Write

an extensible text editor for the Oberon system

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Clemens A. Szyperski

Write

An Extensible Text Editor for the Oberon System

January 1991

151

Author's address:

Computersysteme
ETH-Zentrum
CH-8092 Zurich, Switzerland

e.mail: szyperski@inf.ethz.ch

Write – An Extensible Text Editor for the Oberon System

Clemens A. Szyperski

Abstract

Extensible software systems open the opportunity of reducing complexity by trading off initial functionality. Instead of building everything into a monolythic system, a core system with a certain built–in potential for later extension is separated from an arbitrarily rich set of extensions (and, in principle, extensions of extensions). This report concentrates on the text editor Write and some of its existing extensions. It is discussed how extensibility is opened but also limited by the core system's design. It is observed that a potentially very rich and flexible basic framework may actually lead to few or no implemented extensions if the chosen model was inadequately complex for the typical extension demands. On the other hand, a too rigid core structure may impose limits preventing useful extensions right from the beginning.

Keywords: Extensibility, Editor, Oberon

Introduction

Making a text editor extensible is a rich source of conceptual variations [e.g. WeGaMa89] and already in itself an interesting project target. However, other aspects need to be considered that are not directly related to the aim of engineering an extensible editor. First of all, the fact that extensions are possible in a given framework needs to be paralleled by sufficient ease of actually implementing extensions. Secondly, for the editor to be useful it is important to integrate it well into its environment. Finally, the interaction of environmental support, potential for extensibility and ease of use need to add up to the user's impression of working with a rather light–weight system. The impression of working with a heavy–weight system should be avoided whenever possible, as the resulting barrier might well doom the newly created tool not to be used.

On the first sight, there is a contradiction between having a rich, extensible, and flexible system, and having a light–weight, easy to master, and easy to understand tool. The key to an adequate compromise lies in the approach to extensibility. An unextensible system with a rich set of well designed, built–in functionality might well be easier to master than an extensible but highly complicated system. For an extensible system to be transparent and manageable, there are two criteria of dominant importance. On one hand, the core system without any (visible) extensions must follow a clean and easy to understand model. This model is the basis for understanding the whole system and the framework to learn more about available extensions. On the other hand, extensions should be restricted to a *few atomic* features of the core system. Otherwise, the user can hardly predict the effects of incorporating a certain extension into the editing tool. The importance of these two points might be stressed by considering that extensions naturally evolve at different places and that vendors of extensions might hardly know about each other or about the potentially harmful interaction of their extensions when all put together.

The Write editor limits extensibility to only two orthogonal domains: commands and characters. The concept of extending the system by adding new commands is already well–established in the Oberon system [WiGu88]. Adding a new command adds new manipulational capabilities applicable by the user. The second domain of extensions is the actual data that commands operate on. To keep the interdependence of commands and data extensions low, data extensions should be organized around the atomic components of the data structures provided by the core framework. In the case of a text editor, the core data structure is text, i.e. a sequence of characters. *As a result, the natural anchor of data extensibility is considered to be the character.* In fact, no other form of data extension is supported by Write; especially, there are no provisions for overlaying a text with additional structures. While this might sound too rigorous, it results in a relatively compact and easy to understand design. Also, the remaining potential for extensibility is easy to explore and still quite rich.

As Write is integrated into the Oberon system, the natural environment are the existing standard text class [Gu90], and all commands operating on such texts. The latter range from simple formatting commands (e.g. changing the font of a selected text stretch) to complex transformations (including the Oberon compiler). While in most aspects, Write integrates well with the Oberon system, there are certain shortcomings. These will be discussed in more detail later. For the time being, it should suffice it to say that the main problem is the traditional procedural interface to the Oberon system's standard data types, including texts, which limits the system's potential for extensibility.

The remainder of this report is organized into two main parts and a set of appendices. The first part concentrates on the core system of Write, its major design decisions and related tradeoffs, and its potential for extensibility. The second part presents the details on how Write extensions work, as well as a series of successively more advanced extensions together with practical and available examples. The appendices cover a user manual for the Write core system, a description of the programmer's interfaces of the Write core modules, and two complete sources of actual extensions. Care has been taken to decouple the various parts of this report. It should be possible to read each of them independently. A few exceptions to this rule are clearly marked using cross–references.

Part 1 – The Write Core System

This part deals with the standard modules of the Write system. The following sections form a stepwise analysis of the provided functionality, the design criteria, and the resulting tradeoffs. The involved modules and levels of abstraction are discussed bottom–up, while the concepts of extensibility are introduced top–down.

1.1 Approaching Extensibility – Coping with Complexity

The importance of limiting extensibility in a system to a few strategically well chosen points has been stressed in the introduction. As mentioned, for a text seen as a sequence of characters, the natural atoms of extensibility are just the characters themselves. Other possibilities are higher–level structures layered over a basic text such as paragraphs, links between text areas, and the like. The Write system restricts itself to the extension of characters. Such extended characters are called *elements*. In order to understand what it actually means "to extend a character", the properties of unextended characters as members of texts are examined in the following.

Standard Oberon Texts

A character is an atomic instance: it can neither be split nor merged with other characters. Associated with a character are certain attributes that are fully independent of the neighbouring characters. Texts of such attributed characters are implemented in the Oberon standard module *Texts*.

attribute	typical values
font family	Times Roman, Syntax, Helvetica, ...
font size	height in points
font style	bold, italic, ...
color	foreground, background, ...
vertical offset	character displacement relative to the base line

Table 1. Oberon Character Attributes.

Furthermore, a character is the natural atomic editing unit of a text. Characters (or sequences thereof) may be inserted, copied, or deleted, and their attributes may be changed. This is achieved by allowing the user to select any subsequence of characters within a text, and by supporting the placement of a caret between any two characters. The user may delete selected text, copy it over to the position marked by the caret, change its attributes, or enter new characters at the caret position. Finally, for the sake of simplicity, characters are also displayed as atomic units, i.e. either the whole character can be displayed within the available screen area, or it is suppressed (clipped) completely.

In order to maintain the very simple and efficient text editing model of the Oberon system, it is desirable to design extension features such that they will not get into the way of the editing model. The only way to achieve this is the restriction to extensions that do not leave the principal conceptual properties of texts seen as a *sequence of characters*. It will be shown later, that within this carefully chosen restriction, the remaining potential for extensions is still quite large.

Another important aspect of Oberon texts is that programs have access to any text displayed on the screen. For example, it is possible to directly compile a displayed text without storing it beforehand. This open accessibility of data structures attached to viewers is an important key to the Oberon system's integration among various applications (better: command packages, as Oberon has no primary notion of distinct applications). Therefore, it is important that a functionally enriched text editing environment interacts and cooperates as seamlessly as possible with the existing text and text editing facilities.

Write Texts

The above arguments lead to the design of a first core module of Write: *WriteTexts*. To reach the desired integration, *WriteTexts* should be an extension of the Oberon standard module *Texts*. To reach the extensibility aimed at, *WriteTexts* introduces the notion of extended characters, so called elements. *Texts* defines two basic types of textual representation: texts and buffers. Buffers are used to accumulate output of commands in order to insert them as a whole into a target text (delayed and atomic output). Furthermore, *Texts* defines readers and writers as access structures to texts and buffers, respectively. *WriteTexts* implements extended buffers and texts representing sequences of potentially extended characters, while it does not extend the access structures of *Texts*. It does, however, add some access primitives to operate on extended characters. Hence, existing clients of *Texts* can use readers and writers (see restrictions below) to access *WriteTexts* texts. The following table is an excerpt of the *WriteTexts* interface.

```
TYPE
   Buffer = POINTER TO BufferDesc;
   BufferDesc = RECORD(Texts.BufDesc) END;
   Text = POINTER TO TextDesc;
   TextDesc = RECORD(Texts.TextDesc) END;

(* Buffers *)
PROCEDURE OpenBuf(B: Buffer);
PROCEDURE CopyBuf(SB: Buffer; VAR DB: Buffer);

(* Texts *)
PROCEDURE Open(T: Text; name: ARRAY OF CHAR);
PROCEDURE Delete(T: Text; beg, end: LONGINT);
PROCEDURE SaveBuf(T: Texts.Text; beg, end: LONGINT; VAR B: Buffer);
PROCEDURE Insert(T: Text; pos: LONGINT; B: Texts.Buffer);
```

There are no primitives available in *WriteTexts* to read or write plain characters. The standard procedures defined in module *Texts* can be used with the exception that a writer's buffer must not be inserted into a Write text using the *Texts.Insert* procedures but must be inserted using the corresponding procedures of *WriteTexts*. The same holds for the other shown text and buffer manipulation procedures. It is a shortcoming of the procedural interface of *Texts* that extensions cannot override functionality. A future version of Write might well be based on a different version of *Texts* that either has a class–centered interface (i.e. methods instead of procedures), or has a built–in notion of "extensible characters".

It is now possible to augment the *WriteTexts* interface by certain procedures to insert and retrieve elements from a text. As elements should not get into the way of clients of plain texts, each element is represented by a special character (ASCII Code $1C_H$). Whenever a *Reader* returns this special character, an application knowing about elements might retrieve the associated element. This way all standard character attributes are attached to the representing character and elements thus inherit them. Therefore procedures of *Texts* that merely change attributes of character ranges within a text need no changes at all and thus have no counterparts in *WriteTexts*. The following definition gives the base type of elements, as well as a set of procedures that may be used to operate on elements. After changing an element, the programmer has to call *ChangedElem*, which propagates a notification to potentially update the display. Note that the interface hides the internal structure used to hold elements of a text. (In the current implementation, a linear list is used.)

```
CONST
    ElemChar = 1CX;
TYPE
    Elem = POINTER TO ElemDesc;
    Handler = PROCEDURE(E: Elem; VAR msg: Display.Message);
    ElemDesc = RECORD
        DX, W, H: LONGINT;    (*DX >= W*)
        handle: Handler
    END;
    CopyMsg = RECORD
        e: Elem
    END;

PROCEDURE OpenElem(E: Elem; handle: Handler; dx, w, h: LONGINT);
PROCEDURE InsertElem(T: Text; pos: LONGINT; E: Elem);
PROCEDURE ElemAt(T: Text; pos: LONGINT): Elem;    (*returns NIL if not found*)
PROCEDURE ChangedElem(E: Elem);
```

Elements are active objects: An installed handler allows object–centered interpretation of messages sent to an object. The messages understood by an element will be defined in the course of this report when needed. To enable delegation of existing display–system messages, the used basetype is the standard Oberon type *Display.Message*. Like characters, elements are characterized by a bounding box, plus some offset to the next character. (Cf. Fig. 1. Actually, the geometric model used for characters is slightly more complicated. This will be discussed in the section on *WriteFrames*.) Both, handler and geometry are set when opening an element. The element may then be inserted. Elements floating in a text can be retrieved by means of their current position in the text. Typically, a client asks for an element at a certain position after a *Reader* returned the special character *WriteTexts.ElemChar*.

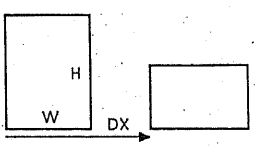

Figure 1. Geometry of an Element.

Generic copying of elements is implemented by sending a copy message. The recipient allocates and initialized a copy of itself. To support delegating a partially processed copy message, a new element should only be allocated and assigned to the message field *e*, if *e* is NIL. (Note that the message record is a reference parameter to the receiving handler.)

1.2 Load & Store – The Write File Format

Loading and storing texts that contain arbitrary extensions is a bit more involved than dealing with fixed file formats. Three points stood in the foreground when deciding on the file format for Write. First of all, element designers should be able to decide on their own, how to store an elements data. Secondly, Write should be able to load existing plain text files, and, conversely, existing clients of plain text files should be able to open a Write text in a meaningful way. Thirdly, Write should be able to gracefully degrade upon detecting unloadable elements, henceforth called *aliens*, within a Write file.

The generic loading and storing of elements is implemented by sending special messages to elements.

Since an element cannot receive a load message unless it already exists, loading has to be separated from allocation. When storing itself, an element first writes out the name of an Oberon command. This command is called at loadtime to allocate an instance of the element to load, followed by a load message sent to the new element. Loading can fail if the implementing module is not available on the local machine. If loading fails, the resulting alien element is reduced to an anonymous frame–shaped element remembering its data block in the original file. (To make this possible, the file contains the length of each element block.) Such an alien has the original element's frozen bounding box and can be copied or deleted freely. It will return to its normal behaviour as soon as the containing text is loaded with the implementing module available. (In the following and in the remainder of the report, the convention is used that definitions are qualified using the defining module. As above, qualification is left out, if the definition belongs to the module treated in the corresponding section.)

```
TYPE
   WriteTexts.LoadMsg = RECORD(Display.FrameMsg)
      r: Files.Rider
   END;
   WriteTexts.StoreMsg = RECORD(Display.FrameMsg)
      r: Files.Rider
   END;
```

The compatibility to the existing *Texts* file format is reached by true extension, i.e a Write file block consists of a *Texts* file block followed by a Write extension block. Customers of *Texts* file blocks will see a projection of a Write file that reduces all contained elements to a single character. Storing such a text with a non–Write editor will of course remove all Write specific data including all contained elements. The following syntax defines the Write file block format. All values are written as portable compact numbers as defined in [Te90]. The values pos, DX, W, and H are handled outside of the generic load/store mechanism and the byte count *bytes* always takes four bytes. Both together is used to correctly handle alien elements. The tag values are exported by *WriteTexts*.

```
TextBlock = Texts.TextBlock textTag ElemBlock { ElemBlock }.
ElemBlock = elemTag pos DX W H bytes { byte }bytes.
   (*The load command belongs to the element specific data, i.e. is contained in the { byte } sequence*)
```

1.3 Visual Interpretation of Write Texts – WriteFrames

The visual interpretation of a data structure covers displaying (part of) the structure within a frame and accepting display dependent edit operations. As *WriteFrames* behaves very much like the standard Oberon TextFrames, this section concentrates on the casting of individual displayed text lines. For any given line, a left margin, a maximal line width, a minimal line height *lsp*, a base line offset (descender) *dsr*, and a grid option are defined. How these values are associated with text lines is explained in the next section. If the grid option is not set, the resulting line height is at least *lsp*, but extended to the above and below to fully cover all characters (and elements) falling into that line (cf. Fig. 2). If the grid option is set (fig. 3), the line height and base line offset are incremented in steps of the minimal line height *lsp*, such that lines extended due to larger contained characters (or elements) fall onto the same grid as non–extended lines. From a typographical point of view, the resulting line spacing has nicer properties, while at the same time it may cause irritation by introducing unexpected white space into a text.

