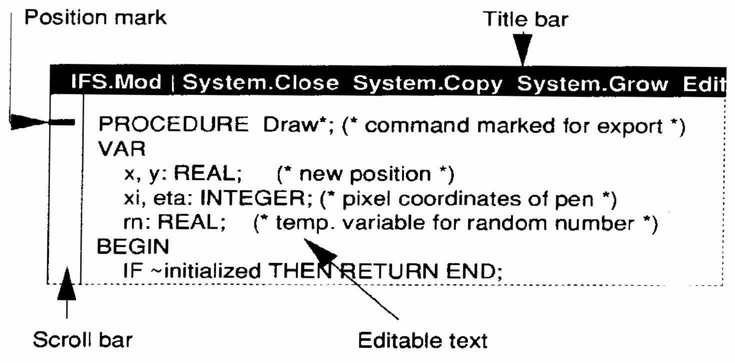
Programming in Oberon

7.4.2 The role of texts

One of the first surprises of the novice gaining acquaintance with a computer is the discovery that text displayed on the monitor is *modal*. Some text is merely meant to be looked at. It is a volatile entity on the screen, written by the system. Other text, written in a special place, will be interpreted as commands, instructing the system what to do. The special place is known as the *command line*. Still a third kind – close to what a naive user would expect as normal – is text ready to be edited, changed, stored or printed.

In Oberon, there is only one kind of text: it is editable, storable, printable, and can be interpreted as commands. Texts exist in windows called *text viewers*. In every text viewer, a simple editor is available. *The text modes are abolished.*

The layout of a text viewer is depicted in Figure 7.3. The perimeter of the viewer is marked with a thin line. On top is a *title bar*, rendered in reverse video. A scroll bar with a position mark is placed at left. The title bar contains the viewer name separated by a vertical bar " | " from the commands *System.Close, System.Copy, System.Grow, Edit.Locate* and *Edit.Store*. These commands can be executed with the mouse and perform operations on the viewer or on the text contained in the viewer.



```
PROCEDURE Draw*; (* command marked for export *)
VAR
   x, y: REAL;    (* new position *)
   xi, eta: INTEGER; (* pixel coordinates of pen *)
   rn: REAL;   (* temp. variable for random number *)
BEGIN
   IF ~initialized THEN RETURN END;
```

Figure 7.3 Layout and elements of an Oberon text viewer.

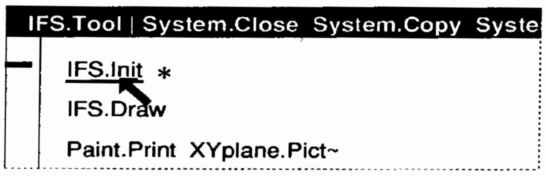
Command activation

Commands are identified through a qualified identifier of the form *Mod.Proc*, where *Proc* is the name of the command and *Mod* denotes the module in which *Proc* is declared. Oberon provides the facility to execute any command by simply pointing at its name – typed anywhere in a text – and pressing the execute key of the mouse.

Tool texts

The fact that commands are activated out of text viewers allows the user to create a highly efficient working environment. One simply types the group of commands that comprise the current work into a text viewer and stores the text on disk. Such a text is appropriately termed a *tool*. Figure 7.4 portrays such a tool, written to execute the commands from our sample module *IFS*. Three commands are prepared: *IFS.Init,* *IFS.Draw* and *Paint.Print*, the last will print the file *XYplane.Pict* that can be created from a *XYplane* viewer. The mouse pointer (arrow) is over *IFS.Init*, which appears underlined. That means the user is pressing the execute key. Upon release of that key, *IFS.Init* will execute.

Note that tool texts in text viewers are *like menus* of conventional user interface designs – except that they offer a lot more flexibility.



IFS.Tool | System.Close System.Copy Syste

IFS.Init *

IFS.Draw

Paint.Print XYplane.Pict~

Figure 7.4 A tool viewer; the command *IFS.Init* is about to be executed.

Text output

If commands produce output, a new text viewer is opened and the output text displayed within its perimeter. Again, the text can be modified, printed and stored. Module *Out* can be used to create such text output.

7.4.3 Modules *In* and *Out*

The abstractions of modules *In* and *Out* are general enough to be implementable on any computer. Such implementations differ in the way the input and output streams are defined. The input stream may be the keyboard or a file – the output stream a display device or again a file. In the Oberon system *both streams are texts that are displayed in viewers.*

The output stream

Invocation of *Out.Open* opens an empty text viewer with title *Out.Text*. The text displayed in that viewer represents the output stream. Each time a write procedure is called, the textual addition becomes visible in the viewer. The important thing to remember is that, even though viewer *Out.Text* is home of the output stream, it is also a normal text viewer. That means that the displayed text can be edited, stored or printed at any time the system admits input from mouse and keyboard. Such modifications do not interfere with the fact that further calls of write procedures add output at the end of the text.

The input stream

The input stream is also embedded in a text displayed in a text viewer. There are three possibilities, depending on the character that follows

22.3. Texts, Readers and Writers

Texts, basically, are sequences of characters. In the Oberon System, however, texts have some additional properties. As they can be displayed and printed, they must carry properties which determine their style and appearance. In particular, a *font* is specified. This is a style, the patterns by which characters are represented. Subsequences of characters in a text may have their individual fonts. Furthermore, Oberon texts specify color and vertical offset (allowing for negative offset for subscripting, and positive offset for superscripting). Texts are typically subjected to complicated editing operations, which require a flexible, internal data representation. Therefore Texts in Oberon differ substantially from Files in many respect. However, in their essence they are also sequences, and the basic operations strongly resemble those of files.

Our special interest in texts is justified by the fact, that humans communicate with computers mostly via texts, be they typed on a keyboard or displayed on a screen. As a consequence, reading and writing of texts usually includes a change of data representation. For example, integers will have to be changed into sequences of decimal digits on output, and the reading of integers requires the reading of a sequence of digits and the computation of the represented value. These conversions are included in the read/write procedures of module *Texts*, which we have used in many examples in preceding chapters. The following is its definition, which evidently resembles that of *Files*. In place of the single type *Rider* we find the two types *Reader* and *Writer*.

```
DEFINITION Texts;
IMPORT Files, Fonts;
 CONST replace = 0; insert = 1; delete = 2;
  Inval = 0; Name = 1; String = 2; Int = 3; Real = 4; Char = 6;
 TYPE Text = POINTER TO RECORD
   len: INTEGER
  END ;
  Buffer;
  Reader = RECORD
   eot: BOOLEAN;
   fnt: Fonts.Font;
   col, voff: INTEGER
  END ;
  Scanner = RECORD (Reader)
   nextCh: CHAR;
   line, class, i: INTEGER;
   x: REAL;
   c: CHAR;
   len: INTEGER;
   s: ARRAY 32 OF CHAR;
  END ;
  Writer = RECORD
   buf: Buffer;
   fnt: Fonts.Font;
   col, voff: INTEGER;
  END ;
 PROCEDURE Open(t: Text; name: ARRAY OF CHAR);
 PROCEDURE Delete(t: Text; beg, end: LONGINT);
 PROCEDURE Insert(t: Text; pos:INTEGER; b: Buffer);
 PROCEDURE Append(t: Text; b: Buffer);
 PROCEDURE ChangeLooks
          (T: Text; beg, end: INTEGER; sel: SET; fnt: Fonts.Font; col, voff: INTEGER);
 PROCEDURE OpenReader (VAR r: Reader; t: Text; pos: INTEGER);
 PROCEDURE Read (VAR r: Reader; VAR ch: CHAR);
 PROCEDURE Pos (VAR r: Reader): INTEGER;
 PROCEDURE OpenScanner (VAR S: Scanner; t: Text; pos: INTEGER);
 PROCEDURE Scan (VAR S: Scanner);
 PROCEDURE OpenWriter (VAR w: Writer);
 PROCEDURE SetFont (VAR w: Writer; fnt: Fonts.Font);
 PROCEDURE SetColor (VAR w: Writer; col: INTEGER);
 PROCEDURE SetOffset (VAR w: Writer; voff: INTEGER);
 PROCEDURE Write (VAR w: Writer; ch: CHAR);
 PROCEDURE WriteLn (VAR w: Writer);
 PROCEDURE WriteString (VAR w: Writer; s: ARRAY OF CHAR);
 PROCEDURE WriteInt (VAR w: Writer; x, n: INTEGER);
 PROCEDURE WriteHex (VAR w: Writer; x: INTEGER);
 PROCEDURE WriteReal (VAR w: Writer; x: REAL; n: INTEGER);
 PROCEDURE WriteRealFix (VAR w: Writer; x: REAL; n, k: INTEGER);
 PROCEDURE Load (VAR r: Files.Rider; t: Text);
 PROCEDURE Store (VAR r: Files.Rider; t: Text)
END Texts.
```

Procedure *OpenWriter* uses as defaults a standard font, black color, and zero offset.

