))
          ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)
          END
        END
      ELSIF M IS Display.ModifyMsg THEN
        IF M.F = F.dsc THEN ModifyContent(F, M(Display.ModifyMsg))
        ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)
        END
      ELSIF M IS Display.ConsumeMsg THEN
        WITH M: Display.ConsumeMsg DO
          IF (M.id = Display.drop) & (M.F = F.dsc) & (F.dsc IS Skeleton.Frame) THEN
            RemoveObj(M.obj(Display.Frame));
            PutObj(F, M.obj(Display.Frame));
            M.res := 0
          ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)
          END
        END
      ELSE ToContent(F, M.x + F.X, M.y + F.Y, M)
      END
    END
  END
  ELSIF M IS Objects.AttrMsg THEN Attributes(F, M(Objects.AttrMsg))
  ELSIF M IS Objects.BindMsg THEN
    F.dsc.handle(F.dsc, M); Gadgets.framehandle(F, M)
  ELSIF M IS Objects.CopyMsg THEN
    WITH M: Objects.CopyMsg DO
      IF M.stamp = F.stamp THEN M.obj := F.dlink (*non–first arrival*)
      ELSE (*first arrival*)
        NEW(F1); F.stamp := M.stamp; F.dlink := F1; Copy(M, F, F1); M.obj := F1
      END
    END
  ELSIF M IS Objects.FileMsg THEN
    WITH M: Objects.FileMsg DO
      IF M.id = Objects.store THEN
        Gadgets.WriteRef(M.R, F.lib, F.dsc)
      ELSIF M.id = Objects.load THEN
        Gadgets.ReadRef(M.R, F.lib, obj);
        IF (obj # NIL) & (obj IS Gadgets.Frame) THEN F.dsc := obj(Gadgets.Frame)
        ELSE Skeleton.New; F.dsc := Objects.NewObj(Gadgets.Frame)
        END
      END;
      Gadgets.framehandle(F, M)
    END
  ELSE Gadgets.framehandle(F, M)
  END
 END
END Handle;

PROCEDURE New*;
 VAR F: Portrait;
BEGIN
 NEW(F); F.handle := Handle; F.W := 50; F.H := 50;
 Skeleton.New; F.dsc := Objects.NewObj(Display.Frame);
 Objects.NewObj := F
END New;

END Portraits.
```

5.11.8 The document design pattern

A document is a container gadget with a single descendant. A TextDoc contains a TextGadget, a PanelDoc contains a Panel, a RembrandtDoc contains a RembrandtFrame etc. The document gadget provides a file name, a storage mechanism, an icon, and has the capability of generating a menu bar. Programming a new document normally involves class inheritance, that is, the document handler is exchanged and no type extension is made. A document has a generator procedure to generate an empty instance of that document type. In addition, two methods for loading and storing, implement the persistency mechanism. These two methods should not be confused with the FileMsg, which in this case stores nothing more than the document name, an attribute of the document.

A document can either be stored locally on disk, be present on remote machines, or be generated on the fly at load time. Locally stored documents often use the newer Oberon document format, which prepends a standard header to the document file:

```
0F7X  7X  "Generator string"  X Y W H   (document content follows)
```

This header contains the generator of the loading document instance. The X Y W H fields are used for determining a preferred position and size on the display, and can be copied to the document coordinates. This is visible in the Load and Store methods defined below.

Compatibility with documents in the existing document formats is obtained by defining a lookup table consisting of (generator, file extension) pairs. Remote documents are identified using uniform resource locators (URL) known from the world–wide web. The appropriate document generators are defined in a lookup table consisting of (generator, URL format) pairs, where URL format is "http", "ftp" etc. The tables of (generator, file extension) and (generator, URL format) pairs are managed by the Documents module. For testing purposes, the tables can be extended by adding entries to the file Oberon.Text for Native Oberon or to the Registry for the other Oberon implementations. This file is parsed once the Documents module is loaded.

The DocumentSkeleton module implements a trivial example of document containing a Panel, only the color of which is stored. The meaning of the piece of code in bold typeface is explained later.

```
MODULE DocumentSkeleton; (* jm 25.10.93 *)
IMPORT Attributes, Desktops, Display, Documents, Files, Gadgets,
            Links, Oberon, Objects, Texts, ColorDriver;
CONST Menu = "Desktops.StoreDoc[Store] DocumentSkeleton.Cycle[Cycle]";

VAR W: Texts.Writer;

PROCEDURE Cycle*;
   VAR doc: Documents.Document; F: Gadgets.Frame; col: LONGINT;
BEGIN
   doc := Desktops.CurDoc(Gadgets.context);
   IF (doc # NIL) & (doc.dsc IS Gadgets.Frame) THEN
      F := doc.dsc(Gadgets.Frame);
      Attributes.GetInt(F, "Color", col);
      Attributes.SetInt(F, "Color", (col + 1) MOD 4);
      Gadgets.Update(F)
   END
END Cycle;

PROCEDURE NextColor (doc: Documents.Document; col: INTEGER);
   VAR F: Gadgets.Frame;
BEGIN
   F := doc.dsc(Gadgets.Frame);
   Attributes.SetInt(F, "Color", col);
   Gadgets.Update(F)
END NextColor;

PROCEDURE Load (D: Documents.Document);
   VAR
   obj: Objects.Object;
   tag, x, y, w, h, col: INTEGER;
   name: ARRAY 64 OF CHAR; F: Files.File; R: Files.Rider; ch: CHAR;
   CM: Gadgets.CmdMsg;
BEGIN
   obj := Gadgets.CreateObject("Panels.NewPanel");
   WITH obj: Gadgets.Frame DO
      x := 0; y := 0; w := 250; h := 200; col := 1; (* default *)
```

```
            F := Files.Old(D.name);
            IF F # NIL THEN
                Files.Set(R, F, 0);
                Files.ReadInt(R, tag);
                IF tag = Documents.Id THEN
                    Files.ReadString(R, name);
                    Files.ReadInt(R, x); Files.ReadInt(R, y); Files.ReadInt(R, w); Files.ReadInt(R,
h);

                    Files.Read(R, ch);
                    IF ch # 0F7X THEN (* attachments *)
                        Documents.LoadAttachments(R, D.attr, D.link);
                        IF D.link # NIL THEN
                            CM.cmd   :=   "PREPARE";   CM.res   :=   -1;   CM.dlink   :=   D;
Objects.Stamp(CM);
                            Links.Broadcast(D.link, CM)
                        END;
                        Files.ReadInt(R, col)
                    END
                END
            ELSE (* COPY("DefaultName", D.name) *)
            END;
            D.X := x; D.Y := y; D.W := w; D.H := h;
            Attributes.SetInt(obj, "Color", col);
            Documents.Init(D, obj)
        END
    END Load;

    PROCEDURE Store (D: Documents.Document);
        VAR obj: Gadgets.Frame; F: Files.File; R: Files.Rider; col: LONGINT;
    BEGIN
        obj := D.dsc(Gadgets.Frame);
        Texts.WriteString(W, "Store "); Texts.Append(Oberon.Log, W.buf);
        IF D.name # "" THEN
            F := Files.New(D.name);
            IF F = NIL THEN HALT(99) END;

            Files.Set(R, F, 0);
            Files.WriteInt(R,         Documents.Id);         Files.WriteString(R,
"DocumentSkeleton.NewDoc");
            Files.WriteInt(R, D.X); Files.WriteInt(R, D.Y);
            Files.WriteInt(R, D.W); Files.WriteInt(R, D.H);

            IF (D.attr # NIL) OR (D.link # NIL) THEN (* attachments *)
                Documents.StoreAttachments(R, D.attr, D.link)
            END;
            Attributes.GetInt(obj, "Color", col);
            Files.WriteInt(R, SHORT(col));
            Files.Register(F);
            Texts.Write(W, 22X); Texts.WriteString(W, D.name); Texts.Write(W, 22X)
        ELSE Texts.WriteString(W, "[Untitled document]")
        END;
        Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
    END Store;

    PROCEDURE Handler (D: Objects.Object; VAR M: Objects.ObjMsg);
    BEGIN
        WITH D: Documents.Document DO
            IF M IS Objects.AttrMsg THEN
                WITH M: Objects.AttrMsg DO
                    IF M.id = Objects.get THEN
                        IF M.name = "Gen" THEN
                            M.class := Objects.String; M.s := "DocumentSkeleton.NewDoc"; M.res
:= 0
                        ELSIF M.name = "Adaptive" THEN
                            M.class := Objects.Bool; M.b := FALSE; M.res := 0
                        ELSIF M.name = "Icon" THEN
                            M.class := Objects.String; M.s := "Icons.Tool"; M.res := 0
                        ELSE Documents.Handler(D, M)
                        END
                    ELSE Documents.Handler(D, M)
                    END
                END
```

```
      ELSIF M IS Objects.LinkMsg THEN
         WITH M: Objects.LinkMsg DO
            IF (M.id = Objects.get) & (M.name = "DeskMenu") THEN
               M.obj := Gadgets.CopyPublicObject("TestMenus.DeskMenu", TRUE);
               IF M.obj = NIL THEN M.obj := Desktops.NewMenu(Menu) END;
               M.res := 0
            ELSIF (M.id = Objects.get) & (M.name = "SystemMenu") THEN
               M.obj := Gadgets.CopyPublicObject("TestMenus.DeskMenu", TRUE);
               IF M.obj = NIL THEN M.obj := Desktops.NewMenu(Menu) END;
               M.res := 0
            ELSIF (M.id = Objects.get) & (M.name = "UserMenu") THEN
               M.obj := Gadgets.CopyPublicObject("TestMenus.DeskMenu", TRUE);
               IF M.obj = NIL THEN M.obj := Desktops.NewMenu(Menu) END;
               M.res := 0
            ELSE Documents.Handler(D, M)
            END
         END
      ELSIF M IS ColorDriver.ColorMsg THEN
         NextColor(D, M(ColorDriver.ColorMsg).col)
      ELSIF M IS Display.DisplayMsg THEN
         WITH M: Display.DisplayMsg DO
            IF (M.device = Display.printer) & (M.id = Display.contents) & (D.dsc # NIL)
THEN
               (* print *)
            ELSE Documents.Handler(D, M)
            END
         END
      ELSE Documents.Handler(D, M)
      END
   END
END Handler;

PROCEDURE NewDoc*;
   VAR D: Documents.Document;
BEGIN
   NEW(D); D.Load := Load; D.Store := Store; D.handle := Handler;
   D.W := 250; D.H := 200; Objects.NewObj := D
END NewDoc;