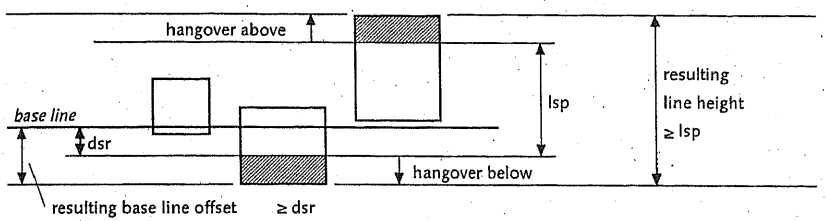

Figure 2. Line casting without grid option.

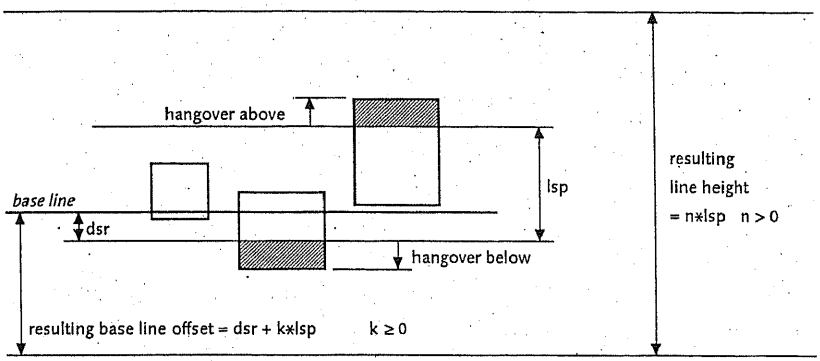

Figure 3. Line casting with grid option.

An element to be displayed within a line receives a prepare message. In turn, the element might change its geometry to adopt to its current environment (for examples, cf. 2.4). Then, the element is casted into a line and clipped against the frame boundaries. If the element is fully visible, it receives a draw message and in turn displays itself. Prepare and draw messages contain the font and color set for the element. The vertical offset attribute for elements is directly interpreted by *WriteFrames*.

```
TYPE
    WriteFrames.PrepareMsg = RECORD(Display.FrameMsg)
        fnt: Fonts.Font;
        col: SHORTINT;
        unit: LONGINT;    (*units per device pixel*)
        indent: LONGINT;    (*width already consumed in line, in units*)
        printing: BOOLEAN;
        pno: INTEGER;    (*page number, valid if printing*)
    END;
```

```
WriteTexts.DrawMsg = RECORD(Display.FrameMsg)
    fnt: Fonts.Font;
    col: SHORTINT;
    unit: LONGINT;   (*units per device pixel*)
    frame: Display.Frame;   (*containing frame in device space*)
    X0, Y0: INTEGER   (*absolute left-bottom coordinate in device space*)
END;
```

The messages have a special unit field that reflects the fact that *WriteTexts* has no knowledge of particular output devices and their resolution. Hence, all measures maintained within *WriteTexts* are in device independent units. For a precise definition of device independent units refer to appendix B. A typical conversion of, say, the width of an element *elem* to device space upon receiving message *msg* is a simple division: elem.W DIV msg.unit.

WriteFrames contains a sophisticated screen update mechanism to avoid unnecessary drawing to the screen. Typical edit operations, like character insertion and deletion, have local effects only. If necessary, preservable but ill-positioned portions of the frame contents are moved using fast bitblock transfers. The screen updating is the most complex operation contained in the Write system and thus has been optimized for frequent operations, only. Less frequent operations like viewer repositioning have not been fully optimized. However, in nearly all cases the implementation tries to avoid unnecessary screen flicker, for some rarely used operations this is done at the cost of increased processing time.

The screen update mechanism is based on a descriptor list attached to each frame. The list contains bounding box and positional information for each line displayed in the frame. When a change notification is received from the displayed text, the frame update proceeds in two phases. First of all, it is checked whether the change can at all affect the frame. If so, it is checked whether the frame origin (i.e. the position of the first displayed character) is affected. If it is affected, it is forced to a line start, and the whole frame is recasted. Hence, editing in the first displayed line can become rather inefficient. If the origin is not affected, two synchronization points in the descriptor list are seeked: The last descriptor which is still valid (as seen from top to bottom), and the first descriptor which is again valid if the corrsponding line is displaced by a certain offset. Between these two points and perhaps behind the last old descriptor, new descriptors are computed, i.e. the affected lines are casted. Once the descriptor list is reestablished, it is determined what amount of screen update needs to be performed and what screen blocks merely need to be shifted using bitblock moves. (Currently, it is not tried to preserve an unchanged suffix of an affected line.)

1.4 From Elements to Parcs – Introduction of Paragraph Structure

The previous section introduced several new attributes (e.g. line heights) that are no longer logically attached to a single character but to a range of characters (falling into a certain range of lines). It is tempting to introduce a new structure, overlayed over the existing text model, to implement such "paragraph" attributes. This temptation is resisted to stay with the initial principle not to introduce any extensions to texts other than extended characters. Hence, if no added explicit structures are to be used, it is necessary to introduce implicit structures.

A first approach to map paragraph attributes into a character stream would be the introduction of opening and closing bracket-like characters. However, this has the main disadvantage of unclear semantics in the case of overlapping paragraph ranges. Another problem is the switch of paragraph attributes in the middle of a line. The way chosen in Write is to indeed introduce special paragraph control characters, called parcs for short. The range of a parc is defined implicitly as reaching up to the next following parc or to the end of the text if there is no successor. To avoid the dilemma of inserting a parc into the middle of a line, a parc is always as wide as a whole line and hence is forced by the line casting algorithm onto the next line when not encountered at the beginning of a line.

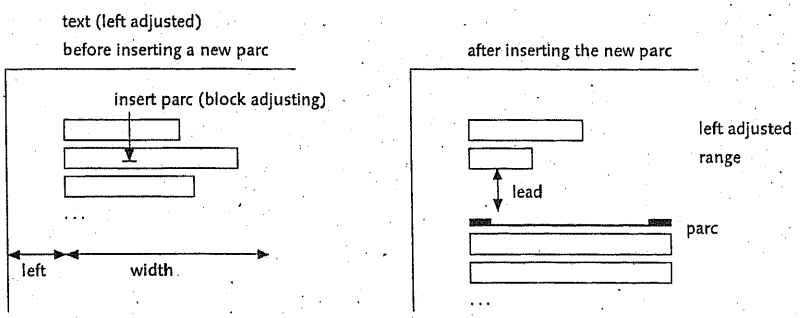

Figure 4. Effect of inserting a parc into a text.

The definition of a parc and operations thereon are listed below. A special procedure *WriteTexts.ParcBefore* allows for retrieving the parc valid for a certain text position. If the position is before the first parc within a text, a default parc attached to every Write text is returned.

```
TYPE
    Parc = POINTER TO ParcDesc;
    ParcDesc = RECORD(ElemDesc)
        left, width, lead, lsp, dsr: LONGINT;   (*in units*)
        opts: SET;
        nofTabs: INTEGER;
        tab: ARRAY MaxTabs OF LONGINT   (*in units*)
    END;

PROCEDURE ParcBefore(T: Text; pos: LONGINT): Parc;
    (*returns T.defParc if none found*)
PROCEDURE ParcExtent(P: Parc): LONGINT;
    (*parc P extends over range [ElemPos(P), ParcExtent(P) )*)
```

The many attributes associated with a parc can be queried and set using state messages. The commands Write.Get and Write.Set send such messages to the selected parc. However, for some attributes, direct manipulation would be preferable. Examples are the settings of tabulators, the left margin, the maximal line width, and the like. This is implemented in module *WriteParcs* as an extension of the parcs defined in *WriteTexts*. The precise meaning of all parc attributes and whether they can be set interactively is defined in Appendix A. While implementing a separated extension (the interactive changing of parc attributes), *WriteParcs* also defines a global variable used as prototype for default parcs when opening a new text. Therefore, *WriteParcs* is known by the text opening command *Write.Open* and thus belongs to the Write core system.

```
TYPE
    StateMsg = RECORD(Display.FrameMsg)
        id: INTEGER;
        par: Texts.Scanner;   (*scanned by receiver to extract individual parameters*)
        log: Texts.Text;   (*used by receiver to log error messages*)
        unit: LONGINT;
        frame: WriteFrames.Frame
    END;
```

1.5 Printing – Options and Compromises

When printing a text edited with an interactive, screen oriented editor, one often asks for "what you see is what you get" (wysiwyg) functionality, i.e. the image visible on the screen is more or less the same as what will be printed. Of course, this is limited by the screen resolution which is typically far lower than that of a modern laser printer. However, far worse is the effect of different resolution when considering rastered fonts. Since the current Write release does not reside on some device independent font model (e.g. PostScript or Display–PostScript [Ad85]), and the used pre–rastered fonts have been tuned by hand for optimal results on screen and on printer, each single character introduces an error in the order of 10%. As this is a systematic effect of the font tuning, the error does not average out, and assuming that optimal spacing is used on screen and on printer, a line of text is about 10 to 15% longer when printed than it is on screen. Note that a similar effect occurs if the fonts used on the screen are substituted by some other fonts when printing, as is done in some Oberon implementations e.g. for the Macintosh [Fr90] and Sparc [Te90] machines.

There are several possible solutions to this problem, and the solution chosen for Write is fully encapsulated in module Write itself (and could be changed easily). One could use the screen coordinates as dominating grid and force each printed character slightly besides its optimal position. This leads to optimal screen but sub–optimal printer output, as a misadjustment of >10% is visible for certain character constallations. The other way round, the printer coordinates could be chosen to dominate, leading to a sub–optimal screen output. This seems unacceptable, as the screen is the medium the author permanently looks at. Finally, a mixed strategy can be chosen. For example, one might decide to tolerate certain adjustment errors on the screen if text is formatted in a left flush fashion, etc. In Write a different approach has been chosen: When printing a text it is recasted for optimal printer results and thereby allowing for both, optimal screen and printer outputs. However, the price paid is a certain loss of wysiwyg functionality, as line breaks on the screen do no longer correspond to line breaks on the printer.

To be able to recast printed text according to printer font metrics requires some minimal information about printer metrics to be available to the print algorithm. Hence, Write uses a special metrics file per font family containing the horizontal displacement information for each character in each style available in the corresponding font family. These files are taken from the Leda document editor [Hi91] for the Oberon system and parallel the rastered font files.

Printing of elements is close to the process of displaying elements on the screen. After sending a prepare message, a print message is sent and the element in turn prints itself. (The print message contains the current page number which an element may use to produce page number dependent informations such as an index.)

```
TYPE
   PrintMsg = RECORD(Display.FrameMsg)
      fnt: Fonts.Font;
      col: SHORTINT;
      unit: LONGINT;   (*units per device pixel*)
      pno: INTEGER;   (*page number*)
      X0, Y0: INTEGER   (*absolute bottom–left coordinate in device space*)
   END;
```

Part 2 – Elements : Extensions to the Write System

The second part of this report concentrates on the main theme of an extensible system: It's extensions. For the Write system, extensions are called elements. As elements are objects in the sense of object–oriented programming, the first sections study the standard structure of such objects and the messages that need be understood. Building on this general view, a series of sections introduces successively more advanced element features together with available elements that make use of these features. For each of the discussed elements, a new problem and its solution is shown. Thereby it is demonstrated that the complexity introduced by an element is proportional to the level of functionality provided by that element.

2.1 The Structure of an Element Implemention

Elements are the key to the Write system's extensibility. Using the object–oriented style of programming, i.e. by implementing elements as objects, the functionality of possible elements is roughly circumscribed by the set of messages sent to an element by the Write core system. This section describes the typical structure of a module implementing an element. Two sample implementations of Write elements are listed in full detail in the appendix. The messages sent by the core system are summarized in the next section.

An element's implementation is typically encapsulated in a single module, although, in principle, it could take more than one module. Also, one might want to package multiple, perhaps correlated elements into a single module. Without loss of generality, it is assumed that a single element is implemented in a single module. Such a module needs to implement three basic groups of functionality. First of all, it provides a set of commands allowing a user to at least insert a new element of the implemented class into a Write text. Secondly, it contains the message handling for all messages understood by the implemented elements. Thirdly, it contains the hook for Write's generic dynamic loading facility. Each of these three functions will be explained in the following.

The Oberon system uses the concept of commands to link arbitrary modules to flexible user interfaces. By issueing a command, the user forces the system to execute a (parameterless) procedure exported by a particular module. If necessary, the required module is loaded. In the Write framework, elements found in opened texts cause their implementing modules to be loaded. To insert new elements into a text, each module is expected to provide appropriate commands. A typical insertion command takes some parameters to produce a customized element of the desired class. The created element is than inserted at the current caret position.

Once an element has been inserted into a Write text it is a potential receiver of messages sent by the Write core system, or sent by one of the participating element implementations. The used message handling framework corresponds closely to that used for the Oberon viewer system [WiGu88]. Messages are extensible records [Wi88a] dispatched to objects via object specific handlers. A handler uses a series of type tests applied to the message record to determine which action to take. It is possible for an object to ignore messages or to delegate them to some other object or module. This instance–centered style [Wi89] incorporates little conceptual overhead and opens designs with great flexibility. However, often the flexibility is more than one actually wants and the price paid in terms of weaker specifications and less clear system structure may be considered too high. While in princpiple, Write could adopt a class–centered style for its extension space (i.e. the element interface), this has not been done for the current version. The main rational for staying with the object–centered style is conformance with the Oberon system: It was a major design goal of Write to aim at a rather seamless integration with existing Oberon applications.

The handler associated with an element dispatches received messages using type tests. In order to be able to extend an element implementation, the handler should be exported. For each message to be handled, a handler should call a separately exported procedure (a handler component), as this enables an

extension to selectively call components of the overriden handler. Cf. appendix C for examples on element handler implementations. The following table summarizes the standard messages sent to elements by the Write system.

defining module	message	semantics
WriteTexts	Copy	return copy of receiver
	Draw	draw self to frame
	Load	load self from file
	Print	print self
	Store	store self to file
WriteFrames	Notify	used for broadcasts to visible elements
	Prepare	prepare self for drawing/printing
	Track	handle mouse tracking and mouse clicks
WriteParcs	State	get/set parc attributes

Table 2. Summary of Standard Messages to Write elements.