Simple, sequential reading and writing of a text now follow the patterns of reading and writing a file. Let us assume the following declarations of variables:

```
T: Texts.Text;
R: Texts.Reader;
W: Texts.Writer;

NEW(T); Texts.Open(T, “MyName”); Texts.OpenReader(R, T, 0); Texts.Read(R, ch);
WHILE ~R.eot DO process(ch); Texts.Read(R, ch) END

NEW(T); Texts.Open(T, “MyName”); Texts.OpenWriter(W); generate(ch)
WHILE more DO Texts.Write(W, ch); generate(ch) END ;
Texts.Append(T, W.buf)
```

Note that the Oberon paradigm is to write a text, or a piece of text, first into a buffer, and thereafter insert or append it to a text. This is done for reasons of efficiency, because the possibly needed rendering of the text, for example on a display, can be done once upon insertion of the buffered piece of text rather than after generating each character.

Very often one wants to read a text consisting of a sequence of items which are not all of the same type, such as numbers, strings, names, etc. When using procedures reading items of a fixed type, the programmer must know the exact sequence in which the various items will appear in the text. Typically one does not know, and even if a specific order is specified, mistakes may occur. So one should like to have available a reading mechanism that reads items one at a time, but lets the program determine what type of item was read, and take further steps accordingly. This facility is provided in the Oberon text module by the machanism called *scanning*. In place of a *Reader* we use a *Scanner*. Procedure *Scan(S)* then reads the next item. Its kind can be determined by inspecting the field *S.class*, and its value accordingly is given by *S.i* in the case an integer was read, *S.x* if a real number was read, *S.s* if a name or a string was read. This scheme has proven to be most useful because of its flexibility. It defines the following syntax for texts to be scanned. Blanks and line ends serve to separate consecutive items.

```
item    =  name | integer | real | longreal | string | SpecialChar.
name    =  letter {letter | digit}.
integer =  [sign] digit {digit} | digit {digit | hexdig} "H“.
hexdig  =  "A“ | "B“ | "C“ | "D“ | "E“ | "F“.
sign    =  ["-“].
real    =  [sign] digit {digit} „.“ digit {digit} [("E“|"D") [sign] digit {digit}].
```

A string is any sequence of any characters (except quotes) enclosed in quotes. Special characters are are all characters except letters, digits, and the quote mark.

22.4. Standard Input and Output

In order to simplify the description of input and output of texts in simple cases, the Oberon system introduces some conventions and global variables. We call this source of input and sink of output *standard* input and output.

Standard output sink is the text *Log* defined as global variable in module *Oberon*. Given a global writer W, the text, previously written by Write procedures into the writer’s buffer (W.buf) is sent to the displayed *Log* text by the statement

```
Texts.Append(Oberon.Log, W.buf)
```

The standard input is often assumed to be the text following the command which invoked execution of a given procedure. This input text is identified by the variable *Par* in module *Oberon*. It is considered as parameter of the invoked command, and it is typically read by the scanning mechanism described above. The necessary statements are:

```
Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S)
```

The first item thus read may be the first item of the desired text, or it may be the name of the text file to be scanned, or something else, according to the specification of the individual command. The following conventions have established themselves for the designation of input texts. Assume that the command (procedure) name is followed by

- an identifier (possibly qualified). Then this is the name of the input text (input file),
- an asterisk (*). Then the marked viewer (window) contains the input text,
- an @ symbol. Then the most recent text selection is the beginning of the input text,
- an arrow (^). Then the most recent selection denotes the file name of the input text.

These conventions are expressed by the following function procedure yielding the designated input text:

```
PROCEDURE This*(): Texts.Text;
  VAR beg, end, time: INTEGER;
    S: Texts.Scanner; T: Texts.Text; v: Viewers.Viewer;
BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
IF S.class = Texts.Char THEN
  IF S.c = "*" THEN  (*input in marked viewer*)
    v := Oberon.MarkedViewer();
    IF (v.dsc # NIL) & (v.dsc.next IS TextFrames.Frame) THEN
      T := v.dsc.next(TextFrames.Frame).text; beg := 0
    END
  ELSIF S.c = "@" THEN  (*input starts at selection*)
    Oberon.GetSelection(T, beg, end, time);
    IF time < 0 THEN T := NIL END
  ELSIF S.c = "↑" THEN  (*selection is the file name*)
    Oberon.GetSelection(T, beg, end, time);
    IF time >= 0 THEN (*there is a selection*)
      Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
      IF S.class = Texts.Name THEN  (*input is named file*)
        NEW(T); Texts.Open(T, S.s); pos := 0
      END
    END
  END
ELSIF S.class = Texts.Name THEN  (*input is named file*)
  NEW(T); Texts.Open(T, S.s); pos := 0
END ;
RETURN T
END This;
```

Let us assume that this function procedure is defined in module *Oberon*, together with the global variable *pos*, indicating the position in the text where input is to start. The following program for copying a text may serve as a pattern for selecting the input according to the conventions postulated above.

```
MODULE ProgramPattern;
  IMPORT Texts, Oberon;
  VAR W: Texts.Writer;  (*global writer*)

  PROCEDURE copy(VAR R: Texts.Reader);
    VAR ch: CHAR;
  BEGIN Texts.Read(R, ch);
    WHILE ~R.eot DO Texts.Write(W, ch); Texts.Read(R, ch) END
  END copy

  PROCEDURE SomeCommand*;
    VAR R: Texts.Reader;  (*local reader*)
  BEGIN Texts.OpenReader(R, Oberon.This.text, Oberon.Par.pos);
    copy(R); Texts.Append(Oberon.Log, W.buf)
  END SomeCommand;

BEGIN Texts.OpenWriter(W)
END ProgramPattern.
```

Module *Oberon* can be considered as the core of the Oberon system housing a small number of global resources. These include the previously encountered output *Log*, and the record *Par*, specifying the viewer, frame, and text in which the activated command lies, and hence providing access to its input parameters. Here we show only an excerpt of the definition of module Oberon:

```
TYPE ParList = POINTER TO RECORD
          vwr: Viewers.Viewer;
          frame: Display.Frame;
          text: Texts.Text;
          pos: INTEGER;
  END ;

VAR Log: Texts.Text;
  Par: ParList;

PROCEDURE Time (): LONGINT;

PROCEDURE AllocateUserViewer (DX: INTEGER; VAR X, Y: INTEGER);
PROCEDURE AllocateSystemViewer (DX: INTEGER; VAR X, Y: INTEGER);
  (*provide coordinates X and Y for a new viewer to be allocated*)

PROCEDURE GetSelection (VAR text: Texts.Text; VAR beg, end, time: LONGINT);
```

To conclude this introduction to Oberon conventions about input and output, we show how a new viewer (window) is opened, and how the output text is directed into this text viewer. Optimal positioning of the new viewer is achieved by procedure *Oberon.AllocateViewer*, with the specified set of standard, frequently needed commands in the title bar.

```
PROCEDURE OpenViewer(T: Texts.Text);
  VAR V: MenuViewers.Viewer; X, Y: INTEGER;
BEGIN Oberon.AllocateUserViewer(Oberon.Mouse.X, X, Y);
  V := MenuViewers.New(TextFrames.NewMenu(
      "Name", "System.Close  System.Copy  System.Grow  Edit.Search  Edit.Store"),
      TextFrames.NewText(T, 0), TextFrames.menuH, X, Y)
END OpenViewer;
```

The Oberon System

User Guide and Programmer's Manual

14 Module Texts

Module Texts provides the abstract data type *Text* which is a model of a *sequence of characters* with their associated properties: *font*, *color* and *vertical offset*. The text's data structure is hidden. It is accessed or changed through a set of procedures which comprise the *data manager*.