BEGIN Texts.OpenWriter(W)
END DocumentSkeleton.
```

```
Desktops.OpenDoc (DocumentSkeleton.NewDoc)
```

The generator procedure generates an empty instance of `Documents.Document` and installs the load and store methods and the handler which are all typical in document modules. The document possesses two "read–only" attributes *Adaptive* and *Icon*. The *Adaptive* attribute set to FALSE in the present case indicates that a fixed size camera–view of the document will be presented. When the value TRUE is returned, the document will adapt its size to that of the viewer. Also when the viewer is resized, the document is resized. Normally, a TextDoc or a LogDoc are adaptive, whereas a PanelDoc or a Columbus document are not adaptive. The *Icon* attribute indicates what public object should be regarded as its pictorial icon representation. The document should return a string attribute in the form `L.O` where `L` identifies the public library, and `O` the object in that library. The gadget identified this way is then packed by the desktop inside an Icon gadget in a Finder gadget or when `Desktops.MakeIcon` command is executed.

Each document requires a menu bar with commands associated with the document type when opened with `Desktops.OpenDoc`. The menu bar is gathered from the links "DeskMenu", "SystemMenu" and "UserMenu" when the command is executed. The menu can be constructed with the procedure `Dekstops.NewMenu` or can be taken from a public library. The string given as parameter in the procedure must contain a sequence of Oberon commands. By immediately following a menu command with a word in square brackets, that word will be used as the menu bar button caption. A typical menu string might look as follows:

"MyDoc.Search[Search] MyDoc.Save[Store]"

In the example given, the menu string appears in the Menu constant and that menu is indeed used as long as no TestMenus public library exists. The Desktops.NewMenu procedure automatically adds the Buttons [Close], [Hide] and [Grow] and a NamePlate to the menu bar.

For more flexibility, documents may also define their own menu bars by "exporting" them as public objects from a public library. The public library should contain three menu bars for the Desktop, System track and User track respectively. These menus should have the names "DeskMenu", "SystemMenu" and "UserMenu" respectively. For example, the text documents have such a library (called "TextDocs.Lib"). When the library is missing, the default menu bars are used. Programmers must add support for this feature in their Document handlers. The desktop uses the LinkMsg to request the document to return its menu bar. You should *always* return a *deep copy* of the menu bar from the library. It is best to lock the menu bars and to set the Panel's *Border* attribute to 0. The menu bar can have any height and content.

The example document supports attachments, a concept which has been introduced and used earlier in this chapter.

5.11.9 Defining a new message type

Up to now we have been exploiting the message collection belonging to the standard Oberon distribution. Creating an ad–hoc message was listed among the implementation steps for a new gadget. The following ColorDriver introduces a new message type ColorMsg and provides the capability to change the color of all the open documents having DocumentSkeleton.NewDoc as generator. To this effect the handler must be capable of interpreting the message: the program modifications appear in bold typeface in the source text of DocumentSkeleton.

```
MODULE ColorDriver;
IMPORT Display;
TYPE
   ColorMsg* = RECORD (Display.FrameMsg)
      col*: INTEGER
   END;
VAR Color: INTEGER;

PROCEDURE NextCol*;
VAR M: ColorMsg;
BEGIN
   Color := (Color + 1) MOD 4;
   M.F := NIL; M.col := Color; Display.Broadcast(M)
END NextCol;

BEGIN Color := 1
END ColorDriver.NextCol
```

The color could also be controlled by a background task by replacing the ColorDriver module by:

```
MODULE ColorDriver;
IMPORT Display, Input, Modules, Oberon;
TYPE
   ColorMsg* = RECORD (Display.FrameMsg)
      col*: INTEGER
   END;
VAR Color: INTEGER; task : Oberon.Task;

PROCEDURE ColorTask(me: Oberon.Task);
VAR M: ColorMsg;
BEGIN
   Color := (Color + 1) MOD 4;
   M.F := NIL; M.col := Color; Display.Broadcast(M);
   me.time := Input.Time() + Input.TimeUnit * 3
END ColorTask;

PROCEDURE Cleanup;
```

```
BEGIN
    Oberon.Remove(task)
END Cleanup;

BEGIN
    Color := 1;
    NEW(task); task.handle := ColorTask; task.safe := TRUE; Oberon.Install(task);
    Modules.InstallTermHandler(Cleanup)
END ColorDriver.
```

The background task is immediately activated when a document is opened, because this module is imported and thus automatically loaded. This example also shows how the task can be correctly removed (Cleanup) when a command `System.Free ColorDriver` is executed.

5.11.10 The camera view design pattern

Camera views are programmed in practically the same way as container gadgets. Here only the differences will be sketched. Camera views have to be type extensions of `Gadgets.View` and should preferably reference their contents through the `obj` field in `Gadgets.Frame` instead of the `dsc` field typically used for containers. The extension of `Gadgets.View` is required so that the display masks of the camera view descendants are calculated correctly. For the same reason a special message forwarding strategy, encapsulated in the `Gadgets` module, has to be used:

```
PROCEDURE ToModel(F: Frame; x, y: INTEGER; VAR M: Display.FrameMsg);
    VAR obj: Display.Frame;
BEGIN
    IF (F.obj # NIL) & (F.obj IS Display.Frame) THEN
        obj := F.obj(Display.Frame);
        M.x := ... ; M.y := ...;
        Gadgets.Send(F, x, y, obj, M)
    END
END ToModel;
```

The mask handling for the camera view content can be simplified by not taking the camera view mask into account when updating the mask of the content. Instead, the camera view should create a mask for the content in such a way that the content is completely visible. The `Gadgets.MakeMask` procedure automatically clips the mask of a gadget to the masks of the camera views through which it is visible.

An example of a simple camera view gadget can be found in the `ViewSkeleton` module.

```
MODULE ViewSkeleton;    (* jt, 13.12.94 *)
IMPORT Display, Display3, Fonts, Gadgets, Oberon, Objects;
TYPE
    Frame* = POINTER TO FrameDesc;
    FrameDesc* = RECORD (Gadgets.ViewDesc)
        (* view is adjusted to model size, no border *)
    END;

PROCEDURE ToModel (F: Frame; x, y: INTEGER; VAR M: Display.FrameMsg);
    VAR obj: Display.Frame;
BEGIN
    IF (F.obj # NIL) & (F.obj IS Display.Frame) THEN
        obj := F.obj(Display.Frame);
        M.x := x − obj.X; M.y := y + F.H − (obj.Y + obj.H);
        Gadgets.Send(F, x, y, obj, M)
    END
END ToModel;

PROCEDURE Restore (F: Frame; x, y: INTEGER; dlink: Objects.Object);
    VAR R: Display3.Mask; M: Display.DisplayMsg;
BEGIN
    Gadgets.MakeMask(F, x, y, dlink, R);    (* simplified *)
    IF F.obj = NIL THEN
```

```
            Display3.ReplConst(R, Display3.green, x, y, F.W, F.H, Display.replace);
            Display3.String(R, Display3.FG, x + 3, y + 3, Fonts.Default,
                          "empty view", Display3.replace)
        ELSE
            M.device := Display.screen; M.id := Display.full; M.F := NIL;
            M.dlink := dlink; M.res := -1; ToModel(F, x, y, M)
        END ;
        IF Gadgets.selected IN F.state THEN
            Display3.FillPattern(R, Display3.blue, Display3.selectpat,
                          x, y, x, y, F.W, F.H, Display3.paint)
        END
    END Restore;

    PROCEDURE Adjust (F: Frame; X, Y, W, H: INTEGER);
        VAR MM: Display.ModifyMsg;
    BEGIN
        MM.F := F; MM.mode := Display.display;
        MM.dX := X - F.X; MM.dY := Y - F.Y; MM.dW := W - F.W; MM.dH := H - F.H;
        MM.X := X; MM.Y := Y; MM.W := W; MM.H := H;
        Display.Broadcast(MM)
    END Adjust;

    PROCEDURE Consume (F: Frame; x, y: INTEGER; VAR M: Display.ConsumeMsg);
        VAR f: Objects.Object; CM: Display.ControlMsg;
    BEGIN f := M.obj;
        IF (M.id = Display.drop) & (M.F = F) & (F.obj = NIL) & (f IS Gadgets.Frame) THEN
            WITH f: Gadgets.Frame DO
                f.slink := NIL;
                CM.id := Display.remove; CM.F := f; Display.Broadcast(CM);
                F.obj := f; f.X := 0; f.Y := 0; f.mask := NIL;
                F.state := f.state*{Gadgets.transparent};
                Adjust(F, F.X + M.u, F.Y + F.H - 1 + M.v, f.W, f.H); M.res := 0
            END
        ELSE ToModel(F, x, y, M)
        END;
    END Consume;

    PROCEDURE UpdateMask (F: Frame; x, y: INTEGER; VAR M:
Display3.UpdateMaskMsg);
        VAR R: Display3.Mask; O: Display3.OverlapMsg;
    BEGIN
        IF M.F = F.obj THEN
            NEW(R); Display3.Open(R);
            Display3.Add(R, 0, -F.obj(Display.Frame).H+1,
                          F.obj(Display.Frame).W, F.obj(Display.Frame).H);
            O.F := F.obj(Display.Frame); O.x := 0; O.y := 0; O.M := R; O.res := -1;
            O.dlink := NIL; ToModel(F, x, y, O); M.res := 0
        ELSIF M.F = F THEN
            NEW(F.mask); Display3.Open(F.mask);
            Display3.Add(F.mask, 0, -F.H+1, F.W, F.H);
            F.mask.x := 0; F.mask.y := 0
        ELSE ToModel(F, x, y, M)
        END
    END UpdateMask;

    PROCEDURE FrameHandler* (F: Objects.Object; VAR M: Objects.ObjMsg);
        VAR x, y, u, v: INTEGER; F0: Frame;
    BEGIN
        WITH F: Frame DO
            IF M IS Objects.AttrMsg THEN
                WITH M: Objects.AttrMsg DO
                    IF (M.id = Objects.get) & (M.name = "Gen") THEN
                        M.s := "ViewSkeleton.NewFrame"; M.class := Objects.String; M.res := 0
                    ELSE Gadgets.framehandle(F, M)
                    END
                END
            ELSIF M IS Objects.CopyMsg THEN
                WITH M: Objects.CopyMsg DO
                    IF M.stamp = F.stamp THEN M.obj := F.dlink
                    ELSE
                        NEW(F0); F.stamp := M.stamp; F.dlink := F0;
                        Gadgets.CopyFrame(M, F, F0); F0.border := F.border; M.obj := F0
                    END
                END
        ELSIF M IS Objects.FileMsg THEN Gadgets.framehandle(F, M)
        ELSIF M IS Objects.BindMsg THEN Gadgets.framehandle(F, M)
        ELSIF M IS Objects.LinkMsg THEN Gadgets.framehandle(F, M)
        ELSIF M IS Objects.FindMsg THEN Gadgets.framehandle(F, M)
        ELSIF M IS Display.FrameMsg THEN
          WITH M: Display.FrameMsg DO
            x := M.x + F.X; y := M.y + F.Y;
            u := M.x; v := M.y; (* save *)
            IF M IS Display.DisplayMsg THEN
              WITH M: Display.DisplayMsg DO
                IF M.device = Display.screen THEN
                  IF (M.F = NIL) OR (M.F = F) THEN Restore(F, x, y, M.dlink)
                  ELSE ToModel(F, x, y, M)
                  END
                ELSIF M.device = Display.printer THEN
                END
              END
            ELSIF M IS Display.ConsumeMsg THEN
              Consume(F, x, y, M(Display.ConsumeMsg))
            ELSIF M IS Gadgets.UpdateMsg THEN
              WITH M: Gadgets.UpdateMsg DO
                IF M.obj = F.obj THEN Restore(F, x, y, M.dlink)
                ELSE ToModel(F, x, y, M)
                END
              END
            ELSIF M IS Oberon.InputMsg THEN
              WITH M: Oberon.InputMsg DO
                IF F.obj # NIL THEN ToModel(F, x, y, M)
                ELSE Gadgets.framehandle(F, M)
                END
              END
            ELSIF M IS Oberon.ControlMsg THEN ToModel(F, x, y, M)
            ELSIF M IS Display.LocateMsg THEN Gadgets.framehandle(F, M)
            ELSIF M IS Display.SelectMsg THEN
              Gadgets.framehandle(F, M) (* should be more elaborate *)
            ELSIF M IS Display.ModifyMsg THEN
              WITH M: Display.ModifyMsg DO
                IF M.F = F THEN Gadgets.framehandle(F, M)
                ELSIF M.F = F.obj THEN
                  ToModel(F, x, y, M); Adjust(F, F.X + M.dX, F.Y + M.dY, M.W, M.H)
                ELSE ToModel(F, x, y, M)
                END
              END
            ELSIF M IS Display.ControlMsg THEN
              IF (M(Display.ControlMsg).id = Display.remove) & (M.F = F.obj) THEN
                F.obj := NIL; Gadgets.Update(F)
              END
            ELSIF M IS Display3.OverlapMsg THEN Gadgets.framehandle(F, M);
            ELSIF M IS Display3.UpdateMaskMsg THEN
              UpdateMask(F, x, y, M(Display3.UpdateMaskMsg))
            ELSE ToModel(F, x, y, M)
            END;
            M.x := u; M.y := v (* restore *)
          END
        ELSIF F.obj # NIL THEN F.obj.handle(F.obj, M)
        END
      END
    END FrameHandler;

