The Oberon Guide

by Jürg Gutknecht/ revised May 20, 1994

This document, although outdated, provides a historical perspective on System 3.

First Part: The Original System

Abstract

This guide provides a concise and detailed description of the Oberon system on three different levels: the user's level, the level of programmers of tools, and the level of implementors of new viewer classes. In particular, the guide features a complete documentation of standard commands, a commented series of important interface definitions, and a tutorial collection of Oberon programs exemplifying the typical Oberon programming style.

Table of Contents

Introduction

History

Oberon is simultaneously the name of a project and of its outcome. The project was started by Niklaus Wirth and the author late in 1985 with the goal of developing a modern and portable operating system for personal workstations. Its results are an implementation of the system for the *Ceres* computer and a programming language (see section Literature).

The development of the language Oberon needs perhaps a short justification. It became quite inevitable because the type–system of available languages turned out to be too restrictive to express the desired data model in a natural and safe way. We refer to the report for a definition of the new language, and to the third and fourth chapter of this text for some examples of its application.

Design Principles

For the present, we focus on the *system* Oberon, beginning with a brief overview of its design principles. The underlying dynamic model is extremely simple. There exists a single process acting as a common carrier of multiple tasks. This process repetitively interprets commands, which are the official 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.

The following typical examples indicate the bandwidth covered by the concept of command: Placing the caret, inserting a character into a text, selecting a piece of text, deleting a selected piece of text, applying a new font to a piece of text, searching a pattern in a text, compiling a software module, opening a viewer, backing up a sequence of files to diskette, and displaying a directory. We emphasize that the execution of a command always results in non–volatile information. For example, in the last example, this means that the 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 complete 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 *viewer*, to which its further handling is delegated. A viewer in Oberon is a rectangular area on the screen that can display any kind of data. Most viewers feature a thin frame and a title–bar containing a menu. Any mouse–oriented input is handled by the viewer the mouse points to. Data from the keyboard is immediately passed over to the current so–called *focus–viewer*. 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 viewer type is a very powerful but also quite far–reaching method to extend the Oberon system. It is only appropriate in conjunction with the installation of a new class of displayable objects (for example graphics or tables). The fourth chapter will provide more insight into this topic.

A more moderate way to increase the system's functionality consists in adding new *commands* operating on objects of an already existing class (for example a language compiler operating on text). We shall see in the third chapter 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.

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 by designing and implementing private commands and facilities. Little is "hardwired" in the system. However, there are several general conventions and existing tools. They are presented in the next chapter.

Acknowledgement

I gratefully acknowledge Niklaus Wirth's initiative and willingness for travelling through the adventures of designing and building a new workstation, a new language, and a new operating system, escpecially under the given severe restrictions of personal resources. Without his competence and extraordinary commitment this truly mammoth–sized project could not have been successfully completed. I am also very grateful for the possibility to include an extract of Niklaus Wirth's Draw guide in the first chapter. My thanks further go to the Oberon user's community, in particular to Martin Reiser, Hans–Peter Mössenböck, and Beverly Sanders for their constructive critisism and valuable suggestions for improvements.

User's Guide

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 objects in Oberon.

This manifests itself immediately after system startup, when, besides of the log–viewer, a so–called *tool viewer* is automatically opened on the screen. It contains a list of command names (*commands* in short), some of them followed by parameters. Commands in Oberon are of the 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 middle mouse key. For example, activating the command *Edit.Open* will result in a new viewer showing some default text.

More often than not, the execution of a command is parameterized. For example, the opening of another tool needs the specification of its name, as in *System.Open Edit.Tool* or in *System.Open Net.Tool*. 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. In principle, a text obeying 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 a global star–shaped pointer. Some commands even allow different ways of paramewter specification. For example, if *System.Open* or *Edit.Open* is called with a "↑"–symbol instead of a file name following the command name, then the file name is taken from the most recent selection. In general, a "↑"–symbol following a command name always refers to the current selection.

It is noteworthy that tools are ordinary texts distinguishing themselves from more usual texts only by their structure and contents. In particular, tools are amenable to editing operations. Looking at this differently, we recognize that commands like *Edit.Open 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 *System.Tool* as ancestor and the tools listed in the *System.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.

We now discuss editing operations. Recall first that most commands are interpreted individually by viewers. There are, however, a few more universal operations, which are handled directly by the central system. For example, when you type the *Escape* key, all marks on the display are removed, including caret and text selections. Or, when you type *Ctrl Shift Del* the system immediately terminates the execution of the current command and opens a trap–viewer displaying the state of the interrupted process. Notice that we have just identified you as the reader of this guide with a user of the system. In order to simplify phrasing, we shall henceforth occasionally do so tacitly.

Next we turn to viewers and display–specific operations. Yçou can put your primary display screen into any one of three different modes by hitting one of the function keys F1, F5 and F6. F5 specifies white script on a black background, F6 specifies black script on a white background and F1 turns the display off. Oberon uses a *tiling viewer* system. The display is divided into vertical *tracks*, and each track is further subdivided into *viewers*. 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 reestablished when the overlaying track is later removed.

Although the global screen layout can be changed, we relate our current explanations to the standard layout showing two tracks, a larger *user track* on the left and a narrower *system track* on the right. In principle, viewers are allocated automatically by the respective commands using a little 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 *pointer* or *marker* at the location where you desire the top of the new viewer to be placed. The pointer is a star–shaped marker, and it is placed by moving the mouse to the desired location and then hitting the *Setup* key. In order to change the size of an existing viewer, simply point to its title–bar, press the left mouse key, and move the mouse up or down accordingly. You can also conveniently move a viewer to any different place on the display screen by starting exactly as just explained, then interclicking the middle mouse–key, dragging the mouse to the new location, and releasing all keys there.

Interclicking means clicking (pressing and releasing) a secondary mouse button at an arbitrary time while a primary mouse 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 in the following chapters. Perhaps the easiest and most important rule says that the current command is nullified, if all remaining mouse–keys have been interclicked (not necessarily simultaneously) during the action.

By convention, most viewers are *menu–viewers*. They show a header consisting of a title and a list of selected commands (*menu*). These commands automatically refer to their own viewer. In the case of ordinary text viewers, commands are included from the *System* tool–package and the *Edit* tool–package. *System.Close* removes the viewer, *System.Copy* opens a new viewer displaying the same instance of text, *System.Grow* lets the viewer grow to the size of a full track or of the whole display, *Edit.Locate* locates a text position, and *Edit.Store* stores the text on file. We shall explain these commands in greater detail in the following sections on tools.

First, we fix some terminology and general conventions. We shall call *marked* an object or a location if it is visibly or invisibly marked by the earlier introduced star–shaped pointer. Visibility of the pointer is irrelevant in most cases. As an exception we mention the explicit allocation of a viewer, which requests the pointer to be visible. Note that an explicitly allocated viewer is automatically marked. Also note that the pointer is initially invisible and placed in the lower left corner of the display.

By convention, the title of a viewer is normally either the name of the displayed object or the name of the command that opened this viewer. Further, if a list of names is passed to a command as a parameter, it must be terminated by a symbol other than a name, for example by the character "~" which will be referred to as *list terminator*. By another convention, the upper right corner of the display is reserved for the *log viewer* reporting on the progress and result in the execution of a command. Finally, there are several *default extensions* of file names. Among them are *Text*, *Graph* and *Pict* for file copies of texts, graphics, and bitmap pictures respectively, and *Scn.Fnt* for screen font files.

In the following sections 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.

The Edit Tool Package

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 edit tool package providing an extensible set of powerful editing commands. Nevertheless, several built–in commands are interpreted directly by text frames. They include positionning 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 attributes and, most importantly, executing an arbitrary command which is specified by its name. We should point out that

all of these built–in commands are applicable in particular to menu–bars (which, in fact, are ordinary text frames featuring an inverted background color).

Mouse Commands

Text positionning. In order to reposition a text within a viewer, move the mouse into the viewer's scrolling–zone. This is a vertical bar along the left borderline of about 0.5 cm width. You can scroll forward by pressing the left mouse 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 middle mouse key at the location where you want the crossbeam to be. If you wish the beginning of the text to be displayed, you can alternatively simply click the right mouse key anywhere within the scroll–bar.

Placing the caret. If you want to place the caret, move the mouse to the desired text, press the left mouse button and, while keeping it down, move the caret to the desired position. Any subsequently typed characters are then inserted at this position.

Selecting text. You can select any contiguous stretch of text by moving the mouse to the desired beginning, pressing the right mouse button and, while holding it down, dragging the selection to the end. If you click twice at the beginning, the selection is automatically extended to the origin of that text line. If a piece of text is too large to be selectable within a single viewer, use *System.Copy* to open an adjacent second viewer. Then select the beginning of the stretch of text in the upper viewer and its end in the lower viewer separately. Note that a separate selection may be active for each displayed text section, including headers of viewers. If several selections exist simultaneously on the display, commands normally refer to the most recent one or to the most recent ones.

There are the following interesting interclick–variants of caret placing and text selection that combine these marking operations effectively with text editing. Remember the general rule saying that any mouse–controlled operation that is currently under execution can be nullified by interclicking all remaining mouse–keys.

Copying text. If you interclick the middle mouse button while you are placing the caret, the most recent selection is automatically copied to the caret's position as soon as you release the left button. 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 middle mouse key while you are selecting a piece of text, the selected text is copied to the caret's position when you release the select–button. This option is most conveniently used in order to copy a given string to various places. In order to neutralize (undo) any "interclick", simply click the remaining third button while still holding down the primary key.

Copying attributes. If you interclick the right mouse button while you are placing the caret, the *character attributes* (font, color, vertical offset) of the character by the caret is automatically applied to the most recent selection as soon as you release the left button.

Deleting text. If you click the left mouse–button while selecting a text, the selected text is eventually deleted. Here also, the interclick can be undone by an additional interclick of the middle mouse–key.

Notice that the above described editing operations are applicable to a header of a viewer only restrictedly. A header cannot be changed. Nor can the caret be placed in a header. Further notice that the copy variant and the delete variant of the select command apply also in the case of large selections involving split viewers.

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 middle mouse key. Sometimes (e.g. in a test 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, simply interclick the left key while you are pressing the middle mouse key and pointing to the command's name.

If command execution fails, the system falls into a *trap*. There is no interactive debugger currently available under Oberon. However, a trap handler is automatically called whenever a trap has occurred. It displays the state of the interrupted process, including the stack of procedure activations.

The following table summarizes the basic meanings of the three mouse–keys: The left key is the *point–key*. It is used to focus a certain location, i.e. to place the caret. The middle key is the *execute–key*. Pressing and releasing it causes the appropriate command interpreter to be called. The right key is the *select–key*. It is used to select objects within a viewer.

Summary of mouse commands

primary button	left	middle	right	secondary button
none	set caret	execute	select	
left	–	free & execute	delete	
middle	copy selection	–	copy this selection	
right	copy attributes	–	–	

Text viewers display text in a standard line–oriented way. In particular, they do not support any non–trivial formatting, such as automatic line–breaking or right–justifying paragraphs, for example. Font variation, color specification, and vertical offset, are however possible. We recommend the following fonts to be used in connection with text viewers: *Oberon10.Scn.Fnt* (default font), *Oberon10i.Scn.Fnt* (italics variant), *Oberon10b.Scn.Fnt* (bold face variant), and *Courier8.Scn.Fnt* (monospaced font).

According to Oberon's basic scheme, additional functionality is provided by the text edit tool package. It contains the following commands.

Edit Commands

Edit.Open
opens a viewer in the user track displaying the specified text. The text 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. If none exists, a default name is taken. In order to override automatic allocation, place the pointer anywhere on the screen.

Edit.Show M.X
opens a viewer in the user track displaying the specified object X of module M. If the implementation of M is available, the implementation of X is shown, otherwise X's definition is displayed.

Edit.Store
writes the text in the marked viewer to the file with the name defined by the parameter, or, if called from the menu line of a text viewer, writes the displayed text to the file with the name of the viewer.

Edit.Recall
inserts the most recently deleted piece of text at the position of the caret.

Edit.CopyFont
transfers the font from the marked location to the most recent text selection.

Edit.ChangeFont
applies the font specified by the parameter to the most recent text selection.

Edit.ChangeColor
applies the color specified by the parameter to the most recent text selection.

Edit.ChangeOffset
applies the vertical offset specified by the parameter to the most recent text selection.

Edit.Search
searches a pattern in the marked 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. The initial value of the pattern is the space character.

Edit.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.

Edit.Print ServerName ["%"] { (TextFileName | "") ["/" NofCopies] }*
sends all texts specified by the parameter list to the print server whose name is taken from the first entry in the parameter list. Names in the parameter list refer to text files, the symbol * to the text in the marked viewer. The symbol % specifies the vanilla–print option. If active, the texts are printed in a single monospaced small Font (*Gacha10l*). This option is typically used for printing source program listings. *NofCopies* optionally specifies the desired number of copies. It must be a single–digit number. This command assumes that the correct user identification has previously been installed (by calling *System.SetUser*).

The System Tool Package

This package features system related commands. Among them are procedures to display tools in the form of viewers, to close viewers and tracks, to define and display all kinds of system oriented parameters, and to provide basic utilities for the manipulation of named files. In addition, the system tool package contains a trap handler that is implicitly called after a trap has occured. It displays the current state of the system stack.

System.Open
opens a viewer in the system track displaying the specified tool. The tool is alternatively specified by a parameter on the command line or, in the case of a "↑" following the command name, by the most recent selection of a name. If none exists, a default name is taken..

System.OpenLog
opens a viewer in the system track displaying the system–wide log. Notice that the log is updated regardless of its visibility. Therefore, the log always shows the complete history of the current session.

System.Copy
opens a copy of the original viewer displaying the same instance of contents.

System.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. Note that the original constellation will be reestablished when the grown viewer is later removed.

System.Close
removes the marked viewer from the display, or, if called from the menu line of a text viewer, removes the own viewer.

System.CloseTrack
closes the marked track, i.e. removes all viewers in this track.

System.Recall
reopens the most–recently (perhaps erronously) closed viewer.

System.Time
displays the current date and time. If date and time parameters *dd.mm.yy hh.mm.ss* (day, month, year and hour, minute, second) immediately follow the command name, *System.Time* first sets date and time accordingly.

System.Watch
displays the amount of currently used disk space and memory resources.

System.Collect
initiates a subsequent garbage collection.

System.Free
unloads every module specified by the parameter list. If a module name is immediately followed by *, imported modules are also unloaded.

System.ShowModules
displays a map of all currently loaded modules.

System.ShowCommands ModName
displays a list of all commands exported by this module.

System.State ModName
displays the global data of the specified module.

System.SetUser
accepts the user's identification in the form *UserName "/" Password* without echoing it on the display. *UserName* is up to eight characters long (initials in most cases). The password is an arbitrary string.

System.Directory
displays the selection of all disk files whose name match the template specified by the parameter. The parameter is a string and may contain the symbol "*" as a wildcard. If option "d" is specified ("/d" immediately following the parameter) additional information about file sizes and dates is displayed. In the case of a "↑" following the command name the parameter is taken from the current selection.

System.CopyFiles
processes a parameter list of pairs *A => B*. Copies each file A to B. In the case of a "↑" following the command name the parameter is taken from the current selection.

System.RenameFiles
processes a parameter list of pairs A => B. Renames each file A to B. In the case of a "↑" following the command name the parameter is taken from the current selection.

System.DeleteFiles
deletes each file specified by the parameter list. In the case of a "↑" following the command name the parameter (a single file name) is taken from the current selection.

System.SetFont
applies the font specified by the parameter to subsequently typed characters.

System.SetColor
applies the color specified by the parameter to subsequently typed characters.

System.SetOffset
applies the vertical offset specified by the parameter to subsequently typed characters.

The CompilerTool Package

This package contains the Oberon compiler. The result of compilations is shown in the log viewer. Possible errors are listed in the log viewer together with their position in the source text. In order to locate the error within the source text, use

Edit.Locate in the log viewer's title menu. The compiler tool exports a single command:

Compiler.Compile
compiles all texts specified by the parameter list. Names in the parameter list refer to text files, the symbol * to the text in the marked viewer. In the case of a "↑" following the command name the parameter list is taken from the current selection. The following options are available: "/x" (index check off), "/o" (integer overflow off), /"t" (type guards off), "/s" (allow change of symbol file) and "/d" (locate trapped statement).

The Miscellaneous Tool Package

This package provides utilities to convert files and determine statistical data.

Miscellaneous.CountLines
counts the lines of all text files specified by the parameter list. "↑" following the command name refers to the current selection (a single file name).

Miscellaneous.GetObjSize
extracts code size and data size from all object files specified by the parameter list. "↑" following the command name refers to the current selection (a single file name).

Miscellaneous.Snapshot name
takes a snapshot of the current contents of the screen and saves it on file under the given name.

Miscellaneous.MakeAscii pairlist ~
converts a set of (possibly formatted) texts to pure Ascii. The parameter is a list of pairs *fromName => toName* .