2.2 Temporary Elements – Error Elements

The minimal form of a Write element are temporary elements. Such elements are never stored and hence need neither methods to load or store themselves nor a generic allocation mechanism. Typically, temporary elements do not print either and thus understand little more than Draw and perhaps Prepare messages. Upon storing a text all temporary elements vanish automatically.

An example for temporary elements are *ErrorElems* as listed in full source in appendix C. When using Write to edit programs, error elements can be used to translate error lists produced by a compiler into special marking elements within the source text. Since elements move when editing text, it is possible to find all error locations even when the text has been edited. By clicking on an error element, the element expands itself and displays the corresponding error message. (Cf. sections 2.4 and 2.6.)

```
MODULE DoubleBug;
    PROCEDURE One;
    END Two; ■
BEGIN Three
END [undeclared identifier] DoubleBug.
```

The following list summarizes the commands implemented by *ErrorElems*. The same format will be used throughout this report. For the conventions used to describe the parameters taken, refer to appendix A.

command	*explicit parameters*	*implicit parameters*
ErrorElems.Mark	("↑" | error–list)	marked viewer

Takes the list of errors produced by a compiler (position, error code pairs) and inserts error marks into the marked text. If error marks were already present in that text, they are removed beforehand.

ErrorElems.Unmark — marked viewer

Remove all error elements from the marked text.

ErrorElems.LocateNext — marked viewer, caret

Locate the next error element starting the search at the caret position. If no caret is set in the marked viewer, the first error element is located.

2.3 Static Elements – Chart Elements *(ChartElems due to C. Pfister)*

When adding interpretation of standard messages for loading, storing, and printing an element, the straightforward class of so called static elements results. Such static elements are very close to plain characters in that their size and content are fixed at insertion time.

A typical static element implementation are *ChartElems*. Taking a list of specially marked numbers, *ChartElems* generates a series of simple rectangular elements arranged as a horizontal or vertical bar chart. Each of the chart elements displays (and prints) itself as a frame scaled to the value consumed when creating it.

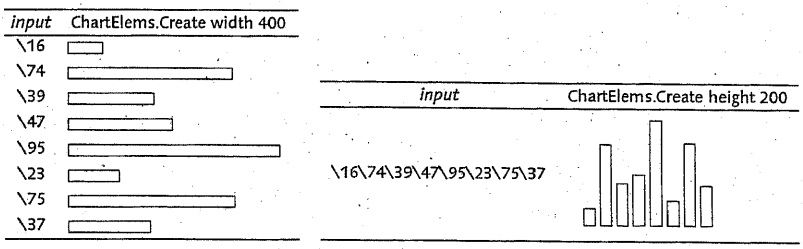

Figure 5. Horizontal and Vertical Bar Charts.

command	*explicit parameters*	*implicit parameters*
ChartElems.Create	("width" | "height") number	selection

Starting from the most recent selection, the text is scanned until a "~" is found. All numbers prefixed with a backslash are replaced by a chart element. Everything else remains untouched.

2.4 Adaptive Elements – Line Elements

For certain elements it is meaningful to automatically adapt to the current environment when getting displayed or printed. For example, an element may have different looks when displayed and when printed. This can be used to implement elements like bookmarks that are completely suppressed when printed. Also, an element may automatically adapt to the available line height, line width, or remaining space on a partially casted line. This latter feature is used by *LineElems* to provide simple horizontal or vertical lines.

An element adapts to the display or printing environment by interpreting the prepare message. The message contains information about the space already consumed in the currently casted line, whether the element is about to be printed or displayed, and if it is going to be printed, on which page that will happen. Thus an element may even adapt to the page number during the print process. Furthermore, an element receiving a prepare message can retrieve the parc valid for its position and thereby change its geometry depending on all parc attributes, like line width and tabulator settings. The following table gives some examples.

width	height	first tab	second tab
auto	1		
auto	5		
auto	auto		
tab	5		
tab	auto		
1	auto		
5	auto		
20	auto		

Table 3. Various Line Elements.

command	*explicit parameters*	*implicit parameters*
LineElems.Insert	("↑" | (("auto" | "tab" | W) ("auto" | H)))	caret

Inserts a line of given or automatically adopted width and height into a Write text. Width and height may be set in units of 1/10 mm. Otherwise, the width may be set to extend to the end of the current line ("auto") or to the next tabulator position ("tab"). The height may be set to equal the paragraph's line height (determined by the parc's line attribute).

2.5 Active Elements – Clock Elements, Icon Elements *(Clock–, IconElems due to R. Griesemer)*

Besides passively adapting to the display or print environment, elements may be active in the sense that some text external events like the passing of time affect a displayed element. Two examples (which of course are not visibly active in this printed report) are clock and icon elements. The basic mechanism is the separation of model and view, where the actual element floating in the text takes over the role of a view to some connected model. In the case of clock and icon elements, the external model is little more than the real–time clock, i.e. the absolute time. As shown below, clock elements simulate an analog clock, while icon elements display kind of an icon sequence film. (As an added feature, icon elements act as screen savers when being clicked on.)

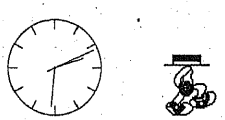

Figure 6. A Clock and an Icon Element.

The model needs to notify its views when some update needs to be performed. As the model (e.g. the clock–polling task) has no control over the location and number of currently visible views (i.e. displayed elements), a special message broadcast mechanism is used. The model broadcasts a notify message to all viewers (using the Oberon viewer system's broadcast) which delegate it to their subframes. If a *WriteFrames* frame receives such a notify message it adds its own identity to the message and broadcasts it to all elements in the visible range (using the procedure *WriteTexts.Broadcast*). This is done using the range broadcast mechanism of *WriteTexts*.

```
TYPE
    WriteFrames.NotifyMsg = RECORD(Display.FrameMsg)
        unit: LONGINT;
        frame: Frame
    END;
```

A view (an active element) receiving a notify message in turn asks the containing frame for its current location, and whether it is fully visible. (Remember that elements are completely clipped when not fully visible.) This is done by calling *WriteFrames.LocateElem*. If the receiving element indeed is fully visible, it queries its model and thereby learns about the new state to be displayed. In turn, the element may redraw (part of) itself. If an element needs to resize itself as a reaction to the changes, it does change its geometry and state immediately and triggers a full redraw within a recasted environment by notifying the containing text of its change. To do so, the element calls *WriteTexts.ChangedElem*. In the former case, i.e. when redrawing in place, the element should use inverse mode drawing operations, only, as the element may be selected and therefore displayed in reverse video. In the latter case, to avoid multiple view updates it must be checked whether the model has really changed: The notify message might be received by multiple views showing the same element, but the first one will update all others by calling *WriteTexts.ChangedElem*.

Storing an active element and its model requires additional thought. As several elements might relate to the same model, multiple storage of the model should be avoided and the N:1 relation should be re-established at load-time. To distinguish between elements receiving a store message during a single store action, and elements receiving a store message in consecutive stores, variable *WriteTexts.storeTime* can be used. It is set to Oberon.Time() at the beginning of each call to *WriteTexts.Store*. This can be used to reset a model dictionary used during element storage whenever a timestamp associated with the dictionary fails to match *WriteTexts.storeTime*.

command	*explicit parameters*	*implicit parameters*
ClockElems.Insert	[number]	caret
Insert a clock (with radius given in pixels, default is 32, minimum is 12) at the caret position.		
IconElems.Insert		caret
Inserts a new Icon Element at the caret position.		

2.6 Interactive Elements – Popup and Style Elements *(PopupElems due to M. Franz)*

Several of the elements discussed so far already added some degree of user interaction by interpreting certain mouse clicks. This can be generalized to fully interactive elements as a *WriteFrames* frame delegates all mouse clicks that happen within a displayed and fully visible element to that element by sending a track message. By tracking the mouse and thereby consuming the mouse event, an element can handle mouse click combinations. If it ignores the track message or a certain mouse click combination, the tracking is performed by the enclosing frame.

From a user's point of view, nested editable objects add a nonneglectable amount of complexity. If floating elements support full editing in place, the user has to distinguish three different kinds of mouse clicks interpreted in potentially very narrow or even overlapping screen areas. First of all, certain mouse clicks are on the level of the containing frame, and ask for placing the caret or selecting some text stretch. Secondly, other mouse operations are required to operate on a floating element, like resizing it. Thirdly, again other mouse operations are meant to operate within an element, such as editing its contents. Things get even worse if arbitrary nesting of interactive objects is to be allowed.

Within Write the convention is adopted not to support direct editing of nested objects, i.e. elements

floating within a text. This is not a limitation of the framework, as each element can easily implement in–place editing if desired. However, experience has shown that such in–place operations get complex and unhandy, especially for the casual user of the system. Hence, the much simpler method is preferred to open a separate editing viewer when clicking on an element. Usually the middle mouse button is used as it has no obvious semantics for floating elements, anyway.

Two rather different examples for such interactive elements are *PopupElems* and *StyleElems*. *PopupElems* implement popup menus containing arbitrary and editable Oberon commands. Using the middle mouse button, a popup menu is opened; a middle–right interclick opens a viewer displaying the commands to be edited. *StyleElems* extend standard parcs (which already are interactive) by giving them a name: All style parcs within a text that have the same name change synchronously. When copying a style parc from one text to another, the parc will be copied as is, if the target text does not already contain a style of that name. If it does, the copied parc will be changed to conform to the existing style. Note that *StyleElems* give an example on how to extend extensions (in this case parcs), while all other examples given in this report are extensions of the element base type.

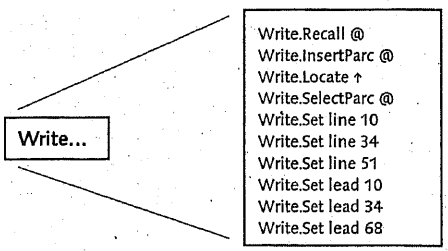

Figure 7. A Popup Element.

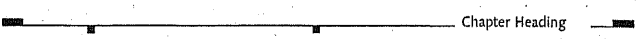

Figure 8. A Style Element named "Chapter Heading".

command	*explicit parameters*	*implicit parameters*

PopupElems.Insert string caret

Inserts a new popup element with an initially empty menu. The menu may be edited by opening an editor using a middle–right interclick.

StyleElems.Insert ("↑" | name | string) caret

Insert a new style parc with the given name at the caret position. A quoted string may be used to assign names consisting of multiple words. If the target text already contains a style with the given name, the inserted parc will adopt that existing style. Otherwise, the style of the target text's default parc will be used.

StyleElems.Rename ("↑" | name | string) selected parc

Renames the selected parc to the given name. If the text already contains a style with the target name, the selected parc will be changed to conform to that existing style.

2.7 Wrapper Elements – Graphics and Picture Elements *(PictElems due to H. Marais and K. Rege)*

Having interactive elements, a straightforward consequence is to add elements that enclose some existing editable object class not originally designed to be encapsulated in an element. Such elements are called

wrappers, as they merely provide the glue required to "wrap" some object and let it appear as an element. Here, the convention developed above, that it is preferable to edit elements in a separate viewer instead of in-place, gets a new strengthening argument. As the existing object classes necessarily do not know about elements, utility commands developed to edit them will not work as they cannot be applied to a floating element. However, the separate editing viewer opened for such elements behaves like the standard viewer used for the corresponding object class opened and is therefore directly accepted by such commands.

Currently, there are two wrapper elements available: *GraphicElems* and *PictElems*, wrapping the standard Oberon line drawing system Draw and the picture editing system Paint, respectively. *GraphicElems* have already been used extensively throughout this report to add all kind of illustrations.

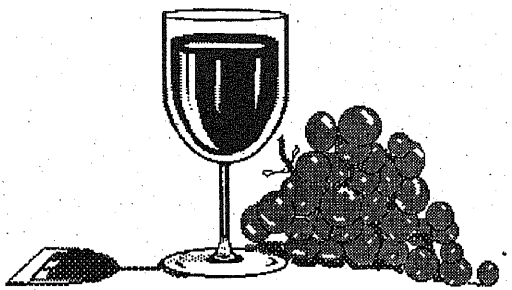

Figure 9. A Picture Element.

command	*explicit parameters*	*implicit parameters*
GraphicElems.Insert	("↑" | "*" | name)	caret, marked viewer

Inserts a graphics element sized to show the graph stored in graphics file "name". If an open graphics viewer is marked, the command can be used to insert all or part of a displayed graph: if a selection exists in the marked graphics viewer's graph, only the selection is copied into the graphics element.

PictElems.Insert	name ["scaled"]	caret

Inserts the named picture at the caret position. If "scaled" is added, the picture will be scaled to fit into a 3 by 3 cm area. Otherwise, the picture will be displayed in full size. The middle mouse button may be used to open a Paint editing viewer. When clicking into the lower right corner, a scaled picture can be resized.

2.8 Structuring Elements – Fold Elements *(FoldElems due to H. Mössenböck)*

Like parcs, elements may be conceived that operate on their environment. While formatting modes must be known to the formatting instance, i.e. the frame and the print mechanism, other structuring effects can be introduced freely. However, such structures should be defined implicitly as the stretch between certain elements floating in a text, as editing operations cannot be constrained and hence consistency of explicit structures cannot be guaranteed.

A rather mighty example are fold elements which allow to structure a text into hierarchically folded segments. Folds can be expanded or collapsed at will. The idea is to support zooming into areas of interest while at the same time preserving an overview picture when collapsing folds. There are two different kinds of fold elements: Fold opening and closing ones. To avoid explicit structures, the opening elements contain all the relevant information attached to a fold, while the closing elements are matched on demand using a simple syntactic rule. Missing closing elements are catched by the text's end,

superfluous closing elements are ignored. Thus, the user can freely edit the folded text, and inconsistent states are not possible.