    PROCEDURE InitFrame* (F: Frame);
    BEGIN F.W := 100; F.H := 100; F.border := 0; F.handle := FrameHandler
    END InitFrame;

    PROCEDURE NewFrame*;
      VAR F: Frame;
    BEGIN NEW(F); InitFrame(F); Objects.NewObj := F;
    END NewFrame;

    END ViewSkeleton.

    Gadgets.Insert ViewSkeleton.NewFrame ~
```

5.11.11 Further perspectives

This concludes our discussion of gadgets programming. The code of the examples presented is included in the Oberon release:

- Skeleton.Mod: a simple visual gadget
- Portraits.Mod: a simple container gadget
- DocumentSkeleton.Mod: a document gadget containing a Panel
- ViewSkeleton.Mod: a camera view gadget

In addition, the source code of existing gadgets, which is also included in this release, constitutes an invaluable reference from where to collect ideas on how to implement Gadgets. Advanced readers might also be interested in understanding the motivations behind some of the design and implementation decisions made during the development of the Gadgets system. For those, the full text of the thesis which emanated from this development [Mar96] is included on the CD–ROM.

Chapter Six

Applications and Examples

6.1 Introduction

In this chapter, we present:

– an application of interest to graphical user interface designers,
– an application for assisting software developers in their daily work,
– the complete source code of an application serving as yet another example,
– an overview of the applications delivered with the system.

6.2 Composing Gadgets with the Layout Language LayLa

LayLa is a functional layout language for constructing arbitrarily complex gadgets. The gadgets can be placed at specific x– and y–coordinates or can be arranged automatically in containers. Layouts saved as text files can be reused in new layout texts.

Each text describes one object. Such a description is a list enclosed in parentheses. The list type is defined by its first element, the operator. The following elements are the operator's arguments which can be lists themselves. All characters between "{" and "}" are comments. Comments may be nested.

Example:

```
(HLIST Panel
   (LAYOUT (SET border 8))
         { a border of 8 pixels in width must be left free of components }
   (ATTR (SET Locked TRUE))
   (NEW Button
      (LAYOUT (SET w 50) (SET h 20))
      (ATTR (SET Caption "Open") (SET Cmd "Desktops.OpenDoc &File.Value"))
   )
   (NEW TextField (ATTR (SET Name "File")))
)
```

The outermost list consists of the operator "HLIST" and its arguments "Panel", "(LAYOUT ...)", "(ATTR ...)", "(NEW Button ...)" and "(NEW TextField ...)". All these arguments except for "Panel" are lists too.

6.2.1 Simple Objects: NEW

A new object is generated with the operator NEW. The first argument of NEW is a generator of the form `Module.NewProcedure` defining the object type. Usually the alias is used instead.

The subsequent arguments are: the list of layout parameters with the operator LAYOUT, the list of attributes with the operator ATTR (or ATTRIBUTES) and the list of links with the operator LINKS. Each of these operators takes as arguments assignment lists of the form (SET name val). In a list of layout parameters the parameter *name* is set to *val*, in a list of attributes, the attribute *name* is set to *val*, and in a list of links the object *val* is inserted as link *name*.

A simple object has the following layout parameters (the type "Size" is explained in section 6.1.1.4)

name	type	description	default
w	Size	gadget width	the width and height set in the generator

h	Size	gadget height	or [1] for virtual objects

Example:

```
(NEW BasicGadgets.NewButton    { <– or    NEW Button }
   (LAYOUT (SET w 50))         { <– or    w=50 }
   (ATTR (SET Caption "Guide") (SET Cmd "Desktops.OpenDoc LayLa.Guide.Text"))
)
```

6.2.1.1 Syntactic Sugar

"(SET name val)" can be written as "name=val". Also the operator LAYOUT is optional. Thus, (LAYOUT (SET name1 val1) (SET name1 val2)) can be shortened to (name1 = val1 name2 = val2).

6.2.1.2 Virtual Objects: VIRTUAL

If you don't want to build a new container for a list or a table, or if you want some blank space (e.g. an empty cell in a table), then you need virtual objects. A virtual object is generated with the generator VIRTUAL. VIRTUAL as an argument (not as a generator) stands short for (NEW VIRTUAL (LAYOUT (SET w []) (SET h []))) or also (NEW VIRTUAL (w=[] h=[])).

Example:

```
(HLIST Panel (border=4 dist=0)
   (NEW Button (h=[]) (ATTR Caption="One"))
   (NEW VIRTUAL (w=10))
   (VLIST VIRTUAL (sameSize=TRUE dist=0)
      (NEW Button (ATTR Caption="Two"))
      VIRTUAL
      (NEW Button (ATTR Caption="Three"))
   )
)
```

6.2.1.3 Using Public Library Objects: SCOPY, DCOPY and L.O

SCOPY makes a shallow copy and DCOPY a deep copy of a public object. The first argument is the object's name of the form `L.O`The following arguments are, as for the operator NEW, the list of layout parameters, the list of attributes and the list of links. A public object `L.O`may be used also directly in an argument of a list of links.

Example:

```
(VLIST Panel (border=10 hjustify=CENTER)
   (NEW Icon (w=50 h=50)
      (ATTR Caption="Diskette")
      (LINKS Model=Icons.Diskette2)
   )
   (SCOPY LayLaTest.Slider (w=75 h=15))
)
```

6.2.1.4 Size

The width "w" and the height "h" of a gadget are of the type:

Size = [(Integer | "DEFAULT")] "[" [Integer] "]" | Integer.

When no width or no height value is specified, these values are assigned by the generator. DEFAULT may also be used instead.
"[" [Integer] "]" is called the *expansion factor*. When it is omitted, 0 is assumed, meaning that the gadget has a fixed width or height (w=50 is the same as w=50[0]). When it is simply "[]", 1 is assumed ([] is equivalent to [1]). For a virtual object, the expansion factor is assumed to be [].

Example:

w=50 [3] : Here *50* is the minimal width of the gadget and *3* the expansion factor. This means that the gadget gets at least 50 pixels wide. If the gadget is located in a wider container, the gadget is expanded according to the expansion factor value. The expansion factor's function is best shown with an example:

```
(HLIST Panel (w=120 dist=0 vjustify=CENTER)
   (NEW Button (w=30[]))
   (NEW TextField (w=[2]))
)
```

Since "Button" has the expansion factor 1 and "TextField" has the expansion factor 2, "TextField" gets twice as wide as "Button" and both expand to occupy the full width of 120 pixels in a 1 to 2 ratio.

```
(HLIST Panel (w=120 dist=0 vjustify=CENTER)
   (NEW Button (w=50[]))
   (NEW TextField (w=[2]))
)
```

If "TextField" was twice as wide as "Button" here, "Button" would become smaller than its minimal size. However, "Button" is at least 50 pixels wide, and "TextField" gets only the remaining 70 pixels. "Button" starts to grow only when "Panel" is more than 150 pixels wide (and the size of "TextField" is twice the size of "Button").

The expansion factor of a row or of a column in a container (see below) is the largest expansion factor of all the gadgets in that row or respectively column. The expansion factor in a row is zero, if any gadget inside the row has a fixed width. The expansion factor in a column is zero, if any gadget inside the column has a fixed height.

6.2.2 Containers

The operators HLIST, VLIST, TABLE and PLACE generate containers. Each of these operators defines a particular way of inserting the components. As with simple objects, the first argument is the generator, followed by the list of layout parameters, the list of attributes and the list of links. The description of the components, which may be simple objects or containers, terminates the argument list.

The default width and height of a container are as big as the bounding box of its components plus the border defined by the container's layout parameters:

name	type	description	default
hborder	Integer	left and right border size	0
vborder	Integer	top and bottom border size	0
border	Integer	border size	0

6.2.2.1 Lists: VLIST and HLIST

VLIST inserts components in a vertical list and HLIST inserts components in a horizontal list. Besides the vborder, hborder and border parameters introduced above, these operators accept further layout parameters:

name	type	description	default
hdist	Integer	horizontal distance between components ignored in VLIST	5
vdist	Integer	vertical distance between components ignored in HLIST	5
dist	Integer	common value for hdist and vdist	5
sameWidth	Boolean	TRUE: all components are made as wide as the widest	FALSE
sameHeight	Boolean	TRUE: all components are made as high as the highest	FALSE

sameSize	Boolean	common value for sameWidth and sameHeight	FALSE
hgrid	STATIC	STATIC: all columns are made as wide as the widest	DYNAMIC
	DYNAMIC	(not the components themselves)	
vgrid	see hgrid	STATIC: all rows are made as high as the highest	DYNAMIC
grid	see hgrid	common value for hgrid and vgrid	DYNAMIC
hjustify	LEFT	the components are placed at the left	LEFT
	RIGHT	at the right	
	CENTER	in the middle (horizontal placement)	
vjustify	TOP	the components are placed at the top	BOTTOM
	BOTTOM	at the bottom	
	CENTER	in the middle (vertical placement)	

The hjustify and vjustify parameter values assigned to a container can be overridden locally by a component with:

name	**type**	**description**	**default**
hjustifyMe	= hjustify	overrides the hjustify of this gadget's container	hjustify
vjustifyMe	= vjustify	overrides the vjustify of this gadget's container	vjustify

Example:

```
(HLIST Panel (border=5 vjustify=TOP hgrid=STATIC)
   (NEW Button)
   (NEW List)
   (NEW Button (vjustifyMe=CENTER hjustifyMe=CENTER))
   (NEW TextGadget)
   (NEW Button)
)
```

6.2.2.2 Tables: TABLE

TABLE inserts components row–wise in a horizontally oriented table (HOR), and column–wise in a vertically oriented table (VER). The number of rows in a horizontally oriented table is determined by counting the components to align in each row (cols). Equally, the number of columns in a vertically oriented table is determined automatically. The layout parameters of VLIST and HLIST also apply to TABLE and are extended by:

name	**type**	**description**	**default**
orientation	HOR	horizontally oriented table	HOR
	VERT	vertically oriented table	
cols	Integer	number of columns (ignored in VERT table)	1
rows	Integer	number of rows (ignored in HOR table)	1

6.2.2.3 Breaking the Formatting of Tables: SPAN und BREAK

The formatting of tables can be broken with SPAN and BREAK. (SPAN rows cols Object) reserves *rows* rows and *cols* columns for the object *Object*.

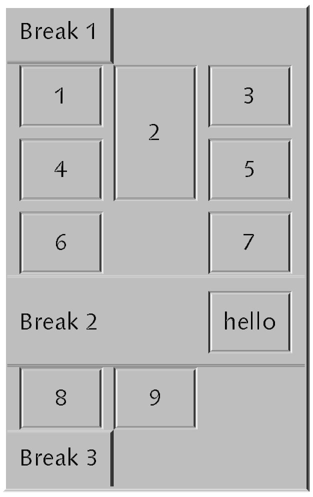
If the components are inserted row–wise, (BREAK Object) replaces the distance to the following row with the single object *Object*. The next component of the table appears at the beginning of the next line. In a vertically oriented table, BREAK replaces the distance to the next column with the single object *Object*. If the table is empty except for a BREAK, the container is made empty. SPAN and BREAK can also be used in lists generated with the operators VLIST and HLIST.

Example:

```
(TABLE Panel (border=8 cols=3)
   (ATTR Locked=TRUE)
   (BREAK (HLIST Panel (border=8) (NEW Caption (ATTR Value="Break 1"))))
   (NEW Button (ATTR Caption = "1"))
   (SPAN 2 1 (NEW Button (w=[1] h=[1]) (ATTR Caption = "2")))
   (NEW Button (ATTR Caption = "3"))
```

```
      (NEW Button (ATTR Caption = "4"))
      (NEW Button (ATTR Caption = "5"))
      (NEW Button (ATTR Caption = "6"))
      VIRTUAL
      (NEW Button (ATTR Caption = "7"))
      (BREAK (HLIST Panel (h=44 w=[1] border=8)
         (NEW Caption (vjustifyMe=CENTER) (ATTR Value="Break 2"))
         (NEW Button (hjustifyMe=RIGHT) (ATTR Caption="hello"))
      ))
      (NEW Button (ATTR Caption = "8"))
      (NEW Button (ATTR Caption = "9"))
      (BREAK (HLIST Panel (border=8) (NEW Caption (ATTR Value="Break 3"))))
   )))
```

which results in:



6.2.2.4 Placing the Components by Hand: PLACE

Experience has shown that ordering components in horizontal or vertical lists or in tables covers all practical needs required for designing containers. However, in very special cases, components can be placed "by hand" in a container using the PLACE operator, though such non–regular ordering is not recommended. The position at which PLACE inserts a component in a container is defined by a vector specified in the component itself. The vector's origin is the lower left corner of the container. The size of the border is automatically taken into account. The position of the lower left corner of the component is specified by two further layout parameters:

name	type	description	default
x	Integer	x–coordinate	0
y	Integer	y–coordinate	0

Example:

```
   (PLACE Panel
      ( border = 5 )    { without syntactic sugar: (LAYOUT (SET border 5)) }
      (ATTR Locked=TRUE)    { (ATTRIBUTES (SET Locked TRUE)) }
      (NEW List (y=30 w=70 h=100))
      (NEW Button (x=80 y=80 (SET w 60) h=20) (ATTR Caption="Press Me"))
   )
```

6.2.3 Configurations: CONFIG

CONFIG defines values and objects which can be used repeatedly inside a LayLa description. For example, you can define an Integer object and use it in two gadgets as a shared model in LINKS lists.

The last argument of the CONFIG operator is the object to construct; all the other arguments are definitions of the form (DEF name value), where *value* is optional. *name* can be used in subsequent definitions and also in the object to construct. For example, if *value* is an object and *name* is used twice, the *same* object is inserted twice. This can cause serious problems and should be done in LINKS lists *only*!

If *value* is an object, (NEW *name*) makes a deep copy of *value*. In this context, no list of layout parameters, list of attributes or list of links may appear. However, if *value* is a configuration, its definition can be modified by means of a parameter list starting with the operator PARAMS. This parameter list is an additional argument of the operator NEW. The operator PARAMS, like LAYOUT, is optional.

Example:

```
 (CONFIG
    (DEF IntGadget    { The value of IntGadget is a configuration }
         (CONFIG
            (DEF Int (NEW Integer (ATTR Value=25)))    { The value of Int is an object
}
            (VLIST VIRTUAL (dist=0)
               (NEW Slider (w=100 h=20) (LINKS Model=Int))
               (NEW TextField (w=100 h=20) (LINKS Model=Int))
            )
         )
    )
    (HLIST Panel (border=10 dist=10)
       (NEW IntGadget)
       (NEW IntGadget (PARAMS Int = (NEW Integer (ATTR Value=75))))
    )
 )
```

6.2.4 Reusing Layouts: INCLUDE

INCLUDE reads the file whose name is given as the first argument and inserts it in the text. If the object in this file is a configuration, parameters can be passed to it as a second argument.

Example:

```
 (CONFIG
    (DEF int1 (NEW Integer (ATTR Value=75)))
    (HLIST Panel (border=8)
       (VLIST VIRTUAL (border=4)
          (NEW Slider (w=100 h=20) (LINKS Model=int1))
          (NEW TextField (w=100 h=20) (LINKS Model=int1)))
       (INCLUDE LayLa.Include.Config (PARAMS (SET int int1) (SET MyBorder 4)))
       (INCLUDE LayLa.Include.Config (MyBorder=4))
    )
 )
```

6.2.5 LayLa tool

The LayLa tool interprets a description text written in the LayLa language, glues gadgets together and inserts these gadgets in the display space or in a public library. The interpreter is written in basic Oberon, without the need for defining new object extensions requiring new messages and handlers.

The `LayLa.Insert` command inserts a gadget at the caret. The gadget is constructed according to the description supplied.

`LayLa.Insert ^`
reads the description starting at the most recent selection.

`LayLa.Insert *`
reads the description from marked text.

`LayLa.Insert @ fileName`

reads the description from the named file.

`LayLa.Insert descriptionText`
reads the description which follows the command.

In the same fashion:

```
LayLa.OpenAsDoc ^
LayLa.OpenAsDoc *
LayLa.OpenAsDoc @fileName
LayLa.OpenAsDoc descriptionText
```
reads the description and places the new gadget in an unnamed PanelDoc, even if the top–most container is VIRTUAL.

```
LayLa.AddToLibrary L.O ^
LayLa.AddToLibrary L.O *
LayLa.AddToLibrary L.O @fileName
LayLa.AddToLibrary L.O descriptionText
```
reads the description and inserts it into the library `L` under the name `O`. If an object with that name already exists, it is replaced. The object's *Name* attribute is assigned the value `O`. If the top–most container is VIRTUAL, only the first gadget is inserted.

Most Panel documents delivered with this release have been constructed using the LayLa tool.

6.2.6 LayLa Description Debugging

The LayLa parser issues an error message in the Oberon log when it detects an error. Here is an example:

```
pos 179  err SET operator expected
```

Very much like for an Oberon compiler error, the log line starting with *pos* indicates an error at approximately that character position in the description, followed by an error indication. To set the caret at that position, mark the description text, select the entire error line and use the `[Locate]` button in the log menu bar.

The parser's function is to detect syntactical errors only. It does not recognize all semantical errors.

6.2.7 Customized Menu Bar Example

Customized menu bars may be created for the various document classes provided with the system (refer to Chapter 3). The following LayLa description text mimics the default menu bars. Use the `LayLa.AddToLibrary` command to store your own creations in the ad–hoc public libraries under the correct names.

```
LayLa.AddToLibrary TextDocs.SystemMenu ^
LayLa.Insert ^

(CONFIG
   (DEF BW 39)   { Button width }
   (DEF BH 18)   { Button height }
   (DEF BW2 80)   { Button width in Panel }
   (DEF Popup
      (NEW Iconizer (w=BW h=16)
         (ATTR Popup=TRUE Pin=FALSE)
         (LINKS
            Closed=(HLIST Panel (w=BW h=16 hjustify=CENTER)
                  (NEW Caption (ATTR Value="Menu")))
            Open=(NEW TextNote)                    { later replaced by a Panel }
         )
```

```
        )
    )
    (HLIST Panel (border=0 dist=0 w=384 h=21 vjustify=CENTER) (ATTR Border=0)
        (NEW NamePlate (w=110 h=20))
        (NEW Button (w=BW h=BH) (ATTR Caption="Close" Cmd="Desktops.CloseDoc"))
        (NEW Button (w=BW h=BH) (ATTR Caption="Hide" Cmd="Finder.Minimize"))
        (NEW Button (w=BW h=BH) (ATTR Caption="Grow" Cmd="Desktops.Grow"))
        (NEW DigitalClock (w=BW h=BH))
        (NEW Popup)
    )
)
```

This menu bar features three classical Buttons, a DigitalClock and a pop–up menu implemented in an Iconizer (refer to the description in Chapter 4). It is not possible to further define the menu text under LayLa. To proceed with the implementation of the TextNote one has to use the `Libraries.Panel` and Columbus.

1 – Select the (CONFIG line
2 – Execute the LayLa command to add the defined object to the ad hoc library
3 – Open the `Libraries.Panel`
4 – Display the Directory of libraries in Memory
5 – MM click on the library name (in the example `TextDocs`)
6 – MM click on the object name (in the example `SystemMenu`)
7 – MM click on "Reference"
8 – Set the caret at any convenient place and retrieve the object
9 – Edit the pop–up menu with Columbus
10 – Store the library

The new customized menu bar is ready for use.

To get around the difficulty of finalizing the text in the TextNote (or TextGadget) one can choose to place a Panel on the "Open" side with the advantage that the construction can be finalized with the help of LayLa.