Module Texts exports:

- The abstract data type *Text*. Instances of *Text* are active Oberon objects with an installed procedure, called *notifier*, which is invoked whenever the text changes.
- The abstract data type *Buffer* to assemble and hold temporary texts.
- The abstract data type *Reader* to read the characters of a text sequentially.
- The abstract data type *Scanner* to read symbols (integers, reals, strings etc.) from texts and translate them into an internal representation.
- The abstract data type *Writer* to create texts in buffers from variables of several basic types (CHAR, INTEGER, REAL etc.)
- Procedures for changing, opening and closing texts including reading and writing to disk.

Readers, scanners and writers operate sequentially on the sequence of characters. Like riders on files, they have an implicit property, the *position*, which is incremented at each operation. In fact, they are extensions of the type *Files.Rider*.

As a preliminary, module Fonts is described which exports the abstract data type *Font*, which provides the raster data for screen fonts.

```
DEFINITION Fonts;

IMPORT Display;

TYPE
  Font = POINTER TO FontDesc;
  FontDesc = RECORD
    name: Name;
    height, minX, maxX, minY, maxY: INTEGER;
    raster: Display.Font
  END;
  Name = ARRAY 32 OF CHAR;

VAR Default: Font;

PROCEDURE This (name: ARRAY OF CHAR): Font;

END Fonts.
```

```
DEFINITION Texts;

IMPORT Display, Files, Fonts;

CONST
  Inval = 0; Name = 1; String = 2; Int = 3; Real = 4; LongReal = 5;
  Char = 6;
  replace = 0; insert = 1; delete = 2;

TYPE
  Buffer = POINTER TO BufDesc;
  BufDesc = RECORD
    len: LONGINT
  END;

  Notifier = PROCEDURE (T: Text; op: INTEGER; beg, end: LONGINT);

  Reader = RECORD (Files.Rider)
    eot: BOOLEAN;
    fnt: Fonts.Font;
    col, voff: SHORTINT
  END;

  Scanner = RECORD (Reader)
    nextCh: CHAR;
    line, class: INTEGER;
    i: LONGINT;
    x: REAL;
    y: LONGREAL;
    c: CHAR;
    len: SHORTINT;
    s: ARRAY 32 OF CHAR
  END;
```

```
    Text = POINTER TO TextDesc;
    TextDesc = RECORD
      len: LONGINT;
      notify: Notifier
    END;

    Writer = RECORD (Files.Rider)
      buf: Buffer;
      fnt: Fonts.Font;
      col, voff: SHORTINT
    END;

  PROCEDURE Append(T: Text; B: Buffer);
  PROCEDURE ChangeLooks(T: Text; beg, end: LONGINT; sel: SET;
                              fnt: Fonts.Font; col, voff: SHORTINT);
  PROCEDURE Copy(SB, DB: Buffer);
  PROCEDURE Delete(T: Text; beg, end: LONGINT);
  PROCEDURE Insert(T: Text; pos: LONGINT; B: Buffer);
  PROCEDURE Load(T: Text; f: Files.File; pos: LONGINT;
                       VAR len: LONGINT);
  PROCEDURE Open(T: Text; name: ARRAY OF CHAR);
  PROCEDURE OpenBuf(B: Buffer);
  PROCEDURE OpenReader(VAR R: Reader; T: Text; pos: LONGINT);
  PROCEDURE OpenScanner(VAR S: Scanner; T: Text; pos: LONGINT);
  PROCEDURE OpenWriter(VAR W: Writer);
  PROCEDURE Pos(VAR R: Reader): LONGINT;
  PROCEDURE Read(VAR R: Reader; VAR ch: CHAR);
  PROCEDURE Recall(VAR B: Buffer);
  PROCEDURE Save(T: Text; beg, end: LONGINT; B: Buffer);
  PROCEDURE Scan(VAR S: Scanner);
  PROCEDURE SetColor(VAR W: Writer; col: SHORTINT);
  PROCEDURE SetFont(VAR W: Writer; fnt: Fonts.Font);
  PROCEDURE SetOffset(VAR W: Writer; voff: SHORTINT);
  PROCEDURE Store(T: Text; f: Files.File; pos: LONGINT;
                       VAR len: LONGINT);
  PROCEDURE Write(VAR W: Writer; ch: CHAR);
  PROCEDURE WriteDate(VAR W: Writer; t, d: LONGINT);
  PROCEDURE WriteHex(VAR W: Writer; i: LONGINT);
  PROCEDURE WriteInt(VAR W: Writer; i, n: LONGINT);
  PROCEDURE WriteLn(VAR W: Writer);
  PROCEDURE WriteLongReal(VAR W: Writer; y: LONGREAL;
                                n: INTEGER);
  PROCEDURE WriteLongRealHex(VAR W: Writer; y: LONGREAL);
  PROCEDURE WriteReal(VAR W: Writer; x: REAL; n: INTEGER);
  PROCEDURE WriteRealFix(VAR W: Writer; x: REAL; n, k: INTEGER);
  PROCEDURE WriteRealHex(VAR W: Writer; x: REAL);
  PROCEDURE WriteString(VAR W: Writer; s: ARRAY OF CHAR);

  END Texts.
```

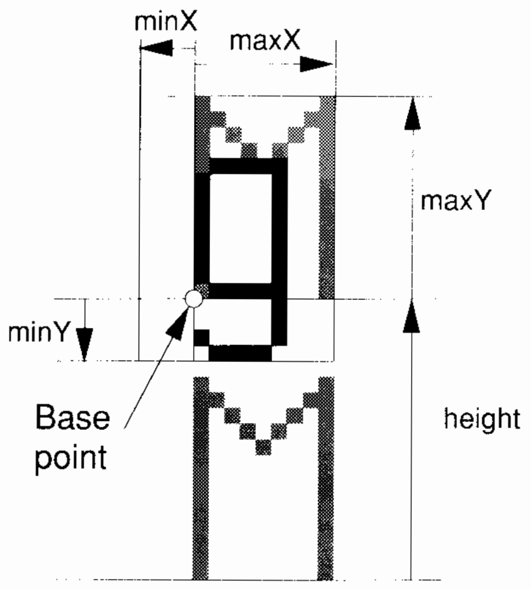
14.1 Module Fonts

The term *font* refers to the set of characters of a certain design and size. The abstract data type *Font* has the following definition:

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

where:

- *name* is the name of the file which contains font data.
- *raster* is the set of patterns used in procedure *Display.GetChar*.
- *height* is the minimum distance between text lines.
- *minX, maxX, minY, maxY* are the extremal values of the box which encloses the raster points of all characters of the font when their base points are aligned at (0, 0).



Note: The extremal values are algebraically defined; that is, *minX* ≤ 0 and *minY* ≤ 0. Module Fonts exports a variable designating the default font which is initialized from Syntax10.Scn.Fnt:

```
VAR Default: Font;
```

This

PROCEDURE This(name: ARRAY OF CHAR): Font;

Initializes the returned font from data stored in a file whose name is indicated in parameter *name*. If the file does not exist, the default font is returned.