Folding a text stretch causes its replacement by another, usually shorter or empty text stretch. Unfolding a folded stretch reveils the original text stretch while hiding its replacement. It is possible to unfold or to fold all folds existing in a text. Also, a (partially) folded text may be compiled as if it where fully unfolded using a utility module *FoldComp*. *FoldComp* also includes the functionality of *ErrorElems*, as defined elsewhere in this report, but uses a different file *Oberon.Errors* to translate error codes into error messages. *Oberon.Errors* also contains the name of the compiler to be used by *FoldComp*.

```
PROCEDURE GetName(VAR n: Name);                    ►◄ closed fold

PROCEDURE GetName(VAR n: Name);                    ▷ opened fold
   VAR s: Texts.Scanner;
BEGIN
   Texts.OpenScanner(s, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(s);
   IF s.class = Texts.Name THEN COPY(s.s, n) ELSE n := "" END
END GetName;
◁
```

Figure 10. A Text containing Fold Elements.

command	*explicit parameters*	*implicit parameters*

FoldElems.Insert — selection
Make selection foldable by inserting an appropriate pair of opening and closing fold elements before and after the selected text stretch, respectively.

FoldElems.Expand — marked viewer
Unfolds (expands) all folds contained in the marked text.

FoldElems.Collapse — marked viewer
Folds (collapses) all folds contained in the marked text.

FoldElems.Marks ("↑" | "on" | "off") — marked viewer
Make all fold elements in the marked text visible or invisible.

FoldElems.Search — selection, marked viewer
Search for the selected text stretch in the (partially) folded marked text starting at the caret position. Does search within folded parts and opens folds if necessary to display found text stretches. If no selection is present, the last search pattern is used again. If no caret is present, the search starts at the text's beginning.

FoldComp.Compile ("*" [compiler–options]) — marked viewer
Compile the marked text containing folds. The compiler is started on the *fully expanded* view of the text.

FoldComp.ShowError — caret, marked viewer
Show next error after the caret position. If no caret is present, the first error in the marked text is displayed.

2.9 Formatting Elements – Table Elements

The elements discussed so far added new object types embedded in Write elements. By introducing Write texts as element contents, recursion and the potential of recursive constructors can be implemented. By doing so, a Write element can incorporate its own world of text representation and formatting to serve special purposes. A quite powerful example that makes use of recursively embedded Write texts are table elements. A table element automatically translates a primitive data format into tables, examples for which can be found all over this report. The last generic element class introduced in this report is intentionally exemplified using a rather heavy–weight extension: The quite mighty table elements demonstrate the potential available within Write's extension scheme.

Starting from a defining Write text containing arbitrary strings separated by tabulator and carriage–return characters, a table is constructed. Tabulators separate columns, carriage–returns separate rows. The separated strings form the contents of table cells. Each such string may in turn contain arbitrary Write elements and especially it may contain nested tables. The defining text is bound to a table element and can be edited by clicking on a table with the middle mouse button. In return, a standard Write viewer is opened containing an update command in its menu bar. The syntax of the defining text is as follows, table options will be defined later.

```
table = {option} "/table" {line}.
line = cell {TAB cell} CR.
cell = ["#" | "&"] <string of characters not containing TAB or CR>.
```

A string starting with a digit, a period (.), or a sign (+ or –) is considered numeric, otherwise it is considered nonnumeric. A number sign (#) prefixing a string is suppressed and enforces numerical interpretation; likewise, an ampersand (&) prefixing a string is suppressed and enforces nonnumerical interpretation. The default table formatting causes all cells in the first column to be formatted left flush. In all other columns, the default format for numeric cells is period–aligned and that for nonnumeric cells is centered. All cells in a row are by default aligned to a common base line. Also, the default format causes the table to be framed and separating lines between rows and columns to be drawn. Finally, for a table with more than two rows or two columns the first row or column separatation line is drawn using two parallel lines, respectively. Most of the default formats can be changed using the provided option set.

For table elements the printing dilemma discussed in section 1.5 needs to be reconsidered. Following the strategy developed for Write texts, a table would be recasted when printing. However, one of the primary attributes of a table are its measures, i.e. the space taken for individual cells of the table and the arrangement of the cells with respect to each other. It is not acceptable that a table gets completely redimensioned when printed. Hence, for table elements a different printing strategy has been choosen: the displayed table on the screen follows – down to the character level – the metrics of the printer fonts. The resulting screen image looks a bit distorted, however, the displayed measures correspond as closely as possible to the printed ones.

cell type \ col. format	left	center	right	numeric
alpha	abc	abc	abc	abc
integer	42	42	42	42
real	1.3	1.23	1.3	1.3
negative	–12.34	–12.34	–12.34	–12.34

Table 4. Column Formatting.

row format \ cell type	alpha	subtable 1	subtable 2	subtable 3	subtable 4
top	Yepp	single line	double line	even one more	double w/ offset
center	Yepp	single line	double line	even one more	double w/ offset
line	Yepp	single line	double line	even one more	double w/ offset
bottom	Yepp	single line	double line	even one more	double w/ offset

Table 5. Row Formatting.

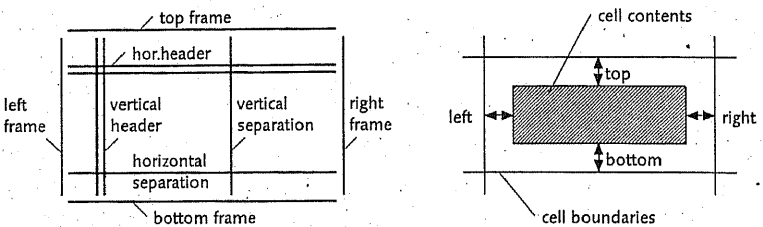

Figure 11. Table Lines and Cell Margins.

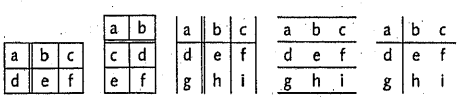

Figure 12. Frame Formatting.

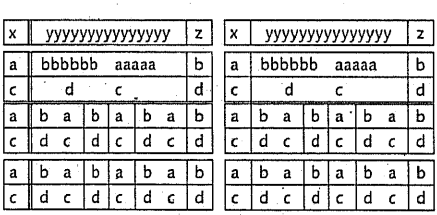

Figure 13. A Complicated Composed Table.

command	*explicit parameters*	*implicit parameters*
TableElems.Insert	("↑" | name)	caret

Inserts a table using the named text as defining text. If no such text exists, an empty table is inserted.

option	effect (measures in 1/10 mm)	argument values
/columns string	control column formatting string = {L|C|R|N|–}.	L left flush C centered R right flush N numeric (period adjusted) – default
/rows string	control row formatting string = {B|L|C|T|–}.	B bottom flush L common baseline C centered T top flush – default
/noheads string	turn header lines off string = {H|V |*}.	H horizontal V vertical * all
/nolines string	turn separation lines off string = {L|R|B|T|H|V|*}.	L left frame R right frame B bottom frame T top frame H horizontal separation V vertical separation * all
/period string	first char redefines period (e.g. ",")	
/left integer	left cell margin	
/right integer	right cell margin	
/bottom integer	bottom cell margin	
/top integer	top cell margin	
/grid integer	cell width grid *)	

*) A parc at the beginning of the table defining text introduces a minimal cell height as well as a cell height grid (if the grid option of the parc is set)

Table 6. Overview of Formatting Options.

2.10 Special Purpose Elements: Building an Application using Write *(Example due to B.Heeb)*

Beyond extending Write's usability as a text editor by providing new elements, Write can be used as a framework for rather specific applications. By decomposing the visual elements of an application into fragments and implementing these fragments as Write elements, a large part of the code typically required to implement interactive applications can be avoided. By interpreting a Write text as an application specific "panel" the available editing operations can be used to freely arrange the application's visible components. To close the second part of this report, an example of such an application is given.

To interactively simulate electronic circuits different signals found in the circuit are to be displayed as trace diagrams showing the change of a signal over simulation time. The Debora simulator is part of an ongoing research project. It uses a special Write element per trace. When displaying itself a trace element scans the name written in front of it to dynamically associate a simulated signal to be displayed. Furthermore, a trace element uses the tab setting of the valid parc to derive the time unit, i.e. the scale at which to display the trace. In turn, the user can freely arrange traces annotated with their natural signal names and select the wished time base. Trace elements are also interactive and can serve as input to a simulation, too, as it is possible to edit a trace in several ways.

As this section is not meant to be the documentation of the Debora toolset still under development, no further details of trace elements will be looked at. However, it should be noted that the mere implementation of trace elements is sufficient to construct an application that allows free composition of signal traces together with arbitrary Write texts as annotations. The resulting active texts can be frozen

and printed at any time.

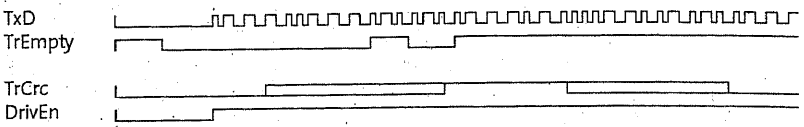

Figure 14. Debora Simulation Elements.

Summary and Conclusions

The delicate balance between a rich basis for arbitrary extensions and carefully chosen restrictions to certain simple concepts determines the usefulness of an extensible system. It should be clearly noted that the unrestricted extension potential tends to be overshadowed by the massive burden that every actual extensions needs to carry, even if it is meant to perform a rather straightforward task. A compromise is necessary to enable the common case of simple extensions at a low implementational and conceptual price.

This report concentrated on the extensible text editor Write which attempts to limit extensions to a single atomic concept: The character. As texts are naturally thought of as a (formatted) sequence of (attributed) characters, this model is quite intuitive in capturing the idea of extending the *attribute space* of a character. However, some further techniques are required to capture the *formatting space* of a text. In Write both concepts have been coerced by introducing special characters, called parcs (paragraph controls), that implicitly define formatting attributes over character ranges. In this sense, parcs may be compared to *rulers* found in many other text editors. Indeed, at least the LisaWrite editor [Wi83] and its successors (like MacWrite) treated rulers nearly as characters in that all standard character editing operations can be applied to rulers.

It is very important that parcs are realized without adding a new structure to the system. Instead, the existing structure enabling all element extensions of the editor, can be queried to find the parc responsible for a certain position in a Write text. Likewise, all other extensions illustrated in this report work without any additional structures superimposed on the text carrying the extensions. In all cases, required information is derived from the implicit relation of elements positioned in a text. The most advanced example for this technique are folding elements which by their nature always act in pairs. Instead of linking a pair of elements, a fold element seeks its partner element on demand. If the elements would contain links, maintaining consistency would lead to necessary modifications or restrictions of the user's editing model. Consider the case where the user deletes a fold element but not its partner element. Or even worse: One of the two elements is moved to a different text but not the other. To cope with such cases in the case of an explicit structure (e.g. pointers between elements not known to Write), the system would have to either *restrict* the edit operations, or the edited elements need to be notified in order to *repair* the text, e.g. to make it consistent again.

The large number and variety of the existing element extensions discussed in this report serve to strengthen the point. The simple model does indeed *enable* a quite rich world of extensions. The simplicity of two sample implementations of element extensions given in the appendix clearly demonstrates that it is also *easy* to add new extensions. The latter argument is further supported by the large number of different authors that already implemented Write extensions. Finally, the set of nontrivial extensions presented (like popup, fold, graphics, picture, or table elements) shows that the Write extension world is not limited to toy examples. Last but not least, the approach to restricted extensibility led to a relatively simple core system. The major gain is its robustness: After the first release no destructive bugs have been found. Also, most bugs found corresponded to the frame update mechanism which is indeed one of the most complex parts.

Write has been implemented in the Oberon system using the Oberon language. Both proved to be valuable tools to rapidly design and implement a functioning and useful application. However, certain aspects of the Oberon system hinder a free extension even at levels as those of the Write system. Especially, the large number of statically bound procedural interfaces to central components make certain extensions inherently unsafe. A class–centered style based on a dynamic binding of procedures would help; the ongoing Ethos project investigates this further as a general principle of structuring an operating system [Szy90a, b].

The reader might notice that this report is kind of an existence proof of Write, as it has been written and printed using solely Write and all of the mentioned Write elements. The whole report has been handled as a single Write file with good performance on a Ceres–2 machine (25MHz National Semiconductor 32532 Processor). It contains 16635 words, 109393 chars, 180 elements, and takes

156707 bytes to store. Appendix B adds some measures on the system's code size.

Acknowledgements

I would like to express my deep appreciation to the large number of early users of the Write system. Providing a useful system as the result of a short–termed project is a challange and requires a certain amount of "guinea–pigs" to test results on. (However, the fact that everybody stayed alive and that many of the Write users actually became implementors by adding new elements is quite encouraging.) Also, I would like to thank H. Mössenböck and J. Templ for careful reading and many valuable comments on this report.

References

[Ad85] Adobe Systems, Inc. *PostScript Language Reference Manual.* Addison–Wesley, Reading, MA, 1985.

[Fr90] M. Franz. *The Implementation of MacOberon.* Tech. Report 141, Institut für Computersysteme, ETH Zurich, October 1990.

[Gu90] J. Gutknecht. *The Oberon Guide. System Release 1.2.* Tech. Report 138, Institut für Computersysteme, ETH Zurich, October 1990.

[Hi91] U. Hiestand. *Leda – Das Dokumentenverarbeitungssystem für Oberon.* Tech. Report to appear, Institut für Computersysteme, ETH Zurich.

[Szy90a] C.A. Szyperski. *Towards Object–Oriented Structures for Open Operating Systems.* Presented at the Workshop on Object Orientation in Operating Systems at OOPSLA–ECOOP '90, Ottawa, Oct 1990.

[Szy90b] C.A. Szyperski. *The Carrier / Rider Separation. A New Structuring Concept for Open Operating Systems.* Submitted to ECOOP '91, Geneva, July 1991.

[Te90] J. Templ. *SPARC Oberon – User's Guide and Implementation.* Tech. Report 133, Institut für Computersysteme, ETH Zurich, June 1990.

[WeGaMa89] A. Weinand, E. Gamma, R. Marty. *Design and Implementation of ET++, a Seamless Object–Oriented Application Framework.* Structured Programming, 10:2, 63–87, February 1989.

[Wi83] G. Williams. The Lisa Computer System. BYTE 2, February 1983.

[Wi88a] N. Wirth. *Type Extensions.* ACM Trans. on Progr. Lang. and Systems. 10:2, 204–214. April 1988.

[Wi88b] N. Wirth. *The Programming Language Oberon.* Software – Practice and Experience. 18:7, 671–690. July 1988.

[Wi89] N. Wirth. Modula–2 and Object–Oriented Programming. First International Conf. on Modula–2, Bled, Yugoslavia, Oct 1989. (also: Tech. Report 117, Institut für Computersysteme, ETH Zurich, October 1989.)