```
Open=(VLIST Panel (w=84 hjustify=CENTER vjustify=TOP vdist=1 border=1)
        (ATTR Locked=TRUE)
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="SearchDiff ↑↑" Cmd="TextDocs.SearchDiff \w"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="RecallText" Cmd="TextDocs.Recall"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="RecallDoc" Cmd="Desktops.Recall"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="Columbus" Cmd="Columbus.Inspect ~"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="Count *" Cmd="EditTools.Words *"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="Controls *" Cmd="TextDocs.Controls *"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="DeleteObjs *" Cmd="EditTools.RemoveObjects *"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="OpenMod ↑" Cmd="TextDocs.Show ↑"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="OpenUnix ↑" Cmd="EditTools.OpenUnix ↑"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="OpenAscii ↑" Cmd="EditTools.OpenAscii ↑"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="StoreUnix" Cmd="EditTools.StoreUnix"))
    (NEW Button (w=BW2 h=BH)
        (ATTR Caption="StoreAscii" Cmd="EditTools.StoreAscii"))
)
```

Since the entire menu bar is based on a Panel, one can infer that it is quite feasible to reshape the conventional menu bars provided to one's wildest compositions whilst limited by one's own imagination.

6.3 The TextPopups tool

The module TextPopups uses the ListGadget, linked to a ListModel, to display a simple popup menu of Oberon commands, document names, procedure names and type names in TextDocs. This facility is particularly interesting for Oberon software developers: it allows rapid access to text pieces of interest in a source text. The popup menu is controlled by a menu description stored in the TextPopups.Text file which may be edited and customized at will. When a new system is installed, this file contains the following text:

```
[Mod]
Compiler.Compile *
Compiler.Compile *\f
Compiler.Compile *\s
Analyzer.Analyze *
–––––––––––––––––––
<5 Recent Files>
–––––––––––––––––––
<Procedures>

[Text]
TextDocs.SearchDiff \w
–––––––––––––––––––
<5 Recent Files>

[Tool]
<5 Recent Files>
```

The text is divided into sections identified by the suffix of the document names to which the specifications must apply. The previous text contains three sections. A section may contain the following optional elements:

– an enumeration of Oberon commands
– <i any text> where i is an integer specifying a number of file names
  The "Recent Files" used above is nothing more than a user hint.
– <Procedures>
– <Types>
– a separation line "–––––––––––––"

The menu pops up when the MM key is pressed, but only if the mouse focus is positioned outside of text or positioned on a space, a carriage return or a tab. The mouse focus is then automatically positioned approximately in the menu middle. The menu items are made of Oberon commands, the names of the i last document names recently opened (the most recent appears on top), procedure names alphabetically ordered and type names also ordered. Dragging on the MM key places the focus on other menu items which are underlined. When the key is released, if the focus is on a:

– command: the command is executed
– file name: the corresponding document is opened
– procedure name: the text is re–positioned to make the name visible
– type name: the text is re–positioned to make the name visible.

Commands defined in TextPopups.Mod

`TextPopups.Install`
activates the facility. It is convenient to add this command on a line in the Configuration.Text file (or in the System.InitCommands section of the PC Native Oberon Oberon.Text file).

`TextPopups.Remove`
removes the facility without unloading the module.

6.4 The ColorSystem tool

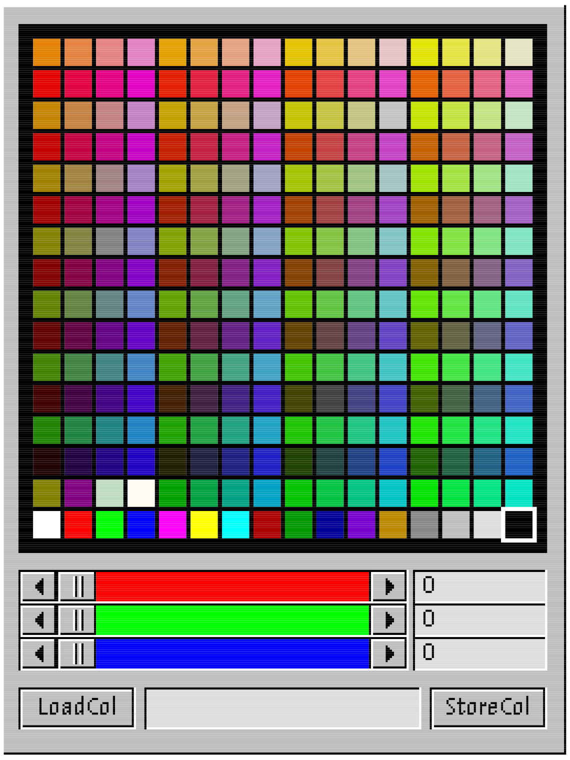


Figure 6.1 The ColorSystem

The ColorSystem tool is provided for editing the color palette, for saving a palette to a palette file and for loading a palette from a file. You edit the current color palette from the ColorSystem.Panel. The tool uses the RGB method for defining the colors that appear on the computer monitor. For that purpose, it calls the central procedures Display.GetColor and Display.SetColor exported by the Display module. A number of colors are preset and cannot be modified. This number varies with the platform. When Oberon is started, the palette is set by loading the Default.Pal file.

```
MODULE ColorSystem (*JM/ JG 10.8.94/JM 27.4.95*);

IMPORT Desktops, Display, Display3, Documents, Effects, Files, Gadgets,
  Input, Oberon, Objects, Out, Printer, Printer3, Strings, Texts;

TYPE
  Color* = POINTER TO ColorDesc;
  ColorDesc* = RECORD (Gadgets.ObjDesc)
    col*: INTEGER
  END;

  Frame* = POINTER TO FrameDesc;
  FrameDesc* = RECORD (Gadgets.FrameDesc)
    col*: INTEGER;
  END;

VAR NC, SC: INTEGER;

PROCEDURE HandleObj* (obj: Objects.Object; VAR M: Objects.ObjMsg);
  VAR obj1: Color; red, green, blue: INTEGER;
BEGIN
  WITH obj: Color DO
    IF M IS Objects.AttrMsg THEN
      WITH M: Objects.AttrMsg DO
        IF M.id = Objects.get THEN
          IF M.name = "Gen" THEN
            M.class := Objects.String; COPY("ColorSystem.NewObj", M.s); M.res := 0
          ELSIF M.name = "col" THEN M.class := Objects.Int; M.i := obj.col; M.res := 0
          ELSIF M.name = "red" THEN Display.GetColor(obj.col, red, green, blue);
          M.class := Objects.Int; M.i := red; M.res := 0
        ELSIF M.name = "green" THEN Display.GetColor(obj.col, red, green, blue);
          M.class := Objects.Int; M.i := green; M.res := 0
        ELSIF M.name = "blue" THEN Display.GetColor(obj.col, red, green, blue);
          M.class := Objects.Int; M.i := blue; M.res := 0
        ELSE Gadgets.objecthandle(obj, M)
        END
      ELSIF M.id = Objects.set THEN
        IF M.name = "col" THEN obj.col := SHORT(M.i); M.res := 0
        ELSIF M.name = "red" THEN Display.GetColor(obj.col, red, green, blue);
          IF M.class = Objects.String THEN Strings.StrToInt(M.s, M.i) END;
          Display.GetColor(obj.col, red, green, blue);
          Display.SetColor(obj.col, SHORT(M.i), green, blue); M.res := 0
        ELSIF M.name = "green" THEN Display.GetColor(obj.col, red, green, blue);
          IF M.class = Objects.String THEN Strings.StrToInt(M.s, M.i) END;
          Display.GetColor(obj.col, red, green, blue);
          Display.SetColor(obj.col, red, SHORT(M.i), blue); M.res := 0
        ELSIF M.name = "blue" THEN Display.GetColor(obj.col, red, green, blue);
          IF M.class = Objects.String THEN Strings.StrToInt(M.s, M.i) END;
          Display.GetColor(obj.col, red, green, blue);
          Display.SetColor(obj.col, red, green, SHORT(M.i)); M.res := 0
        ELSE Gadgets.objecthandle(obj, M)
        END
      ELSIF M.id = Objects.enum THEN
        M.Enum("col"); M.Enum("red"); M.Enum("green"); M.Enum("blue");
        Gadgets.objecthandle(obj, M)
      END
    END
  ELSIF M IS Objects.CopyMsg THEN
    WITH M: Objects.CopyMsg DO
      IF M.stamp = obj.stamp THEN M.obj := obj.dlink    (* copy msg arrives again *)
      ELSE (* first time copy message arrives *)
        NEW(obj1); obj.stamp := M.stamp; obj.dlink := obj1;
        obj1.handle := obj.handle; obj1.col := obj.col;
        M.obj := obj1
      END
    END
  ELSE Gadgets.objecthandle(obj, M)
  END
 END
END HandleObj;

PROCEDURE **NewObj***;
VAR obj: Color;
BEGIN
 NEW(obj); obj.handle := HandleObj; obj.col := Display.FG; Objects.NewObj := obj
END NewObj;

PROCEDURE **LoadColors***;
 VAR obj: Objects.Object; M: Objects.AttrMsg;
  T: Texts.Text; S: Texts.Scanner;
  f: Files.File; R: Files.Rider;
  beg, end, time: LONGINT;
  col: INTEGER; red, green, blue: CHAR;
BEGIN
 M.id := Objects.get; M.name := "Value";
 obj := Gadgets.FindObj(Gadgets.context, "PalName");
 obj.handle(obj, M);
 IF (M.id # Objects.String) OR (M.s[0] = 0X) THEN
  Oberon.GetSelection(T, beg, end, time);
  IF time >= 0 THEN
   Texts.OpenScanner(S, T, beg); Texts.Scan(S); COPY(S.s, M.s)
  END
 END;
 f := Files.Old(M.s); Files.Set(R, f, 0);
 IF f # NIL THEN col := 0;
  REPEAT
   Files.Read(R, red); Files.Read(R, green); Files.Read(R, blue);
   Display.SetColor(col, ORD(red), ORD(green), ORD(blue));
   INC(col)
  UNTIL col = NC
 END
END LoadColors;
```

```
PROCEDURE **StoreColors***;
  VAR obj: Objects.Object; M: Objects.AttrMsg;
    T: Texts.Text; S: Texts.Scanner;
    f: Files.File; R: Files.Rider;
    beg, end, time: LONGINT;
    col: INTEGER; red, green, blue: INTEGER;
BEGIN
  M.id := Objects.get; M.name := "Value";
  obj := Gadgets.FindObj(Gadgets.context, "PalName");
  obj.handle(obj, M);
  IF (M.class # Objects.String) OR (M.s[0] = 0X) THEN
    Oberon.GetSelection(T, beg, end, time);
    IF time >= 0 THEN
      Texts.OpenScanner(S, T, beg); Texts.Scan(S); COPY(S.s, M.s)
    END
  END;
  IF M.s # "" THEN
    Out.String("ColorSystem.StoreColors ");
    f := Files.New(M.s); Files.Set(R, f, 0);
    IF f # NIL THEN col := 0;
      REPEAT
        Display.GetColor(col, red, green, blue);
        Files.Write(R, CHR(red));
        Files.Write(R, CHR(green));
        Files.Write(R, CHR(blue));
        INC(col)
      UNTIL col = NC
    END;
    Files.Register(f);
    Out.String(M.s); Out.Ln
  END
END StoreColors;

PROCEDURE HandleAttributes (F: Frame; VAR M: Objects.AttrMsg);
BEGIN
  IF M.id = Objects.get THEN
    IF M.name = "Gen" THEN
      M.class := Objects.String; COPY("ColorSystem.NewFrame", M.s); M.res := 0
    ELSIF M.name = "Color" THEN
      M.class := Objects.Int; M.i := F.col; M.res := 0
    ELSE Gadgets.framehandle(F, M)
    END
  ELSIF M.id = Objects.set THEN
    IF M.name = "Color" THEN
      IF M.class = Objects.Int THEN
        F.col := SHORT(M.i); M.res := 0
      END
    ELSE Gadgets.framehandle(F, M);
    END
  ELSIF M.id = Objects.enum THEN
    M.Enum("Color"); M.Enum("Cmd"); Gadgets.framehandle(F, M)
  END
END HandleAttributes;