Miscellaneous.MakeDOSText pairlist ~
converts a set of (possibly formatted) texts to DOS Ascii texts. The parameter is a list of pairs *fromName => toName*.

Miscellaneous.MakeOberonText pairlist ~
converts a set of DOS texts to Oberon texts. The parameter is a list of pairs *fromName => toName*.

Miscellaneous.Do namelist ~
executes a series of commands (implements "command files").

The Backup Tool Package

This package handles the transfer between main disk and diskette.

Backup.Directory[/d]
displays the directory of the currently loaded diskette. Option /d gives extended information about the size of the files and about the date and time of their creation.

Backup.DeleteFiles namelist ~
deletes all files specified by the parameter list from the currently loaded diskette.

Backup.ReadAll
reads all files from the currently loaded diskette.

Backup.ReadFiles namelist ~
reads all files specified by the parameter list from the currently loaded diskette.

Backup.WriteFiles namelist ~
writes all files specified by the parameter list to the currently loaded diskette.

Backup.Format[/H]
formats a diskette to normal density (DD) or high density (HD). Option /H specifies high density.

Backup.InitDOS name
initializes a standard DOS diskette with a volume label.

Backup.InitOberon name
initializes an Oberon diskette with a volume label. The empty name generates compatibility with earlier system versions.

The Net Tool Package (excerpt)

This package handles communication between two computers connected via network. It assumes that the correct user identification has been installed by calling *System.SetUser*.

Net.Directory server prefix

displays a list of all files on the server with specified prefix.

Net.ReceiveFiles server files ~
gets a sequence of files from a remote file server station. The first name in the parameter list identifies the desired file server and the remaining names define the sequence of desired files.

Net.SendMsg partner "message"
sends a message to the specified partner.

Net.GetTime server
gets the time from the time server and adjusts the local clock.

Guide for Programmers of Commands

In Oberon's modular hierarchy we recognize the following structural entities: The *inner core*, the *outer core*, the *text system*, the *graphic system*, the *picture system*, and a collection of *tools*.

Oberon's module hierarchy

```
Net  Backup  Compiler   System   Miscellaneous   ColorSystem
                        Edit          Draw           Paint
                        TextFrames    GraphicFrames  PictureFrames
                                      Graphics       Pictures
                        MenuViewers
                         Outer Core
                   Printer        Oberon
              Inner Core                Texts
                   Modules  Fonts
                   Files
                   FileDir  Math   MathL  Reals  Viewers
Kernel  Diskette        V24   SCC   Input      Display
```

The responsability of the inner core comprises *memory management*, *file management*, and *program loading*. The outer core additionally provides *device drivers* for network ports, *keyboard*, *mouse*, and *display screens*. Other parts of the outer core are *viewer manager*, elementary *text management*, and support for (remote) *printing*. Module *Oberon* represents both the *task dispatcher* and the main *interface* between the outer core and its clients. It includes sections that are devoted to the current system configuration, to default strategies for track allocation and viewer placement, and to the support of command execution.

Module *Display* stands at the bottom of the display system hierarchy. The display area is considered as a plane with x and y coordinates. It includes both a black–and–white area and a color area. *Raster operations* are used to generate and copy rectangular areas on the display plane. Sections of the plane can be made visible by display control procedures. The visible parts of the display plane are structured as tracks and viewers, and they are managed by the viewer manager *Viewers*. Module *Oberon* defines a standard layout featuring one user track and one system track per display screen. Finally, module *MenuViewers* is a high–level viewer manager for standard menu–viewers consisting of a title bar and a rectangular main area surrounded by a thin frame. Both title bar and main area are so–called *frames*. While the title bar is almost always a text frame (see next paragraph), the type of the main frame depends on the kind of viewer.

The *text system*, the *graphic system*, and the *picture system* are identical in their structure. Each consists of a triple of linearly dependent modules. In the case of texts they are called *Texts*, *TextFrames*, and *Edit*. *Texts* defines the object type *Text* and exports intrinsic operations on texts. *TextFrames* defines the object type *TextFrames.Frame* and handles representations of texts within sub–frames of viewers. *Edit* provides additional (non–built–in) text–editing operations.

Modules at the top (like *Edit*) are *tool packages*. Typically, a tool package merely exports a collection of commands in the form of parameterless procedures. Tool modules make intensive use of facilities provided by lower level modules, in particular by the viewer system, the text system, and the central system module Oberon. It is essential that usual commands strictly operate on texts or graphics instead of accessing keyboard or screen directly.

We understand this chapter partly as a tutorial on implementing tool packages. First, we now give a commented overview of the definitions of the most important lower–level modules. Then, we shall exemplify their usage by some typical excerpts from existing tools.

The Display System

```
DEFINITION Display; (*display driver*)
  CONST black = 0; white = 15;
    replace = 0; paint = 1; invert = 2; (*operation modes*)
  TYPE
    Frame = POINTER TO FrameDesc;
    FrameMsg = RECORD END; (*base type of messages to frames*)
    Pattern = LONGINT; (*pointer to pattern descriptor*)
    (*PatternDesc = RECORD
        w, h: SHORTINT;
```

```
      raster: ARRAY (w + 7) DIV 8 * h OF BYTE
      END*)
   Font = POINTER TO Bytes;
   Bytes = RECORD END;
   Handler = PROCEDURE (Frame, VAR FrameMsg);
   FrameDesc = RECORD (*base type of frames*)
      dsc, next: Frame;
       X, Y, W, H: INTEGER;
      handle: Handler
    END;
   VAR
    Unit: LONGINT; (*RasterUnit = Unit/36000 mm*)
     Left,          (*left margin of black–and–white maps*)
     ColLeft,       (*left margin of color maps*)
     Bottom,        (*Bottom of primary map*)
     UBottom,    (*Bottom of secondary map*)
     Width,       (*map width*)
     Height:       (*map height*)
        INTEGER;
    arrow, star, cross, downArrow, hook: Pattern;

   PROCEDURE Map (X: INTEGER): LONGINT; (*address of map at X*)
   PROCEDURE SetMode (X: INTEGER; s: SET); (*set mode at X*)
 (*color display*)
   PROCEDURE SetColor (col, red, green, blue: INTEGER);
   PROCEDURE GetColor (col: INTEGER; VAR r, g, b: INTEGER);
   PROCEDURE SetCursor(mode: SET);
   PROCEDURE InitCC;
   PROCEDURE InitCP;
   PROCEDURE DefCC (X, Y, W, H: INTEGER);
   PROCEDURE DefCP (VAR raster: ARRAY OF BYTE);
   PROCEDURE DrawCX (X, Y: INTEGER);
   PROCEDURE FadeCX (X, Y: INTEGER);  (*fade color cursor at X, Y*)
 (*fonts*)
   PROCEDURE GetChar(f: Font; ch: CHAR; VAR dx, x, y, w, h: INTEGER;
        VAR p: Pattern); (*get box x, y, w, h, width dx, and raster data p
        of character ch in font f*)
 (*raster operations*)
   PROCEDURE CopyBlock (SX, SY, W, H, DX, DY, mode: INTEGER);
        (*copy source block SX, SY, W, H to destination DX, DY using
        operation mode. A block is given by its lower left corner X, Y
        and its dimension W, H*)
   PROCEDURE CopyPattern (col: INTEGER; pat: Pattern; X, Y,
        mode: INTEGER);
        (*copy pattern p in color col to X, Y using operation mode
        col = 0: black; col = 15: white*)
    PROCEDURE ReplPattern (col: INTEGER; pat: Pattern; X, Y, W, H,
        mode: INTEGER);
        (*replicate pattern p in color col into block X, Y, W, H
        suing operation mode, proceeding from left to right and
        from bottom to top, starting at lower left corner*)
   PROCEDURE ReplConst (col: INTEGER; X, Y, W, H, mode: INTEGER);
        (*place "ones" in color col into block X, Y, W, H using operation
        mode*)
END Display.
```

Remarks:

1. If a computer features a monochrome display, then its position (lower left corner) is specified by the variables *Left* and *Bottom*, and whose width and height are given by the variables *Width* and *Height*.

2. If a color display is installed, the module's raster procedures can be used to generate and copy areas on the color screen. The position of the color area (lower left corner) is specified by the variables *ColLeft* and *Bottom*; its width and height are the same as for the monochrome display.

3. The postulated preconditions upon procedure parameters are not checked by the module; this is left to the calling modules which are held responsible for robustness.

```
––––––––––––––––––––––––––––––––––––––––––––––––––––––––––DEFINITIONViewers; (*viewer
manager*)
  IMPORT Display;
```

```
  CONST
    restore = 0; modify = 1; suspend = 2;
      (*message ids referring to the following message type*)
  TYPE
      Message = RECORD (*message sent to viewers on viewer events*)
          (Display.FrameMsg)
          id: INTEGER;
          X, Y, W, H: INTEGER;
          state: INTEGER
        END;
    Viewer = POINTER TO ViewerDesc;
    ViewerDesc = RECORD
        (*viewer descriptor extends Display.FrameDesc*)
        (Display.FrameDesc)
          state: INTEGER
      END;
    (*state > 1: displayed
        state = 1: filler
        state = 0: closed
      state < 0: suspended*)
  VAR curW, minH: INTEGER;
    (*current width of logical display, minimum viewer height*)
  PROCEDURE InitTrack (W, H: INTEGER; Filler: Viewer);
    (*append to current logical display and init track of width W
    and height H and install Filler*)
  PROCEDURE OpenTrack (X, W: INTEGER; Filler: Viewer);
    (*open new track overlaying span of [X, X + W[*)
  PROCEDURE CloseTrack (X: INTEGER);
    (*close track at X and restore overlaid tracks*)
  PROCEDURE Locate (X, H: INTEGER; VAR fil, bot, alt, max: Display.Frame);
    (*in the track at X locate the following viewers:
      filler fil, bottom viewer bot,
      an alternative viewer alt of height >= H,
      viewer max of maximum height*)
  PROCEDURE Open (V: Viewer; X, Y: INTEGER);
    (*open new viewer V with top at Y in track at X*)
  PROCEDURE Change (V: Viewer; Y: INTEGER);
    (*expand or shrink viewer V to new top Y*)
  PROCEDURE Close (V: Viewer);
    (*remove viewer V from the display*)
  PROCEDURE Recall (VAR V: Viewer);
    (*recall most recently closed viewer*)
  PROCEDURE This (X, Y: INTEGER): Viewer;
    (*return viewer at X, Y*)
  PROCEDURE Next (V: Viewer): Viewer;
    (*return next upper neighbour of V*)
  PROCEDURE Broadcast (VAR M: Display.FrameMsg);
    (*send message M to all visible viewers*)
END Viewers.

-------------------------------------------------DEFINITIONMenuViewers;

  IMPORT Display, Viewers;

  CONST extend = 0; reduce = 1; (*message ids*)

  TYPE
    Viewer = POINTER TO ViewerDesc;

  ViewerDesc = RECORD (Viewers.ViewerDesc)
      menuH: INTEGER (*height of menu frame*)
    END;

   ModifyMsg = RECORD (Display.FrameMsg)
      id: INTEGER; (*extend or reduce*)
      dY, Y, H: INTEGER (*translation vector dY; new Y and H*)
    END;

  VAR Ancestor: Viewer; (*current menu viewer*)

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

```
    (*standard handler for menu viewers*)


  PROCEDURE New (Menu, Main: Display.Frame; menuH,
    X, Y: INTEGER): Viewer; (*create and open at X, Y new
    menu viewer containing frames Menu and Main*)

END MenuViewers.
```

Remark:

Messages to menu viewers not affexting size and position are passed on to their subframes. The ancestor viewer is made available to the subframe handlers via the variable Ancestor. MenuViewers also creates new messages of type ModifyMsg requesting subframes to change size or vertical position (or both). dY represents a vertical translation vector, and Y and H specify the new position and height respectively.

The Text System

```
DEFINITION Fonts; (*font loader*)
  IMPORT Display;
  TYPE
    Name = ARRAY 32 OF CHAR;
    Font = POINTER TO FontDesc;
    FontDesc = RECORD
      name: Name; (*file name*)
      height, minX, maxX, minY, maxY: INTEGER;
        (*characteristic data*)
      raster: Display.Font (*raster data*)
    END;
    (*height = minimum distance between text lines,
    minX, maxX, minY, maxY are minima and maxima of X and Y,
    if all character boxes of the font are placed at the origin 0, 0*)

  VAR Default: Font; (*the default font*)

  PROCEDURE This (name: ARRAY OF CHAR): Font;
    (*font with name given*)
END Fonts.

_____________________________________________________DEFINITIONTexts; (*text manager*)
  IMPORT Files, Fonts;
  CONST
    (*symbol classes, see def. of type Scanner*)
    Inval = 0;        (*invalid symbol*)
    Name = 1;         (*name s (length len)*)
    String = 2;       (*literal string s (length len)*)
    Int = 3;          (*integer i (decimal or hexadecimal)*)
    Real = 4;         (*real number x*)
    LongReal = 5;     (*long real number y*)
    Char = 6;         (*special character c*)
    replace = 0; insert = 1; delete = 2; (*op-codes*)
  TYPE
    Text = POINTER TO TextDesc;
    Notifier = PROCEDURE (T: Text; op: INTEGER; beg, end: LONGINT);
    TextDesc = RECORD
      len: LONGINT; (*text length*)
      notify: Notifier (*of editing operations*)
    END;
    Reader = RECORD
      (Files.Rider)
      eot: BOOLEAN;
      fnt: Fonts.Font; (*font of current character*)
      col: SHORTINT; (*color of current character*)
      voff: SHORTINT(*vertical offset*)
    END;
    Scanner = RECORD
      (Reader)
      nextCh: CHAR;
      line: INTEGER;
      class: INTEGER;
```

```
    i: LONGINT;
    x: REAL;
    y: LONGREAL;
    c: CHAR;
    len: SHORTINT;
    s: ARRAY 32 OF CHAR
  END;
   (*used to convert a text into a stream of symbols.
   Symbol classes are defined under CONST*)

  Buffer = POINTER TO BufDesc;
  BufDesc = RECORD
    len: LONGINT (*buffer length*)
  END;

  (*used to write a stream of textual data in a buffer*)
  (*used to store a stretch of a text*)
  Writer = RECORD
    (Files.Rider)
    buf: Buffer; (*associated buffer*)
    fnt: Fonts.Font; (*current font*)
    col: SHORTINT; (*color of current character*)
    voff: SHORTINT (*vertical offset*)
  END;

PROCEDURE Load (T: Text; f: Files.File; pos: LONGINT;
  VAR len: LONGINT);
  (*load text block from file f at position pos to text T*)
PROCEDURE Open (T: Text; name: ARRAY OF CHAR);
  (*open text T from disk file specified by name; open new text
  if name = ""*)
PROCEDURE OpenBuf (B: Buffer);
  (*open new text buffer B*)
PROCEDURE OpenReader (VAR R: Reader; T: Text; pos: LONGINT);
  (*open text reader R and set it up at position pos in text T*)
PROCEDURE Read (VAR R: Reader; VAR ch: CHAR);
  (*read next character in ch*)
PROCEDURE Pos (VAR R: Reader): LONGINT;
  (*return reader's position within its text*)
PROCEDURE Store (T: Text; f: Files.File; pos: LONGINT;
  VAR len: LONGINT);
  (*store text T on disk file f at position pos*)
PROCEDURE Save (T: Text; beg, end: LONGINT; B: Buffer);
  (*append stretch [beg, end[ of text T to buffer B*)
PROCEDURE Copy (SB, DB: Buffer);
  (*append copy of source buffer SB to destination buffer DB*)
PROCEDURE ChangeLooks (T: Text; beg, end: LONGINT; sel: SET;
  fnt: Fonts.Font; col, voff: SHORTINT);
  (*change character attributes within stretch [beg, end[ of text T.
  sel selects attributes to be changed.
   0, 1, 2 IN sel = fnt, col, voff selected*)
PROCEDURE Insert (T: Text; pos: LONGINT; B: Buffer);
  (*insert buffer B in text T at position pos*)
PROCEDURE Append (T: Text; B: Buffer);
  (*append buffer B to text T*)
PROCEDURE Delete (T: Text; beg, end: LONGINT);
  (*delete stretch [beg, end[ of text T*)
PROCEDURE Recall (VAR B: Buffer);
  (*recall previously deleted text*)
PROCEDURE OpenScanner (VAR S: Scanner; T: Text; pos: LONGINT);
  (*open text scanner S and set it up at position pos in text T*)
PROCEDURE Scan (VAR S: Scanner);
  (*read next symbol*)
PROCEDURE OpenWriter (VAR W: Writer);
  (*open new writer W*)
PROCEDURE SetFont (VAR W: Writer; fnt: Fonts.Font);
  (*set writer W to font fnt*)
PROCEDURE SetColor (VAR W: Writer; col: SHORTINT);
  (*set writer W to color col*)
PROCEDURE SetOffset (VAR W: Writer; voff: SHORTINT);
  (*set writer W to vertical offset voff*)
```

```
  PROCEDURE Write (VAR W: Writer; ch: CHAR);
    (*write character ch to W's buffer*)
  PROCEDURE WriteLn (VAR W: Writer);
    (*write end-of-line to W's buffer*)
  PROCEDURE WriteInt (VAR W: Writer; x, n: LONGINT);
    (*write integer x to W's buffer. Right adjust to n positions*)
  PROCEDURE WriteHex (VAR W: Writer; x: LONGINT);
    (*write integer x to W's buffer in hexadecimal form.
  PROCEDURE WriteString (VAR W: Writer; s: ARRAY OF CHAR);
    (*write string s to W's buffer*)
  PROCEDURE WriteReal (VAR W: Writer; x: REAL; n: INTEGER);
    (*write real number x to W's buffer. Use n positions*)
  PROCEDURE WriteRealFix (VAR W: Writer; x: REAL; n, k: INTEGER);
    (*write real number x to W's buffer in fixed-point form,
      using k positions for decimal fractions and n positions in total*)
  PROCEDURE WriteRealHex (VAR W: Writer; x: REAL);
    (*write real number x to W's buffer in hexadecimal form*)
  PROCEDURE WriteLongReal (VAR W: Writer; x: LONGREAL;
    n: INTEGER);
    (*write long real number x to W's buffer. Use n positions*)
  PROCEDURE WriteLongRealHex (VAR W: Writer; x: LONGREAL);
    (*write long real number x to W's buffer in hexadecimal form*)
END Texts.
```

Remark:

Open does not create a text object nor does it install a notifier procedure. Both actions are left to the calling modules. Typically, a calling module first creates a text object (or an extension of it) by using NEW, and then installs a notifier procedure. The main purpose of notifier procedures is requesting the display to re-establish consistency after a change in a text has occurred.