[WiGu88] N. Wirth, J. Gutknecht. *The Oberon System.* Tech. Report 88, Institut für Computersysteme, ETH Zurich, July 1988.

Appendix A – Write User Manual

Write is an extensible text editor for the Oberon system. It is based on a few simple concepts and aims at typical writing tasks, such as memos and reports. It does not try to support a document model, and it is not fully "wysiwyg". To arrive at a rather simple implementation, the few concepts provided have been carried out with all consequences. This should be kept in mind before considering a certain behaviour of the editor to be a "bug". The following tutorial assumes that the reader already knows how to use the Oberon system, especially, how to handle viewers, files, and commands. Many important Write extensions, called elements, are documented in the second part of this report.

Principles

A Write text models a sequence of characters. Besides standard characters, Write supports special user extensible characters. Such characters, called elements, float in the text just as ordinary characters do. Typical elements support the integration of graphics and the like into texts. A standard extension of elements allows for separating the text into paragraphs. Such elements are called paragraph controls or parc for short. A parc defines the paragraph attributes for all characters following it up to the next parc or the end of the text.

Mouse commands are interpreted by Write just as defined by Edit. Exceptions to that occur when clicking on an element. Here, the element is free to consume (and act on) the mouse click, overriding the standard behaviour. This is used by parcs to support interactive setting of paragraph attributes.

Insertion of new characters into the text requires setting the caret left to the character before which to insert, or to the end of the text. Selecting a stretch of the text corresponds to selecting a range of characters. The visible selection extends from the beginning of the first selected character to the beginning of the first character behind the selection or the end of the text. Hence, selections always reflect the exact area which is affected when changing or deleting.

Editing

Basic editing functionality is just as in Edit. This includes scrolling, setting the caret, selecting text, inserting, copying, and deleting text stretches, and changing attributes of characters. The *cursor left/right* keys move the caret. To support editing indented text (like programs), the *Linefeed* key can be used instead of Carriage–Return. This will indent the next line to the same level as the previous one. The *cursor left/right* keys will shift indented text if it is selected in the focus viewer with the caret invisible. Furthermore, when extending a selection over two consecutive viewers, Write adds visual feedback. This avoids the danger of extending a selection by mistake and then deleting or copying a far larger stretch of text than was intended. See below for a description of commands that further support basic editing.

In order to insert a parc into a text, place the caret at the appropriate position and press BREAK. This will copy the parc above the caret position (or the default parc if there is none above). A parc is displayed as a separation line corresponding to the width set for that paragraph. Above that separation line, one or two marks signal the formatting mode of the paragraph: a single mark at the very left, in the middle, or at the very right indicates left flush, centered, or right flush formatting, respectively. Two marks at both ends indicate block (fully justified) formatting. Below the separation line, set tabs are indicated by marks. Pressing Shift–BREAK inserts a parc which forces a page break (attribute break=before). Such parcs are displayed using a solid separation line.

A parc defines a special behaviour for *middle–button* mouse clicks. It has two separate sensitive areas. One above and one below the separation line. The following table shows how various middle mouse button clicks and interclicks affect the paragraph attributes bound to a parc. Again, see the commands below for further manipulations of parcs.

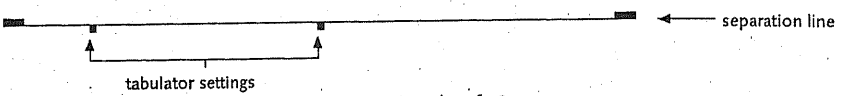

Figure 15. Display of a Parc.

where	*buttons*	*effect*
above separation line, left end	middle	set left margin
above separation line, left end	middle + right	symmetrically set left & right margin
above separation line, right end	middle	set right margin
above separation line, right end	middle + right	symmetrically set left & right margin
above separation line, towards left end	middle	left flush formatting
above separation line, in the middle	middle	centered formatting
above separation line, towards right end	middle	right flush formatting
above separation line	middle + left	block formatting
below separation line, no tab mark hit	middle	create new tab
below separation line, tab mark hit	middle	move tab
below separation line, tab mark hit	middle + right	move tab and all subsequent tabs in synch
below separation line, tab mark hit	middle + left	delete tab

Table 7. Interactive Parc Manipulation.

Printing

When printing a text, Write reformats individual paragraphs for the printer. Write preserves all user setable measures, while individual words may be placed differently. This allows for optimal display of texts on the screen, while at the same time fully exploiting the printer resolution. Hence, it *should not be tried* to affect the placing of individual words by inserting additional blanks or the like! Furthermore, Write ignores all empty space at the beginning of a page except when preceded by a parc with enforced page break attribute. White space in this sense are empty lines (a single blank makes a line non–empty!) and lead–space defined by a parc. Refer to the print command description below for details on how to initiate a printout.

In order to achieve good results, the *Lm3 metrics files* for the used font families should be available. For example, when using the fonts Syntax12, Math12, Syntax14, and LetterHead, the metrics files Syntax.Lm3.Fnt, Math.Lm3.Fnt, and LetterHead.Lm3.Fnt are required. If a metrics file for a used font is missing, the printer metrics are estimated using a simple heuristics based on the screen font metrics. *Beware*: this leads to at most draft quality of the printed text.

Commands in the Viewer Menu

Write.Search
Takes the most recent selection as search argument. If the selection is older than the latest one used for Write.Search, the previously set search argument will be used. Write.Search starts searching at the current caret position, or at the beginning of the text, if no caret is set. When starting the search behind the last occurence of the search pattern, the search will automatically wrap around and start over again at the beginning of the text.

Write.Replace
Takes the most recent selection as replacement argument. If the selection is older than the latest one

used for Write.Replace, the previously set replacement argument will be used. Write.Replace verifies that the pattern right to the current caret position matches the current search argument. If so, it is replaced and Write.Replace automatically searches for the next occurence. Write.Replace does not wrap around when searching.

Write.ReplaceAll
To make Write.ReplaceAll usable, the separating space in the menu has to be deleted. Write.ReplaceAll operates much the same way as Write.Replace does. Additionally, the replacing process is carried on to the end of the text. Write.ReplaceAll does not wrap around when searching.

Write.Store
Stores the text. After completing, Write.Store writes out the number of characters in the text. Whenever the text displayed in a Write viewer has been changed but not yet stored, the menu bar contains an exclamation mark following the Write.Store command. If a text is stored under a name that already exists, the existing file *X* is renamed to *X.Bak*.

Commands in the Write Tool

The following commands are supported by Write. Generally, the standard Write tool contains several examples on how to use the Write commands, most of which should be self–explaining. For each command, the parameters directly following the command are given. Furthermore, most of the commands take further implicit parameters, like the current selection, the current focus (the caret), and the selection in the currently marked viewer. Such parameters are also listed below. Whenever the marked viewer (or the selection in its body frame) is taken as a parameter, the viewer is expected to display a Write text. The selection symbol "↑" adds a level of indirection by taking the explicit parameters of the command from the current selection. Explicit parameters may be *names* (sequences of characters starting with a letter, followed by letters, digits, or periods), *strings* (sequence of characters enclosed by quotes), or *numbers* (sequences of characters starting with a minus or a digit, followed by digits).

command	*explicit parameters*	*implicit parameters*

Write.Open ("↑" | name)
Opens a Write viewer displaying the named text.

Write.SysOpen ("↑" | name)
Opens a Write viewer in the system track displaying the named text. The default formatting uses a reduced width and a regular tabulator setting every 5mm.

Write.Store name marked viewer
Takes the text displayed in the marked viewer and stores it under the given name. The total number of bytes taken by the created file is written to the log.

Write.Print server–name ("↑" | "*" {option} | {name {option}} "~")
Prints a list of Write texts. The print options are explained in a separate table below. Unless specified otherwise (using the /p option), Write.Print will assign consecutive page numbers to all texts printed with a single print command. As feedback, Write.Print writes the name and the number of copies of each printed text to the system log. Also, a period (.) is written to the log for each page sent to the printer and an apostrophe (') is written for each page formatted but skipped (due to an /s option).

Write.Recall caret
Inserts the most recently deleted text stretch at the current caret position. Write.Recall can be used repeatedly to insert a deleted portion at several places. (Edit.Recall does nothing when applied to a Write

text.)

Write.InsertParc caret
Inserts a new default parc at the current caret position.

Write.Locate ("↑" | number) marked viewer
Locates a character position and makes it visible in the marked viewer. (Do not use Edit.Locate to locate a position in a Write text!)

Write.SelectParc caret
Make visible the parc corresponding to the paragraph containing the caret and select it.

Write.ChangeFont ("↑" | name) selection in marked viewer
Change the font attribute of the selected characters in the marked viewer.

Write.ChangeColor ("↑" | number) selection in marked viewer
Change the color attribute of the selected characters in the marked viewer. (Has no visible effect on monochrome monitors, on color monitors; 0 is the background and 15 the foreground color.)

Write.ChangeOffset ("↑" | number) selection in marked viewer
Change the vertical offset attribute of the selected characters in the marked viewer (–128..127). The offset is specified in 1/64th of the font height of the affected character.

Write.Set ("↑" | {attribute–name [values]}) selected parc in marked viewer
Sets paragraph attributes bound to the selected parc. The paragraph attributes are explained in a separate table below.

Write.Get ("↑" | attribute–name) selected parc in marked viewer
Gets paragraph attributes bound to the selected parc. The paragraph attributes are the same as for Write.Set. If no special paragraph attribute is selected, all attributes will be shown.

Write.MakeDefault selected parc in marked viewer
Sets the default formatting of the marked text to the attributes of the selected parc.

Write.ShowAliens marked viewer
Searches the marked text for undefined elements. Such elements occur, if a text is opened while some defining modules of extended elements are not available. Write displays undefined elements uniformly as an empty frame. Write.ShowAliens lists the position of such elements, as well as the names of the missing modules. Note that undefined elements can be selected, copied and deleted freely. As soon as the missing modules become available, it is sufficient to reopen affected texts and the elements will again behave as originally intended.

Print Options

By supplying print options to the Write.Print command, certain print conventions may be controlled. Options may be given in any order, but note that certain options may depend on values set by others. For example, "/p 5/s 5 7" starts page numbering with 5 and causes printing pages 5 to 7, i.e. the first three pages of the text. Hence, the range 5...7 of the select option corresponds to the origin set by the page number option ("s 5 7/p 5" has the same effect).

option	*function*
"/a"	alternating pages – page number on right side for even pages
"/c" number	request a certain number of copies to be printed (1..9)
"/f" name	select font (other than the system default) for headers and page numbers
"/h"	print a header line (file name plus print date) on each page
"/l" number	shift print image on page (correction, in 1/10 mm)
"/p" number	start page numbering at a certain number (otherwise it starts at 0)
"/p" "f"	suppress page number and header on first page (as set by /p)
"/p" "n"	suppress page numbers
"/s" number [number]	select page(s) to print (numbers corresponds to the ones set by /p)

Table 8. Print Options.

Paragraph Attributes

All numerical values are to be specified in 1/10 mm units (e.g. the value 150 corresponds to 1.5 cm). For lead and line space attributes, font names may be used. In this case, the font height is used to compute an optimal value.

attribute	*value(s)*	*default*	*function*
"adjust"	block | center | left | right	left	formatting mode
"break"	before | normal	normal	if before: force page break before parc
"grid"	on | off	off	line grid
"lead"	name | number | "default"	0	leading paragraph space
"left"	number | "default"	0	left margin
"line"	name | number | "default"	34	minimal line height
"tabs"	("*" number | {number} "~")	~	tab spacing (every n, or enumerated)
"width"	number | "default"	1650	maximal line width

Table 9. Paragraph Attributes.

The line spacing model assumes a minimal line height for each paragraph. If a line exceeds its minimal line height, it is automatically adjusted to avoid clutter between it and the line above it. If the line grid is turned on for that paragraph (default), the line is adjusted to an integer multiple of the minimal line height.

Advise: In order to reach a typographically sound layout, in most cases a single line height should be selected for all paragraphs of the text. To get good results when combining a font for body text with larger ones for titles and section headings and smaller ones for captions, it is useful to set the line height to the body font and turn the line grid on. In most cases, it is preferable to set line height and paragraph leadings using the name of the predominant font ("model font").

Utility Module WriteTools

A collection of utility commands most of which operate equally well on standard texts and Write texts. (The application to Leda texts is not recommended – the standard Leda Attributes Viewer should be used in conjunction with Leda.Search and Leda.Replace.)

command	*explicit parameters*	*implicit parameters*
WriteTools.GetAttr		selected character

Get textual attributes of the selected character: ASCII code, font name, color, and vertical offset.

WriteTools.IncSize ("↑" | number) selection
Increment font sizes in selected range (negative increments may be used to decrement sizes). If a computed font is not available, the old font is retained.

WriteTools.ChangeSize ("↑" | {number "=>" number} "~") selection
Change font sizes absolutely within the selected range. If a computed font is not available, the old font is retained.

WriteTools.ChangeFamily ("↑" | {name "=>" name} "~") selection
Change font families within the selected range. If a computed font is not available, the old font is retained.

WriteTools.Change ("↑" | {name "=>" name} "~") selection
Change fonts within the selected range. If a font is not available, the old font is retained.

WriteTools.Words ("↑" | {name} "~")
Counts characters and words in the listed files.

WriteTools.Elements ("↑" | {name} "~")
Counts characters, words, and elements in the listed files.

WriteTools.Elements ("↑" | {name} "~")
Counts characters, words, and elements in the listed (Write) files.

WriteTools.Cleanup ("↑" | {name} "~")
Scans a Write text and removes alien and ill–sized elements. (Restricted in use to Write texts.)

Appendix B – Interfaces of the Write Core Modules

This appendix documents the interfaces of the Write core modules. For two complete and commented source listings of actual element implementations, cf. appendix C. For hints on how to implement Write elements in a way conforming to established conventions, see the second part of this report. The following table gives some impression on the code size of the Write system and some of its extensions. When comparing against the standard module triple *Texts*, *TextFrames*, and *Edit*, one has to take into account that *WriteTexts* heavily builds on module *Texts*, while *WriteFrames* and *Write* are fully independent. Also, the module *WriteParcs* could, in principle, be left out, leading to a version of Write with non-interactive parcs.

module	lines of source	object code [bytes]	functionality
WriteTexts	450	5 500	elements, load/store
WriteFrames	950	14 800	selfcontained frame
WriteParcs	400	6 700	interactive parcs
Write	750	10'900	commands & printing
Texts	n/a	6 900	attributed texts
TextFrames	n/a	10 000	selfcontained frame
Edit	n/a	4 800	commands & printing
LineElems	100	1 600	various lines
GraphicElems	200	2 300	encapsulated graphics
PopupElems	200	3 500	popup menu buttons
TableElems	500	8 500	table formatting

Table 10. Size of Write Core System, Standard Text System, and Extensions.