PROCEDURE Restore (F: Frame; Q: Display3.Mask; x, y, w, h: INTEGER);
  VAR model: Color; col, i, j, xcur, ycur, wfld, hfld, wmarg, hmarg: INTEGER;
BEGIN
  model := F.obj(Color);
  Display3.ReplConst(Q, F.col, x, y, w, h, Display.replace);
  wfld := (w - (SC + 1)*2) DIV SC; wmarg := (w - SC*(wfld + 2) - 2) DIV 2;
  hfld := (h - (SC + 1)*2) DIV SC; hmarg := (h - SC*(hfld + 2) - 2) DIV 2;
  col := 0; j := 0; ycur := y + hmarg + 2;
  REPEAT i := 0; xcur := x + wmarg + 2;
    REPEAT
      Display3.ReplConst(Q, col, xcur, ycur, wfld, hfld, Display.replace); INC(col);
      INC(i); xcur := xcur + wfld + 2
    UNTIL i = SC;
    INC(j); ycur := ycur + hfld + 2
  UNTIL j = SC;
  i := model.col MOD SC; j := model.col DIV SC;
  Display3.Rect(Q, Display.FG, Display.solid,
    x + wmarg + i*(wfld + 2), y + hmarg + j*(hfld + 2), wfld + 4, hfld + 4, 2,
    Display.invert);
  IF Gadgets.selected IN F.state THEN
```

```
    Display3.FillPattern(Q, Display3.white, Display3.selectpat, x, y, x, y, w, h,
    Display.paint)
  END
END Restore;

PROCEDURE Print (F: Frame; VAR M: Display.DisplayMsg);
  VAR Q: Display3.Mask;

  PROCEDURE P (x: INTEGER): INTEGER;
  BEGIN RETURN SHORT(x * LONG(10000) DIV Printer.Unit)
  END P;

BEGIN
  Gadgets.MakePrinterMask(F, M.x, M.y, M.dlink, Q);
  Printer3.ReplConst(Q, F.col, M.x, M.y, P(F.W), P(F.H), Display.replace)
END Print;

PROCEDURE **Copy*** (VAR M: Objects.CopyMsg; from, to: Frame);
BEGIN to.col := from.col; Gadgets.CopyFrame(M, from, to)
END Copy;

PROCEDURE SelectColor (F: Frame; VAR M: Oberon.InputMsg; Q: Display3.Mask;
                                    x, y, w, h: INTEGER);
  VAR model: Color; keysum: SET;
    i, j, k, l, x0, y0, w0, h0, wfld, hfld, wmarg, hmarg, col: INTEGER;
BEGIN
  model := F.obj(Color);
  wfld := (w - (SC + 1)*2) DIV SC; wmarg := (w - SC*(wfld + 2) - 2) DIV 2;
  hfld := (h - (SC + 1)*2) DIV SC; hmarg := (h - SC*(hfld + 2) - 2) DIV 2;
  x0 := x + wmarg; y0 := y + hmarg; w0 := wfld + 2; h0 := hfld + 2;
  keysum := M.keys;
  REPEAT
    i := (M.X - x0) DIV w0; j := (M.Y - y0) DIV h0;
    IF i < 0 THEN i := 0 ELSIF i >= SC THEN i := SC - 1 END;
    IF j < 0 THEN j := 0 ELSIF j >= SC THEN j := SC - 1 END;
    col := i + j*SC;
    IF (model.col < 0) OR (col # model.col) THEN
      Oberon.FadeCursor(Oberon.Mouse);
      IF model.col >= 0 THEN
        k := model.col MOD SC; l := model.col DIV SC;
        Display3.Rect(Q, Display.FG, Display.solid, x0 + k*w0, y0 + l*h0, w0 + 2, h0 + 2,
        2, Display.invert)
      END;
      Display3.Rect(Q, Display.FG, Display.solid, x0 + i*w0, y0 + j*h0, w0 + 2, h0 + 2, 2,
      Display.invert);
      model.col := col
    END;
    Oberon.DrawCursor(Oberon.Mouse, Oberon.Mouse.marker, M.X, M.Y);
    Input.Mouse(M.keys, M.X, M.Y);
    keysum := keysum + M.keys
  UNTIL M.keys = {};
  Oberon.RemoveMarks(x, y, w, h);
  Gadgets.Update(model)
END SelectColor;

PROCEDURE **HandleFrame*** (F: Objects.Object; VAR M: Objects.ObjMsg);
  VAR x, y, w, h: INTEGER; F1: Frame; Q: Display3.Mask;
BEGIN
  WITH F: Frame DO
    IF M IS Display.FrameMsg THEN
      WITH M: Display.FrameMsg DO
        IF (M.F = NIL) OR (M.F = F) THEN (* message addressed to box *)
          x := M.x + F.X; y := M.y + F.Y; w := F.W; h := F.H;
          IF M IS Display.DisplayMsg THEN
            WITH M: Display.DisplayMsg DO
              IF M.device = Display.screen THEN
                IF (M.id = Display.full) OR (M.F = NIL) THEN
                  Gadgets.MakeMask(F, x, y, M.dlink, Q);
                  Restore(F, Q, x, y, w, h)
                ELSIF M.id = Display.area THEN
                  Gadgets.MakeMask(F, x, y, M.dlink, Q);
                  Display3.AdjustMask(Q, x + M.u, y + h - 1 + M.v, M.w, M.h);
                  Restore(F, Q, x, y, w, h)
                END
            ELSIF M.device = Display.printer THEN Print(F, M)
            END
          END
        ELSIF M IS Oberon.InputMsg THEN
          WITH M: Oberon.InputMsg DO
            IF (M.id = Oberon.track) & Gadgets.InActiveArea(F, M) & (M.keys # {}) THEN
              Gadgets.MakeMask(F, x, y, M.dlink, Q);
              Oberon.RemoveMarks(x, y, w, h);
              SelectColor(F, M, Q, x, y, w, h);
              M.res := 0
            ELSE Gadgets.framehandle(F, M)
            END
          END
        ELSIF M IS Oberon.ControlMsg THEN Gadgets.framehandle(F, M)
        ELSIF M IS Display.ModifyMsg THEN Gadgets.framehandle(F, M)
        ELSIF M IS Display.SelectMsg THEN Gadgets.framehandle(F, M)
        ELSIF M IS Display.ConsumeMsg THEN Gadgets.framehandle(F, M)
        ELSE Gadgets.framehandle(F, M)
        END
      END
    END
  ELSIF M IS Objects.AttrMsg THEN HandleAttributes(F, M(Objects.AttrMsg))
  ELSIF M IS Objects.FileMsg THEN
    WITH M: Objects.FileMsg DO Gadgets.framehandle(F, M) END
  ELSIF M IS Objects.CopyMsg THEN
    WITH M: Objects.CopyMsg DO
      IF M.stamp = F.stamp THEN M.obj := F.dlink (* msg arrives again *)
      ELSE (* first time msg arrives *)
        NEW(F1); F.stamp := M.stamp; F.dlink := F1;
        Copy(M, F, F1); M.obj := F1
      END
    END
  ELSE Gadgets.framehandle(F, M)
  END
 END
END HandleFrame;

PROCEDURE NewFrame*;
VAR F: Frame;
BEGIN
  NEW(F); F.W := 50; F.H := 50; F.col := Display.FG; F.handle := HandleFrame;
  NewObj; F.obj := Objects.NewObj; Objects.NewObj := F
END NewFrame;

BEGIN
  IF Display.Depth(0) >= 8 THEN NC := 256; SC := 16
  ELSE NC := 16; SC := 4
  END
END ColorSystem.
```

6.5 The Oberon System 3 Applications Collection

Oberon System 3 is delivered with a collection of applications which are either already installed or, for the majority of them, contained in archives (application packages).

6.5.1 PC Native Oberon Applications

All the applications are contained in one package compressed with Compress. It suffices to install this package to install all the applications.

Application name	Description
ASCIITab	ASCII table gadget
Backdrops	Wallpaper generator
Calc	Simple calculator
Coco	Scanner and parser generator
CUSeeMe	CUSeeMe video receiver (requires Net)

Diff	Text difference tool
Dim3	3D–engine
EditKeys	Keyboard macro utility for TextGadgets
Find	String searcher
FontEditor	Oberon raster font editor
Games	A collection of games: Asteroids, Freecell, MineSweeper, Scramble, Shanghai, Sokoban, Solitaire, Spider, Tetris
Hex	Binary file editor
Histogram	Histogram gadget (see Chapter 4)
HPCalc	RPN calculator
HTML	Simple text to HTML converter
LayLa	Layout language (see section 6.1)
LayoutPanels	LayoutPanels with formatting constraints
LPRPrinter	Remote (LPR) printer utility (requires Net)
Magnifier	Pixel magnifier
PictConverters	Picture format converters
RX	Regular expression searcher
Snapshot	Makes snapshot of gadget, viewer, document, screen
Sort	Line–based sorter
Sound	Sound and CD tool (Soundblaster)
TextPopups	Popup menu for TextDocs (see section 6.2)
V24Terminal	Simple V24 terminal
WTS	V4 text to System 3 text converter

6.5.2 Oberon System 3 for Windows Applications

Application packages can be installed by executing the Installer tool command:

```
        Installer.Install archive [Script] ~
```

This command automatically processes the archive according to the specifications contained in the script file named in the parameter list. The specifications may instruct the Installer to compile the source code files, to generate definition files (`.Def`), and to copy application related files. A script file named `Install.Script`, included in each archive, is used by default. If you do not have a standard Oberon directory layout, you may specifiy your sub–directories in the registry section [Installer]. The standard setting is:

```
[Installer]
SYSTEM := C:/Oberon/System    ; system files
OBJ := C:/Oberon/Obj    ; compiled obj files
SRC := C:/Oberon/Src    ; source code files
APPS := C:/Oberon/Apps    ; miscellaneous application files and packages
DOCU := C:/Oberon/Docu    ; tutorials and other documentation files
```

Archive file (.Arc)	**Description**
ASCIITab	ASCII table gadget
Backdrops	Wallpaper generator
Calc	Simple calculator
Coco	Scanner and parser generator
Diff	Text difference tool
Dim3	3D–engine
EditKeys	Keyboard macro utility for TextGadgets
FileUtils	A file backup tool (see section 2.7)
Find	String searcher
Games	A collection of games: Asteroids, Freecell, MineSweeper, Scramble, Shanghai, Sokoban, Solitaire, Spider, Tetris
Hex	Binary file editor
Histogram	Histogram gadget (see Chapter 4)
HPCalc	RPN calculator
HTML	Simple text to HTML converter
JuiceCDK	Juice Authoring Toolkit

LayLa	Layout Language (see section 6.1)
LayoutPanels	LayoutPanels with formatting constraints
Leonardo	A tool for drawing illustrations
Log	Another Out module
LPRPrinter	Remote (LPR) printer utility
Magnifier	Pixel magnifier
MultiMail	Send e–mail with MIME attachments
OldFiles	Reads Oberon files from DOS and Win32s Systems (see section 2.7)
PictConverters	Picture format converters
Pr2Fnt	Printer fonts in 200 dpi resolution (not an application)
Pr6Fnt	Printer fonts in 600 dpi resolution (not an application)
PS	Picture to Postscript converter
RX	Regular expression searcher
Sort	Line–based sorter
TextPopups	Popup menu for TextDocs (see section 6.2)
V24Terminal	Simple V24 terminal
Win.Audio	CDAudioPlayer, FileAudioPlayer, Mixer and other tools
Win32.Backup	Ceres and PC Native Oberon Backup
Win.CUSM	CUSeeMe video receiver
Win.FontTools	Oberon raster font editor, Converter to Windows FON resources
Win.ODBC	ODBC Interface
Win.Snapshot	Makes snapshot of gadget, viewer, document, screen
WTS	V4 text to System 3 text converter

6.5.3 Oberon System 3 for Linux and Macintosh Applications

The applications are installed in the same manner as Windows applications. The applications are those listed in section 6.5.2, but a few of them are not available:

Dim3, FileUtils, OldFiles, V24Terminal and all the Win* applications.

Appendix A

Configuring the System

A.1 Introduction

A personal computer needs to be configured and tailored to a given environment and to a personal profile. For this purpose, special tools are available in Oberon. They are described in the following sections separately for each implementation. Further platform–dependent details must be obtained from the specific installation guides.

A.2 PC Native Oberon

For PC Native Oberon, the configuration tool is a simple text called `Oberon.Text` that can be edited freely. The configuration data is structured according to a simple, self–explanatory and recursive syntax whose formal definition in EBNF (Extended Backus Naur Form) is as follows:

```
Configuration =  Group.
Group = { Entry } { Token }.
Entry = [ Name "=" ] Value.
Value = Token | "{" Group "}".
Token = any token from Texts.Scanner, where "{" and "}" must occur pairwise.
```

Example:

```
{ The Oberon Configuration Text }

System = {
   InitCommands = {
      { System.OpenLog }
      { System.Open System.Tool }
   }
}

Printer = {
   Resolution = 300
   Layout = {
      Width = {210 mm}
      Height = {297 mm}
   }
}

NetSystem = {
   Hosts = {
      Domain = { "domain", "" }
      DNS1 = { "", "" }
   }
   NewsFont=Courier10.Scn.Fnt
}

SLIP = {
   Host = { "<slipserver>" }
   Init = { COM1 19200 }
   Dial = {
      "ATZ"
      10 "OK"
      "ATD <phonenumber>"
      60 "CONNECT"
      10 "login:"
```

```
        USER
        20 "password"
        PASSWORD
        20 "enabled"
        START }
}
```

For programmed access to Oberon.Text a single procedure Oberon.OpenScanner in combination with the standard text scanning facility suffices.

```
    PROCEDURE OpenScanner (VAR S: Texts.Scanner; entry: ARRAY OF CHAR);
```

This procedure is used within programs to position the text scanner at any desired entry in the Oberon.Text. It takes two parameters, a scanner and an entry name, where the name actually denotes a path and may be arbitrarily nested by qualification.

For example, the name Printer.Layout.Width positions the scanner to the width specification "210 mm" and SLIP.Dial positions it to the dial–in code sequence.

Note that this concept of system configuration is open in many senses. Not only can new entries be added freely but the local syntax of entries (within group braces "{" "}") is completely open as well. Also note that there are a few low–level configuration data that are used on levels below the text system and therefore cannot make use of it in the mentioned way. For these cases (that users never have to care about) a low–level extension of the above explained mechanism is provided. It is based on a set of pairs (name, value) that are stored in some initial part of the boot file.

A.2 Oberon System 3 for Windows

The registry provides a consistent interface for programs to store configuration data. The data is organized in a two–level hierarchy. The first level is called "section", the second level is called "key". The registry serves to associate a value with a (section, key) pair and its data has the format:

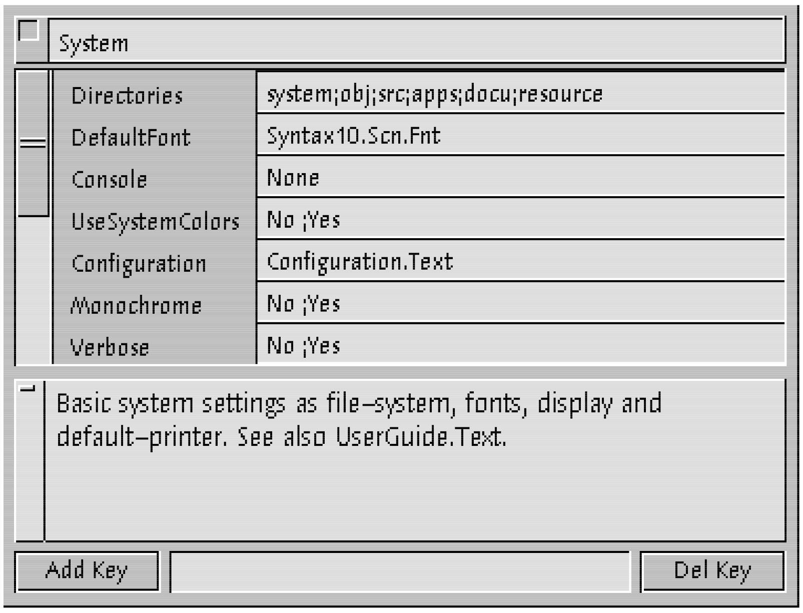
```
    [Section1]
    Key1=Value1
    Key2=Value

    [Section2]
    Key1=Value
    ...
```

For Windows 95, 98 and NT, the registry data is kept in the Windows registry under the key:

```
        HKEY-CURRENT-USER/Software/Oberon System 3/Release 2.x/
```

The registry data is accessed when Oberon is started.

This data can be accessed and maintained by the user in various ways. The first and most expeditive way is by executing the commands System.Get and System.Set (described in Chapter 2) to retrieve and respectively update the registry data. Second, using the IniEdit.Panel, a graphical user interface for maintaining that data.



System

Directories	system;obj;src;apps;docu;resource
DefaultFont	Syntax10.Scn.Fnt
Console	None
UseSystemColors	No ;Yes
Configuration	Configuration.Text
Monochrome	No ;Yes
Verbose	No ;Yes

Basic system settings as file-system, fonts, display and default-printer. See also UserGuide.Text.

Third, with the commands provided by the `RegistryTools` module for storing and loading the Oberon specific registry data in a file, which can be edited as an ASCII file in Oberon. The name of the registry file is the user's choice. Such a file `Oberon.reg` is delivered with the system for prompting the system after its installation without user intervention.

`RegistryTools.Version`
Display software and version information for the currently running version. Example:

```
Software: Oberon System 3
Version: Release 2.3
```

`RegistryTools.Store [\V version] filename`
Store all the entries in [HKEY-CURRENT-USER\Software\<software>\<version>] in the named registry file. By default, the registry data of the currently running version is stored. To store the data of another version use the V option. This file is best edited after opening it with `EditTools.OpenAscii` and after that it should be stored back with `EditTools.StoreAscii`. Such a file can serve as backup copy.

Example: `RegistryTools.Store \V "Release2.1" old.reg`

`RegistryTools.Load [\V version] filename`
Load the registry entries in the named registry file to [HKEY-CURRENT-USER\Software\<software>\<version>]. By default the registry of the currently running version is overwritten. To overwrite entries of another version use the V option.

Fourth, from outside of Oberon, by running the Windows Regedit.exe program and by editing the data via the Edit menu. After updating, Oberon must be restarted to apply the new values.

For programmed access, four procedures are provided in the module `Registry`:

```
(* Associates a value with a (section, key) pair. *)
PROCEDURE Set (section, key, value: ARRAY OF CHAR);

(* Retrieves the value associated with a (section, key) pair. *)
PROCEDURE Get (section, key: ARRAY OF CHAR; VAR value: ARRAY OF CHAR);

(* Enumerates all keys and their values in a section. *)
PROCEDURE Enumerate (section: ARRAY OF CHAR; handler: EntryHandler);

(** Enumerates all sections. *)
```

PROCEDURE EnumerateSections*(handler: SectionHandler);

A section called [SystemInfo] contains the following details depending on the host operating system:

Key	Windows 95/NT	Windows 3.1
OS	Windows	Windows
Processor	Intel	Intel
FileSystem	VFAT/NTFS	FAT
HostFilenamePrefix	empty	- (underscore)

Remark: The registry is mostly of use to modules low in the module hierarchy, and its extensive use is discouraged.

The registry data consists of user specific, critical data that can be backed up in two different ways. First, inside of Oberon, using the `RegistryTools` commands described above. Second, by running the Regedit.exe program and by selecting "Export Registry File..." in the Registry menu. Restoring is carried out with an Import operation. This is particularly needed when upgrading an installed Oberon version. Also, when an Oberon is shared by different users, users have to import their personalized registry data, prior to using the system.

For Windows 3.1, the registry data is kept in the `oberon.ini` file. The registry file is accessed when Oberon is started. If no explicit path is specified in the command

```
oberon.exe [path\oberon.ini]
```

starting the Oberon application, the file is searched for in the following directories:

(1) current directory,
(2) Windows system directory,
(3) Windows directory,
(4) in all directories listed in the PATH environment variable.

This file can be edited directly with an ASCII text editor or with an Oberon editor. In the latter case, care must be taken to preserve the ASCII format by using the commands `EditTools.OpenAscii` and `EditTools.StoreAscii`. Be sure to access the `oberon.ini` file in the root directory of your Oberon system and to store it back there, and not in some other current directory! It is much safer to retrieve or update the registry data directly from Oberon with the commands `System.Set` and `System.Get` which are described in Chapter 2. Finally, the `IniEdit.Panel` provides a graphical user interface for maintaining the configuration data. After updating, Oberon must be restarted to apply the new values.

A.3 Oberon System 3 for Linux

The registry provides the same consistent interface for programs to store configuration data as is described in the previous section. There are however a few differences:

o there is no "Directories" key in the registry section [System]. Instead, the directories to search are defined in an environment variable "OBERON".

o Oberon is started with the command `oberon`. The `oberon.ini` file is searched in the directories named in "OBERON".

A section called [SystemInfo] returns the following details:

Key	Value
OS	Linux / X Windows
Processor	Intel

A.3 MacOberon System 3

The configuration data is kept in the resource fork of the Oberon application. In principle the users do not have to edit this data themselves. Data is retrieved and updated directly from Oberon with the commands `System.Set` and `System.Get` which are described in Chapter 2. Changes are applied immediately.

A section called [SystemInfo] returns the following details:

Key	Value
OS	MacOS
Processor	PowerPC (or MC680x0)

A.4 Information which all systems have in common

Even if configuration data is stored in `Oberon.Text` for PC Native Oberon and in the registry for the other implementations, all these systems have a common need for a stock of information. This central information is located in the following registry sections (or in their equivalent group under PC Native Oberon):

[Aliases]	list of aliases for the generator procedures
[Documents]	list of document extensions with their associated generator
[FinderTemplates]	list of document names to appear in a Finder

The 3 former section names are prefixed "Gadgets." in Windows 95, 98 and NT.

[NetSystem]	list of networking specifications
[PictureConverters]	list of acceptable graphical information formats
[System]	list of local system specifications
[SystemInfo]	list of system values

Two keys are of particular interest in the [System] section. They are:
DateFormat and
TimeFormat
which influence the representation of the TimeStamp gadget and of the text string generated by the commands `Clocks.InsertDate` and `Clocks.InsertTime`. The default values are `DD.MM.YY` and `HH:MM:SS`. The section contains further details on the format specifications in the form of comments.

Appendix B

Extended Language and Compiler for PC Native Oberon

The PC Native implementation of Oberon System 3 comes with an improved and slightly extended language and compiler. In the following we enumerate and informally explain the new constructs and features. We should emphasize that the sole purpose of this language extension is an increased expressiveness and that it does not compromise the efficiency of the original language. We consider both style and implementation of the extensions as completely compatible with the "spirit" of Oberon and in particular with the principle of "making it as simple as possible but not simpler".

B.1 Methods and initializers within record scopes

Traditionally in the Pascal line, record scopes are poorly used. In the course of a current project (called Active Oberon) with the goal of unifying objects and processes we upgrade record scopes by the following optional ingredients:

(a) a body,
(b) procedure declarations and
(c) an initializer.

The body is used to specify the intrinsic behavior of objects of this type and it typically runs as a separate thread. Type–local procedures support the protected access to objects of this type in a multi–process environment (using built–in exclusive or shared locks) and the initializer guarantees atomic creation and initialization of instances.

Syntactically, record types have thereby been brought into closer line with modules, where the VAR keyword and the repeated name after the END keyword are optional. One of the procedure declarations within a record scope can be distinguished as initializer by adding a "&"–mark after the keyword PROCEDURE. Any corresponding NEW statement must then supply the initializer's actual parameters immediately after the traditional pointer parameter.

In a single–process environment like Oberon System 3 there is no direct use of record bodies. However, type–local procedures and the initializer concept still make perfect sense. Type–local procedures can beneficially be used as methods, because the ordinary object–oriented rules for inheritance and covering apply. The actual benefits in comparison with the original Oberon language are that (a) neither a "self"–parameter nor any qualification for the access of type–local data is needed and that (b) methods are type–centered rather than instance–centered.

Example:

```
TYPE
  RP = POINTER TO R;
  R = RECORD
    VAR a, b: INTEGER;
    PROCEDURE P (i: INTEGER): BOOLEAN;
    BEGIN RETURN i <= a + b
    END P;
    PROCEDURE& Q (a0, b0: INTEGER);
    BEGIN a := a0; b := b0
    END Q;
  END R;

VAR r: RP; a0, b0: INTEGER;

NEW(r, a0, b0)
```

B.2 Dynamic Arrays

Unless they appear as open procedure parameters, arrays in Oberon need a specified size at compile time. In our extended language, we allow two kinds of dynamic arrays.

The first kind still requires an explicit size specification in the declaration that, however, may now be a variable expression. Because the value of this expression must be well–defined at scope activation time, this kind of dynamic array is applicable within local scopes only.

For the second kind of dynamic array no explicit size specification is needed at declaration time. Syntactically, the size expression is replaced by a "*". However, before using such an array it has to be created explicitly by a NEW statement that specifies the size in the form of an expression. For both kinds of dynamic array an arbitrary dimensionality is possible.

Syntax:

```
Type       = ARRAY [ Interval { "," Interval } ] OF Type.
Interval   = Expr | "*".
```

Example:

```
TYPE A = ARRAY * OF REAL;
VAR a: A;
.. NEW(a, 100); ...

TYPE B = ARRAY *, * OF REAL;
VAR i: INTEGER; b: B;
.. NEW(b, i, i+1); ...

TYPE C = ARRAY * OF ARRAY * OF REAL;
VAR i, j: INTEGER; c: C;
.. NEW(c, i+1); ...; NEW(c[i–1], j); ...

PROCEDURE P (n, i: INTEGER);
  VAR a: ARRAY n+1 OF REAL;
BEGIN ...
END P;
```

B.3 Abstract Operators

Oberon supports the definition of abstract data types but lacks a corresponding support for the definition of abstract infix operators. This is unfortunate, because the ordinary procedural notation is clumsy in combination with nesting. Therefore, we allow the overloading of operators by redefinition. Syntactically, a redefinition is identical to a procedure declaration, where the procedure name is replaced by the operator symbol, for example by "*". It should be noted that the identification of operators in the context of an expression is done at compile time, that is, it does not depend on the dynamic types of the operands. The identification algorithm is: (a) identify all matching declarations, i.e. declarations whose formal parameter types are (direct or indirect) base types of the corresponding actual parameter types and (b) select the matching operator that lexicographically minimizes the "type–distance vector", where the components of this vector are the level differences of actual type and corresponding formal type from left to right. Typically, Oberon does not allow structured types for function return values. In the interest of nestable abstract operators, this restriction is removed in our extended compiler.

Syntax:

```
ProcDecl  =  PROCEDURE {ProcTags} (IdentDef | '"'OpDef'"' ["*"])
             [FormalPars] ";" Scope (ident | '"'OpDef'"').
OpDef     =  Relation | AddOp | MulOp | "~".
Relation  =  "=" | "#" | "<" | "<=" | ">" | ">=" | IN.
AddOp     =  "+" | "–" | OR.
MulOp     =  " * " | "/" | DIV | MOD | "&".
```

Example:

```
TYPE
  A = RECORD ... END;
  B = RECORD ... END;
  A1 = RECORD (A) ... END;
  B1 = RECORD (B) ... END;

PROCEDURE "*" (a: A; b: B): A;
BEGIN ... (* implementation 1 *)
END "*";

PROCEDURE "*" (a: A1; b: B): B1;
BEGIN ... (* implementation 2 *)
END "*";

PROCEDURE "*" (a: A1; b: B1): B;
BEGIN ... (* implementation 3 *)
```

```
END "*";

VAR a: A; b: B; a1: A1; b1: B1;

a*b identifies implementation 1
a1*b identifies implementation 2
a*b1 identifies implementation 1
a1*b1 identifies implementation 3
(a*b)*b identifies  implementation 1 twice
a1*(a1*b1) identifies implementations 3 and 2
```

B.4 Additional Compiler Features

1. The Native Oberon compiler has been improved, so that "forward"–declarations are no longer needed. They are, however, still accepted for compatibility reasons.

2. If types RP and R are connected by the pair of declarations (DD), then RP is now consistently regarded by the compiler as a representative of R in type tests and type guards. This allows types of pointer–based records to be kept anonymous and (DD) to be replaced by a single declaration (D).

```
(DD)    TYPE RP = POINTER TO R; R = RECORD ... END;
(D)     TYPE RP = POINTER TO RECORD ... END;
```

3. The language extension described above is a functional superset of Oberon–2. However, in the interest of compatibility with existing Oberon–2 programs, the PC Native Oberon compiler accepts Oberon–2 constructs under the option "\2".

Appendix C

How to get Oberon System 3

Oberon System 3 can be obtained via anonymous file transfer:
ftp://ftp.inf.ethz.ch/pub/Oberon/System3/

The subdirectory names for the different implementations are:

Native	for Intel–based PC
Win32	for Windows 95 and Windows NT (supports long file names)
Win32s	for Windows 3.1, WfWG 3.11, Windows 95 and Windows NT
Linux	for Linux on Intel–based PC
Macintosh	for MacOS on PowerMac and 68K–based Apple Macintosh II

MacOberon can also be obtained from:
ftp://ftp.ics.uci.edu/pub/oberon/System3/Macintosh/

The Oberon home page URL is: http://www.oberon.ethz.ch
The Oberon System 3 home page URL is: http://www.oberon.ethz.ch/system3
The Oberon newsgroup is: news:comp.lang.oberon

For further information, please contact

Institute for Computer Systems
ETH Zentrum
CH–8092 Zürich
Switzerland

Telephone +41 (0) 1 632 7311   Facsimile +41 (0) 1 632 1307
E–mail: oberon@inf.ethz.ch

Bibliography

[Caf96] Max Caflisch. *Die Entstehung der Syntax–Antiqua*
OFFICINA, Mitteilungen des Hauses Schwabe & Co., Basel, 1996.

[Car86] Luca Cardelli. *Building User Interfaces by Direct Manipulation*
Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, vol. 20: 233–240, 1986.

[Cre95] R. B. J. Crelier. *Separate Compilation and Module Extension*
PhD Thesis, Institut für Computersysteme, ETH Zürich, 1995.

[Der96] Jörg Derungs
LayLa – Layout Language – eine Beschreibungssprache für Gadgets
Semesterarbeit, Institut für Computersysteme, ETH Zürich, 1996.

[Ebe87] Hans Eberle
Development and Analysis of a Workstation Computer
PhD Thesis, Institut für Computersysteme, ETH Zürich, 1987.

[FK96] Michael Franz and T. Kistler. *Slim Binaries*
Technical Report No. 96–24, Department of Information and Computer Science, University of California, Irvine; June 1996.

[Fra94] Michael Franz
Code Generation On–The–Fly: A Key to Portable Software
PhD Thesis, Institut für Computersysteme, ETH Zürich, 1994.

[Gut94] Jürg Gutknecht
Oberon System 3: Vision of a Future Software Technology
Springer – Software – Concepts and Tools, 15: 26–33, 1994.

[Mar94] Johannes L. Marais. *Oberon System 3*
Dr. Dobb's Journal, #220: 42–50, October 1994.

[Mar96] Johannes L. Marais
Design and Implementation of a Component Architecture for Oberon
PhD Thesis, Institut für Computersysteme, ETH Zürich, 1996.

[ML97] J.R. Mühlbacher, B. Leisch, B. Kirk, U. Kreuzeder
Oberon–2 – Programming with Windows
Springer Verlag, 1997.

[Mös93] Hanspeter Mössenböck
Object–Oriented Programming in Oberon–2
Springer Verlag, 1993.

[Mös96] Hanspeter Mössenböck and Niklaus Wirth
The Programming Language Oberon
Institut für Computersysteme, ETH Zürich, 1996.
Available in Oberon.Report.Text and in
http://www.ics.uci.edu/~oberon/report.html

[Rei91] Martin Reiser
The Oberon System – User Guide and Programmer's Manual
Addison–Wesley Publishing Company, 1991.

[RW92] Martin Reiser and Niklaus Wirth
Programming in Oberon – Steps beyond Pascal and Modula
Addison–Wesley Publishing Company, 1992.

[Sal95] Patrick Saladin. *Watson – A Smart Browsing Tool*
Semesterarbeit, Institut für Computersysteme, ETH Zürich, 1995.

[Sal96] Patrick Saladin. *Columbus – Die Entwicklung eines neuen Objekt–Inpektors für Oberon System 3 und Gadgets*
Master's Thesis, Institut für Computersysteme, ETH Zürich, 1996.

[WG92] Niklaus Wirth and Jürg Gutknecht
Project Oberon – The Design of an Operating System and Compiler.
Addison–Wesley Publishing Company, 1992.

[Wir88] Niklaus Wirth. *The Programming Language Oberon*
Springer – Software – Practice and Experience, 19(9), 1988.

[Wir73] Niklaus Wirth. *Systematic Programming: An Introduction*
Prentice–Hall, Inc. , 1973

[Zel93] Emil Zeller. *Data Compression Techniques*
Semesterarbeit, Institut für Computersysteme, ETH Zürich, 1993.

[Zel97] Emil Zeller.
Seamless Integration of Online Services in the Oberon Document System – Joint Modular Languages Conference 1997, Linz, Austria.
Springer – Lecture Notes in Computer Science – Vol. 1204, 1997.

An extensive publication list is found at: http://www.oberon.ethz.ch/books.html