```
-----------------------------------------------------DEFINITIONTextFrames; (*text
display*)
  IMPORT Display, Texts;
  TYPE
    Location = RECORD
      org, pos: LONGINT; (*line origin, position*)
      dx, x, y: INTEGER (*width and position of located character*)
    END;
    Frame = POINTER TO FrameDesc;
    FrameDesc = RECORD
      (Display.FrameDesc)
      text: Texts.Text; (*displayed text*)
      org: LONGINT; (*position in text of first displayed character*)
      col: INTEGER; (*background color*)
      lsp, asr, dsr: INTEGER; (*line spacing, ascender, descender*)
      left, right, top, bot: INTEGER; (*margins*)
      markH: INTEGER; (*margin width, position of mark*)
      time: LONGINT; (*time of latest selection*)
      mark, car, sel: INTEGER; (*state of mark, caret, selection*)
      carloc: Location; (*caret location*)
      selbeg, selend: Location (*locations of begin and end
      of selection*)
    END;
    (*mark < 0: arrow mark
      mark = 0: no mark
      mark > 0: position mark
      car = 0: caret not set
      car > 0: caret set
      sel = 0: no selection active
      sel > 0: selection active*)

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

    VAR menuH, barW, left, right, top, bot, asr, dsr, lsp: INTEGER; (*standard sizes*)
```

```
  PROCEDURE Restore (F: Frame);
     (restore frame F*)
  PROCEDURE Suspend(F: Frame);
     (*suspend frame F*)
  PROCEDURE Extend (F: Frame; newY: INTEGER);
   (*extend frame F to bottom newY*)
  PROCEDURE Reduce (F: Frame; newY: INTEGER);
   (*reduce frame F to bottom newY*)
  PROCEDURE Mark (F: Frame; mark: INTEGER);
   (*mark frame F as specified by mark*)
  PROCEDURE Show (F: Frame; pos: LONGINT);
   (*show text part containing position pos in frame F*)
  PROCEDURE Pos (F: Frame; X, Y: INTEGER): LONGINT;
   (*convert coordinates X, Y to text position*)
  PROCEDURE SetCaret (F: Frame; pos: LONGINT);
   (*set caret in frame F at position pos*)
  PROCEDURE TrackCaret (F: Frame; X, Y: INTEGER;
   VAR keysum: SET);
   (*track caret in frame F, starting from X, Y, and return
   mouse-keys pressed*)
  PROCEDURE RemoveCaret (F: Frame);
   (*remove caret from frame F*)
  PROCEDURE SetSelection (F: Frame; beg, end: LONGINT);
   (*select text stretch [beg, end[ in F*)
  PROCEDURE TrackSelection (F: Frame; X, Y: INTEGER;
   VAR keysum: SET);
   (*track selection in frame F, starting from X, Y, and return
   mouse-keys pressed*)
  PROCEDURE RemoveSelection (F: Frame);
   (*remove selection from frame F*)
  PROCEDURE TrackLine (F: Frame; X, Y: INTEGER;
   VAR org: LONGINT; VAR keysum: SET);
   (*track text line in frame F, starting from X, Y, and return
   line-origin and mouse-keys pressed*)
  PROCEDURE TrackWord (F: Frame; X, Y: INTEGER;
   VAR pos: LONGINT; VAR keysums: SET);
   (*track text word in frame F, starting from X, Y,
  and return starting position and mouse-keys pressed*)
  PROCEDURE Replace (F: Frame; beg, end: LONGINT);
   (*text stretch [beg, end[ was replaced; update frame F*)
  PROCEDURE Insert (F: Frame; beg, end: LONGINT);
   (*text stretch [beg, end[ was inserted; update frame F*)
  PROCEDURE Delete (F: Frame; beg, end: LONGINT);
   (*text stretch [beg, end[ was deleted; update frame F*)

 (*------------------ message handling ------------------*)
  PROCEDURE NotifyDisplay (T: Texts.Text; op: INTEGER;
    beg, end: LONGINT);
   (*notify display manager of text status change*)
  PROCEDURE Call* (F: Frame; pos: LONGINT; new: BOOLEAN);
  (*call command specified at pos in frame F. new forces loading
  of newest version*)
  PROCEDURE Write* (F: Frame; ch: CHAR; fnt: Fonts.Font;
  col, voff: SHORTINT);
  (*write character ch with given attributes at caret position*)
  PROCEDURE Defocus* (F: Frame); (F: Frame; ch: CHAR;
   fnt: Fonts.Font; col, voff: SHORTINT); (*remove caret*)
  PROCEDURE Neutralize* (F: Frame); (*remove marks*)
  PROCEDURE Modify* (F: Frame; id, dY, Y, H: INTEGER);
  (*vertically translate and extend or reduce frame F. id indicates
  type (extension or reduction),
  dy is a translation vector, and Y, H specify new location and
  height respectively*)
  PROCEDURE Open* (
      F: Frame; H: Display.Handler; T: Texts.Text; org: LONGINT;
      col, left, right, top, bot, asr, dsr, lsp: INTEGER);
   (*open new text frame F displaying text T starting from
    position org, with background color col,
    margins left, right, top, bot, and line geometry asr, dsr,
    lsp = ascender, descender line spacing.
```

```
        Install notifier H*)
    PROCEDURE Copy* (F: Frame; VAR F1: Frame);
      (*generate copy F1 of frame F. Initialize to empty frame*)
    PROCEDURE CopyOver* (F: Frame; text: Texts.Text;
      beg, end: LONGINT);
      (*copy over text stretch [beg, end[ to caret position in frame F*)
    PROCEDURE GetSelection* (F: Frame; VAR text: Texts.Text;
      VAR beg, end, time: LONGINT);
      (*get current text selection in frame F (if any)*)
    PROCEDURE Update* (F: Frame; VAR M: UpdateMsg);
      (*update display after editing operation*)
    PROCEDURE Edit* (F: Frame; X, Y: INTEGER; Keys: SET);
      (*track mouse and interpret editing commands*)
    PROCEDURE Handle* (F: Display.Frame;
      VAR M: Display.FrameMsg);
      (*standard handler for text frames*)
    PROCEDURE Text* (name: ARRAY OF CHAR): Texts.Text;
      (*create new displayed text from named file.
      Empty file name means empty text*)
    PROCEDURE NewMenu* (name, commands: ARRAY OF CHAR):
      Frame; (*create new menu frame containing listed commands*)
    PROCEDURE NewText* (text: Texts.Text; pos: LONGINT): Frame;
      (*create new standard text frame*)
END TextFrames.

_____________________________________________________TheOberon Core

DEFINITION Math; (*math library for reals*)
  CONST pi = 3.14159265; e = 2.71828182;
  PROCEDURE sqrt(x: REAL): REAL;
  PROCEDURE exp(x: REAL): REAL;
  PROCEDURE ln(x: REAL): REAL;
  PROCEDURE sin(x: REAL): REAL;
  PROCEDURE cos(x: REAL): REAL;
  PROCEDURE arctan(x: REAL): REAL;
END Math.

_____________________________________________________DEFINITIONMathL; (*math library
for longreals*)
  CONST pi = 3.141592653589793D0; e = 2.718281828459045D0;
  PROCEDURE sqrt(x: LONGREAL): LONGREAL;
  PROCEDURE exp(x: LONGREAL): LONGREAL;
  PROCEDURE ln(x: LONGREAL): LONGREAL;
  PROCEDURE sin(x: LONGREAL): LONGREAL;
  PROCEDURE cos(x: LONGREAL): LONGREAL;
  PROCEDURE arctan(x: LONGREAL): LONGREAL;
END MathL.

_____________________________________________________DEFINITIONFiles; (*file manager*)
  TYPE
    File = POINTER TO Handle;
     (*A file is a sequence of bytes, accessed via (a pointer to) a  handle.
       Files are stored on disk and
       may be referenced through a name entered in the file directory*)
     Rider = RECORD
        res: INTEGER;
        eof: BOOLEAN
      END ;
    (*Elements  of files are accessed through a rider,
      which has a position that is advanced when
      reading or writing data. The position is an integer between
      0 and the length of the file to which the rider is attached.
     The fields eof and res serve as result parameters of
      file procedures.*)

  PROCEDURE Old(name: ARRAY OF CHAR): File;
    (*the file with the given name. NIL if the name is not
    in the directory*)
  PROCEDURE New(name: ARRAY OF CHAR): File;
    (*a new file with given name*)
  PROCEDURE Register(f: File);
```

```
   (*Close file f and register it under its name in the directory.
    If the name exists already, the corresponding old file
    is unregistered*)
  PROCEDURE Close(f: File);
  PROCEDURE Purge(f: File);
  PROCEDURE Length(f: File): LONGINT;  (*the number of bytes
   in the file*)
  PROCEDURE Set(VAR r: Rider; f: File; pos: LONGINT);
   (*Associate rider r with file f at position pos. r.eof := FALSE*)
  PROCEDURE Pos(VAR r: Rider): LONGINT;
  PROCEDURE Read(VAR r: Rider; VAR x: BYTE);
   (*read byte and advance rider by one position.
   If at end, r.eof := TRUE and x := 0X*)
  PROCEDURE ReadBytes(VAR r: Rider; VAR x: ARRAY OF BYTE; n: INTEGER);
   (*read n bytes and advance rider by n positions.
    If at end, r.eof := TRUE and r.res := no. of bytes requested
    but not read.*)
  PROCEDURE ReadBool (VAR R: Rider; VAR x: BOOLEAN);
  PROCEDURE ReadInt (VAR R: Rider; VAR x: INTEGER);
  PROCEDURE ReadLInt (VAR R: Rider; VAR x: LONGINT);
  PROCEDURE ReadLReal (VAR R: Rider; VAR x: LONGREAL);
  PROCEDURE ReadNum (VAR R: Rider; VAR x: LONGINT);
     (* read variable length number *)
  PROCEDURE ReadReal (VAR R: Rider; VAR x: REAL);
  PROCEDURE ReadSet (VAR R: Rider; VAR x: SET);
  PROCEDURE ReadString (VAR R: Rider; VAR x: ARRAY OF CHAR);
  PROCEDURE Write(VAR r: Rider; x: BYTE);
   (*write byte and advance rider by one position*)
  PROCEDURE WriteBytes(VAR r: Rider; VAR x: ARRAY OF BYTE;
   n: INTEGER);
   (*write n bytes and advance rider by n positions*)
  PROCEDURE WriteBool (VAR R: Rider; x: BOOLEAN);
  PROCEDURE WriteInt (VAR R: Rider; x: INTEGER);
  PROCEDURE WriteLInt (VAR R: Rider; x: LONGINT);
  PROCEDURE WriteLReal (VAR R: Rider; x: LONGREAL);
  PROCEDURE WriteNum (VAR R: Rider; x: LONGINT);
   (* write variable length number *)
  PROCEDURE WriteReal (VAR R: Rider; x: REAL);
  PROCEDURE WriteSet (VAR R: Rider; x: SET);
  PROCEDURE WriteString (VAR R: Rider; x: ARRAY OF CHAR);

  PROCEDURE Rename(old, new: ARRAY OF CHAR;
   VAR res: INTEGER); (*res = 0: renamed;
    res = 1: new existed already and now denotes the renamed file;
    res = 2: old name not in directory;
    res = 3: name is illegal;
    res = 4: name is too long *)
  PROCEDURE Delete(name: ARRAY OF CHAR; VAR res: INTEGER);
   (*res = 0: deleted;
    res = 2: name not in directory;
    res = 3: name is illegal;
    res = 4: name is too long *)
END Files.
```

——DEFINITIONDiskette; (*diskette manager*)

```
  TYPE EntryHandler* = PROCEDURE (name: ARRAY OF CHAR; date, time: INTEGER; size: LONGINT);

  VAR res: INTEGER; (*result of file-oriented operation,
    error occurred = (res # 0)*)
    err: SHORTINT; sect: LONGINT; busy: BOOLEAN;
    (*state of device driver*)

  (*device driver*)
  PROCEDURE Reset;
  PROCEDURE GetSector (sec: INTEGER; VAR buf: ARRAY OF BYTE;
    off: INTEGER);
  PROCEDURE PutSector (sec: INTEGER; VAR buf: ARRAY OF BYTE;
    off: INTEGER);
  PROCEDURE Format;
```

```
  (*directory handler*)
  PROCEDURE InitDir (format: CHAR); (*format for future extension*)
  PROCEDURE ReadDir;
  PROCEDURE WriteDir;
  PROCEDURE GetData (VAR date, time, nofFiles,
     nofClusters: INTEGER); (*get volume data*)
  PROCEDURE Enumerate (proc: EntryHandler);

  (*file handler*)
  PROCEDURE ReadAll;
  PROCEDURE ReadFile (name: ARRAY OF CHAR);
  PROCEDURE WriteFile (name: ARRAY OF CHAR);
  PROCEDURE DeleteFile (name: ARRAY OF CHAR);

END Diskette.

_____________________________________________________DEFINITIONInput; (*keyboard and
mouse driver*)
  PROCEDURE Available(): INTEGER;
    (*the number of characters available from the keyboard*)
  PROCEDURE Read (VAR ch: CHAR);
    (*next character from keyboard*)
  PROCEDURE Mouse (VAR keys: SET; VAR x, y: INTEGER);
    (*current coordinates and key setting of mouse.
      0 IN keys = right key pressed
      1 IN keys = middle key pressed
      2 IN keys = left key pressed*)
  PROCEDURE SetMouseLimits (w, h: INTEGER);
    (* define width and height of rectangle in which mouse moves*)
  PROCEDURE Time(): LONGINT;
    (* current system time in units of 1/300 sec*)
END Input.

_____________________________________________________DEFINITIONSCC; (*SCC driver*)
  (*Serial Communications Controller driver module (Zilog Z8530)
    Data are transmitted in blocks. Each block contains two parts: header and data *)
  TYPE Header =
    RECORD valid: BOOLEAN; dadr, sadr, typ: SHORTINT;
      len: INTEGER; (*of data following header*)
      destLink, srcLink: INTEGER  (*link numbers*)
    END ;
  (*dadr is the receiver's machine number, len is the length
    (number of bytes) of
    the data part. typ, destLink, and srcLink are not interpreted by SCC*)
  PROCEDURE Start(filter: BOOLEAN);
    (*initialise the SCC*)
  PROCEDURE Send(VAR head, buf: ARRAY OF BYTE);
    (*send buf[0] ... buf[head.len-1] to head.adr*)
  PROCEDURE Available(): INTEGER;
    (*number of bytes available from receiver buffer.
      Buffer contains stream of
      received bytes, including headers and data parts*)
  PROCEDURE ReceiveHead(VAR head: ARRAY OF BYTE);
    (*read a header from the receiver buffer*)
  PROCEDURE Receive(VAR x: BYTE);
    (*read a byte from the receiver buffer*)
  PROCEDURE Skip(m: INTEGER);
    (*skip m bytes in the receiver buffer*)
  PROCEDURE Stop;  (*turn SCC off*)
END SCC.