14.2 Text and buffer

14.2.1 Text

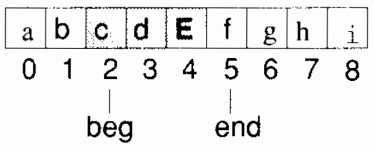
A text is an active object which is an instance of the abstract data type *Text*, which implements the notion of a sequence of characters with their associated properties. Characters may be retrieved based on their position with respect to the start of the text.

Numbering

In line with the Oberon language conventions used for arrays, the first text element has ordinal number 0.

Stretch

A *stretch* denotes a subsequence of a text beginning with element *beg* up to but not including *end*. The shorthand notation [*beg*, *end*) is used. The length of the stretch is always *end* − *beg*. For example, in the following diagram, the stretch [2, 5) consists of the elements *c*, *d* and *E*.



The abstract type *Text* is defined by:

```
TYPE
  Text = POINTER TO TextDesc;
  TextDesc = RECORD
    len: LONGINT;
    notify: Notifier
  END;
```

It has the properties:

- *len*: the text's length (in characters.)
- *notify*: a procedure invoked when the text is changed.

A text element has the properties:

- *fnt*: the character's font (type *Fonts.Font*.)
- *color*: the character's color numberer[1] (type SHORTINT.)
- *voff*: vertical offset in pixels (type SHORTINT.)

Notifier The procedure value of *notify*, termed the text's notifier, is called whenever the text is changed. It is of type:

```
TYPE Notifier = PROCEDURE(T: Text; op: INTEGER;
                                        beg, end: LONGINT);
```

where:

- *T* is the text changed prior to the call of the notifier.
- *op* is an operation code defining the nature of the change.
- *beg*, *end* are the stretch [*beg*, *end*) which were changed.

Module Texts exports the following named constants defining the operation codes in a self-explanatory manner:

```
CONST replace = 0; insert = 1; delete = 2;
```

Open PROCEDURE Open(T: Text; name: ARRAY OF CHAR);
Loads the text stored on disk in the file whose name is contained in the parameter *name*. Initializes the text *T*. The field *T.len* is set to the text's length (in characters.)

If name = "" (empty string) or if no file with that name exists, a new text is created.

The file must be marked either as a text file or an ASCII file. A text file is preceded by a text block identifier 0F001H (−4095 decimal) two bytes in length. Files created with *Edit.Store* are so marked. If it is an ASCII file, the default font color and vertical offset values are applied.

For example, an instance of type *Text* is created as follows:

```
NEW(Txt); Txt.notify := TextFrames.NotifyDisplay;
Texts.Open(Txt, "InputFile");
```

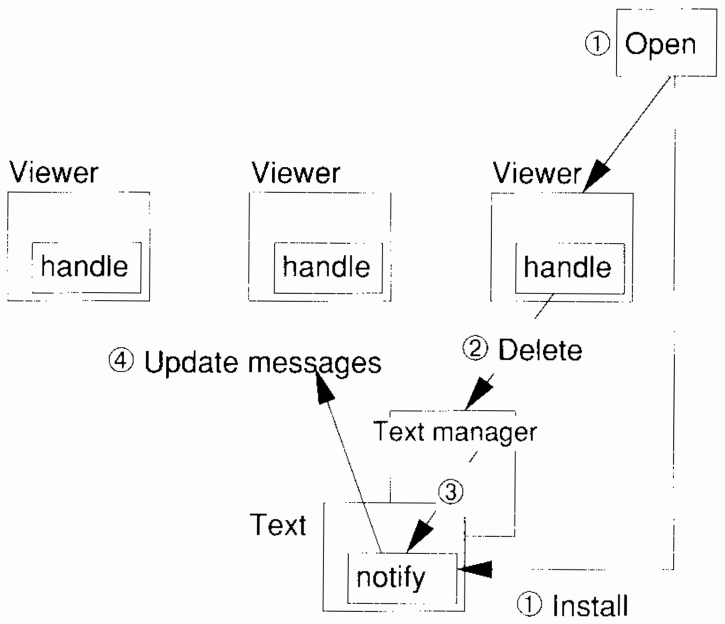
TextFrames.NotifyDisplay is a notifier for standard texts.

[1] See Chapter 12.

14.2.2 Display of texts in viewers

Texts are typically displayed in a text viewer. The displayed view of the text must reflect all changes. The task of synchronizing the viewers is complicated if the same text is displayed in more than one viewer. The notifier concept provides an elegant solution. The notifier is called whenever the text is modified. Texts which are to be displayed in standard text viewers use the notifier *TextFrames.NotifyDisplay* which broadcasts messages of type *TextFrames.UpdateMsg*, indicating the nature of the change and allowing the display managers of the affected viewers to update their views of the text.

The following diagram depicts creation of a text viewer, installation of the notifier and broadcast of update messages.



(1) The command *Open* creates the viewer, opens the displayed text and installs the notifier.

(2) Later, the handler may invoke a procedure which changes the text (for example, *Delete*.)

(3) The text manager performs the operation on the text's data structure and calls *notify*.

(4) The procedure *notify* sends update messages to all visible viewers including the one where the text was modified. These viewers may actualize their display if they show the changed text.

We wish to point to the great generality of the scheme. Texts make no assumption about message sending. It is the programmer of the viewer class who chooses the appropriate update mechanism.

14.2.3 The buffer

A buffer is another abstract data type describing a sequence of characters with their properties. It differs from a text in two respects:

(1) There is no notifier.

(2) No procedures are provided to store buffers on disk.

Buffers provide, therefore, temporary text data structures which exist no longer than the duration of a session. The absence of a notifier makes a buffer more efficient when assembling the characters comprising a text. Buffers are, therefore, used by writers for that purpose.

The abstract data type *Buffer* has the following definition:

```
TYPE
  Buffer = POINTER TO BufferDesc;
  BufferDesc = RECORD
    len: LONGINT (* Length of the buffer *)
  END;
```

OpenBuf

```
PROCEDURE OpenBuf(B: Buffer);
```

Initializes the buffer *B*. The field *B.len* is set to 0. As in the case of text, *B* must be created first with NEW(*B*).

14.2.4 Operations on texts and buffers

ChangeLooks

```
PROCEDURE ChangeLooks(T: Text; beg, end: LONGINT; sel: SET;
                      fnt: Fonts.Font; col, voff: SHORTINT);
```

Changes the attributes color, font or vertical offset of the stretch [*beg*, *end*) in text *T*. The notifier *T.notify* is called. Results are undefined if the stretch [*beg*, *end*) is not valid.

Which of the attributes is changed is directed by the parameter *sel*:

- 0 IN *sel*: the font is changed to *fnt*.
- 1 IN *sel*: the color is changed to *col*.
- 2 IN *sel*: the vertical offset is changed to *voff*.

Delete

```
PROCEDURE Delete(T: Text; beg, end: LONGINT);
```

Deletes stretch [*beg*, *end*) in text *T*. The notifier *T.notify* is called. Results are not defined if the stretch is invalid.

Copy

```
PROCEDURE Copy(SB: Buffer; DB: Buffer);
```

Appends a copy of buffer *SB* to buffer *DB*. The source buffer is not cleared.

Recall

```
PROCEDURE Recall(VAR B: Buffer);
```

Recalls the most recently deleted stretch of text in buffer *B*. There is no need to open the buffer prior to passing it as actual parameter.

Save

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

Appends stretch [*beg*, *end*) in text *T* to the end of buffer *B*.

Insert

```
PROCEDURE Insert(T: Text; pos: LONGINT; B: Buffer);
```

Inserts the contents of buffer *B* into text *T* at position *pos*. After completion, the first character of *B* occupies position *pos* in *T*. The buffer *B* is cleared. The notifier *T.notify* is called. Results are undefined if *pos* is outside of the text.

