Differences between Experimental Oberon and Original Oberon

Andreas Pirklbauer

12.12.1990 / 9.5.2016

Experimental Oberon[1] is a revision of the Original Oberon operating system[2]. It contains a number of simplifications, generalizations and enhancements of functionality, the module structure and the boot process.

1. Continuous fractional line scrolling with variable line spaces

Continuous fractional line scrolling enables smooth, pixel-by-pixel scrolling of texts with variable lines spaces and dragging of viewers with continuous refreshing of content.

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. The conceptual hierarchy of the display system consists of the triplet *(display, track, viewer)*. The extended base module *Viewers* exports routines to add and remove logical *displays*, to open and close *tracks* within logical displays and to open and close individual *viewers* within tracks[3]. There are no restrictions on the number of displays, tracks or viewers that can be created. Messages can be sent to one or several displays. Text selections, central logs and default looks are defined for each display. The scheme naturally maps to systems with multiple physical monitors.

3. Streamlined viewer message type hierarchy

The viewer message type hierarchy has been streamlined. A number of message types (or minimal subsets thereof) that appeared to be generic enough to make them generally available to *all* types of viewers have been merged and moved from higher-level modules to the base module *Viewers*, resulting in fewer module dependencies in the process. Most notably, module *TextViewers* no longer depends on module *MenuViewers*, making it possible to 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. Unified viewer concept

The concepts of *frame* and *viewer* have been merged into a single, unified *viewer* concept. The notion of *frame* has been eliminated. The original distinction between *frames* and *viewers* (a viewer's state of visibility) appeared to be hardly justified in Oberon, which uses hierarchical tiling rather than overlapping windows. The nesting properties of the original *frame* concept have been fully preserved and been made explicit through the addition of a

[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 Zurich)*
[2] *http://www.projectoberon.com (Original Oberon, 2013 Edition)*
[3] *Copying a viewer using System.Copy while holding down the "n" key will also create a new logical display area (and open the cloned viewer in its user track)*

parent field for each viewer, making the implementation of certain operations – such as identifying the enclosing viewer of a sub viewer – both simpler and more efficient. The unified viewer concept is not only simpler, but also more flexible than the original scheme, as *each* sub viewer (previously called a *frame*) can now have its *own* state of visibility.

5. Refined module structure

The routines implementing the *tasking system* have been moved from module *Oberon* to a new module *Tasks*. This makes it easier to change the task model of the existing viewer manager or to add new viewer managers with their own task models.

The routines for *cursor handling* have been moved from module *Oberon* to a new module *Cursors.* The viewer manager and cursor handler share and operate on an abstract logical display rather than on individual physical monitors (one can imagine two parallel planes on top of a logical display area, one displaying viewers and one displaying cursors). Viewer management and cursor handling can therefore be viewed as clients of the module that implements the raster operations. This fact is now also reflected in the module structure.

6. Simplified boot process

The number of stages in the *boot process* has been reduced from 3 to 2. The first stage continues to be the transfer of the *boot file* (a file containing the pre-linked modules of the *inner core*) from a valid boot source into main memory by the *boot loader* (a small program permanently resident in read-only store), when the system is started. It has not been modified, except for a small change in module *Modules* to load an enlarged outer core (now including module *System* and all its imports) during the second boot stage.

7. System building tools

A minimal version of the Oberon *building tools* has been added, consisting of the modules *Linker*[4] and *Builder*. They provide the necessary tools to establish the prerequisites for the regular startup process[5].

The command *Linker.Link* links a set of object files together and generates a *regular boot file* from them, which is 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 command *Builder.CreateBootTrack*[6] loads a pre-linked regular boot file, as generated by the command *Linker.Link*, onto the local disk's boot area (sectors 2-63 in Oberon 2013). 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.

The format of the *regular* Oberon 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 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

[4] *The file 'Linker.Mod' has been included from a different source (https://github.com/charlesap/io). It was slightly adapted for Experimental Oberon.*
[5] *Currently not implemented is a tool to prepare a disk initially – which in Oberon is a single 'Kernel.PutSector' statement that initializes the root page of the file directory (sector 1).*
[6] *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..*

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. This is precisely what the *Oberon boot loader* does.

In sum, to create a new Oberon *inner core* and load it onto the local disk's boot area, one can execute the following commands (on the system whose inner core 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
```

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

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" is used. It contains additional facilities for communication with a "host" system and is loaded by the boot loader over a serial line from the host computer, when the user selects a certain switch on the 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).

8. Tools to modify the boot loader and the inner core

In general, there is no need to modify the Oberon *boot loader*, a small standalone program permanently resident in read-only store. 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 *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. Loading the generated image into the permanent, read-only store of the target system typically requires the use of proprietary tools from the hardware manufacturer.

When modifying the Oberon *boot file* containing the pre-linked modules of the *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.*[7]

When adding 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

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

transfers control to its top module. But it does *not* call the initialization bodies of the *other* modules 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 that is called from the inner core's top module. An alternative solution is to extract the starting address of the module's initialization body from the module descriptor and simply call it, as shown in procedure *InitMod* below (see chapter 6 of the book *Project Oberon* for a description of the format of a *module descriptor*; 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 chosen as the top module of the *outer core*.

```
MODULE Modules; ...                    (*old top module of the inner core, now just a regular module*)
BEGIN ...                              (*no longer loads module System, 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                                  (*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(„Cursors“);
   InitMod(„Viewers“);
   InitMod(„Fonts“);
   InitMod(„Texts“);
   InitMod(„Tasks“);
   Modules.Load(„System“, Mod); ...    (*load the outer core using the regular Oberon loader*)
   Tasks.Loop                          (*transfer control to the Oberon central loop*)
END Oberon.


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