Module WriteTexts

WriteTexts extends module *Texts* by adding the capability of arbitrary elements floating in the text just as ordinary characters do. In particular, *WriteTexts* extends the types *Texts.Text* and *Texts.Buffer*. The extension suffers from the impossibility of overriding procedures defined in *Texts*. When using *WriteTexts*, the following *Texts* procedures must be considered with special care. Procedures *Texts.Delete*, *Texts.Insert*, *Texts.Append*, *Texts.Load*, and *Texts.Open must not* be used on a *WriteTexts* text. Procedures *Texts.Save*, *Texts.Recall*, and *Texts.Store* have reduced semantics when applied to a *WriteTexts* text, as they ignore floating elements. All other procedures, especially those defined on Readers, Writers, and Scanners can be used freely. This includes procedure *Texts.ChangeLooks* to modify character attributes of a text range.

WriteTexts is not coupled to any particular device and especially to no particular resolution. Hence, all measures contained in elements and parcs are in device independent units. The used unit is defined as follows:

1 mm = 36'000 units		1 point = 12'700 units	1 inch = 914'400 units
dots per inch	units per dot	examples	
72	12'700		
91	10'000	Ceres–1/2 monochrome monitors; Ceres *.Scn.Fnt files	
144	6'350		
200	4'572		
240	3'810		
300	3'048	Ceres laser printer; Ceres *.Pr3.Fnt and *.Lm3.Fnt files	
400	2'286		
600	1'524		
900	1'016		
1200	762		

Table 11. Pixel Resolution vs. Device Independent Units.

Since the Ceres–Oberon screen fonts (*.Scn.Fnt files) are tuned for 91 dpi resolution, the scale factor used for other monitors attached to the Ceres (color monitors, Ceres–3 monitor) should be set to 10'000, anyway. Likewise, the existing printer fonts (*.Pr3.Fnt and *.Lm3.Fnt files) are tuned for 300 dpi resolution.

When changing an element or a parc inserted into a Write text, calling procedures *WriteTexts.ChangedElem* or *WriteTexts.ChangedParc*, respectively, initiates an appropriate change notification updating all existing views.

Variable *WriteTexts.storeTime* can be used to distinguish among different store actions when receiving a store message. This is useful when storing elements sharing a common model, as is explained for active elements in section 2.5.

```
DEFINITION WriteTexts;
   IMPORT
      Display, Files, Fonts, Texts, Oberon;

   CONST
      replace = 0; insert = 1; delete = 2;
      mm = 36000;   (*units per mm*)
      ElemChar = 1CX;
      MaxTabs = 32;
      TextTag = 0F5X; ElemTag = 0F6X;

   TYPE
      Elem = POINTER TO ElemDesc;
      Handler = PROCEDURE(E: Elem; VAR msg: Display.FrameMsg);
      ElemDesc = RECORD
         DX, W, H: LONGINT;   (*in units; DX >= W*)
         handle: Handler;
         temp: BOOLEAN   (*suppress storage of element*)
      END;
```

```
    Parc = POINTER TO ParcDesc;
    ParcDesc = RECORD(ElemDesc)
        first, left, width, lead, lsp, dsr: LONGINT;    (*in units*)
        opts: SET;
        nofTabs: INTEGER;
        tab: ARRAY MaxTabs OF LONGINT    (*in units*)
    END;

    Buffer = POINTER TO BufferDesc;
    BufferDesc = RECORD(Texts.BufDesc) END;

    Text = POINTER TO TextDesc;
    TextDesc = RECORD(Texts.TextDesc)
        defParc: Parc
    END;

    LoadMsg = RECORD(Display.FrameMsg)
        r: Files.Rider    (*DX, W, H loaded automatically*)
    END;
    StoreMsg = RECORD(Display.FrameMsg)
        r: Files.Rider    (*DX, W, H stored automatically*)
    END;
    CopyMsg = RECORD(Display.FrameMsg)
        e: Elem (*receipient should allocate e iff e = NIL*)
    END;

    DrawMsg = RECORD(Display.FrameMsg)
        fnt: Fonts.Font;
        col: SHORTINT;
        unit: LONGINT;    (*units per device pixel*)
        frame: Display.Frame;    (*containing frame in device space*)
        X0, Y0: INTEGER    (*anchor in device space*)
    END;
    PrintMsg = RECORD(Display.FrameMsg)
        fnt: Fonts.Font;
        col: SHORTINT;
        unit: LONGINT;    (*units per device pixel*)
        pno: INTEGER;    (*page number*)
        X0, Y0: INTEGER    (*anchor in device space*)
    END;

    AllocPar = POINTER TO AllocParDesc;
    AllocParDesc = RECORD(Oberon.ParRec)
        e: Elem (*receipient should allocate element e and install handler*)
    END;

VAR
    storeTime: LONGINT;    (*timestamp of most recent Store*)

(* Elements *)
PROCEDURE OpenElem(E: Elem; handle: Handler; dx, w, h: LONGINT);
    (*forces dx >= w, e.temp = FALSE*)
```

```
PROCEDURE CopyElem(SE: Elem; VAR DE: Elem);
    (*returns copy of SE in DE using CopyMsg; forced copying of DX, W, H, handle, and temp*)
PROCEDURE ElemClass(E: Elem; VAR unknown: BOOLEAN; VAR mod: ARRAY OF CHAR);
    (*unknown => mod valid*)
PROCEDURE ElemBase(E: Elem): Text;
    (*returns NIL if E is not within a text*)
PROCEDURE ElemPos(E: Elem): LONGINT;
    (*returns -1 if E is not within a text*)
PROCEDURE ElemSucc(E: Elem): Elem;
    (*returns NIL if no successor or not within a text*)

(* Parcs – Paragraph Controls *)
PROCEDURE ParcBefore(T: Text; pos: LONGINT): Parc;
    (*returns T.defParc if none found*)
PROCEDURE ParcExtent(P: Parc): LONGINT;
    (*parc P extends over range [ElemPos(P), ParcExtent(P) )*)
PROCEDURE LoadParc(VAR r: Files.Rider; P: Parc);
    (*load parc attributes*)
PROCEDURE StoreParc(VAR r: Files.Rider; P: Parc);
    (*store parc attributes*)
PROCEDURE CopyParc(SP, DP: Parc);
    (*copy parc specific fields from SP to DP*)
PROCEDURE HandleParc(E: Elem; VAR msg: Display.FrameMsg);
    (*primitive parc handler, calls LoadParc, StoreParc, and CopyParc*)
PROCEDURE AllocParc;
    (*allocate plain parc for loading*)

(* Buffers *)
PROCEDURE OpenBuf(B: Buffer);
    (*prepare & clear B*)
PROCEDURE CopyBuf(SB: Buffer; VAR DB: Buffer);
    (*allocates DB if set to NIL; DB := copy(SB); SB unmodified*)
PROCEDURE SaveBuf(T: Texts.Text; beg, end: LONGINT; VAR B: Buffer);
    (*allocates B if set to NIL; B := copy(T[beg, end])*)
PROCEDURE AppendBuf(SB, DB: Buffer);
    (*DB := DB + copy(SB); SB := empty*)
PROCEDURE RecallBuf(B: Buffer);
    (*B is copy of last deleted stretch*)

(* Texts *)
PROCEDURE Delete(T: Text; beg, end: LONGINT);
PROCEDURE InsertElem(T: Text; pos: LONGINT; E: Elem);
PROCEDURE Insert(T: Text; pos: LONGINT; B: Texts.Buffer);
PROCEDURE AppendElem(T: Text; E: Elem);
PROCEDURE Append(T: Text; B: Texts.Buffer);
PROCEDURE ElemAt(T: Text; pos: LONGINT): Elem;
    (*returns NIL if not found*)
PROCEDURE FirstElem(T: Text): Elem;
    (*returns NIL if no elems in T*)
PROCEDURE Broadcast(T: Text; beg, end: LONGINT; VAR msg: Display.FrameMsg);
    (*broadcast to all elems in range [beg, end) of T*)
```

```
    PROCEDURE ChangedElem(E: Elem);
        (*notify about replacement of range [ElemPos(E), ElemPos(E)+1)*)
    PROCEDURE ChangedParc(P: Parc);
        (*notify about replacement of range [ElemPos(P), ParcExtent(P) )*)

    (* Text Files *)
    PROCEDURE Load(T: Text; f: Files.File; pos: LONGINT; defParc: Parc; VAR len: LONGINT);
        (*load Texts or WriteTexts block at (f, pos) into T; use defParc for Texts blocks;
        returns block length in len*)
    PROCEDURE Store(T: Text; f: Files.File; pos: LONGINT; VAR len: LONGINT);
        (*store T as WriteTexts block at (f, pos); returns block length in len*)
    PROCEDURE Open(T: Text; name: ARRAY OF CHAR; defParc: Parc);
        (*open T by trying to load (Files.Old(name), 0); can load plain-ascii files;
        creates empty text otherwise*)
END WriteTexts.
```

Module WriteFrames

WriteFrames extends module *TextFrames* by adding numerous formatting and editing capabilities. Care must be taken when using *WriteFrames*, as the procedures of module TextFrames *must not* be applied to a *WriteTexts* text. The most visible side effect is the non-functioning of command Edit.Locate which results in a trap (without doing any harm to the Write frame or its text).

WriteFrames contains the screen formatter. It defines a special prepare message (*WriteFrames.PrepareMsg*) sent to elements about to be displayed. This allows for adapting to remaining space on a line and the like. Hence, an element is allowed to change its bounding box upon receipt of a prepare message.

WriteFrames tries to delegate mouse clicks to elements hit by the cursor. This is done by sending a *WriteFrames.TrackMsg*. By tracking the mouse until all mouse buttons have been released again, the element can selectively consume mouse clicks. It is strongly recommended to restrict the set of consumed mouse clicks to middle button clicks and interclicks, i.e. the command click combinations. These are undefined for elements, anyway. Consuming left or right button clicks or interclicks causes interference with the caret and selection controlling clicks interpreted by Write frames.

It is possible to attach elements to some external model, i.e. to use elements as views showing some shared model. To update these views, a model change should cause a notification. For this purpose, a special *WriteFrames.NotifyMsg* is provided, which is broadcasted by a Write frame to all elements in the frame's visible range. An element receiving a notification message should call *WriteFrames.LocateElem* to find out whether it is truely visible (or clipped) and if so, what screen coordinates correspond to the element.

Opening a *WriteFrame* takes besides the handler to be used and the text to be displayed a bunch of additional parameters. For each of these a default variable is exported. The parameters define the border width defined around a displayed text in a Write frame (left, right, top, bot), the scroll bar width (barW, if barW > left : scroll bar is not displayed and not available), and a set of options. Currently, frame options *WriteFrames.scrollOpt* and *WriteFrames.fontOpt* are defined. If the scroll option is set, the frame scrolls automatically when user interactions (editing, moving the cursor, but not backward deleting) reach the frame bounds. Setting the font option causes the frame to take fonts from neighbouring characters when accepting keyboard input. (Both options are set by default.)

Although a *WriteFrames.Frame* is an extension of a *TextFrames.Frame* for backward compatibility reasons (e.g. the compiler accepts a WriteFrame as source), not all fields of a *TextFrames.Frame* record are supported by a *WriteFrames* frame. The fields *col*, *mark*, *car*, *sel*, *carloc*, *selbeg*, and *selend* are not used. The background color is always Display.black, caret and selection are reflected by the fields *hasCar*, *hasSel*, *carLoc*, *selBeg*, and *selEnd*. The field *text* is expected to refer to a WriteTexts text. To install a new text in an