For example, if *T* has textual value 'This␣is␣the␣Oberon␣guide', and *B* = 'new␣', where the character '␣' denotes a space, then *Texts.Insert*(*T*, 12, *B*) yields a text *T* with value:

```
"This␣is␣the␣new␣Oberon␣guide"
```

Append

```
PROCEDURE Append(T: Text; B: Buffer);
```

Appends the contents of buffer *B* to text *T*. The buffer *B* is cleared (*B.len* = 0.) The notifier *T.notify* is called. Append is an abbreviation for *Insert*(*T*, *T.len*, *B*).

14.3 Reading from texts, writing to buffers

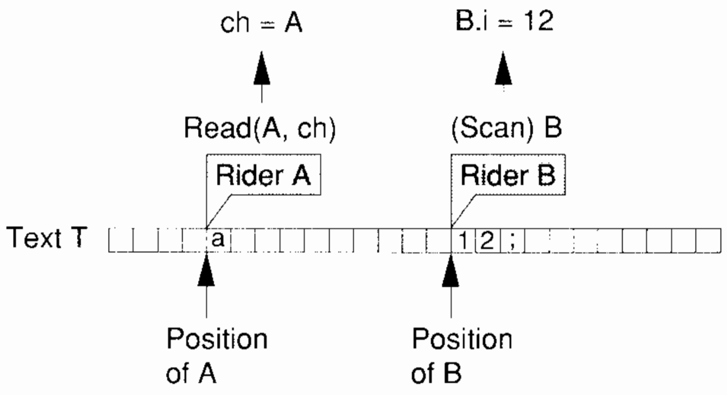
Texts and buffers are sequential data structures with many similarities to files. In fact, they are implemented as extensions of the Oberon file system. As with the file's rider, the read/write operations on texts are performed through instances of abstract data types. There are three:

(1) *Reader*: to read characters from a text.

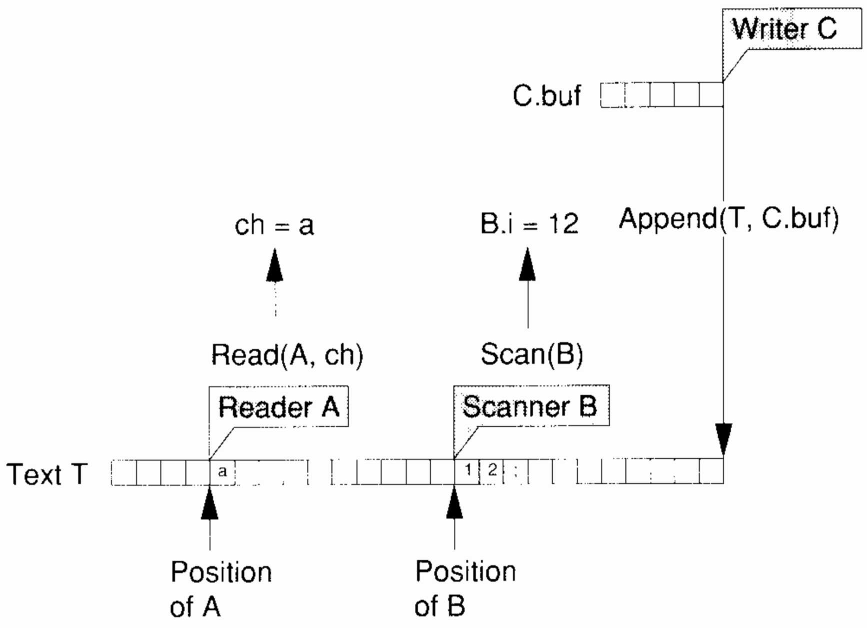
(2) *Scanner*: to read symbols (integers, reals etc.) and translate them to an internal representation.

(3) *Writer*: to assemble text data in buffers.

It is the reader, the scanner and the writer which possess the property *position*, not the object text. Several readers and scanners may operate on the same text.



Writers only operate on an associated buffer. The written characters are always appended. When a chunk of text is assembled, the buffer is typically inserted into a text. At that moment, the notifier is called. If the text is displayed and uses message passing, all views are updated.



14.3.1 The reader

A reader is an instance of the abstract data type *Reader* and affords sequential read access to the elements of an associated text. It has the following definition:

```
TYPE
  Reader = RECORD
    (Files.Rider)
    eot: BOOLEAN;
    fnt: Fonts.Font;
    col, voff: SHORTINT
  END;
```

The reader's properties are:

- *an associated text* on which the reader operates (hidden.)
- *a position* in the text (hidden.)
- *eot*: an end-of-text condition.
- *fnt*: the font of the character which was last read.
- *col*: the color number of the character which was last read.
- *voff*: the vertical offset of the character which was last read.

Note: The reader also inherits properties from the rider which it extends. However, these play no role in the application of the reader.

OpenReader

```
PROCEDURE OpenReader(VAR R: Reader; T: Text; pos: LONGINT);
```

Initializes reader *R* and sets it at position *pos* in text *T*. The field *R.eot* is set to FALSE; the attribute fields remain unspecified. The first read operation after the reader is opened will return the character at position *pos* in text *T*.

If *pos* is outside the text ($pos \geqslant T.len$), the result is unspecified. The programmer cannot rely on *R.eot* being TRUE.

Read

```
PROCEDURE Read(VAR R: Reader; VAR ch: CHAR);
```

The character *ch* found at the position of the reader *R* is returned and that position is incremented by 1. If an attempt is made to read beyond the text's length, then *R.eot* = TRUE and *ch* remains undefined. Otherwise, the reader's fields are set:

- *R.eof* to FALSE.
- *R.fnt*, *R.col* and *R.voff* to the attributes of *ch*.

Pos

```
PROCEDURE Pos(VAR R: Reader): LONGINT;
```

Returns the position of reader *R* in its associated text. At the end of text, the result is unspecified.

14.3.2 The scanner

A scanner is an instance of the abstract data type *Scanner*. It parses an associated text for tokens of the following kind:

- numbers (of various types);
- strings;
- names;
- special symbols.

Tokens are scanned sequentially, translated to an internal representation and returned by the scanner in one of its result fields.

A scanner has the following definition:

```
TYPE
  Scanner = RECORD
    (Reader)
    nextCh: CHAR;
    line, class: INTEGER;
    i: LONGINT;
    x: REAL;
    y: LONGREAL;
    c: CHAR;
    len: SHORTINT;
    s: ARRAY 32 OF CHAR
  END;
```

The properties of a scanner are:

- *an associated text* on which the scanner operates (hidden.)
- *a position* in the text (hidden.)
- *nextCh*: the character in the text which immediately follows a scanned symbol.
- *line*: the line number of the scanned symbol.
- *class*: a code indicating the type of the scanned symbol (see definition later.)
- *i*, *x*, *y*, *c*: fields of appropriate type in which results are returned.
- *len*: the length of a name or string.
- *s*: field in which names or strings are returned.

Note: The scanner also inherits properties from the underlying reader and rider. Except possibly for *eot*, thesé are not productive in the use of the scanner.

Scanner class codes

The field *class* which identifies the type of the scanned symbol admits values 0 . . 6. The following constants define the meaning of *class*:

```
CONST Inval = 0; Name = 1; String = 2; Int = 3; Real = 4; LongReal = 5;
          Char = 6;
```

Scanned symbols

The scanner parses the associated text for tokens. While parsing, blanks (SP or 20X) and tab characters (HT or 09X) are ignored. Carriage return characters (CR or 0DX) are also skipped but they have the effect of incrementing the line count. Tokens are defined by the lexicographic syntax:

Name = *NamePart* { "." *NamePart* }.
NamePart = *letter* { *letter* | *digit* }.
String = """ { *letter* | *digit* | *specialChar* } """ |
"'" { *letter* | *digit* | *SpecialChar* } "'".
Integer = ["+" | "−"] *digit* { *digit* } |
["+" | "−"] *digit* { *hexDigit* } "*H*".
Real = ["+" | "−"] *digit* { *digit* } "." *digit* { *digit* }
["E" ["+" | "−"] *digit* { *digit* }].
LongReal = ["+" | "−"] *digit* { *digit* } "." *digit* { *digit* }
["D" ["+" | "−"] *digit* { *digit* }].[2]
Char = *specialChar* | *ctlChar*.[3]
specialChar = Printable ASCII characters other than letters or number ("~" | "!" | "@" | "#" | "$" ...).
ctlChar = The control character of the ASCII code (that is, ordinal number < 20X) except HT or CR.