_____________________________________________________DEFINITIONPrinter; (*printer
interface*)
    VAR res: INTEGER; (*result*)
        PROCEDURE Open(VAR name, user: ARRAY OF CHAR;
          password: LONGINT);
            (*res = 0: opened, 1: no printer, 2: no link, 3: bad response,
          4: no permission*)

      PROCEDURE Circle (x0, y0, r: INTEGER); (*place circle*)
```

```
    PROCEDURE ContString (VAR s, fname: ARRAY OF CHAR);
    (*place continuation string*)
    PROCEDURE Ellipse (x0, y0, a, b: INTEGER); (*place ellipsis*)
    PROCEDURE Line (x0, y0, x1, y1: INTEGER);
    (*place line of general direction*)
    PROCEDURE Page (nofcopies: INTEGER); (*print current page*)
    PROCEDURE Picture (x, y, w, h, mode: INTEGER; adr: LONGINT);
    (*place picture*)
    PROCEDURE ReplConst (x, y, w, h: INTEGER);
    (*place horizontal or vertical line*)
    PROCEDURE ReplPattern (x, y, w, h, col: INTEGER); (*shade area*)
    PROCEDURE Spline (x0, y0, n, open: INTEGER;
    VAR X, Y: ARRAY OF INTEGER); (*place spline*)
    PROCEDURE String (x, y: INTEGER; VAR s, fname: ARRAY OF CHAR);
    (*place string*)
    PROCEDURE UseListFont (VAR name: ARRAY OF CHAR);

    PROCEDURE Close; (*close connection*)

END Printer.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~DEFINITIONOberon; (*system
manager*)
 IMPORT Display, Viewers, Texts;
 CONST
  consume = 0; track = 1; (*ids for input messages*)
  defocus = 0; neutralize = 1; mark = 2; (*ids for control messages*)

 TYPE
  Painter = PROCEDURE (x, y: INTEGER);
  Marker = RECORD Fade, Draw: Painter END;

  Cursor = RECORD
    on: BOOLEAN; m: Marker; X, Y: INTEGER
  END;

  ParList = POINTER TO ParRec;
  ParRec = RECORD
    vwr: Viewers.Viewer; (*caller's viewer*)
    frame: Display.Frame; (*caller's sub-frame*)
    text: Texts.Text; (*parameter list*)
    pos: LONGINT (*starting position of parameter list*)
  END;

  InputMsg = RECORD
    (Display.FrameMsg)
    id: INTEGER; (*message id*)
    modes, keys: SET; (*current modes and mouse keys*)
    X, Y: INTEGER; (*current location of the mouse*)
    ch: CHAR (*current char*)
  END;

  ControlMsg = RECORD
    (Display.FrameMsg)
    id: INTEGER; (*message id*)
    X, Y: INTEGER (*current location of the mous*)
   END;

  SelectionMsg = RECORD
    (Display.FrameMsg)
    time: LONGINT;
    text: Texts.Text;
    beg, end: LONGINT
  END;

   CopyOverMsg* = RECORD
    (Display.FrameMsg)
    text*: Texts.Text;
    beg*, end*: LONGINT
  END;
```

```
  CopyMsg* = RECORD
    (Display.FrameMsg)
    F*: Display.Frame
  END;

 Task = POINTER TO TaskDesc; (*installable task*)
  Handler = PROCEDURE;
  TaskDesc = RECORD
    safe: BOOLEAN; (*safe tasks are not removed after trap*)
    handle: Handler
  END;

VAR
(*configuration*)
  FocusViewer: Viewers.Viewer; (*current focus viewer*)
  Log: Texts.Text; (*system log text*)
  Par: ParList; (*actual parameters for next command*)
  User: ARRAY 8 OF CHAR; Password: LONGINT; (*current user*)

  CurFnt, CurCol:, CurOff SHORTINT;
    (*current font, color, vertical offset*)
   Arrow, Star: Marker;
   Mouse, Pointer: Cursor;
(*user identification*)
 PROCEDURE SetUser (VAR user, password: ARRAY OF CHAR);
(*clocks*)
  PROCEDURE GetClock (VAR t, d: LONGINT);
  PROCEDURE SetClock (t, d: LONGINT);
  PROCEDURE Time (): LONGINT; (*in units of 1/300 sec*)
(*cursor handling*)
  PROCEDURE OpenCursor (VAR c: Cursor);
  PROCEDURE FadeCursor (VAR c: Cursor);
  PROCEDURE DrawCursor (VAR c: Cursor; VAR m: Marker;
    X, Y: INTEGER);
 (*display management*)
  PROCEDURE OpenDisplay (UW, SW, H: INTEGER);
     (*initialize new display with user track width UW,
     system track width SW, and height H*)
  PROCEDURE DisplayWidth (X: INTEGER): INTEGER;
     (*get width of display at X*)
  PROCEDURE DisplayHeight (X: INTEGER): INTEGER;
     (*get height of display at X*)
   PROCEDURE OpenTrack (X, W: INTEGER);
     (*open a new track of width W at X*)
   PROCEDURE UserTrack (X: INTEGER): INTEGER;
     (*get left margin of user track at X*)
   PROCEDURE SystemTrack (X: INTEGER): INTEGER;
     (*get left margin of system track at X*)
   PROCEDURE AllocateUserViewer (DX: INTEGER;
     VAR X, Y: INTEGER);
     (*allocate new user viewer within display at DX*)
   PROCEDURE AllocateSystemViewer (DX: INTEGER;
     VAR X, Y: INTEGER);
     (*allocate new system viewer within display at DX*)
  PROCEDURE PassFocus (V: Viewers.Viewer);
     (*pass focus to viewer V*)
    PROCEDURE RemoveMarks (X, Y, W, H: INTEGER);
     (*remove marks within given rectangle*)
  PROCEDURE MarkedViewer (): Viewers.Viewer;
     (*returns viewer marked by star–shaped pointer*)
(*command interpretation*)
   PROCEDURE ShowMenu (VAR cmd: INTEGER;
     X, Y: INTEGER; menu: ARRAY OF CHAR);
    (* menu = {command "|"} command.
       Six commands allowed, 6 > cmd >= –1.
       cmd  = 5:  first command selected
       cmd = 0:  last command selected
       cmd = –1:  no selection *)
    PROCEDURE Call (VAR name: ARRAY OF CHAR; par: ParList;
     new: BOOLEAN; VAR res: INTEGER);
     (*call command name and pass parameter list par.
```

```
    Option new requests loading of module.
    Done = (res = 0)*)
  PROCEDURE GetSelection (VAR text: Texts.Text;
    VAR beg, end, time: LONGINT);
  (*get most recent text selection.
    Text selection exists = (time >= 0)*)
    PROCEDURE Install (T: Task);
     (*install new task T*)
  PROCEDURE Remove (T: Task);
    (*remove installed task T*)
  PROCEDURE Collect;
    (*demand garbage collector*)
  PROCEDURE SetFont* (fnt: Fonts.Font);
    (*set current font*)
  PROCEDURE SetColor* (col: SHORTINT);
    (*set current color*)
  PROCEDURE SetOffset* (voff: SHORTINT);
    (*set current vertical offset*)

END Oberon.
```

Remark:

Installed tasks are considered to be *background activities*. They are activated by the central loop when no input events have been detected. For example, the *garbage collector* is implemented as an installed task. Notice that installed tasks may be invalidated after their host module has been unloaded (or replaced). Unsafe tasks are automatically removed after a system trap in order to avoid an infinite repetition of the same error.

Tutorial Examples

Write time stamp to system log

```
PROCEDURE TimeStamp;
BEGIN
  Texts.WriteString(W, "TimeStamp ");
  Texts.WriteInt(W, Oberon.Time(), 1); Texts.WriteLn(W);
  Texts.Append(Oberon.Log, W.buf)
END TimeStamp;
```

where

```
  VAR W: Texts.Writer; is globally defined and initialized
   by Texts.OpenWriter(W).
```

Remarks:

1. Normally, one (global) writer per module is sufficient.

2. If you desire a specific part of the output text to appear in a new font, for example in italics variant *Oberon10i.Scn.Fnt*, call *Texts.SetFont(W,Fonts.This("Oberon10i.Scn.Fnt"))* before writing this part and *Texts.SetFont(W,Fonts.Default)* before continuing to write ordinary text.

Process selected text

```
PROCEDURE CountWords;
  VAR T: Texts.Text; R: Texts.Reader;
    beg, end, pos, time: LONGINT; words: INTEGER; ch: CHAR;
BEGIN words := 0;
 Oberon.GetSelection(T, beg, end, time); (*get most recent selection*)
  IF time >= 0 THEN (*if it exists*)
    Texts.OpenReader(R,T, beg); pos := beg;
      (*setup reader and initialize pos*)
    Texts.Read(R, ch); INC(pos); (*read next character*)
    IF (pos # end) & (ch > " ") THEN
       REPEAT Texts.Read(R, ch); INC(pos)
       UNTIL (pos = end) OR (ch <= " ");
       INC(words)
    END;
    WHILE pos # end DO
       (*(pos # end) & (ch <= " ")*)
       REPEAT Texts.Read(R, ch); INC(pos)
```

```
      UNTIL (pos = end) OR (ch > " ");
      IF pos # end THEN
        REPEAT Texts.Read(R, ch); INC(pos)
        UNTIL (pos = end) OR (ch <= " ");
        INC(words)
      END
    END
  END;
  Texts.WriteString(W, "WordCount = ");
  Texts.WriteInt(W, words, 1); Texts.WriteLn(W);
  Texts.Append(Oberon.Log, W.buf) (*append to system log*)
END CountWords;
```

where again

```
VAR W: Texts.Writer;
```

is globally defined and initialized by *Texts.OpenWriter(W)*.

Open a viewer in system track, generate, and display text data

```
PROCEDURE Directory;
  VAR Menu, Main: TextFrames.Frame; T: Texts.Text;
    V: Viewers.Viewer; X, Y: INTEGER;
BEGIN
  T := TextFrames.Text("");
   (*generate new (and empty) text to be displayed in a frame*)
  Menu := TextFrames.NewMenu("Directory", StandardMenu);
   (*generate standard menu frame*)
  Main := TextFrames.NewText(T, 0);
   (*generate standard text frame*)
  Oberon.AllocateSystemViewer(Oberon.Par.vwr.X, X, Y);
  V := MenuViewers.New(Menu, Main, TextFrames.menuH, X, Y);
   (*open standard menu viewer*)
  TextFrames.Mark(Main, -1); (*setup vertical arrow mark*)
  Diskette.Enumerate(Lister);
   (*pass over Lister-procedure to enumerator*)
  Texts.Append(T, W.buf);
   (*append writer to T and display written text*)
  TextFrames.Mark(Main, 1) (*restore position mark*)
END Directory;
```

where

```
CONST
  StandardMenu =
    "System.Close System.Copy System.Grow Edit.Search Edit.Store";
VAR T: Texts.Text; W: Texts.Writer;
```

are globally defined, W is globally initialized by *Texts.OpenWriter(W)*, and *Lister* is an (upcalled) procedure displaying directory entries:

```
PROCEDURE* Lister (name: ARRAY OF CHAR;
  date, time: INTEGER; size: LONGINT);
BEGIN
  Texts.WriteString(W, name);
  Texts.Write(W, " "); Texts.WriteInt(W, size, 1);
  Texts.Write(W, " "); Texts.WriteDate(W, time, date);
  Texts.WriteLn(W)
END Lister;
```

Remarks:

1. The above program generates its whole output text before displaying it. Alternatively, if you move the statement *Texts.Append(T, W.buf)* into the *Lister*-procedure, every generated directory entry is displayed immediately.

2. *Oberon.AllocateSystemViewer(Oberon.Par.vwr.X, X, Y)* is a standard proposal for the placing of a new system viewer within the track from which the command was called. Of course, individual algorithms are possible as well. For example, if the new viewer is desired to cover the bottom most viewer, except if the pointer overrides this, the algorithm is

```
PROCEDURE AllocateSystemViewer (DX: INTEGER;
  VAR X, Y: INTEGER);
  VAR bot: Viewers.Viewer;
```

```
BEGIN
  IF Oberon.Pointer.on THEN
    X := Oberon.Pointer.X; Y := Oberon.Pointer.Y
  ELSE bot := Viewers.This(Oberon.SystemTrack(DX),0);
    X := bot.X; Y := bot.H – Viewers.minH
  END
END AllocateSystemViewer;
```

3. *TextFrames.NewText* generates a standard text frame. The following statement sequence produce a text frame with an individual handler and a customized geometry.

```
NEW(F);
Open(F, Handle, text, pos, col, left, right, top, bot, asr, dsr, lsp);
```

where F is of type *TextFrames.Frame*.

Open a viewer in user track and display existing text

```
PROCEDURE OpenText;
  VAR par: Oberon.ParList; Text: TextFrames.Frame; S: Texts.Scanner;
  V: Viewers.Viewer; X, Y: INTEGER;
BEGIN
  par := Oberon.Par; (*access parameters*)
  Text := par.frame(TextFrames.Frame); (*calling frame*)
  TextFrames.Mark(Text, –1); (*arrow mark*)
  Texts.OpenScanner(S, par.text, par.pos);
    (*open scanner at position of parameter list*)
  Texts.Scan(S); (*get symbol*)
  IF S.class = Texts.Name THEN
    Oberon.AllocateUserViewer(par.vwr.X, X, Y);
    V := MenuViewers.New(
      TextFrames.NewMenu(S.s, StandardMenu);
      TextFrames.NewText(TextFrames.Text(S.s),0);
      TextFrames.menuH, X, Y);
  END;
  TextFrames.Mark(Text, 1) (*restore position mark*)
END OpenText;
```

Remark:

Oberon.AllocateUserViewer(par.vwr.X, X, Y) is a standard proposal for the placing of a new viewer in the caller's user track. Again, individual algorithms are possible as well.

Grow viewer

```
PROCEDURE Grow;
  VAR V, newV: Viewers.Viewer; M: Oberon.CopyMsg; N: Viewers.ViewerMsg; DH: INTEGER;
BEGIN
  V := Oberon.Par.vwr; (*get originator viewer*)
  DH := Oberon.DisplayHeight(V.X); (*get height of this track*)
  IF V.H < Oberon.DisplayHeight(V.X) THEN (*if viewer is small*)
    Oberon.OpenTrack(V.X, V.W); (*open overlaying track*)
    V.handle(V, M); newV := M.F(Viewers.Viewer);
    (*get a copy of the viewer*)
    Viewers.Open(newV, V.X, DH); (*open new big viewer*)
    N.id := Viewers.restore; newV.handle(newV, N)
    (*ask new viewer to draw itself*)
  END
END Grow;
```

Remark:

The *Grow* command is generic in the sense that it can handle viewer instances of any (current or future) class. Typically (and unavoidably) generic commands use message passing instead of ordinary procedure calls. This object–oriented style will be explained in more detail in the next chapter. Also notice that actually a copy of the original viewer is opened in the new track. When this track is being closed later, the original viewer will reappear.

Process viewer text or sequence of texts, depending on context

```
PROCEDURE ProcessText;
  VAR par: Oberon.ParList; Main: TextFrames.Frame;
```

```
    S: Texts.Scanner; T: Texts.Text;
  BEGIN
    par := Oberon.Par; (*access parameters*)
    IF par.frame = par.vwr.dsc THEN (*command in menu frame*)
      IF par.vwr.dsc.next IS TextFrames.Frame THEN
        Main := par.vwr.dsc.next(TextFrames.Frame); (*main text frame*)
        TextFrames.Mark(Main, -1) (*set arrow mark*)
        Process(Main.text); (*process displayed text*)
        TextFrames.Mark(Main, 1) (*restore position mark*)
      END
    ELSE (*command in main text frame*)
      Main := par.frame(TextFrames.Frame);
      TextFrames.Mark(Main, -1) (*set arrow mark*)
      Texts.OpenScanner(S, par.text, par.pos);
        (*open scanner at position of parameter list*)
      Texts.Scan(S); (*get first symbol*)
      WHILE S.class = Texts.Name DO
        Texts.Open(T, S.s); (*open text from file*)
        Process(T); (*process this text*)
        Texts.Scan(S) (*get next symbol*)
      END;
      TextFrames.Mark(Main, 1) (*restore position mark*)
    END
  END ProcessText;
```

Delete selected part of text in marked viewer

```
  PROCEDURE Delete;
    VAR Main: TextFrames.Frame; V: Viewers.Viewer;
  BEGIN
    V := Oberon.MarkedViewer(); (*get marked viewer*)
    Main := V.dsc.next(TextFrames.Frame);
      (*main text frame of marked viewer*)
    IF Main.sel > 0 THEN (*if there exists a selection*)
      Texts.Delete(Main.text, Main.selbeg.pos, Main.selend.pos)
      (*delete text*)
    END
  END Delete;
```

Copy most recently selected text part to caret's position

