Differences between Experimental Oberon and Original Oberon

Andreas Pirklbauer

12.12.1990 / 29.6.2016

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 scrollbars may not exist for all viewers. 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 display system consists of the triplet *(display, track, viewer)*. 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. 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.

3. Simplified viewer message type hierarchy

A number of viewer message types (e.g., *ModifyMsg*) and identifiers (e.g., *extend, reduce)* have been eliminated. 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 united 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. The nesting properties of the original *frame* concept have been fully preserved. 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.

The command *System.Clone*, displayed in the title bar of every menu viewer, creates a new logical display on the fly and displays a copy of the initiating viewer *there*, avoiding the need to create overlaying tracks[4]. The 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 command *System.Copy* creates a copy of the viewer in the *same* logical display, i.e. without creating a new one first.

Alternatively, the user can use the command *System.Expand* to expand a viewer as much as possible by reducing all other viewers in its track to their minimum heights. The 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].

When the power to a computer is turned on or the reset button is being pressed, the computer's *boot firmware* (called *boot loader* in Oberon) is activated. The *boot firmware* is a small standalone program permanently resident in the computer's read-only store,

[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).*

such as a standard read-only memory (ROM) or a field-programmable read-only memory (PROM), which is part of the computer's hardware.

The main task of the Oberon *boot loader* is to load the regular Oberon *boot file* – a pre-linked binary containing the four modules *Kernel, FileDir, Files* and *Modules* (which are said to constitute the Oberon *inner core*) – from a valid boot source into the computer's main memory and then transfer control to its top module (module *Modules*). Then its job is done until the next time the computer is restarted or the reset button is pressed.

Tools to modify the Oberon boot loader

In general, there is no need to modify the Oberon *boot loader* (*BootLoad.Mod*). Notable exceptions include situations with special requirements, e.g., in embedded systems, or when there is a justified need to add network code allowing one to boot the Oberon system over an IP-based network instead of from the local disk or over the serial line.

To generate a new object file of the Oberon *boot loader,* one needs to mark its source code with an asterisk immediately after the symbol MODULE and compile it with the *regular* Oberon compiler. This will cause the compiler to generate modified starting and ending sequences that make the boot loader suitable for execution as a standalone program (the first instruction will branch to the module's initialization code, and the last instruction will branch to memory location 0). Loading the generated image of the *boot loader* into the permanent read-only store of the target system typically requires the use of proprietary tools from the hardware manufacturer.

Tools to modify the Oberon boot file

The command *Linker.Link* links a set of Oberon files together and generates a *regular boot file* from them, used to start the Oberon system from the local disk. 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 modules in main memory.

The format of the *regular* boot file is *defined* to *exactly* mirror the standard Oberon main memory layout, starting at location 0 (see chapter 8 of the book *Project Oberon* for a detailed description of Oberon's memory layout). In particular, the first location in the boot file (and later in main memory once it has been loaded by the boot loader) contains a branch instruction to the initialization code of the *top* module of the modules that constitute the boot file. Therefore, one can simply transfer the regular boot file byte for byte from a valid boot source into main memory and branch to memory location 0 to pass control to its *top* module – which is precisely what the *Oberon boot loader* does.

The command *Builder.CreateBootTrack*[7] loads a regular Oberon *boot file*, as generated by the command *Linker.Link*, onto the local disk's boot area (sectors 2-63 in Oberon 2013), which is one of the two valid Oberon boot sources (the other one being the serial line). From there, the *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 create a new Oberon *boot file* and load it onto the local disk's boot area, one can execute the following commands (on the system whose *boot file* is to be modified):

[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 only.*

```
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
```

Note that the last command overwrites the disk's boot area of the *running* Oberon system. A disk backup prior to running it is thus recommended. In addition, all modules required to successfully restart the Oberon system (module *System* and all its imports) *and* the Oberon compiler must also be recompiled *before* system restart, in order to prevent *invalid module key* errors after it. Other modules may be recompiled later.

When modifying the Oberon *boot file* containing the pre-linked modules of the Oberon *inner core*, it is recommended to keep it as small as possible. It is stored in the disk's boot area and thus can only be altered using a special tool (*Builder.CreateBootTrack*). Also note that the boot area is rather small (sectors 2-63 in Oberon 2013 with a sector size of 1 KB, or 62 KB in total) – probably by design. Thus, if one wants to extend the *inner core*, one may have to increase the size of the boot area as well. While it is trivial to do so (it requires adding a single statement in procedure *Kernel.InitSecMap* to mark the additional disk sectors as allocated), existing disks cannot be reused by systems running the new *inner core*[8].

When adding new modules to the *inner core*, the need to call their module initialization bodies during stage 1 of the boot process may arise. Recall that the boot loader merely *transfers* the pre-linked *boot file* byte for byte from the disk's boot area 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 the *inner core* is to move its initialization code to an exported procedure and call it from the inner core's top module. 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 address of the module's initialization body from the module descriptor in main memory and simply call it, 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*.

[8] *Although one could, in principle, traverse the file directory and reassign new sectors to the files that occupy the area to be reallocated to the enlarged boot area.*

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, i.e. now part of the boot file*)
  PROCEDURE InitMod (name: ...); ...  (*see above; calls the initialization body of the specified module*)
BEGIN (*initialize inner core modules*)  (*boot loader will branch to 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 new outer core using the regular Oberon loader*)
  Loop                              (*transfer control to the Oberon central loop*)
END Oberon.
```

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

```
MODULE System;                      (*new top module of the outer core*)
  IMPORT MenuViewers, TextFrames... (*outer core modules are initialized by the regular Oberon loader*)
  …
END System.
```

In order to prepare the bare metal of a new system *initially*, i.e. for the very first time, a special version of the boot file called the “build up boot file” must be used. It contains a few additional facilities for communication with a “host” system and is loaded by the Oberon boot loader over a serial line from the host computer, when the user selects a certain switch on the target system to be built. This *initial* “system build” process is currently not supported in Experimental Oberon (see chapter 14 of the book *Project Oberon* for a description of such a facility).