OpenScanner

PROCEDURE OpenScanner(VAR S: Scanner; T: Text; pos: LONGINT);

Opens scanner *S* and sets it at position *pos* in text *T*. The field *S.eot* is initialized to FALSE; the output fields remain unspecified. The scan operation after a call of *OpenScanner* will return the symbol starting at position *pos* in text *T*.

If *pos* is outside the text ($pos \geqslant T.len$), the result is unspecified.

Scan

PROCEDURE Scan(VAR S: Scanner);

After each call to the procedure *Scan*, the associated text is parsed for a symbol, starting at the current position of the scanner in its associated text. Blanks and carriage return characters (CR or 0DX) are skipped. After a symbol is found, it is translated into an internal representation and assigned to the output field of the matching type. The type of symbol being scanned is identified by the field *S.class*.

The character immediately following the scanned symbol is the value of *S.nextCh*. The new position of the scanner is one more than the position of *S.nextCh*.

The scanner counts lines starting at 0 for the first one scanned. Each occurrence of a carriage return character (CR or 0DX) increments the field *S.line* by one.

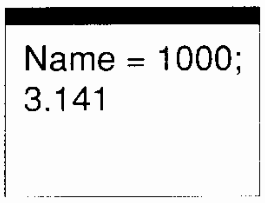
[2] A number is also of type LONGREAL if the number of digits after the decimal point exceeds the precision of reals.

[3] Other than SP, HT or CR.

The Boolean variable *S.eot* (in base type reader) is accessible and can be tested for the end-of-text condition.

Note: Unlike the case of reader, *eot* may be TRUE when the last symbol is scanned, not at the attempt to scan beyond the text. This has to be taken into account when using *eot*.

For example, let *S* be a scanner operating on the text *Txt* with textual value 'Name␣=␣1000; ¶3.141', where '␣' denotes a space and '¶' a line break control character. In a text viewer, this text would look as follows:


Name = 1000;
3.141

After opening *S* at position 0 through a call to *Texts.OpenScanner* (*S*, *Txt*, 0), successive calls to *Scan*(*S*) yield the following:

Scan(*S*)	1st	2nd	3rd	4th	5th	6th
S.eot	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE
S.nextCh	" "	" "	";"	"¶"	Undef	Undef
S.line	0	0	0	0	1	Undef
S.class	Name	Char	Int	Char	Real	Undef
S.i	Undef	Undef	1000	Undef	Undef	Undef
S.x	Undef	Undef	Undef	Undef	3.141	Undef
S.y	Undef	Undef	Undef	Undef	Undef	Undef
S.c	Undef	"="	Undef	";"	Undef	Undef
S.len	4	Undef	Undef	Undef	Undef	Undef
S.s	"Name"	Undef	Undef	Undef	Undef	Undef

Note: 'Undef' means that the programmer should not rely on the value. In the current implementation, once a field is set, it keeps its value until it is changed the next time.

If, in our example, the scanner is opened at the third character by means of *Texts.OpenScanner*(*S*, *Txt*, 2), the first call of *Scan*(*S*) yields:

```
S.class = Texts.Name
S.len = 2
S.s = "me"
```

14.3.3 The writer

A writer is an instance of the abstract data type *Writer*. The writer appends characters or symbols to the end of an associated buffer. Many of the write procedures serve to translate from the internal representation of numbers of the basic types (SHORTINT, INTEGER, LONGINT, REAL, LONGREAL) to textual symbols.

A writer has the following definition:

```
TYPE
  Writer = RECORD
    (Files.Rider)
    buf: Buffer;
    fnt: Fonts.Font;
    col, voff: SHORTINT
  END;
```

The properties of the writer are:

- *buf*: the associated buffer in which text data is assembled.
- *fnt*: the font in current use.
- *col*: the color number in current use.
- *voff*: the vertical offset in current use.

Note: While the writer also inherits the properties of the underlying rider, these are not used.

OpenWriter

```
PROCEDURE OpenWriter(VAR W: Writer);
```

Opens writer *W*. The associated buffer *W.buf* is also opened. The characteristics are *W.fnt* = *Fonts.Default*, *W.col* = *Display.white* and *W.voff* = 0.

Set attributes

```
PROCEDURE SetColor(VAR W: Writer; col: SHORTINT);
PROCEDURE SetFont(VAR W: Writer; fnt: Fonts.Font);
PROCEDURE SetOffset(VAR W: Writer; voff: SHORTINT);
```

Sets the character attributes of writer *W*. Subsequently written characters or symbols possess these attributes.

Write

```
PROCEDURE Write(VAR W: Writer; ch: CHAR);
```

Appends character *ch* to the end of the buffer *W.buf* of writer *W*.

WriteHex

```
PROCEDURE WriteHex(VAR W: Writer; i: LONGINT);
```

Converts integer *i* to a sequence of eight hexadecimal characters preceded by a blank. The resulting string is appended to the end of the buffer *W.buf* of writer *W*.

WriteInt

PROCEDURE WriteInt(VAR W: Writer; i, n: LONGINT);

Converts integer *i* to a character string representing *i* in decimal form. The resulting string is padded with blanks on the left up to a length of *n* and appended to the end of the buffer *W.buf* of writer *W*. The field size is adjusted if chosen too small.

For example, if *W.buf* has textual value "abc" and *i* = 17, then after *WriteInt*(*W*, *i*, 4), *W.buf* contains "abc␣␣17" (␣ represents a blank.)

WriteLn

PROCEDURE WriteLn(VAR W: Writer);

Appends a carriage return character (CR or 0DX) to the end of the buffer *W.buf* of writer *W*.

WriteLongReal

PROCEDURE WriteLongReal(VAR W: Writer; y: LONGREAL; n: INTEGER);

Converts long real *y* to a character string representing *y* in decimal form. The resulting string is padded with blanks on the left up to a length of *n* and appended to the end of the buffer *W.buf* of writer *W*. The field size is adjusted if chosen too small.

WriteLong-RealHex

PROCEDURE WriteLongRealHex(VAR W: Writer; y: LONGREAL);

Converts long real *y* to a sequence of 16 hexadecimal characters preceded by a blank. The resulting character string is appended to the end of the buffer *W.buf* of writer *W*.

WriteReal

PROCEDURE WriteReal(VAR W: Writer; x: REAL; n: INTEGER);

Converts real *x* to a character string representing *x* in decimal form. The resulting string is padded with blanks on the left up to a length of *n* and appended to the end of the buffer *W.buf* of writer *W*. The field size is adjusted if chosen too small.

WriteRealFix

PROCEDURE WriteRealFix(VAR W: Writer; x: REAL; n, k: INTEGER);

Appends the fixed-point decimal character representation of real variable *x*, right justified in a field of *n* places with *k* places for the decimal fraction to the end of the buffer *W.buf* of writer *W*. The field size is adjusted if chosen too small.

For example, if *W.buf* has textual value "abc" and *pi* = 3.14159.. (up to machine precision), then after *WriteRealFix*(*W*, *pi*, 6, 3), *W.buf* is "abc␣3.141" (the symbol ␣ represents blanks.)

WriteRealHex

PROCEDURE WriteRealHex(VAR W: Writer; x: REAL);

Converts real *x* to a sequence of eight hexadecimal characters preceded by a blank. The resulting character string is appended to the end of the buffer *W.buf* of writer *W*.