open Write frame proceed as follows. Set the text field of the frame descriptor to the new text. Set the *trailer* field to NIL. Call *WriteFrames.Show(org)*, where *org* is the wished origin from where the new text is to be displayed. *WriteFrames.Show* automatically *decrements* org to begin at a valid line start position.

```
DEFINITION WriteFrames;
   IMPORT
      Display, Fonts, Oberon, Texts, TextFrames, WriteTexts;

   CONST
      scrollOpt = 1; fontOpt = 2;   (*frame options*)
      gridAdj = 0; leftAdj = 1; rightAdj = 2; pageBreak = 3;   (*parc options*)

   TYPE
      TextLine = POINTER TO TextLineDesc;
      TextLineDesc = RECORD
         next: TextLine;
         eot: BOOLEAN;   (*contains end of text*)
         w, h, dsr: INTEGER;   (*bounding box clipped to frame*)
         org, len, span: LONGINT   (*len w/o; span w/ trailing CR or white space, if any*)
      END;
      Location = RECORD
         org, pos: LONGINT;
         x, y, dx, dy: INTEGER
      END;

      Frame = POINTER TO FrameDesc;
      FrameDesc = RECORD(TextFrames.FrameDesc)
         (*text, org, lsp, left, right, top, bot, markH, time inherited from TextFrames.FrameDesc*)
         barW: INTEGER;   (*scroll bar width*)
         hasCar, hasSel: BOOLEAN;   (*caret/selection present*)
         opts: SET;
         carLoc, selBeg, selEnd: Location;   (*locations of caret, selection begin, and end*)
         trailer: TextLine   (*ring of line descriptors, points to trailer after last visible line*)
      END;

      UpdateMsg = TextFrames.UpdateMsg;
      PrepareMsg = RECORD(Display.FrameMsg)
         fnt: Fonts.Font;
         col: SHORTINT;
         unit: LONGINT;   (*units per device pixel*)
         indent: LONGINT;   (*width already consumed in line, in units*)
         printing: BOOLEAN;
         pno: INTEGER   (*page number, valid if printing*)
      END;
      TrackMsg = RECORD(Oberon.InputMsg)
         unit: LONGINT;   (*units per device pixel*)
         frame: Frame;
         X0, Y0: INTEGER   (*receiver origin in device space*)
      END;
```

```
    NotifyMsg = RECORD(Display.FrameMsg)
        unit: LONGINT;   (*units per device pixel*)
        frame: Frame   (*frame containing receiver*)
    END;

VAR
    defLeft, defRight, defTop, defBot, defBarW: INTEGER;
    defOpts: SET;

(* locators *)
PROCEDURE BegOfLine(T: WriteTexts.Text; VAR pos: LONGINT; adjust: BOOLEAN);
    (*set pos to the first character of the line containing pos;
    adjust takes line wrap-around into accound*)

PROCEDURE LocatePos(F: Frame; pos: LONGINT; VAR loc: Location);
    (*return location descriptor for position (F, pos)*).
PROCEDURE LocateLine(F: Frame; y: INTEGER; VAR loc: Location);
    (*return location descriptor for line at (F, y); y in absolute screen coordinates*)
PROCEDURE LocateChar(F: Frame; x, y: INTEGER; VAR loc: Location);
    (*return location descriptor for character at (F, x, y); x, y in absolute screen coordinates*)
PROCEDURE LocateWord(F: Frame; x, y: INTEGER; VAR loc: Location);
    (*return location descriptor for word at (F, x, y); x, y in absolute screen coordinates*)

PROCEDURE LocateElem(F: Frame; E: WriteTexts.Elem; VAR visible: BOOLEAN;
    VAR fnt: Fonts.Font; VAR col: SHORTINT; VAR X0, Y0: INTEGER);
    (*return whether element E is visible in F; if so, return its font and color (fnt, col),
    and base coordinates (X0, Y0)*)
PROCEDURE Pos(F: Frame; x, y: INTEGER): LONGINT;
    (*return position in text at (F, x, y); x, y in absolute screen coordinates*)

(* caret & selection *)
PROCEDURE RemoveSelection(F: Frame);
PROCEDURE SetSelection(F: Frame; beg, end: LONGINT);
    (*forces range to visible bounds*)
PROCEDURE RemoveCaret(F: Frame);
PROCEDURE SetCaret(F: Frame; pos: LONGINT);
    (*only done if within visible bounds*)
PROCEDURE Neutralize(F: Frame);
    (*remove selection and caret*)

(* display range *)
PROCEDURE GetVisibleRange(F: Frame; VAR beg, end: LONGINT);
    (*return visible range [beg, end) of F*)
PROCEDURE NotifyElems(F: Frame; VAR msg: NotifyMsg);
    (*send notify message msg to all elements in visible range of F*)
PROCEDURE Show(F: Frame; pos: LONGINT);
    (*removes global marks as needed and neutralizes F;
    redisplays whole text if F.trailer = NIL*)
PROCEDURE Resize(F: Frame; x, y, w, h: INTEGER);
    (*change frame size*)
```

```
(* contents update *)
PROCEDURE Update(F: Frame; VAR msg: UpdateMsg);
    (*removes global marks as needed*)

(* user interface *)
PROCEDURE TrackLine(F: Frame; VAR x, y: INTEGER; VAR org: LONGINT; VAR keysum: SET);
    (*track mouse to find line*)
PROCEDURE TrackWord(F: Frame; VAR x, y: INTEGER; VAR pos: LONGINT; VAR keysum: SET);
    (*track mouse to find word*)
PROCEDURE TrackCaret(F: Frame; VAR x, y: INTEGER; VAR keysum: SET);
    (*track mouse to place caret*)
PROCEDURE TrackSelection(F: Frame; VAR x, y: INTEGER; VAR keysum: SET);
    (*track mouse to place selection; supports extension to beginning of line and extension
    over subsequent viewers*)
PROCEDURE Call(F: Frame; pos: LONGINT; new: BOOLEAN);
    (*call the command at (F.text, pos) with appropriate arguments;
    force module reloading if new is set*)
PROCEDURE Write(F: Frame; ch: CHAR; fnt: Fonts.Font; col, voff: SHORTINT);
    (*write character ch to F.text at current caret position;
    if no other choice, use (fnt, col, voff) as attributes*)
PROCEDURE Edit(F: Frame; x, y: INTEGER; keysum: SET);
    (*track mouse and support editing in frame F; calls Track*, Call, and Write*)

(* general *)
PROCEDURE Copy(F: Frame; VAR F1: Frame);
    (*return a copy of F in F1*)
PROCEDURE Open(F: Frame; H: Display.Handler; T: WriteTexts.Text; pos: LONGINT;
        left, right, top, bot, barW: INTEGER; opts: SET);
    (*open frame F using handler H to display (T, pos) with border (left, right, top, bot),
    scroll bar of width barW, and options opts*)
PROCEDURE Handle(f: Display.Frame; VAR msg: Display.FrameMsg);
    (*dispatch message msg; calls NotifyElems, Show, Resize, Update, Write, Edit,
    SetCaret, RemoveCaret, SetSelection, Neutralize, and Copy*)

PROCEDURE NotifyDisplay(T: Texts.Text; op: INTEGER; beg, end: LONGINT);
    (*broadcasts an UpdateMsg for (T, op, beg, end) to all viewers*)
PROCEDURE Text(name: ARRAY OF CHAR; defParc: WriteTexts.Parc): WriteTexts.Text;
    (*opens a standard Write text to be displayed in a Write frame*)
PROCEDURE NewText(T: WriteTexts.Text; pos: LONGINT): Frame;
    (*opens a standard Write frame to be installed as content frame of a
    MenuViewers viewer*)
END WriteFrames.
```

Module WriteParcs

WriteParcs implements a proper extension of the parcs (paragraph controls) defined in module *WriteTexts*, adding interactive manipulation features for most parc attributes. Furthermore, the parc handler installed by *WriteParcs* supports querying and changing parc attributes using the message *WriteParcs.StateMsg*. This is used by the Write commands *Write.Set* and *Write.Get*. Hence, extended parcs defining new attributes can be defined that in turn extend the parameters taken by *Write.Set* and *Write.Get*. (The standard *StyleElems* take advantage of this by propagating changes to all parcs in a text that have the same name.)

```
DEFINITION WriteParcs;
   IMPORT
      Display, Texts, WriteTexts, WriteFrames;

   CONST
      set = 0; get = 1;

   TYPE
      StateMsg = RECORD(Display.FrameMsg)
         id: INTEGER;
         par: Texts.Scanner;
         log: Texts.Text;
         unit: LONGINT;   (*units per device pixel*)
         frame: WriteFrames.Frame
      END;

   VAR
      defParc: WriteTexts.Parc;

   PROCEDURE Prepare(P: WriteTexts.Parc; unit, indent: LONGINT; printing: BOOLEAN);
      (*set width in order to occupy whole line*)
   PROCEDURE Draw(P: WriteTexts.Parc; F: Display.Frame; unit: LONGINT;
         col: SHORTINT; x0, y0: INTEGER);
      (*draw parc*)
   PROCEDURE Edit(P: WriteTexts.Parc; F: WriteFrames.Frame; unit: LONGINT;
         x0, y0, x, y : INTEGER; keysum: SET);
      (*edit parc*)
   PROCEDURE SetAttr(P: WriteTexts.Parc; F: WriteFrames.Frame; unit: LONGINT;
         VAR S: Texts.Scanner; log: Texts.Text);
      (*scan attributes and values to be set using S and append error messages to log*)
   PROCEDURE GetAttr(P: WriteTexts.Parc; F: WriteFrames.Frame; unit: LONGINT;
         VAR S: Texts.Scanner; log: Texts.Text);
      (*scan attributes to be queried using S and append result and error messages to log*)
   PROCEDURE Handle(E: WriteTexts.Elem; VAR msg: Display.FrameMsg);
      (*standard parc handler, calls WriteTexts.LoadParc, WriteTexts.StoreParc,
      Prepare, Draw, Edit, SetAttr, and GetAttr*)
   PROCEDURE Alloc;
      (*allocate parc for loading*)
END WriteParcs.
```

Appendix C – Two Sample Element Implementations

This appendix lists two sample element implementations that can be taken as a tutorial. The first, *TestElems*, implements nothing useful but shows a functional and commented element handler at a glance. Also, it can be used to test out the effect of having unloadable elements (called aliens) in a text. Finally, *TestElems* demonstrate the use of messages broadcasted to all visible elements. The second, *ErrorElems*, implements error marking elements that can be used to mark all errors found by a compiler within the source text. This second example shows how a handler should be decomposed and its components be exported in order to allow for further extension of the implemented element. Note that *ErrorElems* are temporary elements and thus never stored. Therefore, the handling of file I/O is missing in the *ErrorElems* implementation.

TestElems

```
MODULE TestElems;
  IMPORT
    Oberon, Input, Display, Viewers, Files, Fonts, Printer, Texts, WriteTexts, WriteFrames, WriteParcs;

  CONST
    mm = WriteTexts.mm;   rightKey = 0; middleKey = 1; leftKey = 2;

  TYPE
    TestElem = POINTER TO TestElemDesc;
    TestElemDesc = RECORD(WriteTexts.ElemDesc)
      data: ARRAY 8 OF CHAR   (*demonstrate use of element specific data*)
    END;
    NotifyMsg = RECORD(WriteFrames.NotifyMsg) END;


  PROCEDURE WriteString(VAR r: Files.Rider; s: ARRAY OF CHAR);
    VAR i: INTEGER;
  BEGIN i := 0;
    WHILE s[i] # 0X DO INC(i) END;
    Files.WriteBytes(r, s, i + 1)
  END WriteString;

  PROCEDURE ReadString(VAR r: Files.Rider; VAR s: ARRAY OF CHAR);
    VAR i: INTEGER; ch: CHAR;
  BEGIN i := 0;
    REPEAT Files.Read(r, ch); s[i] := ch; INC(i) UNTIL (ch = 0X) OR (i = LEN(s));
    IF ch # 0X THEN s[0] := 0X END
  END ReadString;


  PROCEDURE* TestHandle(E: WriteTexts.Elem; VAR msg: Display.FrameMsg);
    VAR e: TestElem; P: WriteTexts.Parc; x, y, w, h: INTEGER; keys, keysum: SET; visible: BOOLEAN;
      fnt: Fonts.Font; col: SHORTINT; X0, Y0: INTEGER;
  BEGIN
    WITH E: TestElem DO
      IF msg IS WriteFrames.PrepareMsg THEN   (*element is to be drawn or printed*)
        WITH msg: WriteFrames.PrepareMsg DO   (*adopt measures from environment*)
          P := WriteTexts.ParcBefore(WriteTexts.ElemBase(E), WriteTexts.ElemPos(E));
          E.H := P.lsp   (*; E.W := P.width - msg.indent   adapts to remaining space in line*)
        END
      ELSIF msg IS WriteTexts.DrawMsg THEN   (*element is fully visible: draw it to the screen*)
        WITH msg: WriteTexts.DrawMsg DO
          Display.ReplConst(Display.white, msg.X0, msg.Y0,
            SHORT(E.W DIV msg.unit), SHORT(E.H DIV msg.unit), Display.replace)
        END
```

```
    ELSIF msg IS WriteTexts.PrintMsg THEN   (*element is fully visible: print it*)
      WITH msg: WriteTexts.PrintMsg DO
        Printer.Line(msg.X0, msg.Y0, SHORT(E.W DIV msg.unit), SHORT(E.H DIV msg.unit))
      END
    ELSIF msg IS NotifyMsg THEN   (*special viewer broadcast message*)
      WITH msg: NotifyMsg DO
        WriteFrames.LocateElem(msg.frame, E, visible, fnt, col, X0, Y0);
        IF visible THEN   (*if not clipped: update the single view in this frame*)
          Display.ReplConst(Display.white, X0 + 1, Y0 + 1,
            SHORT(E.W DIV msg.unit) - 2, SHORT(E.H DIV msg.unit) - 2, Display.invert)
        END
      END
    ELSIF msg IS WriteTexts.LoadMsg THEN   (*load element specific data*)
      WITH msg: WriteTexts.LoadMsg DO
        ReadString(msg.r, E.data)
      END
    ELSIF msg IS WriteTexts.StoreMsg THEN   (*store element specific data*)
      WITH msg: WriteTexts.StoreMsg DO
        WriteString(msg.r, "TestElems.Alloc");   (*write name of allocation procedure first*)
        WriteString(msg.r, E.data)
      END
    ELSIF msg IS WriteTexts.CopyMsg THEN   (*copy element*)
      WITH msg: WriteTexts.CopyMsg DO
        IF msg.e = NIL THEN NEW(e); msg.e := e ELSE e := msg.e(TestElem) END;
        e.data := E.data   (*copy state into new element*)
      END
    ELSIF msg IS WriteFrames.TrackMsg THEN   (*a mouse click hit the element*)
      WITH msg: WriteFrames.TrackMsg DO
        IF msg.keys = {middleKey} THEN keysum := msg.keys;
          w := SHORT(E.W DIV msg.unit); h := SHORT(E.H DIV msg.unit);
          Oberon.RemoveMarks(msg.X0, msg.Y0, w, h);
          Display.ReplConst(15, msg.X0 + 1, msg.Y0 + 1, w - 2, h - 2, Display.invert);
          REPEAT Input.Mouse(keys, msg.X, msg.Y); keysum := keysum + keys;
            Oberon.DrawCursor(Oberon.Mouse, Oberon.Arrow, msg.X, msg.Y)
          UNTIL keys = {};
          Display.ReplConst(15, msg.X0 + 1, msg.Y0 + 1, w - 2, h - 2, Display.invert);
          IF (keysum = {middleKey, leftKey}) & (E.W > 4 * mm) THEN
            DEC(E.W, 2 * mm); E.DX := E.W + 2 * mm; WriteTexts.ChangedElem(E)
          ELSIF msg.keys = {middleKey, rightKey} THEN
            INC(E.W, 2 * mm); E.DX := E.W + 2 * mm; WriteTexts.ChangedElem(E)
          END
        END
      END
    END
  END TestHandle;

  PROCEDURE* MiscHandle(E: WriteTexts.Elem; VAR msg: Display.FrameMsg);
    (*subclass handler of TestHandle*)
  BEGIN
    WITH E: TestElem DO
      IF msg IS WriteTexts.StoreMsg THEN
        WITH msg: WriteTexts.StoreMsg DO
          (*write name of a nonexistent allocation procedure -> cannot be loaded again*)
          WriteString(msg.r, "TestElems.Unknown"); WriteString(msg.r, E.data)
        END
      ELSE TestHandle(E, msg)   (*"super" call to inherited class*)
      END
    END
  END MiscHandle;

  PROCEDURE Alloc*;   (*allocation proc. for class TestElem; also installs handler*)
    VAR e: TestElem;
  BEGIN NEW(e); e.handle := TestHandle; Oberon.Par(WriteTexts.AllocPar).e := e
  END Alloc;
```

```
(** commands **)

PROCEDURE InsertNew*;    (** W H demonstrates behaviour of trivial floating element**)
   VAR S: Texts.Scanner; w: LONGINT;
      e: TestElem; T: WriteTexts.Text; copyover: Oberon.CopyOverMsg;
BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S); w := S.i; Texts.Scan(S);
   NEW(e); WriteTexts.OpenElem(e, TestHandle, (w + 2)*mm, w*mm, S.i*mm); e.data := "testing";
   T := WriteFrames.Text("", WriteParcs.defParc); WriteTexts.AppendElem(T, e);
   copyover.text := T; copyover.beg := 0; copyover.end := T.len;
   Oberon.FocusViewer.handle(Oberon.FocusViewer, copyover)
END InsertNew;

PROCEDURE InsertMisc*;
   (** W H demonstrates handling of elements which cannot be loaded on opening**)
   VAR S: Texts.Scanner; w: LONGINT;
      e: TestElem; T: WriteTexts.Text; copyover: Oberon.CopyOverMsg;
BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S); w := S.i; Texts.Scan(S);
   NEW(e); WriteTexts.OpenElem(e, MiscHandle, (w + 2)*mm, w*mm, S.i*mm); e.data := "testing";
   T := WriteFrames.Text("", WriteParcs.defParc); WriteTexts.AppendElem(T, e);
   copyover.text := T; copyover.beg := 0; copyover.end := T.len;
   Oberon.FocusViewer.handle(Oberon.FocusViewer, copyover)
END InsertMisc;

PROCEDURE Broadcast*;    (**demonstrate effect of special viewer broadcast message**)
   VAR msg: NotifyMsg;
BEGIN Viewers.Broadcast(msg)
END Broadcast;

END TestElems.
```