```
  PROCEDURE CopyText;
    VAR Main: TextFrames.Frame; buf: Texts.Buffer; V: Viewers.Viewer; time: LONGINT;
  BEGIN
    Oberon.GetSelection(T, beg, end, time);
      *get most recent selection*)
    IF time >= 0 THEN (*if it exists*)
      Texts.OpenBuffer(buf);
      Texts.Save(T, beg, end, buf); (*save text in buffer*)
      V := Oberon.FocusViewer; (*get focus viewer*)
      IF (V.dsc # NIL) & (V.dsc.next IS TextFrames.Frame) THEN
        (*if text viewer*)
        Main := V.dsc.next(TextFrames.Frame); (*main text frame*)
        IF Main.car > 0 THEN (*if caret set*)
          Texts.Insert(Main.text, Main.carloc.pos, buf)
          (*insert text at caret's position*)
        END
      END
    END
  END CopyText;
```

Copy font from visibly marked position to text selection

```
  PROCEDURE CopyFont;
    VAR F: TextFrames.Frame; T: Texts.Text; R: Texts.Reader; V: Viewers.Viewer;
      beg, end, time: LONGINT; X, Y: INTEGER; ch: CHAR;
  BEGIN
    Oberon.GetSelection(T, beg, end, time);
      (*get most recent selection*)
    IF (time >= 0) & Oberon.Pointer.on THEN
      (*if found and pointer visible*)
```

```
    X := Oberons.Pointer.X; Y := Oberon.Pointer.Y;
    V := Viewers.This(X, Y); (*marked viewer*)
    IF (V.dsc # NIL) & (V.dsc.next IS TextFrames.Frame) THEN
      F := V.dsc.next(TextFrames.Frame);
      IF (X >= F.X) & (X < F.X + F.W)
      & (Y >= F.Y) & (Y < F.Y + F.H) THEN
        Texts.OpenReader(R, F.text, TextFrames.Pos(F, X, Y));
        (*position reader*)
        Texts.Read(R, ch); (*read marked char*)
        Texts.ChangeLooks(T, beg, end, {0}, R.fnt, 0, 0)
        (*change font alone*)
      END
    END
  END
END CopyFont;
```

Move caret to next character written in italics

```
PROCEDURE SearchItalics;
  VAR Main: TextFrames.Frame; R: Texts.Reader; italic: Fonts.Font; V: Viewers.Viewer;
    pos: LONGINT; ch: CHAR;
BEGIN
  italic := Fonts.This("Oberon10i.Scn.Fnt");
  V := Oberon.FocusViewer; (*get focus viewer*)
  IF (V.dsc # NIL) & (V.dsc.next IS TextFrames.Frame) THEN
    (*if text viewer*)
    Main := V.dsc.next(TextFrames.Frame); (*main text frame*)
    IF Main.car > 0 THEN (*if caret set*)
      Texts.OpenReader(R, Main.text, Main.carloc.pos);
      (*open reader at caret's position*)
      Texts.Read(R, ch);
      WHILE ~R.eot & (R.fnt # italic) DO Texts.Read(R, ch) END;
      (*read char stream*)
      IF ~R.eot THEN (*not end of text*)
        pos := Texts.Pos(R); (*reader's position*)
        TextFrames.RemoveSelection(Main); (*remove all marks*)
        TextFrames.RemoveCaret(Main);
        Oberon.RemoveMarks(Main.X, Main.Y, Main.W, Main.H);
        TextFrames.Show(Main, Max(0, pos - 200));
        (*show text at pos*)
        TextFrames.SetCaret(Main, pos) (*set caret to new position*)
      END
    END
  END
END SearchItalics;
```

where Max is the maximum–function.

Guide for Programmers of new Frame Classes and Viewer Types

Frames as Active Objects

Frames are the basic entities of Oberon's display system. They are more–or–less autonomous rectangular areas on the display screen and may be nested. In particular, the display screen is hierarchically organized like this:

Display > tracks > viewers > data frames

Display, tracks, and viewers are entities to be managed by the viewer handler module *Viewers*. Because of Oberon's tiling approach, all of these frames are totally visible or totally invisible, i.e. there is no partial overlapping on this level. There exists a class of standard viewers called *menu–viewers* (and handled by module *MenuViewers*). Every menu–viewer contains exactly two vertically adjacent data frames: A header frame (including title and menu) and a main data frame. The main frame of a menu–viewer can be regarded as a virtual display terminal representing the user interface to a certain application or task. It may itself be nested, i.e. contain further subframes. Because an individual command interpreter is associated with every frame, implementing a new class of data frames is a most powerful way of adding functionality to the Oberon system.

We have intentionally used the term *class*. As a matter of fact, frames are implemented in Oberon as *objects* in the sense of object–oriented programming. Calls of frame–specific handling procedures are made by system routines in the form of message transfers. In particular, the central *Oberon loop* typically initiates reactions on user–input actions by sending appropriate messages to the respective viewers. We emphasize that employing the object–oriented programming paradigm

in connection with the handling of frames is necessary in order to enable the kernel to call command interpreters whose identity is unknown to it.

Any handler that is associated with a certain viewer class, for example menu–viewers, is expected to handle messages from (at least) three different originators: From the viewer manager, from the document manager, and from the central loop. Messages from the viewer manager report on the state change of the viewer. For example, the viewer manager Viewers sends a message whenever a hidden viewer becomes visible (and must therefore restore its contents), or when the size of a viewer has changed because its lower neighbour was opened, changed, or closed. Messages from the document manager remind the viewer of updating its contents after an editing operation. Finally, messages from module *Oberon* notify the viewer of system–wide events, such as input events (for example, "mouse button i pressed at position X, Y") or an inquiry for the most recent text selection (regarded in Oberon as the standard input). The interpreter–part handling input events should be regarded as an editor that is bound to the viewer class. In the case of viewers displaying text, this is a text editor, in the case of graphics, it is a graphics editor, and in the case of a viewer representing a mailbox it is an editor for electronic mail.

A typical *viewer handler* (like the one associated with menu–viewers) processes viewer–oriented messages directly and delegates the handling of the more data–specific messages by passing them on to the viewer's subframes. Prime examples of viewer–oriented messages are requests to restore a viewer or change its size. Examples of data–specific messages are selecting an object or invoking a command by pointing at its name. Notice that new and finer–grained requests may evolve from processing of viewer–oriented message.s For example, a request to change the size of a menu–viewer is resolved in requests to change the size of one or both of its subframes. In summarizing, viewer handlers normally act both as mediators and originators of messages to be processed by the handlers of their subframes.

The Canonical Decomposition of an Application

Modularizing by separation of concerns is one of the most effective techniques applied to the design of large software systems. Our concept of a class of frames displaying data of a certain kind provides a perfect opportunity to demonstrate this technique. We can extract the following separate topics from the program implementing an interactive application: the *tool package*, the *command interpreter*, the *display handler*, and the *data manager*. The following tasks are assigned to these parts: Providing a collection of powerful and customized commands operating on the data of the given kind (tool package), reacting on input actions within the associated display frame (command interpreter), displaying the visible objects (display handler), and encapsulating the management of the data without any concern of how they are displayed (data manager). Typically, the data part comprises a set of editing operations. It maintains an internal data structure describing the current state of the data.

It is natural to try to assign a separate module to each of these parts. We soon see that the command interpreter part and the display manager are preferably combined in one module because both need to operate on the same associated display frame. This leads to the following canonical module configurations in the cases of standard texts, graphics and pictures:

Tool Package	Edit	Draw	Paint			
ComInt/DispHan	TextFrames	GraphicFrames	PictureFrames	Data Manager	Texts	Graphics
Pictures						

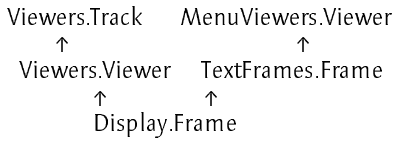
Notice that the same data manager can in principle be used by different display handlers and command interpreters. Data managers serve as links between different applications and thus enable an optimal integration. For example, if the graphic system is based on module *Texts* for the management of text captions, then text can freely be exchanged between frames of these classes: Graphic frames and text frames are integrated.

Tutorial Example: Text Viewers

So far, we have not explained yet how objects and messages are realized in the Oberon language. In contrast to typical object–oriented programming systems, the complete set of messages understood by an object need not be specified together with the definition of the object class. Instead, Oberon explores a more flexible dual approach: Messages are typically defined in *sender* modules. For example, messages of the first of the above mentioned kinds are defined in module *Viewers*, messages of the second kind are defined in the respective editor module (*TextFrames* in the case of text), and messages of the third kind are defined in module *Oberon* which detects input events.

Roughly speaking, the Oberon language supports *subclassing* (by the type extension facility), but messages and message handlers are not institutionalized in the form of a language construct. We have implemented the above outlined scheme by making use of *procedure variables*, and by applying Oberon's *record extension* facility to objects and to messages.

We shall now exemplify this model with the help of standard text–viewers, i.e. menu–viewers whose subframes (menu and main) are text frames. The following exposition may serve as a tutorial and template for implementors of arbitrary frame classes or viewer types. We recall that the display data structure is a hierarchy of display frames. Restricting our explanations to text–viewers we have the following hierarchy of types:



Module *Display* features the base types of frames and frame messages and the type of the message handler. The (empty) base message serves as a root in the (potentially unlimited) hierarchy of messages to be accepted by frames. Module

Viewers provides the definition of type *Viewer*, which is an extension (variant) of type *Display.Frame*. Menu–viewers are a based on *Viewers.Viewer* an defined in module *MenuViewers*.

In definition of *Display*:

```
TYPE
  Frame = POINTER TO FrameDesc;
  FrameMsg = RECORD END;
  Handler = PROCEDURE (Frame, VAR FrameMsg);
  FrameDesc = RECORD
    dsc, next: Frame; (*son, brother*)
    X, Y, W, H: INTEGER;
    handle: Handler
  END;
```

In the definition of *Viewers*:

```
TYPE
  Viewer = POINTER TO ViewerDesc;
  ViewerDesc = RECORD (Display.FrameDesc)
    state: INTEGER
  END;
```

In the definition of *MenuViewers*:

```
TYPE
  Viewer = POINTER TO ViewerDesc;
  ViewerDesc = RECORD (Viewers.ViewerDesc)
    menuH: INTEGER
  END;
```

The following are the declarations of messages that are expected to be handled by every viewer. Note that they are distributed over several modules rather than concentrated in one place (as it would be the case in an "ordinary" object–oriented model).

In definition of *Viewers*:

```
  ViewerMsg = RECORD (Display.FrameMsg)
    id: INTEGER;
    X, Y, W, H: INTEGER; (*new rectangle*)
    state: INTEGER (*new state*)
  END;
  (*signals change of viewer state
    id = 0: restore viewer
         1: modify size at bottom
         2: suspend viewer*)
```

In definition of *TextFrames*:

```
  UpdateMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (*operation*)
    text: Texts.Text; (*edited text*)
    beg, end: LONGINT (*stretch*)
  END;
  (*signals change of contents
    id = 0: stretch [beg, end[ replaced
         1: stretch [beg, end[ inserted
         2: stretch [beg, end[ deleted*)
```

In definition of *Oberon*:

```
  InputMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (*operation*)
    modes, keys: SET; (*mouse*)
    X, Y: INTEGER; (*position*)
    ch: CHAR (*character read*)
  END;
  (*signals input event
    id = 0: track mouse
         1: consume character*)
```

```
ControlMsg = RECORD (Display.FrameMsg)
  id: INTEGER; (*operation*)
  X, Y: INTEGER (*current location of the mous*)
END;
(*signals control action
   id = 0: remove focus
        1: remove all marks
        2: : mark*)
SelectionMsg = RECORD (Display.FrameMsg)
    time: LONGINT;
    text: Texts.Text;
    beg, end: LONGINT
END;
(*signals inquiry for most recent text selection*)

CopyOverMsg = RECORD (Display.FrameMsg)
   text: Texts.Text;
   beg, end: LONGINT
END;
(*receive text stretch [beg, end[*)

CopyMsg* = RECORD (Display.FrameMsg)
   F: Display.Frame
END;
(*request for the creation of a copy of a text frame*)
```

We recall that high–level viewer managers like *MenuViewers* typically pass on most of the received messages to their subframes. In the course of (pre–)processing viewer–oriented requests, high–level viewer managers can also produce new messages. In the case of *MenuViewers* these are

```
ModifyMsg* = RECORD (Display.FrameMsg)
  id: INTEGER; (*operation*)
  dY, Y, H: INTEGER (*vector dY, new Y and H*)
END;
(*vertically move and extend or reduce frame at bottom*)
   id = 0: extend frame
   id = 1: reduce frame*)
```

dY represents a vertical translation vector showing upwards in the extend–case and showing downwards in the reduce–case. By definition, dY is always non–negative. Y and H specify the new position and height respectively.
In summary, essentially the same messages have to be handled by a viewer and its subframes, except that some of the messages are processed or preprocessed by the ancestor viewer already which, in turn, may result in sending new and finer–grained messages to its subframes.

A message is sent to a specific object simply by calling its installed handler. For example, *F.handle(F, M)* has the effect of sending message M to frame F. In case of text–frames, the installed handler is *Handle*. It is elaborated in module *TextFrames*. We now provide a tutorial sketch of this procedure. Notice that *Handle* makes extensive use of Oberon's *type test* (and type guard) facility in order to discriminate among the different message kinds.

```
1  PROCEDURE Handle (F: Display.Frame; VAR M: Display.FrameMsg);
2      VAR F1: Frame;
3  BEGIN
4    WITH F: Frame DO
5     IF M IS Oberon.InputMsg THEN
6       WITH M: Oberon.InputMsg DO
7         IF M.id = Oberon.track THEN Edit(F, M.X, M.Y, M.keys)
8         ELSIF M.id = Oberon.consume THEN
9           IF F.car # 0 THEN Write(F, M.ch, M.fnt, M.col, M.voff) END
12        END
13      END
14    ELSIF M IS Oberon.ControlMsg THEN
15      WITH M: Oberon.ControlMsg DO
16        IF M.id = Oberon.defocus THEN Defocus(F)
17          ELSIF M.id = Oberon.neutralize THEN Neutralize(F)
18        END
19      END
20    ELSIF M IS Oberon.SelectionMsg THEN
21      WITH M: Oberon.SelectionMsg DO
          GetSelection(F, M.text, M.beg, M.end, M.time)
        END
22    ELSIF M IS Oberon.CopyOverMsg THEN
```

```
23      WITH M: Oberon.CopyOverMsg DO
          CopyOver(F, M.text, M.beg, M.end)
        END
24    ELSIF M IS Oberon.CopyMsg THEN
25      WITH M: Oberon.CopyMsg DO Copy(F, F1); M.F := F1 END
26    ELSIF M IS MenuViewers.ModifyMsg THEN
27      WITH M: MenuViewers.ModifyMsg DO
          Modify(F, M.id, M.dY, M.Y, M.H)
        END
28    ELSIF M IS UpdateMsg THEN
29        WITH M: UpdateMsg DO
30        IF F.text = M.text THEN Update(F, M) END
31        END
32      END
33  END
34  END Handle;
```

Explanations:

```
1     procedure of type Display.Handler
2     auxiliary variable for frame copy (line 25).
      Needed because M.F is base type
4     type guard; F must be a text frame
5     type test; is message an Oberon input message?
6     type guard
7     if message demands mouse tracking
8     if message demands consuming then consume a character
9     if caret is active in this text frame
14    type test; is message an Oberon control message?
15    type guard
16    if message demands removing the caret
17    if message demands removing caret and selection
20    type test; is message an Oberon selection inquiry?
21    if so, register own selection if it is newer than candidate
22    type test; is message a copy-over-message?
23    if so, copy text stretch to caret's location in this frame
24    type test; is message a copy-message?
25    if so, produce a copy of this frame (initialized to empty)
26    type test; is message a modify-message?
27    if so, modify size or position of this frame (see below)
28    type test; is message an update message?
30    if so, check if changed text is represented in this frame and then update frame
```

We also provide the following refinement of the procedure *TextFrames.Modify* called in line 27 in *TextFrames.Handle*. It shows well how subframes of menu-viewers should react on a *MenuViewers.ModifyMsg*.