WriteString

```
PROCEDURE WriteString(VAR W: Writer; s: ARRAY OF CHAR);
```

Appends string *s* to the end of the buffer *W.buf* of writer *W*.

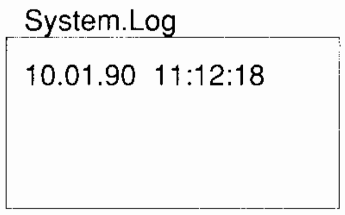
WriteDate

```
PROCEDURE WriteDate(VAR W: Writer; t, d: LONGINT);
```

Appends the date to the buffer *W.buf* of writer *W*. The parameters *t* (time) and *d* (date) are in the format defined by the procedure *Oberon.GetTime*. The format is 'dd.mm.yy hh:mm:ss' with dd: day, mm: month, yy: year, hh: hour, mm: minute and ss: second. All two digit numbers with leading zeros if necessary.

For example, the following writes the date to the system log:

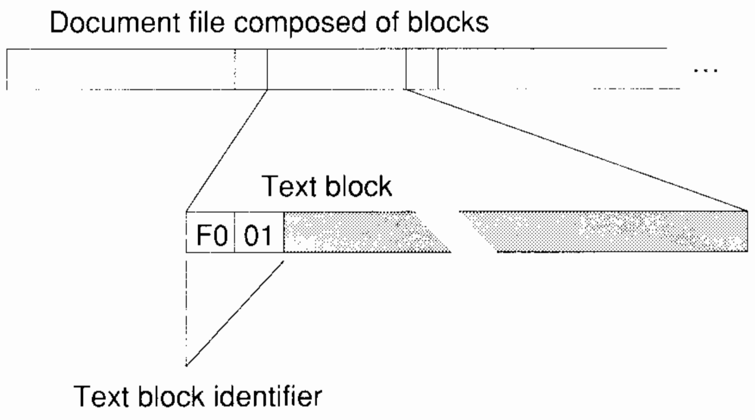
```
VAR t, d: LONGINT; W: Texts.Writer;

Oberon.GetTime(t, d);
Texts.WriteDate(W, t, d);
Texts.WriteLn(W);
Texts.Append(Oberon.Log, W.buf);
```



```
System.Log
10.01.90  11:12:18
```

14.4 Text files

Texts are stored on disk files in the form of *text blocks*. A given file may contain several blocks of different type, of which text blocks are just one. A complex document, for example, may be composed of text blocks, one for each paragraph, graphics blocks and interspersed descriptor blocks.



Each block is preceded by a mark composed of two bytes. The first byte is the block mark identifier (0F0X) and the second one specifies the type of block (01X for text blocks.)

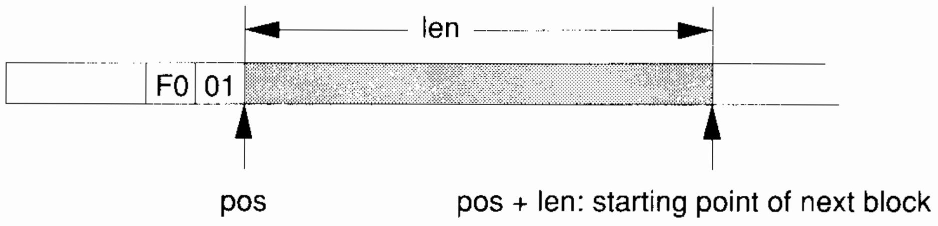
Two procedures are provided to read and write text blocks: *Load* and *Store*.

Load

PROCEDURE Load(T: Text; f: Files.File; pos: LONGINT; VAR len: LONGINT);

Loads the text block stored on disk in file *f* starting at *pos*. The file position *pos* designates the start of text data after the text block identifier (0F001H.) On completion, *len* is set to the length of the block excluding the text block identifier.

It is the caller's responsibility to ensure that *pos* is a valid starting position.



Note: Load knows the length of the text block from information contained in the text data. It reads only one text block (not to the end of file.)

For example, the following loads two consecutive text blocks from file *F*:

```
VAR
  F: Files.File;
  T1, T2: Texts.Text;
  pos, length: LONGINT;

pos := 2;
Texts.Load(T1, F, pos, length);
pos := pos + length + 2;
Texts.Load(T2, F, pos, length);
```

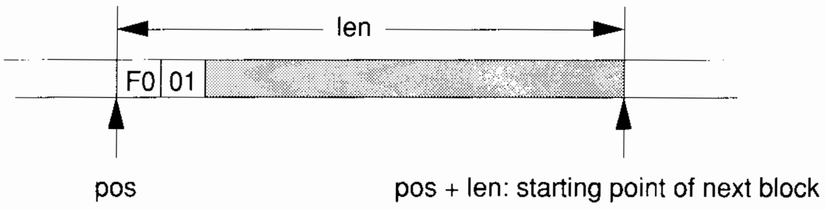
Note: In general, it is prudent to read the block identifier before the call to *Load* and test whether it is really a text block.

Store

PROCEDURE Store(T: Text; f: Files.file; pos: LONGINT; VAR len: LONGINT);

Stores the text *T* in file *f* starting at file position *pos*. The text block identifier (0F001H) is written first. On completion, *len* is set to the length of the block including the text block identifier.

It is the caller's responsibility to ensure that *pos* is a valid starting position. Also, the caller must make sure that no subsequent block is overwritten.



Note: If an existing text block is overwritten, the user must ensure that the newly written text is not longer than the block which is overwritten. If a gap ensues, then *pos* + *length* is no longer the starting position of the next block. It is advisable not to overwrite existing blocks.

For example, the following appends two text blocks at the end of file *F*.

```
VAR
  F: Files.File;
  T1, T2: Texts.Text;
  pos, length: LONGINT;

pos := Files.Length(F);
Texts.Store(T1, F, pos, length);
Texts.Store(T2, F, pos + length, length);
```

18.3 Working with texts

In this section, we discuss common techniques for dealing with texts. The programmer deals frequently with texts since they have many uses in editors and compilers, and commands use them to display (non-volatile) output.

The reader should have a good understanding of the mechanism used by texts to update their display. We recapitulate that texts are active objects. When a standard text is changed, it broadcasts an update message to all visible viewers which indicates the change. If a viewer displays that text, it will update the display in response to the message. Therefore, the programmer does not have to worry about the display of the text – in fact, he or she does not even have to know the viewers in which the text is displayed. In a sense, Oberon displays texts automatically.

Some frequently occurring constructs are as follows:

System log text	Oberon.Log
Create an empty text	text := TextFrames.Text("");
Create a new text and initialize it from disk file *name*	text := TextFrames.Text(name);
Write to system log	Texts.OpenWriter(W);[1] ... Texts.Write(W, ch); ... Texts.Append(Oberon.Log, W.buf);
Insert buffer *buf* at position *pos* in *text*	Texts.Insert(text, pos, buf);
Append buffer *buf* to *text*	Texts.Append(text, buf);
Read sequential characters	Texts.OpenReader(R, text, pos); ... Texts.Read(R, ch); ...

[1] Typically in the body of the module.

Scan sequential symbols	Texts.OpenScanner(S, text, pos);
	... Texts.Scan(S);
	IF S.class = Texts.Int THEN[2]
	   n := S.i (* Process integer *)
	END; ...

Save stretch [*beg*, *end*) of *text* in buffer *buf*	NEW(buf); Texts.OpenBuf(buf); Texts.Save(text, beg, end, buf);

18.3.1 Creating a text

Sometimes, the text on which a command works is given, for example the text of the main frame of a text viewer. Often, however, a new instance of a text (a variable of type *Texts.Text*) needs to be created. Recall that *TextFrames.Text* serves this purpose. Its source text is a good way to learn how a text is generated:

```
PROCEDURE Text*(name: ARRAY OF CHAR): Texts.Text;
VAR text: Texts.Text;
BEGIN
   (* Create an instance and install notifier *)
   NEW(text);
   text.notify := TextFrames.NotifyDisplay;
   Texts.Open(text, name); (* Initialize text from file name *)
   RETURN text
END Text;
```

In most cases, *TextFrames.Text* can be used to generate a text. If it is necessary to create an instance of *Texts.Text* explicitly, it is important *not to forget to install the notifier*. Otherwise, addressing exceptions will result.

