Differences between Experimental Oberon and Original Oberon

Andreas Pirklbauer

12.12.1990 / 30.6.2016

Experimental Oberon[1] is a revision of the Original Oberon operating system (2013 Edition)[2]. It contains a number of simplifications, generalizations and enhancements of functionality, the module structure and the system building tools. Some modifications are purely of experimental nature, while others serve the purpose of exploring potential future extensions, for example to add support for touch display devices.

1. Continuous fractional line scrolling with variable line spaces

Continuous fractional line scrolling enables completely smooth, pixel-by-pixel scrolling of texts with variable lines spaces and dragging of viewers with continuous content refresh. To the purist, such a feature may represent an "unnecessary embellishment" of Original Oberon, but it is simply indispensable if the system is to support touch display devices where a mouse is absent and viewers may not have scrollbars. In such an environment, continuous scrolling is the only (acceptable) way to scroll. As a welcome side effect, the initial learning curve for new users is *considerably* reduced.

2. Multiple logical display areas ("virtual displays")

Original Oberon operates on a *single* abstract logical display area, which is decomposed into a number of vertical *tracks*, each of which is further decomposed into a number of horizontal *viewers*. Experimental Oberon adds the ability to create *several* such display areas on the fly and seamlessly switch between them. The conceptual hierarchy of the extended display system consists of the triplet *(display, track, viewer)*. Consequently, the base module *Viewers* exports routines to add and remove (logical) *displays*, to open and close *tracks* within displays and to open and close individual *viewers* within tracks. There are no restrictions on the number of displays, tracks or viewers that can be created. In addition, text selections, central logs and focus viewers are separately defined for each logical display. The scheme naturally maps to systems with multiple physical monitors. It can also be used to realize fast context switching, for example in response to a swipe gesture on a touch display device.

3. Simplified viewer message type hierarchy

A number of viewer message types (e.g., *ModifyMsg*) and identifiers (e.g., *extend, reduce)* have been eliminated from the Oberon system. The remaining message types and identifiers now have a single, well-defined purpose. For example, restoring a viewer is now accomplished exclusively by means of a *restore* message identifier.

[1] *http://www.github.com/andreaspirklbauer/Oberon-experimental (adapted from the original implementation prepared by the author in 1990 on a Ceres computer at ETH)*
[2] *http://www.projectoberon.com (Original Oberon, 2013 Edition)*

Several viewer message types (or minimal subsets thereof) that appeared to be generic enough to be made generally available to *all* types of viewers have been merged and moved from higher-level modules to the base module *Viewers*, resulting in fewer inter-module dependencies in the process. Most notably, module *TextFrames* no longer depends on module *MenuViewers*, making it now possible to (recursively) embed text viewers in *other* types of composite viewers, for example a viewer consisting of an arbitrary number of text, graphic or picture sub viewers.

4. Enhanced viewer management

The previously separate notions of *frame* and *viewer* have been *united* into a single, unified *viewer* concept. The original distinction between *frames* and *viewers* (a viewer's state of visibility) appeared to be rather marginal and hardly justified in Oberon's display system, which is based on hierarchical tiling rather than overlapping windows. Note that the nesting properties of the original *frame* concept have been fully preserved in the new implementation. We also emphasize that the unified viewer concept is more flexible than the original scheme, as *each* sub viewer (previously called a *frame*) can now have its *own* state of visibility[3]. This makes it possible to bring the concept of *hierarchical tiling* – already used for *top-level* viewers – all the way down to the sub viewer level, enabling multi-column text viewers for example.

The viewer operations *Viewers.Change, MenuViewers.Modify* and *TextFrames.Modify* have been generalized to handle *arbitrary* viewer modifications, including pure vertical translations (without changing the viewer's height), modifying the bottom line, the top line *and* the height of a viewer simultaneously, and moving multiple viewers with a single mouse drag operation.

A new command *System.Clone*, displayed in the title bar of every menu viewer, creates a new logical display area on the fly and displays a copy of the initiating viewer *there*, avoiding the need to create overlaying tracks[4]. The end user can toggle back and forth between the two copies of the viewer (i.e. switch logical display areas) with a single mouse click. By comparison, the Original Oberon command *System.Copy* creates a copy of the viewer in the *same* logical display area, i.e. without creating a new one first.

Alternatively, the user can use the new command *System.Expand* to expand a viewer as much as possible by reducing all the other viewers in its track to their minimum heights. The end user can "switch back" to any of the "compressed" viewers by clicking on *System.Expand* again in any of their (still visible) title bars.

5. System building tools

A minimal version of the Oberon system *building tools* has been added, consisting of the two modules *Linker*[5] and *Builder*. They provide the necessary tools to establish the prerequisites for the regular Oberon startup process[6].