```
PROCEDURE Modify (F: Frame; id, dY, Y, H: INTEGER);
BEGIN
  Mark(F, 0); RemoveMarks(F); (*remove position-bar, caret, and selection*)
  IF id = MenuViewers.extend THEN
    IF dY > 0 THEN (*if frame is to be moved*)
      Display.CopyBlock(F.X, F.Y, F.W, F.H, F.X, F.Y + dY, 0);
      F.Y := F.Y + dY (*move original block*)
    END;
    Extend(F, Y) (*extend moved frame at its bottom*)
  ELSIF id = MenuViewers.reduce THEN
    Reduce(F, Y + dY); (*reduce original frame at its bottom*)
    IF dY > 0 THEN (*if frame is to be moved*)
      Display.CopyBlock(F.X, F.Y, F.W, F.H, F.X, Y, 0); F.Y := Y
      (*move reduced block*)
    END
  END;
  IF F.H > 0 THEN Mark(F, 1) END (*restore position-bar*)
END Modify;
```

Remember that dY is always non-negative, and notice that the reduce-part of the procedure is the exact inverse of the extend-part.

Notice by the way that the above handler is used for the handling of both data-frames and standard header-frames. The two variants of text frames differ only by their background color. This fact testifies for a streamlined system design and is an example of the "today en-vogue" notion of *code reusing*.

The attentative and experienced reader may have noticed that the module *TextFrames* plays two very different roles. Its first role is that of a (late–bound) handler of messages sent to frame instances. The second role is that of a library of procedures operating on text frames. Examples are *Edit*, *Write*, *CopyOverShow*, *SetCaret*, and *TrackWord*. To the two roles of *TextFrames* correspond two different ways of treating text frames, namely as active objects individually reacting on messages, and as passive rectangles to be operated on conventionally by invoking procedures. The second way of treating frames might be of importance in cases where text frames are text boxes belonging to the contents of some more complex document.

Module *TextFrames* offers yet another choice, this time to potential implementors of handlers of subclasses of text frames: Either they implement their own handler by just adding *increments*, and then refer to Handle, or they compose an individual handler from the elements. For example, a designer of *MailFrames* might apply the first strategy. He might just add a few mail–oriented methods catching the mail–oriented messages, and then delegate all further processing to *TextFrames.Handle*. In contrast, an implementor of a new user–interface to text frames might prefer strategy 2.

We conclude this section by explaining briefly the flow of control after, for example, a character has been typed. We assume that the focus viewer is a text viewer and that the caret is set in its main data frame. First, the central loop *Oberon.Loop* detects the new character in the keyboard buffer. It then sends an input consume–message to the focus–viewer which, in turn, passes on this message to its main subframe. After that, the handler (command interpreter) of this subframe locates the caret within the underlying text and subsequently calls the Insert–procedure of the text data manager *Texts*. On that, Insert inserts the character in the text and (up–)calls the associated notifier which, in our case, is residing in *TextFrames*. The notifier then sends a broadcast–message of type *TextFrames.UpdateMsg* to all visible viewers. Every single of these viewers passes on this message to its subframes. If a subframe understands the message and finds itself involved in this update, it restores its contents by accessing the new data, again from the data manager Texts.

We now present some typical examples of implementations.

Tutorial Examples 1: Frame oriented operations

Display text line within text frame

```
PROCEDURE DisplayLine (F: Frame; L: Line; VAR R: Texts.Reader; X, Y: INTEGER; len: LONGINT);
  VAR pat: Display.Pattern; NX, dx, x, y, w, h: INTEGER;
BEGIN NX := F.X + F.W;
  WHILE (nextCh # CarriageReturn) & (R.fnt # NIL) DO
    Display.GetChar(R.fnt.raster, nextCh, dx, x, y, w, h, pat);
    IF (X + x + w <= NX) & (h # 0) THEN
      Display.CopyPattern(R.col, pat, X + x, Y + y, 2)
    END;
    X := X + dx; INC(len); Texts.Read(R, nextCh)
  END;
  L.len := len + 1; L.wid := X + eolW – (F.X + F.margW);
  L.eot := R.fnt = NIL; Texts.Read(R, nextCh)
END DisplayLine;
```

where the types *Line* and *Frame* are defined as follows. Notice that *Line* is a private type and *TextFrames.Frame* is the public projection of type Frame.

```
Line = POINTER TO LineDesc;
LineDesc = RECORD
  len: LONGINT; (*number of characters in this line*)
  wid: INTEGER; (*width of line box*)
  eot: BOOLEAN; (*end of text flag*)
  next: Line (*next lower neighbour*)
END;
Location = RECORD
  org, pos: LONGINT; (*line origin, position*)
  dx, x, y: INTEGER; (*width and position of located character*)
  lin: Line (*associated line*)
END;
Frame = POINTER TO FrameDesc;
FrameDesc = RECORD (Display.FrameDesc)
  text: Texts.Text; (*displayed text*)
  org: LONGINT; (*position in text of first displayed character*)
  col: INTEGER; (*background color*)
  lsp, asr, dsr: INTEGER; (*line spacing, ascender, descender*)
  left, right, top, bot: INTEGER; (*margins*)
  markH: INTEGER; (*margin width, position of mark*)
  time: LONGINT; (*time of latest selection*)
  mark, car, sel: INTEGER; (*state of mark, caret, selection*)
  carloc: Location; (*caret location*)
  selbeg, selend: Location (*locations of begin and end of selection*)
  trailer: Line
```

```
    (*pointer to the associated sequence of line descriptors*)
  END;
```

Track caret

```
PROCEDURE TrackCaret (F: Frame; X, Y: INTEGER; VAR keysum: SET);
  VAR loc: Location; keys: SET;
BEGIN
  IF F.trailer.next # F.trailer THEN
    LocateChar(F, X – F.X, Y – F.Y, F.carloc);
    FlipCaret(F);
    keysum := {};
    LOOP
      Input.Mouse(keys, X, Y);
      IF keys = {} THEN EXIT END;
      keysum := keysum + keys;
      Oberon.DrawCursor(Oberon.Mouse, Oberon.Mouse.marker,
      X, Y);
      LocateChar(F, X – F.X, Y – F.Y, loc);
      IF loc.pos # F.carloc.pos THEN FlipCaret(F);
       F.carloc := loc; FlipCaret(F)
     END
    END;
    F.car := 1
  END
END TrackCaret;
```

where the following auxiliary procedures are used:

```
  PROCEDURE LocateChar (F: Frame; x, y: INTEGER;
  VAR loc: Location);
  VAR R: Texts.Reader;
    pat: Display.Pattern;
    pos, lim: LONGINT;
    ox, dx, u, v, w, h: INTEGER;
BEGIN LocateLine(F, y, loc);
  lim := loc.org + loc.lin.len – 1;
  pos := loc.org; ox := F.left;
  Texts.OpenReader(R, F.text, loc.org); Texts.Read(R, nextCh);
  LOOP
    IF pos = lim THEN dx := eolW; EXIT END;
    Display.GetChar(R.fnt.raster, nextCh, dx, u, v, w, h, pat);
    IF ox + dx > x THEN EXIT END;
    INC(pos); ox := ox + dx; Texts.Read(R, nextCh)
  END;
  loc.pos := pos; loc.dx := dx; loc.x := ox
END LocateChar;

PROCEDURE LocateLine (F: Frame; y: INTEGER; VAR loc: Location);
  VAR T: Texts.Text; L: Line; org: LONGINT; cury: INTEGER;
BEGIN T := F.text;
  org := F.org; L := F.trailer.next; cury := F.H – F.top – F.asr;
  WHILE (L.next # F.trailer) & (cury > y + F.dsr) DO
    org := org + L.len; L := L.next; cury := cury – F.lsp
  END;
  loc.org := org; loc.lin := L; loc.y := cury
END LocateLine;

PROCEDURE FlipCaret (F: Frame);
BEGIN
  IF F.carloc.x < F.W THEN
    IF (F.carloc.y >= 10) & (F.carloc.x + 12 < F.W) THEN
      Display.CopyPattern(white, BigCaret, F.X + F.carloc.x,
        F.Y + F.carloc.y – 10, 2)
    ELSIF (F.carloc.y >= 4) & (F.carloc.x + 6 < F.W) THEN
      Display.CopyPattern(white, SmallCaret, F.X + F.carloc.x,
        F.Y + F.carloc.y – 4, 2)
    END
  END
END FlipCaret;
```

Tutorial Examples 2: Text oriented operations

Save text in buffer

```
PROCEDURE Save (T: Text; beg, end: LONGINT; B: Buffer);
  VAR p, q, qb, qe: Piece; org: LONGINT;
BEGIN
  IF end > T.len THEN end := T.len END;
  FindPiece(T, beg, org, p);
  NEW(qb); qb↑ := p↑;
  qb.len := qb.len – (beg – org);
  qb.off := qb.off + (beg – org);
  qe := qb;
  WHILE end > org + p.len DO
    org := org + p.len; p := p.next;
    NEW(q); q↑ := p↑; qe.next := q; q.prev := qe; qe := q
  END;
  qe.next := NIL; qe.len := qe.len – (org + p.len – end);
  B.last.next := qb; qb.prev := B.last; B.last := qe;
  B.len := B.len + (end – beg)
END Save;
```

where *FindPiece* is the following auxiliary procedure:

```
PROCEDURE FindPiece (T: Text; pos: LONGINT; VAR org: LONGINT;
    VAR p: Piece);
    VAR n: INTEGER;
BEGIN
  IF pos < T.org THEN T.org := –1; T.pce := T.trailer END;
  org := T.org; p := T.pce; (*from cache*)
  n := 0;
  WHILE pos >= org + p.len DO
    org := org + p.len; p := p.next; INC(n)
  END;
  IF n > 50 THEN T.org := org; T.pce := p END (*to cache*)
END FindPiece;
```

and where the types *Piece*, *Text*, and *Buffer* are defined as follows. Notice that *Piece* is a private type, *Texts.Text* is the public projection of type *Text*, and *Texts.Buffer* is the public part of type *Buffer*.

```
Piece = POINTER TO PieceDesc;
PieceDesc = RECORD
    f: Files.File; (*carrier file*)
    off: LONGINT; (*offset in file*)
    len: LONGINT; (*piece length*)
    fnt: Fonts.Font; (*font*)
    col: SHORTINT; (*color*)
    voff: SHORTINT; (*vertical offset*)
    prev, next: Piece (*links to neighbours*)
  END;
  Text = POINTER TO TextDesc;
  Notifier = PROCEDURE (Text, INTEGER, LONGINT, LONGINT);
  TextDesc = RECORD
    len: LONGINT; (*text length*)
    notify: Notifier; (*called after text changes*)
    trailer: Piece; (*sentinel*)
    org: LONGINT; (*cached origin*)
    pce: Piece (*cached piece*)
  END;
  Buffer = POINTER TO BufDesc;
  BufDesc = RECORD
    len: LONGINT; (*buffer length*)
    header, last: Piece (*buffered piece list*)
  END;
```

Insert contents of buffer in text

```
PROCEDURE Insert (T: Text; pos: LONGINT; B: Buffer);
  VAR pl, pr, p, qb, qe: Piece; org: LONGINT;
BEGIN
```

```
    FindPiece(T, pos, org, p); SplitPiece(p, pos – org, pr);
    IF T.org >= org THEN (*adjust cache*)
      T.org := org – p.prev.len; T.pce := p.prev
    END;
    pl := pr.prev; qb := B.header.next;
    IF (qb # NIL) & (qb.f = pl.f) & (qb.off = pl.off + pl.len)
      & (qb.fnt = pl.fnt) THEN pl.len := pl.len + qb.len; qb := qb.next
    END;
    IF qb # NIL THEN qe := B.last;
      qb.prev := pl; pl.next := qb; qe.next := pr; pr.prev := qe
    END;
    T.len := T.len + B.len;
    T.notify(T, insert, pos, pos + B.len); (*call postprocessor*)
    B.last := B.header; B.last.next := NIL; B.len := 0
  END Insert;
```

where *SplitPiece* is

```
 PROCEDURE SplitPiece (p: Piece; off: LONGINT; VAR pr: Piece);
   VAR q: Piece;
 BEGIN
    IF off > 0 THEN NEW(q);
      q.col := p.col; q.fnt := p.fnt;
      q.len := p.len – off;
      q.f := p.f; q.off := p.off + off;
      p.len := off;
      q.next := p.next; p.next := q;
      q.prev := p; q.next.prev := q;
      pr := q
    ELSE pr := p
    END
 END SplitPiece;
```

Literature

Ceres Workstation

H. Eberle. Development and Analysis of a Workstation Computer.
Diss. ETH No. 8431, 1987.
B. Heeb. Design of the Processor–Board for the Ceres–2 Workstation,
Bericht 93, Inst. für Informatik, ETH Zürich, November 1988.

Oberon Language

N. Wirth and M.Reiser: Programming in Oberon.
To be published by Addison–Wesley in early 1992.
N. Wirth. Type Extensions.
ACM Trans. on Prog. Languages and Systems, 10, 2 (April 1988), 204–214.
N. Wirth. From Modula to Oberon.
Software – Practice and Experience, 18, 7, (July 1988), 661–670.
N. Wirth. The Programming Language Oberon.
Software – Practice and Experience, 18, 7, (July 1988), 671– 690.
J. Gutknecht. Variations on the Role of Module Interfaces.
Structured Programming, 10, 1, (Jan. 1989), 40–46.

Oberon System

M. Reiser: The Oberon System, Reference and User's Guide.
Addison Wesley, February 1991.
N. Wirth and J. Gutknecht: Project Oberon.
To be published by Addison–Wesley in early 1992.
N. Wirth: Designing a System from Scratch.
Structured Programming, 10, 1 (Jan. 1989), 10–18.
N. Wirth and J. Gutknecht: The Oberon System.
Software – Practice and Experience, 19, 9 (Sept. 1989), 857–893.

Second Part: Oberon System 3

Introduction and Overview

At the beginning it was a mere vision. A vision with many different facettes, ranging from ordinary texts with integrated pictures and figures to fully grown electronic textbooks, from simple menus and icons to a versatile environment for the use and design of expressive graphical user interfaces, from elementary visual objects all the way to a new paradigm of how to look at software and to construct software, a paradigm that is based on persistent functional units that are interactively usable, editable and composable.

The vision was still rather diffuse when we turned it into a project, and it was by no means clear, if this was really just one project or actually a set of more or less independent projects. Among the more concrete aims, let us for the moment pick the topic "separation of application and user interface programming". The principle we had in mind is simple. It is illustrated in Figure 1 by the example of the simulation of a *waiting line*. The essential point is that the application program (the simulator) is kept absolutely independent of any kind of display (the panel) because it operates on an abstract object (occasionally called model) rather than on any concrete graphical representation (called view).

We decided to realize our ideas by developing *System 3*, an evolutionary version of the original Oberon system as described in Part 1. of this text or in [1]. From our vision and from the mentioned model–view example it followed immediately that System 3 needs to be settled one important step beyond the ordinary state of object–oriented programming, because it needs to be able to organize and manage hundreds or even thousands of *persistent objects* and components that appear in extensive variety, including at least character patterns, formulae of various types, all kinds of figures, pictures, visual user interface elements of different complexity and abstract models. We soon realized that what we really need is an institutionalized, central object management.

Considering *integration* as an absolute highlight of the project, we also decided to develop the system base simultaneously with a small series of typical but sufficiently different application packages. They were later called *Gadgets*, *Script* and *Illustrate*. *Gadgets* is the prime application. Is is an entire toolkit for the support of users, designers and programmers in the use and construction of graphical interfaces of arbitrary complexity and sophistication, including all kinds of visual objects and autonomous functional units. *Script* and *Illustrate* provide fully integrated platforms for text and graphic editing respectively. For example, *Script* allows objects like formulae, functional elements, pictures and formatting control characters to flow within a stream of ordinary characters and, vice versa, Illustrate allows stretches of text to be freely placed in illustrations at any desired position. While *Script* is a rather separate package, *Illustrate* heavily builds on *Gadgets*.

Be Wise– Generalize

It was our declared goal in the System 3 project to enhance the power of the original Oberon kernel, while strictly preserving its spirit of simplicity and its modular architecture and structure. In essence, we have approached and mastered this methodological challenge by exploiting, generalizing and unifying some key concepts and notions of the original system. For example, many of the original ingredients such as character patterns, display frames and viewers are just specialized *object types*, fonts appear as a variant of library and texts are still sequences of objects, though not necessarily sequences of characters. Also, we have adopted the original message *broadcast mechanism* in the display space ([1], Section 5.3) to implement a greatly generalized *Model–View–Controller* (MVC) scheme [2].

Seen structurally, System 3 differs from the original system merely by an additional base module called *Objects* for the handling of persistent objects and libraries. As displayed in Figure 2, this module provides a common platform for the original modules *Display*, *Fonts*, and *Texts*. Even if we also took the opportunity to revise the original system design in detail and to correct some marginal conceptual weaknesses and errors, the discrepancies with the original system are minor only, and original Oberon application programs can typically be adapted to System 3 by a simple recompilation.

In the following sections we shall discuss the new concepts and the rationale behind in detail. We shall also present the complete formal interfaces with clients. Throughout this text we assume the reader to be familiar with the fundamentals of the original Oberon system, in particular with the notion of *Frame* and the abstract data type *Text,* as they are explained e.g. in [1], Sections 4.3 and 5.1.

Objects and Libraries

Objects is a new module introducing two new abstract concepts, *Object* and *Library*, into the Oberon system kernel. Informally, *Object* denotes a class of entities that obey a well–defined message protocol, and *Library* is an abstract class whose instances are mutually disjoint collections of objects. We should immediately add at this point that the two notions are intimately coupled by the membership relation and that they are merely separate aspects of one integrated concept, the concept of persistent objects.

With the aim of achieving a deeper technical understanding of this integrated concept, let us first examine the position of libraries in the global system environment. We start by noticing that libraries come in two variants: *Public* and *private*. Public libraries are named and accessible from any arbitrary authority in the system. Private libraries are anonymous and encapsulated in some higher authority, typically a host document (as depicted in Figure 3). Common to both variants is their structure as *indexed set* of objects. The index elements are called reference numbers. Within the scope of a library, reference numbers are assumed to identify objects uniquely and invariantly.

We can now give a more formal presentation of the public interface that a library presents to its clients in the form of a type declaration, containing a name and an index specification:

```
Library = POINTER TO LibDesc;
LibDesc = RECORD
  name: Name;
  ind: Index;
  maxref: INTEGER
END;

Index = POINTER TO IndexDesc;
IndexDesc = RECORD END;
```

The index is specified by the pair (ind, maxref). *Name* is a fixed–length array of characters. Type *Index* is opaque because its implementation is dependent on the context. *maxref* is the upper limit of reference numbers currently contained in the index. It is used for enumeration purposes.

Functionally, the library interface with clients consists of three sections, devoted to three different topics: *Collective*, *object management* and *dictionary*. The collective section comprises methods *Load* and *Store*. They are used to load (internalize) and store (externalize) the addressed library.

Libraries are considered as dynamic collections. The object management section therefore splits up into the parts *retrieving* and *changing*. Method *GetObj* is used to retrieve the object that is referenced by ref:

```
GetObj: PROCEDURE (L: Library; ref: INTEGER; VAR obj: Object);
```

A set of three methods *GenRef, PutObj* and *FreeObj* serves the purpose of adding and removing objects. *GenRef* generates a new and unused synthetic reference number. *PutObj* installs object obj under the reference number ref and *FreeObj* removes the object with reference number ref:

```
GenRef: PROCEDURE (L: Library; VAR ref: INTEGER);
PutObj: PROCEDURE (L: Library; ref: INTEGER; obj: Object);
FreeObj: PROCEDURE (L: Library; ref: INTEGER);
```

The third and last part of the client interface of libraries is a dictionary. This is a simple tool for both aliasing reference numbers by names and abbreviating names by numerical keys.

Seen technically, a dictionary is a set of corresponding pairs (number, name). Type *Dictionary*, together with four intrinsic operations *PutName*, *GetName*, *GetRef* and *GetKey* is another abstract data type:

```
PROCEDURE PutName (VAR D: Dictionary; ref: INTEGER;
  name: ARRAY OF CHAR);
PROCEDURE GetName (VAR D: Dictionary; key: INTEGER;
  VAR name: ARRAY OF CHAR);
PROCEDURE GetRef (VAR D: Dictionary;
  name: ARRAY OF CHAR; VAR ref: INTEGER);
PROCEDURE GetKey (VAR D: Dictionary;
  name: ARRAY OF CHAR; VAR key: INTEGER);
```

PutName, *GetName* and *GetRef* operate on reference numbers. *PutName* adds a corresponding pair (ref, name) to the dictionary. *GetName* and *GetRef* are used to retrieve the components name and ref respectively. *GetKey* and *GetName* (again) operate on keys. *GetKey* determines a unique key to a given name and enters the corresponding pair (key, name) into the dictionary. *GetName* is used to retrieve the corresponding name from a given key.

Let us now switch our attention to objects. Recalling that (a) the term object in this context is reserved to entities in the system obeying a well–defined message protocol and that (b) each object is a member of at most one library at a time, we arrive at the following formal template of objects that includes a (possibly unused) reference to a library and a message handler:

```
Object = POINTER TO ObjDesc;
ObjDesc = RECORD
  lib: Library;
  ref: INTEGER;
  handle: Handler
END;
```

The reference is represented by the pair (lib, ref). If *lib* points to a library, we call the object *bound* to lib, otherwise we call it unbound or free. Handler is a standard Oberon message–handler type for class *Object* and message base type *ObjMsg*:

```
Handler = PROCEDURE (obj: Object; VAR M: ObjMsg);
```

Before revealing type *ObjMsg*, we emphasize that, in our object oriented environment, messages are rarely handled completely by their first recipient. Much rather, they are passed through a complex network of objects (we shall examine this process more concretely in the section on display frames). For the moment it is sufficient to mention that we make use of two auxiliary facilities for the purpose of message flow control: *Time stamp* and *dynamic link*. The time stamp is used by objects to recognize a recursive receipt of the same message, and the dynamic link typically records the current message path. These control facilities are reflected by two fields *stamp* and *dlink* in *ObjMsg* and in *ObjDesc*:

```
ObjMsg = RECORD
  stamp: LONGINT;
  dlink: Object
END;

Object = POINTER TO ObjDesc;
ObjDesc = RECORD
  stamp: LONGINT;
  dlink: Object;
  lib: Library;
  ref: INTEGER;
  handle: Handler
END;
```

One last addition to type *Object* is a so–called static link that is used to build temporary sets of objects in the form of linked lists.

With that, we get to the following complete public representation of type *Object*:

```
Object = POINTER TO ObjDesc;
ObjDesc = RECORD
  stamp: LONGINT;
  dlink, slink: Object;
  lib: Library;
  ref: INTEGER;
  handle: Handler
END;
```

Having worked out a precise formal definition of type *Object*, we shall henceforth use the term object or real object in the strict sense of "instance of type Object".

It is noteworthy that, in contrast to the fixed set of methods associated with type *Library*, the message interface defined by type *Object* is completely generic and open. However, there exists a small set of basic messages to that most of the real objects in the system are supposed to and prepared to respond appropriately. This is it:

{ AttrMsg, LinkMsg, FindMsg, CopyMsg, BindMsg, FileMsg }.

Roughly spoken, these messages are used in turn to get and set attributes, to create and retrieve functional connections, to find objects in name scopes, to generate a copy of an object, to bind an object to a library and to load and store an object. Notice that *AttrMsg*, *LinkMsg*, *FindMsg* and *CopyMsg* are intrinsic to objects, while *BindMsg* and *FileMsg* are library–oriented. For details we refer to the following explanations.

The Attribute Message

```
AttrMsg = RECORD (ObjMsg)
  id: INTEGER; (* enumerate | get | set *)
  Enum: PROCEDURE (name: ARRAY OF CHAR);
  name: Name;
  res: INTEGER;
  class: INTEGER;
  i: LONGINT;
  x: REAL;
  y: LONGREAL;
  c: CHAR;
  b: BOOLEAN;
  s: ARRAY 64 OF CHAR
END;
```

Before we can actually present the details of this message (type), we need to introduce the concept of *attribute*. Again, this is a generic concept, merely fixing the following: (a) With each object, an individual (and possibly empty) set of attributes is associated and (b) each attribute is specified by its name and class, where the possible classes are the predefined types LONGINT, REAL, LONGREAL, CHAR, BOOLEAN and ARRAY OF CHAR.

Messages of type *AttrMsg* are sent to an object (a) to retrieve the set of its attributes, (b) to get the current values of its attributes and (c) to set new attribute values. The identifier field id selects the operation from the set { enumerate, get, set }. Enum is an enumerator procedure that is to be supplied by the sender in case of enumerate. In the cases of get and set, name and class specify the desired attribute. Depending on its class, the actual value of the attribute is stored in one of the fields i, x, y, c, b or s. res reports about the result.

The two most basic attributes are named *Gen* and *Name*:

The Gen attribute

Objects in the system state space must be generated explicitly. To this purpose, a so–called generator is associated with each object (class). This is a dynamically callable procedure, i.e. an Oberon command. By convention, the generator is assumed to deposit a reference to the newly created object in the global variable *NewObj*. The actual value of the Gen attribute is the name of the associated generator.

The Name attribute

We have learned that bound objects can be identified by a pair (lib, ref) or, alternatively, by a pair (lib, name). In addition, each object can have an *intrinsic name* that is independent of any library scope. The actual value of the *Name* attribute is the intrinsic name (if it exists) or empty.

The LinkMsg

```
LinkMsg = RECORD (ObjMsg)
  id: INTEGER; (* id = enumerate | get | set *)
   Enum: PROCEDURE (name: ARRAY OF CHAR);
  name: Name;
   res: INTEGER;
  obj: Object
END;
```

Messages of type *LinkMsg* are used to establish and retrieve directed functional connections between objects. Recipients of messages of this type create or return a named link to some other object. A typical and important example of such a link is the view –––> model relation. *Enum* is an enumerator procedure that is to be supplied by the sender in case of id = enumerate.

The FindMsg

```
FindMsg = RECORD (ObjMsg)
  name: Name;
  obj: Object
END;
```

Recipients of messages of type *FindMsg* are considered as abstract *name scopes* with one basic function: Find the object with the given name. Typically (though not necessarily) name scopes are or represent collections or sets of objects.

The Copy Message

```
CopyMsg = RECORD (ObjMsg)
  id: INTEGER; (* id = shallow | deep *)
  obj: Object
END;
```

Messages of type *CopyMsg* are used to create an exact copy of the receiving object. The copy operation is much more subtle than perhaps expected at first sight, because real objects are typically compound rather than atomic. In most cases, objects are recursively composed of parts that are complex objects themselves.

We distinguish between *shallow copies* and *deep copies*. If a shallow copy is to be created, as many references to original components as possible are kept unresolved, whereas in case of a deep copy all references are resolved by recursively creating copies of the components. In both cases, the copy message is passed through a part of or through the entire data structure representing the original object.

The Bind Message

```
BindMsg = RECORD (ObjMsg)
  lib: Library
END;
```

Messages of type *BindMsg* are used to bind objects to a given library lib. More precisely, if we call loose an object that is either free or bound to an anonymous library, bind messages require the primary object and all its loose components to

bind themselves. Notice that this message again propagates through the entire data structure representing the primary object.

The File Message

```
FileMsg = RECORD (ObjMsg)
  id: INTEGER; (* id = load | store *)
  len: LONGINT;
  R: Files.Rider
END;
```

The purpose of messages of type *FileMsg* is loading and storing objects from and to a sequential file. Messages of this type are typically sent to all member objects by the library loader and saver at internalization and externalization time respectively. We should immediately clarify that sequentializing needs a preparational step that resolves pointers into reference numbers, their invariant counterparts. Viewed differently, externalizing a library is a two–pass process of the following form (see Figure 4):

Externalize library L = { bind all loose member objects to L; store L }

Let us then summarize the integrated concept of library and object. A library is a logical module encapsulating a set of connected objects and presenting itself to clients as an index of reference numbers. Every object in a library can invariantly be identified and accessed by a pair (lib, ref) or, by dint of the dictionary mechanism, equivalently by a pair (lib, name). Libraries come in two variants: Public and private. Public libraries are part of the global space of system resources. They are accessible by users, programs and objects. Private libraries are typically embedded in some host document and are accessible exclusively by authorities within the scope of this host document.

Recalling that objects typically refer to other objects (their components) we get to a natural relation of dependence among libraries and finally to a hierarchy of libraries. As depicted in Figure 3, this hierarchy is comparable to the import hierarchy of program modules. In particular, users can call objects from named libraries in the very same form *L.O* as they can call Oberon commands *M.P*. We shall refer to such objects as end–user objects.

So far, we might have given the impression that module *Objects* is merely a template defining some basic types in connection with libraries and objects. However, this is not quite true. In total, module Objects contains the following separate sections:

(a) Definition of basic types
(b) Implementation of dictionaries
(c) Implementation of standard libraries
(d) Central library managament
(e) Central time stamp generator

We have discussed section (a) exhaustively above. Sections (b) and (c) are just implementations of section (a) and, as such, they do not introduce any new concept. However, the implementations still do reveal some important decisions and solutions, the most interesting of which is perhaps the handling of the situation of missing code. In order to explain this problem, we start from the observation that libraries are not self–contained in two respects: First, they typically import other libraries and second, they rely on program modules implementing their member objects.

While the situation of missing imported libraries is under individual control of the actual (primary) object to be loaded, the situation of missing code must be handled centrally by the library loader at object generation time. For standard libraries we have chosen a reasonably graceful solution that is based on the creation of a dummy substituting the original object and retaining its identity.

Standard libraries may contain up to maxint (maximum integer) objects. They are further characterized by a coarse–grained loading and storing strategy that is based on loading and storing libraries in their entirety only. In contrast, more sophisticated strategies would allow selective loading and storing of individual objects.

Module *Objects* exports the following procedures for the support of standard libraries

```
PROCEDURE OpenLibrary (L: Library);
PROCEDURE LoadLibrary (L: Library; f: Files.File; pos: LONGINT;
  VAR len: LONGINT);
PROCEDURE StoreLibrary (L: Library; f: Files.File; pos: LONGINT;
  VAR len: LONGINT);
```

OpenLibrary is called to initialize (the descriptor of) a new and empty standard library. The procedures *LoadLibrary* and *StoreLibrary* are provided explicitly for the loading and storing of standard private libraries that are embedded in a larger file. Remember that methods *Load* and *Store* exist to load and store named libraries.

Section (d) provides a system–wide management of named libraries in the form of one main service function

```
PROCEDURE ThisLibrary (name: ARRAY OF CHAR): Library;
```

and the following auxiliary procedures:

```
PROCEDURE Register (ext: ARRAY OF CHAR; new: NewProc);
PROCEDURE Enumerate (P: EnumProc);
PROCEDURE FreeLibrary (name: ARRAY OF CHAR);
```

where

```
NewProc = PROCEDURE (): Library;
EnumProc = PROCEDURE (L: Library);
```

are procedure types.

Roughly spoken, *ThisLibrary* simply returns the (unique) library whose name is given by the parameter. However, the detailed specification of this function is quite intricate. It is based on two assumptions:

(a) There exists a global directory of currently loaded libraries (called library cache)

(b) The kind of a library is uniquely determined by the extension of its name

With that, we are now able to sketch the implementation of function ThisLibrary as follows:

```
ThisLibrary (name) = {
  Search library with given name in cache;
  If ~found then
    use extension to determine kind of library;
    load or generate library of this kind;
    include it in the cache
  end;
  Return library to caller }
```

Also, we can now appreciate the auxiliary procedures Enumerate, *FreeLibrary* and *Register* that are used to enumerate all libraries that are currently in the cache, to free a library from the cache (unload it) and to register a pair of the form (extension, library generator) for subsequent use. Note that the extension *Lib* is pre–registered for standard libraries.

The time stamp mechanism obviously presupposes a central stamp generator, generating unique stamps in strictly ascending order. Exactly to this purpose, procedure *Stamp* is provided as section (e) of module *Objects*:

```
PROCEDURE Stamp (VAR M: ObjMsg);
```

As a side remark we add that procedure *Stamp* generates synthetical time stamps that do not at all reflect any real system time.

This concludes our presentation of module *Objects*.

Display Frames Reinterpreted

In both the original Oberon system and System 3, a single module called *Display* is fully responsible for the basic display management, and a single type of entity called frame is used to define the hierarchical structure of the display space. However, if we compare the ways how the two systems interpret display frames, we recognize an interesting shift of emphasis.

In the original system, frames are rectangular areas in the display space, playing a kind of a double role as (a) a view into some document (model) and (b) an interpreter (controller) of user actions. In other words, the original display frames implement a so–called Model–View–Controller scheme ([1], Section 5.3). It follows that there exist n frames for every n views into a given model, even if some of the views are equivalent. For example, if the same part of a text is displayed twice on the screen, then two different text frames exist. Remembering the nested hierarchy of frames in the display space, we can reformulate the above statement by saying that the frame hierarchy is a tree.

Experience with the development of more complex graphic applications such as the *Gadgets* system has suggested some relaxation of the strict Model–View–Controller scheme. The quintessence of this relaxation is the possibility of sharing a frame instance in case of multiple representations in the display space. In short, the System 3 model of the display space distinguishes itself by the facts that

(a) frames are visual objects
(b) frames are sharable

(a) is formally equivalent to basing type *Frame* on type *Object*, i.e. to defining *Frame* as a type extension of *Object*:

```
Frame = POINTER TO FrameDesc;
FrameDesc = RECORD (Objects.ObjDesc)
  next, dsc: Frame;
  X, Y, W, H: INTEGER
END;
```

Fields *next* and *dsc* are pointers to the next frame on the same level in the hierarchy and to the first frame on the next lower level respectively. W and H are width and height of the described frame respectively and X, Y are the coordinates of the frame, relative to its ancestor. In case of a shared frame, X and Y are irrelevant.

(b) is achieved by allowing different so–called views or view–frames to refer to a common visual object. An immediate consequence is the fact that the structure of the display space in general is a direct *acyclic graph* (dag) rather than a tree.

Our next topic is frame–oriented message passing. In the last section we mentioned that messages in an object–oriented environment are not normally handled by the first recipient completely, but are typically passed through a more or less complex network of objects. We can nicely illustrate this process by the example of message passing in the context of frames.

We first recall that active frames are part of the display space and, as such, are members of the frame hierarchy. The essential aspect of this fact in connection with message passing is the observation that active frames are automatically integrated in a context and that, a fortiori, every action taken by a frame–object potentially depends on its context. Obviously, any appropriate strategy of passing messages to frames should at least respect this structural subordination. Consequently, the strategy of sending messages to target frames directly should be ruled out immediately.

Remember now that the display space of the original Oberon system is tree structured with the display area at its root, hierarchically followed in succession by tracks, viewers and frames of contents. Also remember that, due to frame sharing, the System 3 display space is no longer a tree but still a directed acyclic graph (dag) with a root. Figure 5 depicts an excerpt of such a dag in a simple case, including two views into one and the same visual object.

From Figure 5 again we easily conclude that the most obvious strategy, namely sending a message to the root and have the root sift down the message to the actual target frame is not well–defined because more than one path may lead to the desired target. What is really needed is a *broadcast strategy* within the display space, so that all the paths leading to the desired target are guaranteed to be involved.

A few clarifying and adding remarks on the more formal points and on the implementation of the broadcast strategy are doubtlessly in order. Let us first examine the base type of frame messages. It is called *FrameMsg* and is derived from type *ObjMsg*:

```
FrameMsg = RECORD (Objects.ObjMsg)
  F: Frame;
  x, y, res: INTEGER
END;
```

Component F designates the actual target frame. Messages with no specific target frame (intrinsic broadcast messages) are characterized by F = NIL. x, y are relative coordinates that are accumulated along the current path and res is a result code. By convention, res = 0 indicates that no exception has occurred and res < 0 signals that the message has been handled exhaustively.

The actual process of broadcasting a message in the display space is controlled by procedure (variable) Broadcast that is defined as follows:

```
Broadcast: MsgProc;
MsgProc = PROCEDURE (VAR M: FrameMsg);
```

It is important to be aware that broadcast is no more and no less than a universal rule of the game. Although it is underlying each receipt of a frame message, this rule is not enforced by any global authority. Expressed differently, each recipient is free to override the rule by stopping the message instead of forwarding it to further descendants as, for example, in case of res < 0.

In total, we can enumerate the following cases that potentially cause a stop

(a) The received message has been handled exhaustively (early termination)
(b) The recipient does not contain the target frame (short cut)
(c) This message has arrived at the recipient before (circle cut)

How can these cases be recognized by the recipient? In case (a), the recipient either is the target frame or any authority controlling the target frame. This case is characterized by res < 0. In case (b), the recipient needs concrete knowledge of its contents and, finally, detection of case (c) is made possible by the timestamp mechanism for object messages.

While control of message handling by ancestors of a target frame is built–into the broadcast strategy, the dual control of context dependent message handling by the target frame needs the additional facility of the earlier introduced dynamic link. Context dependent message handling is not only a necessity in cases where the target frame changes its size but opens interesting and far–reaching possibilities in connection with context dependent interaction. For example, we can imagine a visual object reacting differently on user actions in (a) a user's context, (b) a designer's context and (c) a tutorial context (see Figure 6).

In order to further clarify the concept, let us now roughly sketch the handling of messages:

```
Handle message M received by frame F = {
 save dynamic link in F; update dynamic link in M;
 if timestamp of M > backup timestamp in F then
  save timestamp in F;
  accumulate coordinates in M;
  if target frame of M = F THEN (*target frame is me*) handle M
  else
   if target frame of M = NIL THEN
    (*intrinsic broadcast*) handle M
   end;
   while (res >= 0) & more descendants do
    pass M to next descendant
   end
  end
 else special handling in new context
 end }
```

The arsenal of possible frame messages is amazingly small. Let us merely give here an annotated overview. For more detailed explanations and for accompanying examples we refer to the *Gadgets* guide.

Change state of frame

```
ControlMsg = RECORD (FrameMsg)
  id: INTEGER; (* remove | suspend | restore *)
  X, Y: INTEGER
END;
```

Change size of frame

```
ModifyMsg = RECORD (FrameMsg)
  id: INTEGER; (* reduce | extend | move *)
  mode: INTEGER; (* display | state *)
  dX, dY, dW, dH: INTEGER;
  X, Y, W, H: INTEGER
END;
```

Display part of frame

```
DisplayMsg = RECORD (FrameMsg)
  id: INTEGER; (* frame | area *)
  u, v, w, h: INTEGER
END;
```

Print frame

```
PrintMsg = RECORD (FrameMsg)
  id: INTEGER; (* contents | view *)
  pageno: INTEGER
END;
```

Locate frame at position

```
LocateMsg = RECORD (FrameMsg)
  loc: Frame;
  X, Y, u, v: INTEGER
END;
```

Consume object at position

```
ConsumeMsg = RECORD (FrameMsg)
  id: INTEGER; (* drop | integrate *)
  u, v: INTEGER;
```

```
  obj: Objects.Object
END;
```

Return, set or reset object selection

```
SelectMsg = RECORD (FrameMsg)
  id: INTEGER; (* get | set | reset *)
  time: LONGINT;
  sel: Frame;
  obj: Objects.Object
END;
```

We should add that analogous messages to *ConsumeMsg* and *SelectMsg* for text stretches instead of objects are defined in module *Oberon*. In addition, module *Oberon* exports two frame messages *CaretMsg* and *RecallMsg* in connection with texts:

Consume text stretch at position

```
ConsumeMsg = RECORD (Display.FrameMsg)
  id: INTEGER; (* drop | integrate *)
  u, v: INTEGER;
  text: Texts.Text;
  beg, end: LONGINT
END;
```

Return, set or reset selection of text stretch

```
SelectMsg = RECORD (Display.FrameMsg)
  id: INTEGER; (* get | set | reset *)
  time: LONGINT;
  sel: Display.Frame;
  text: Texts.Text;
  beg, end: LONGINT
END;
```

Return, set or reset caret

```
CaretMsg = RECORD (Display.FrameMsg)
  id: INTEGER; (* get | set | reset *)
  car: Display.Frame;
  text: Texts.Text;
  pos: LONGINT
END;
```

Recall text from delete buffer

```
RecallMsg = RECORD (Display.FrameMsg)
END;
```

We conclude this section with a short notification of two additional and more technical ingredients that are provided by module *Display*. The first is a global clipping rectangle together with three operations *SetClip*, *AdjustClip* and *ResetClip* to set, adjust and reset the clipping rectangle:

```
PROCEDURE SetClip (X, Y, W, H: INTEGER);
PROCEDURE AdjustClip (X, Y, W, H: INTEGER);
PROCEDURE ResetClip;
```

The second technical ingredient is an additional procedure *Depth* to get the depth (= number of bits) of a pixel at a given X–coordinate in the display space:

```
PROCEDURE Depth (X : INTEGER): INTEGER;
```

Embedding Objects in Texts

An ordinary text is a pure sequence of characters or, more precisely, a pure sequence of character codes (e.g. Ascii codes). Original Oberon takes a somewhat enhanced view in that Oberon texts are defined as sequences of *attributed characters* with attributes *font*, *color* and *vertical offset*. This allows us to weave into text descriptions a variety of display–oriented looks and so to take full advantage of raster displays and printers.

However, what we really want is generalized texts with integrated figures, pictures, formulae and perhaps even with "generic" elements in the framework of an electronic textbook. In other words, we want a generalized definition of text as a

sequence of arbitrary objects. Diverse models of such a generalized notion of text exist. For example, one rather obvious approach is based on inline encodings of object descriptions in the form of interspersed "escape–sequences". Remarkably, Oberon's enriched text model suggests a much more elegant solution.

The key to this solution is a shift of emphasis in the interpretation of attributed characters. Instead of looking at font as an attribute, we interpret it as a collection. More precisely, we take the pair (code, font) as a reference into an indexed collection or, in different terms, as a reference into a library. And immediately, this new view does not only enable us to allow arbitrary public libraries instead of just fonts, but it also enables us to amalgamate a private library with each text, collecting all the private and possibly mutable objects that are integrated in this specific text. And, notably, it does so in perfect harmony with both the original model and the implementation of text.

Let us quickly summarize our generalized view of text:

(a) A text is a sequence of arbitrary objects (in the strict sense of Section 1).

(b) Every object in the sequence is specified by a pair (ref, lib). In the case of an ordinary character, *ref* is a character code and *lib* is a font.

(c) To every text a private library may be associated with the purpose of collecting private objects that are contained in this text.

In practice, generalized texts typically appear as sequences of ordinary characters that are occasionally broken by (non–character) objects such as pictures or formulae. However, the spectrum of possible objects is by far not restricted to graphical figures. Alternatively, information–oriented objects exist that are interpreted by text scanners rather than displayed. A good example for this kind of object is provided by so–called *formatting control characters* that have been introduced by the *Write* editor [3]. In principle, each formatting control character is loaded with formatting information for the stretch of text following it.

In passing we note that the model of formatting control characters is particularly amenable to a substantial conceptual upgrade in combination with *styles*, i.e. with public libraries consisting of predefined formatting control characters. For details, see the guide of the *Script* text editor.

In the following sections we discuss the implications of our new and generalized view of texts on their presentation as an abstract data type.

Implications on Module Fonts

A (digitized) character is essentially a box containing a *pattern* and a *measure* specifying the distance to the next character, when lined up horizontally. A *font* is a family of characters of a given style and size. It is easily made into an indexed collection by using the Ascii–code as index.

With that, we have arrived at a natural interpretation of characters as objects and fonts as libraries:

Object ––> Character
Library ––> Font

Formally, we derive type *Char* from type *Object* and type *Font* from type *Library*:

```
Char = POINTER TO CharDesc;
CharDesc = RECORD (Objects.ObjDesc)
  dx, x, y, w, h: INTEGER;
  pat: Display.Pattern
END;

Font = POINTER TO FontDesc;
FontDesc = RECORD (Objects.LibDesc)
  type: SHORTINT; (* substitute | font | metric *)
  height, minX, maxX, minY, maxY: INTEGER
END;
```

We distinguish types *font*, *metric* (boxes without patterns) and *substitute* (for non–available font). *height*, *minX*, *maxX*, *minY* and *maxY* are collective metric data that are not applicable to general libraries. We should also note that fonts are specialized libraries in two additional respects:

(a) Fonts are homogeneous, i.e. all member objects are of the same type.

(b) Fonts are static, i.e. member objects are immutable and passive.

It is therefore justified to implement font libraries differently and more efficiently than general standard libraries. However, we emphasize that the client interfaces of fonts and libraries are absolutly compatible. In particular, method *GetObj* returns

the desired character as an object in the case of font libraries. Because character objects are immutable and passive, permanent instantiations are not needed and would amount on a tremendous waste of resources. Therefore, only a single archetypal character object exists in the system. It is recycled at every call to *GetObj*.

For compatibility reasons, procedures *GetChar* and *This* are offered by module *Fonts*. They provide an alternative way to get character data from a given font and a font library from its name respectively.

```
PROCEDURE GetChar (F: Font; ch: CHAR; VAR dx, x, y, w, h: INTEGER;
  VAR pat: Display.Pattern);
PROCEDURE This (name: ARRAY OF CHAR): Font;
```

The class of font libraries is preregistered in the central library management under extension *Fnt*.

Implications on Module Texts

We have learned informally how the original Oberon text model generalizes smoothly to sequences of arbitrary objects. Now, we are prepared to discuss the detailed implications of the generalized view to the formal definition of the abstract data type *Text*.

We first note that type *Text(Desc)* has been slightly adjusted as follows:

```
Text = POINTER TO TextDesc;
TextDesc = RECORD (Objects.ObjDesc)
  len: LONGINT;
  obs: Objects.Library
END;
```

Concretely, we note three modifications against the original type *Text*:

(a) *Text* is now a type extension of *Object*.
(b) A new field *obs* of type *Library* has been added.
(c) The *notifier* field has been removed.

(a) reflects the interpretation of a text document as a (complex) object. The *obs* field in (b) refers to the text's private library. Item (c) is not a direct consequence of our generalization. However, we decided to simplify postprocessing of text update operations by directly broadcasting frame messages of type UpdateMsg in the display space:

```
UpdateMsg = RECORD (Display.FrameMsg)
  id: INTEGER; (* change | insert | delete | replace *)
  text: Text;
  beg, end, len: LONGINT
END;
```

Notice the four possible values of the message id. With the aim of completion we have extended the original set of operators on texts by an operation *Replace*. It allows us to atomically replace an arbitrary stretch of text by a buffer's contents:

```
PROCEDURE Replace (T: Text; beg, end: LONGINT; B: Buffer);
```

In the case of a replace update message the additional field len specifies the length of the replacing text stretch.

Here is a complete table of the basic operations on texts:

Operation	*Parameters*	*Effect*
ChangeLooks	beg, end, sel	Change looks to sel in stretch [beg, end)
Insert	pos, B	Insert contents of B at pos
Delete	beg, end	Delete stretch [beg, end)
Replace	beg, end, B	Replace stretch [beg, end) by contents of buffer B

Type *Buffer* and all access types like *Reader*, *Scanner* and *Writer* have remained essentially unchanged with the obvious exception of fields fnt of type *Font* that have been replaced systematically by fields *lib* of type *Library*. Moreover and independently of the new text model, we have replaced the type dependence of *Reader* and *Writer* from type *Rider* by a *field R* of type *Rider*.

And finally, we have introduced a new access mechanism called *Finder*. Similarly to readers and scanners, finders transform text streams into sequences of well–defined entities, non–character objects in this case. The abstract data type *Finder* consists of one record type and two operations:

```
Finder = RECORD
  eot: BOOLEAN;
  pos: LONGINT
END;

PROCEDURE OpenFinder (VAR F: Finder; T: Text; pos: LONGINT);
PROCEDURE FindObj (VAR F: Finder; VAR obj: Objects.Object);
```

OpenFinder sets up a finder on text T at position pos. *FindObj* returns the next non–character object in the text stream.

The following table summarizes Oberon's access mechanisms on text

Access mechanism	Entity
Reader	Character code and attributes
Scanner	Token of universal class
Finder	Non–character object

A Tool for the Management of Libraries

Libraries need not necessarily be manipulated by a universal user tool. For example, private libraries are accessed implicitly via their host document. Also, public libraries exporting just one complex master object can be naturally identified with this object. Such libraries appear as a single object and are operated by an application–oriented object manager rather than by a general library manager.

Nonetheless, we provide a simple system–wide library manager tool called *Libraries*. It includes a general part (a) and a specialized part (b). Part (a) allows interactive inspection and manipulation of arbitrary libraries. Part (b) is a service for interactively organizing collections of texts and in particular collections of electronic messages within the framework of libraries.

Part (a):

```
PROCEDURE ShowLibs;
PROCEDURE ShowObs;
PROCEDURE Free;
PROCEDURE FreeObj;
PROCEDURE Cleanup;
PROCEDURE Store;
```

ShowLibs lists the names of all currently loaded public libraries, *ShowObs* lists the names of all named objects in the specified library, *Free* unloads the specified public library, *FreeObj* removes the specified object from its library, *Cleanup* collects garbage objects in the specified library and *Store* stores the specified library on file.

Part (b):

```
PROCEDURE InstallText;
PROCEDURE OpenText;
```

InstallText installs a text in the specified library. The text is taken from the main frame of the marked viewer and, in principle, the name is generated from the title bar by omitting the extension. In the case of the text representing an electronic message, a unique synthetic name is generated heuristically. *OpenText* opens a viewer to display a previously installed text.

In Retrospect

What has started relatively harmlessly with a study of how to include pictures in texts has ended with a complete and highly integrated system for the management of persistent "end–user" objects. Most remarkably, we have been able to preserve both Oberon's compactness and its spirit of simplicity. The original Oberon system base has not only proved suitable but really ideal for such an upgrade of functionality. In retrospect, leaving unexploited so much conceptual potential would have been a most deplorable omission.

We decided to develop the basic system hand in hand with three representative though substantially different application packages: An editor for generalized text with integrated objects (*Script*), a graphical user interface kit (*Gadgets*) and an editor for graphical illustrations (*Illustrate*). Our decision has payed well. Without continuous feedback from the applications we would not have been able to find a complete and reasonably small message protocol, nor would we have succeeded in getting the display model right.

The realization of every vision has to be carefully kept in limits, if it is to be completed successfully within a foreseeable period of time. For example, we would have liked to extend the concept of persistent objects and their display model to a local client/server network environment at least. In particular, we would have liked to extend a server's display space to

clients as it is indicated in Figure 7, so that servers can transparently display their objects (e.g. mailbox directories) on client screens.

We are convinced that the concept of persistent objects combining information, control and functionality in one autonomous entity can be driven much further, so that it can lead to a radically new paradigm of what software is, how it is looked at, how it is used and, last but not least, how it is constructed.

Acknowledgement

I am greatly indebted to the designers and implementors of the application packages, in particular to Hannes Marais (Gadgets), Karl Rege (Illustrate) and Ralph Sommerer (Script). Without their enthusiasm and professional expertise, this project could not have been started, leave alone successfully completed.

References

[1] N. Wirth and J. Gutknecht. Project Oberon. Addison–Wesley 1992.

[2] A. Goldberg, D. Robson. Smalltalk–80: The Language and its Implementation. Addison–Wesley 1983.

[3] C. A. Szyperski. Write–ing Applications: Designing an Extensible Text Editor as an Application Framework. Proceedings of the Seventh International Conference on Technology of Object–Oriented Languages and Systems (TOOLS'92), Dortmund, D. Prentice Hall, Englewood Cliffs, NJ. Mar. 1992.