18.3.2 Reading from a text

A reader is used to access the characters in a text in sequential order. The reader is associated with the text and can be set to an arbitrary initial position. Each call returns a character, one after another.

The following program excerpt is typical for the use of a reader. It processes all the characters in a text, starting at a given position *pos*:

[2] Other class codes are *Texts.Char*, *Texts.Inval*, *Texts.LongReal*, *Texts.Name*, *Texts.Real* and *Texts.String* (see Chapter 14.)

```
PROCEDURE ProcessText(text: Texts.Text; pos: LONGINT);
VAR R: Texts.Reader; ch: CHAR;
BEGIN
  IF pos < text.len THEN (* The position is within the text *)
    Texts.OpenReader(R, text, pos);
    Texts.Read(R, ch);   (* Read character at position pos *)
    WHILE ~R.eot DO
      ...                (* Process character ch *)
      Texts.Read(R, ch)  (* Read next character *)
    END;
  END
END ProcessText;
```

If it is not guaranteed that the starting position is within the text, a test must be performed as in our example.

Let us look at a complete example of a procedure which is patterned after the preceding program skeleton. The procedure *GetItalics* searches a text from the initial position *pos* for the first occurrence of an italics font. It returns this position, if it exists, otherwise −1 results.

```
PROCEDURE GetItalics(text: Texts.Text; pos: LONGINT): LONGINT;
VAR R: Texts.Reader; ch: CHAR; Syntax10i: Fonts.Font;
BEGIN
  IF pos < text.len THEN
    Syntax10i := Fonts.This("Syntax10i.Scn.Fnt"); (* The italics font *)
    Texts.OpenReader(R, text, pos);
    Texts.Read(R, ch);                          (* Read first character *)
    WHILE ~R.eot DO
      IF R.fnt = Syntax10i THEN RETURN Texts.Pos(R) - 1 END;
      Texts.Read(R, ch)                         (* Read next character *)
    END
  END;
  RETURN - 1
END GetItalics;
```

Observe that we made use of the field *R.fnt* which reports the font of the last character read.

18.3.3 Scanning a text

The reader provides sequential access to the characters comprising a text. Using a reader, texts can be analyzed and processed as shown in the previous example.

A frequently recurring task, however, is parsing a text for numbers, names, strings and special characters and translating the textual representation of these symbols to internal values. The *scanner* is provided to facilitate these jobs.

Scanners are frequently used to parse parameter lists. Examples of such use will be given in Section 18.4. Another typical use is to read numerical parameters from input files.

The basic principle in using a scanner is simple. A symbol is scanned, its type tested and then the appropriate output field of the scanner is further processed:

```
pos := (* Determine starting position *)
Texts.OpenScanner(S, text, pos); (* Set scanner to starting position *)
Texts.Scan(S);                   (* Scan a symbol *)
IF S.class = Texts.Name THEN     (* Test whether it is a name *)
  name := S.s;
  ...   (* Process name *)
ELSIF S.class = Texts.Int THEN   (* Test whether it is an integer *)
  i := S.i;
  ...   (* Process integer *)
END;
```

where:

```
text: Texts.Text;
pos: LONGINT;
S: Texts.Scanner;
name: ARRAY 32 OF CHAR;
i: INTEGER;
```

Again, if it is not guaranteed that *pos* is within the text, a test must be performed.

If the whole text is to be processed by the scanner, special precautions are required at the end. In the case of a reader *R*, the predicate ~*R.eot* provides a natural stopping condition for the WHILE loop reading all characters of a text. The scanner inherits field *eot* from the reader. However, the end-of-text condition *S.eot* may already yield TRUE while the last valid symbol is returned, not only after an attempt to scan beyond the end of the text. In this case, a WHILE loop using the predicate ~*S.eot* misses the last symbol. Therefore, it is always preferable to terminate a sequence of scan operations with a definite symbol.

18.3.4 Writing a text

The conversion of the internal representation of basic types, such as integers, reals and characters, to textual representation is a frequent operation. The writer performs this conversion using an associated variable of type *Texts.Buffer*. Each call to a write procedure appends the buffer. When a suitable chunk of text is composed, the writer's buffer can be inserted into a text by means of the procedures *Texts.Insert* and *Texts.Append*. At this point, the text's notifier is activated. If it is a standard notifier, it alerts all visible viewers of the change, which will be reflected on the display.

Thus, writing a text to the screen requires:

(1) Installing the text in a viewer and opening that viewer.

(2) Changing the text using a procedure of the text manager.

The manner in which the first is performed will be explained later. In our next example, we will write to the system log. The text *Oberon.Log* already exists and is installed in the log viewer. If the log viewer is visible, then changing the log text will automatically display the change in the system log and all its clones (produced with *System.Copy*, for example.)

Module LogOut

Module *LogOut* exports procedures which write variables to the system log in symbolic form. It is a useful utility for debugging commands.

```
MODULE LogOut;

IMPORT Texts, Oberon;

VAR W: Texts.Writer;

PROCEDURE PutInt*(txt: ARRAY OF CHAR; i: LONGINT);
BEGIN
  Texts.WriteString(W, txt); (* Append string txt to W.buf *)
  Texts.WriteInt(W, i, 1);   (* Convert i to text and append to W.buf *)
  Texts.WriteLn(W);          (* Append a carriage return character to W.buf *)
  Texts.Append(Oberon.Log, W.buf) (* Display W.buf in log *)
END PutInt;

PROCEDURE PutString*(txt: ARRAY OF CHAR);
BEGIN
  Texts.WriteString(W, txt); (* Append string txt to W.buf *)
  Texts.WriteLn(W);          (* Append a carriage return character to W.buf *)
  Texts.Append(Oberon.Log, W.buf) (* Display W.buf in log *)
END PutInt;

(* Other procedures for reals etc. *)

BEGIN
  Texts.OpenWriter(W)
END LogOut.
```

After a call:

```
LogOut.PutInt("i =", i)
```

a line "i = 36" (assuming that i = 36) appears at the end of the system log text in the log viewer.

Only one writer per module

The writer is based on the file system. Opening a writer opens a work file – a relatively complex operation. In most cases, one writer per module is enough. It is, therefore, good practice to open the writer once per session in the module's body, as shown in the foregoing example. *Local writers should be avoided.*

Output of a matrix

Our next example deals with the output of a matrix of real numbers. The procedure *MatrixOut* produces a buffer which is returned as result. The buffer can later be inserted into a text by means of *Texts.Insert* or *Texts.Append*. Since these procedures invoke the notifier, the result will be displayed at that point in time.

```
PROCEDURE MatrixOut(VAR A: ARRAY OF ARRAY OF REAL): Texts.Buffer;
VAR
  i, j: INTEGER;
  Syntax10x: Fonts.Font;
BEGIN
  Syntax10x := Fonts.This("Syntax10x.Scn.Fnt");
  Texts.SetFont(W, Syntax10x);
  i := 0;
  WHILE i < LEN(A, 0) DO
    j := 0;
    WHILE j < LEN(A, 1) DO
      Texts.WriteRealFix(W, A[i, j], 15, 5); (* 15 places, 5 decimal places *)
      INC(j)
    END;
    Texts.WriteLn(W);                         (* Write carriage return *)
    INC(i)
  END;
  RETURN W.buf
END MatrixOut;
```

W is a variable of type *Texts.Writer* which is globally defined and opened in the module's body. The formal parameter *A* is a VAR parameter to avoid copying of an array. Compared to the use of a value parameter, this is more efficient and saves memory, too.