---

[3] *For sub viewers, the field 'state' in the viewer descriptor is not interpreted by the viewer manager, but by the enclosing viewer(s).*
[4] *The command System.Grow would generate a copy of the viewer extending over the entire column (or display), lifting the viewer to an "overlay" in the third dimension.*
[5] *The file 'Linker.Mod' has been included from a different source (https://github.com/charlesap/io). It was slightly adapted for Experimental Oberon.*
[6] *Currently not implemented is a tool to prepare a disk initially – which consists of a single 'Kernel.PutSector' statement that initializes the root page of the file directory (sector 1).*

The command *Linker.Link* links a set of Oberon binary files together and generates an Oberon *boot file* from them. The linker is almost identical to the regular Oberon loader (*Modules.Load),* except that it writes the result to a file on disk instead of loading (and linking) the specified modules in main memory.

The command *Builder.CreateBootTrack*[7] loads a valid *regular boot file*, as generated by the command *Linker.Link*, onto the boot area (sectors 2-63 in Oberon 2013) of the local disk, which is one of the two valid Oberon boot sources (the other one being the serial line). From there, the Oberon *boot loader* will transfer it byte for byte into main memory during stage 1 of the regular boot process, before transferring control to its top module.

In sum, to generate a new *regular* Oberon *boot file* and load it onto the local disk's boot area, one can execute the following commands (*on* the system which is to be modified):

```
ORP.Compile Kernel.Mod FileDir.Mod Files.Mod Modules.Mod ~   … compile the modules of the inner core
Linker.Link Modules ~                                       … create a regular boot file (Modules.bin)
Builder.CreateBootTrack Modules.bin ~                       … load boot file onto the disk's boot area
```

When *adding* new modules to a *boot file*, the need to call their module initialization bodies during stage 1 of the boot process may arise, i.e. when the *boot file* is loaded into main memory by the Oberon *boot loader* during system restart or when the reset button is pressed. Recall that the Oberon *boot loader* merely *transfers* the *boot file* byte for byte from a valid boot source into main memory and then transfers control to its *top* module. But it does *not* call the initialization bodies of the *other* modules that are also transferred as part of the *boot file* (this is why the *inner core* modules *Kernel*, *FileDir* and *Files* don't *have* initialization bodies – they wouldn't be executed anyway).

The easiest way to add a new module *with* a module initialization body to an Oberon *boot file* is to move its initialization code to an exported procedure *Init* and call it from the *top* module of the modules contained in the *boot file*. This is the approach chosen in Original Oberon, which uses module *Modules* as the top module of the *inner core*.

An alternative solution is to extract the starting addresses of the module initialization bodies of the just loaded modules from their module descriptors in main memory and simply call them, as shown in procedure *InitMod* below (see chapter 6 of the book *Project Oberon* for a detailed description of the format of a *module descriptor* in main memory; here it suffices to know that it contains a list of "entries" for exported entities, the first one of which points to the initialization code of the module itself).

```
PROCEDURE InitMod (name: ARRAY OF CHAR);
  VAR mod: Modules.Module; P: Modules.Command; w: INTEGER;
BEGIN mod := Modules.root;
  WHILE (mod # NIL) & (name # mod.name) DO mod := mod.next END ;
  IF mod # NIL THEN SYSTEM.GET(mod.ent, w);
    P := SYSTEM.VAL(Modules.Command, mod.code + w); P
  END
END InitMod;
```

In the following example, module *Oberon* is chosen as the new top module of the *inner core*, while module *System* is configured to be the top module of the *outer core*.

[7] *Historically, the boot file was located on a separate "track" on a spinning hard disk (or floppy disk) on a Ceres computer. We retain the name for nostalgic reasons.*

Stage 1: Modules loaded by the Oberon boot loader (BootLoad.Mod) and initialized by their top module

```
MODULE Modules; ...                          ... old top module of the inner core, now just a regular module
BEGIN ...                                    ... no longer loads module Oberon (as in Original Oberon)
END Modules.

MODULE Oberon; ...                           ... new top module of the inner core, now part of the boot file
  PROCEDURE InitMod (name: ...);             ... see above (calls the initialization body of the specified module)
  BEGIN ...
  END InitMod;

BEGIN                                        ... boot loader will branch here after transferring the boot file
  InitMod(„Modules“);                        ... must be called first (establishes a working file system)
  InitMod(„Input“);
  InitMod(„Display“);
  InitMod(„Viewers“);
  InitMod(„Fonts“);
  InitMod(„Texts“);
  Modules.Load(„System“, Mod); ...           ... load the outer core using the regular Oberon loader
  Loop                                       ... transfer control to Oberon's central loop
END Oberon.
```

Stage 2: Modules loaded and initialized by the regular Oberon loader (Modules.Load)

```
MODULE System;
  IMPORT ..., MenuViewers, TextFrames;
  …
END System.
```