ErrorElems

```
DEFINITION ErrorElems;
   IMPORT
      Display, Fonts,
      WriteTexts;

   TYPE
      Elem = POINTER TO ElemDesc;
      ElemDesc = RECORD(WriteTexts.ElemDesc)
         err: INTEGER;
         msg: ARRAY 128 OF CHAR
      END;

      DeleteMsg = RECORD(Display.FrameMsg) END;
      LocateMsg = RECORD(Display.FrameMsg)
         pos: LONGINT
      END;

   VAR
      font: Fonts.Font;

   PROCEDURE ShowErrMsg(E: Elem; F: Display.Frame; col: SHORTINT; x0, y0, dw: INTEGER);
   PROCEDURE Expand(E: Elem; unit: LONGINT);
   PROCEDURE Reduce(E: Elem);
   PROCEDURE Delete(E: Elem);
   PROCEDURE Handle(E: WriteTexts.Elem; VAR msg: Display.FrameMsg);
   PROCEDURE InsertAt(T: WriteTexts.Text; pos: LONGINT; err: INTEGER);

   (* commands *)
   PROCEDURE Unmark;
   PROCEDURE Mark;
```

```
  PROCEDURE LocateNext;
END ErrorElems.
```

```
MODULE ErrorElems;
  IMPORT
    Display, Input, Files, Fonts, Printer, Oberon, Texts, Viewers, MenuViewers,
    WriteTexts, WriteFrames, WriteParcs;

  CONST
    ErrFile = "Oberon2Errors.Text"; ErrFont = "Syntax8.Scn.Fnt";
    mm = WriteTexts.mm;    middleKey = 1; leftKey = 2;    CR = 0DX;

  TYPE
    Elem* = POINTER TO ElemDesc;
    ElemDesc* = RECORD(WriteTexts.ElemDesc)
      err*: INTEGER;
      msg*: ARRAY 128 OF CHAR
    END;

    DeleteMsg* = RECORD(Display.FrameMsg) END;
    LocateMsg* = RECORD(Display.FrameMsg)
      pos*: LONGINT
    END;


  VAR
    font*: Fonts.Font;
    W: Texts.Writer;
    lastTime: LONGINT;


  PROCEDURE MarkedFrame(): WriteFrames.Frame;
    VAR V: Viewers.Viewer;
  BEGIN V := Oberon.MarkedViewer();
    IF (V IS MenuViewers.Viewer) & (V.dsc.next IS WriteFrames.Frame) THEN
      RETURN V.dsc.next(WriteFrames.Frame)
    ELSE RETURN NIL
    END
  END MarkedFrame;

  PROCEDURE Show(F: WriteFrames.Frame; pos: LONGINT);
    VAR beg, end, delta: LONGINT;
  BEGIN delta := 200;
    LOOP WriteFrames.GetVisibleRange(F, beg, end);
      IF (beg <= pos) & (pos < end) OR (beg = end) THEN EXIT END;
      WriteFrames.Show(F, pos - delta); DEC(delta, 20)
    END
  END Show;


  PROCEDURE Width(E: Elem): INTEGER;
    VAR fnt: Fonts.Font; pat: Display.Pattern; i, px, dx, x, y, w, h: INTEGER;
  BEGIN fnt := Fonts.This(ErrFont); i := 0; px := 0;
    WHILE E.msg[i] # 0X DO
      Display.GetChar(fnt.raster, E.msg[i], dx, x, y, w, h, pat); INC(px, dx); INC(i)
    END;
    RETURN px + 6
  END Width;

  PROCEDURE ShowErrMsg*(E: Elem; F: Display.Frame; col: SHORTINT; x0, y0, dw: INTEGER);
    VAR fnt: Fonts.Font; pat: Display.Pattern; i, px, rm, dx, x, y, w, h: INTEGER; ch: CHAR;
  BEGIN fnt := Fonts.This(ErrFont); i := 0; px := x0 + 3; rm := x0 + dw - 3; INC(y0, 2);
    LOOP ch := E.msg[i]; INC(i);
      IF ch = 0X THEN EXIT END;
```

```
      Display.GetChar(fnt.raster, ch, dx, x, y, w, h, pat);
      IF px + dx > rm THEN EXIT END;
      Display.CopyPattern(col, pat, px + x, y0 + y, Display.invert); INC(px, dx)
    END
  END ShowErrMsg;

  PROCEDURE Expand*(E: Elem; unit: LONGINT);
    VAR S: Texts.Scanner; T: Texts.Text; n: INTEGER; ch: CHAR;
  BEGIN NEW(T); Texts.Open(T, ErrFile); Texts.OpenScanner(S, T, 0);
    REPEAT S.line := 0;
      REPEAT Texts.Scan(S) UNTIL S.eot OR (S.line # 0)
    UNTIL S.eot OR (S.class = Texts.Int) & (S.i = E.err);
    IF ~S.eot THEN Texts.Read(S, ch); n := 0;
      WHILE ~S.eot & (ch # CR) & (n + 1 < LEN(E.msg)) DO E.msg[n] := ch; INC(n); Texts.Read(S, ch) END;
      E.msg[n] := 0X; E.W := Width(E) * unit; E.DX := E.W; WriteTexts.ChangedElem(E)
    END
  END Expand;

  PROCEDURE Reduce*(E: Elem);
  BEGIN E.W := 3 * mm; E.DX := E.W; E.msg[0] := 0X; WriteTexts.ChangedElem(E)
  END Reduce;

  PROCEDURE Delete*(E: Elem);
    VAR T: WriteTexts.Text; pos: LONGINT;
  BEGIN T := WriteTexts.ElemBase(E);
    IF T # NIL THEN pos := WriteTexts.ElemPos(E); WriteTexts.Delete(T, pos, pos + 1) END
  END Delete;


  PROCEDURE Handle*(E: WriteTexts.Elem; VAR msg: Display.FrameMsg);
    VAR e: Elem; pos: LONGINT; w, h: INTEGER; keys, keysum: SET;
  BEGIN
    WITH E: Elem DO w := SHORT(E.W DIV msg.unit); h := SHORT(E.H DIV msg.unit);
      IF msg IS WriteTexts.DrawMsg THEN
        WITH msg: WriteTexts.DrawMsg DO
          Display.ReplConst(15, msg.X0 + 1, msg.Y0 + 2, w - 2, h, Display.replace);
          IF E.msg[0] # 0X THEN ShowErrMsg(E, msg.frame, msg.col, msg.X0, msg.Y0 + 2, w) END
        END
      ELSIF msg IS WriteTexts.PrintMsg THEN
        WITH msg: WriteTexts.PrintMsg DO
          Printer.Line(msg.X0 + 1, msg.Y0 + 2, w - 2, h)
        END
      ELSIF msg IS WriteTexts.CopyMsg THEN
        WITH msg: WriteTexts.CopyMsg DO
          IF msg.e = NIL THEN NEW(e); msg.e := e ELSE e := msg.e(Elem) END;
          e.err := E.err; e.msg := E.msg
        END
      ELSIF msg IS WriteFrames.TrackMsg THEN
        WITH msg: WriteFrames.TrackMsg DO
          IF msg.keys = {middleKey} THEN
            Oberon.RemoveMarks(msg.X0, msg.Y0, w, h);
            Display.ReplConst(15, msg.X0 + 2, msg.Y0 + 3, w - 4, h - 2, Display.invert);
            keysum := msg.keys;
            REPEAT Input.Mouse(keys, msg.X, msg.Y); keysum := keysum + keys;
              Oberon.DrawCursor(Oberon.Mouse, Oberon.Arrow, msg.X, msg.Y);
            UNTIL keys = {};
            Display.ReplConst(15, msg.X0 + 2, msg.Y0 + 3, w - 4, h - 2, Display.invert);
            IF keysum = {middleKey} THEN
              IF E.msg[0] = 0X THEN Expand(E, msg.unit) ELSE Reduce(E) END
            ELSIF keysum = {middleKey, leftKey} THEN Delete(E)
            END
          END
        END
      ELSIF msg IS DeleteMsg THEN Delete(E)
      ELSIF msg IS LocateMsg THEN
```

```
      WITH msg: LocateMsg DO pos := WriteTexts.ElemPos(E);
        IF pos < msg.pos THEN msg.pos := pos END
      END
    END
  END
END Handle;


PROCEDURE InsertAt*(T: WriteTexts.Text; pos: LONGINT; err: INTEGER);
  VAR e: Elem;
BEGIN NEW(e); WriteTexts.OpenElem(e, Handle, 3 * mm, 3 * mm, 3 * mm);
  e.temp := TRUE; e.err := err; e.msg[0] := 0X; WriteTexts.InsertElem(T, pos, e)
END InsertAt;


(** commands **)

PROCEDURE Unmark*;
  VAR F: WriteFrames.Frame; msg: DeleteMsg;
BEGIN F := MarkedFrame();
  IF F # NIL THEN WriteTexts.Broadcast(F.text(WriteTexts.Text), 0, F.text.len, msg) END
END Unmark;

PROCEDURE Mark*;
  VAR F: WriteFrames.Frame; S: Texts.Scanner; T: WriteTexts.Text;
    text: Texts.Text; beg, end, time, pos, delta: LONGINT; err: INTEGER;
BEGIN Unmark; F := MarkedFrame(); Oberon.GetSelection(text, beg, end, time); delta := 0;
  IF (F # NIL) & (time >= lastTime) THEN lastTime := time; T := F.text(WriteTexts.Text);
    Texts.OpenScanner(S, text, beg);
    LOOP S.line := 0;
      REPEAT Texts.Scan(S) UNTIL S.eot OR (S.line # 0) OR (S.class = Texts.Int);
      IF S.eot OR (S.line # 0) THEN EXIT END;
      pos := S.i;
      REPEAT Texts.Scan(S) UNTIL S.eot OR (S.line # 0) OR (S.class = Texts.Int);
      IF S.eot OR (S.line # 0) THEN EXIT END;
      err := SHORT(S.i); InsertAt(T, pos + delta, err); INC(delta);
      REPEAT Texts.Scan(S) UNTIL S.eot OR (S.line # 0)
    END
  END
END Mark;

PROCEDURE LocateNext*;
  VAR F: WriteFrames.Frame; msg: LocateMsg; beg: LONGINT;
BEGIN F := MarkedFrame();
  IF F # NIL THEN msg.pos := MAX(LONGINT);
    IF F.hasCar THEN beg := F.carLoc.pos ELSE beg := 0 END;
    WriteTexts.Broadcast(F.text(WriteTexts.Text), beg, F.text.len, msg);
    IF msg.pos < MAX(LONGINT) THEN Show(F, msg.pos); WriteFrames.SetCaret(F, msg.pos + 1) END
  END
END LocateNext;

BEGIN font := Fonts.This(ErrFont); Texts.OpenWriter(W); lastTime := -1
END ErrorElems.
```

Gelbe Berichte des Departements Informatik

133	J. Templ	SPARC-Oberon User's Guide
134	G. Wong	An Approach on How to Combine Object Recognition Methods
135	C. Wieland	Two Explanation Facilities for the Deductive Database Management System DeDEx
136	H.-J. Schek, M.H Scholl G. Weikum	From the KERNEL to the COSMOS: The Database Research Group at ETH Zürich
137	G. Weikum, C. Hasse, A. Mönkeberg, P. Zabback	The COMFORT Project: A Comfortable Way to Better Performance
138	J. Gutknecht	The Oberon Guide: System Release 1.2
139	P.E. Saylor, D.E. Smolarski	Implementation of an Adaptive Algorithm for Richardson's Method
140	N. Wirth	Die Programmiersprache Oberon
141	M. Franz	The Implementation of MacOberon
142	M. Franz	MacOberon Reference Manual
143	N. Wirth	From Modula to Oberon The Programming Language Oberon (revised edition)
144	J.L. Marais	The GADGETS User Interface Management System
145	J. Mössenböck	She: A Simple Hypertext Editor for Programs
146	H.E. Meier	Schriftgestaltung mit Hilfe des Computers Typographische Grundregeln
147	G. Weikum, P. Zabback, P. Scheuermann	Dynamic File Allocation in Disk Arrays
148	D. Degiorgi	A New Linear Algorithm to Detect a Line Graph and Output its Root Graph
149	A. Moenkeberg, G. Weikum	Conflict-Driven Load Control for the Avoidance of Data-Contention Trashing
150	M.H. Scholl, C. Laasch, M. Tresch	Updatable Views in Object-Oriented Databases