Differences between Experimental Oberon and Original Oberon

Andreas Pirklbauer

12.12.1990 / 12.12.2016

Experimental Oberon[1] is a revision of the Original Oberon[2] operating system, containing several simplifications, generalizations and functional enhancements. Some modifications are purely of experimental nature, while others serve the explicit 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 has been added, enabling completely smooth scrolling of texts with variable lines spaces and dragging of entire viewers with continuous content refresh. Both *far* (positional) and *near* (pixel-based) scrolling are realized[3]. To the purist, such a feature may represent an "unnecessary embellishment" of 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 and presents a more natural user interface. As a welcome side effect, the initial learning curve for users new to Oberon is *considerably* reduced.

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

Original Oberon was designed to operate 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 to seamlessly switch between them. The extended conceptual hierarchy of the display system consists of the triplet *(display, track, viewer)*. The Oberon 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 display area. The scheme naturally maps to systems with multiple *physical* monitors. It can also be used to realize super-fast context switching, for example in response to a swipe gesture on a touch display device.

The command *System.OpenDisplay* opens a new logical display, *System.CloseDisplay* closes an existing one, while *System.ShowDisplays* lists all active displays. The command *System.Clone*, displayed in the title bar of every menu viewer, opens a new logical display on the fly *and* displays a copy of the initiating viewer *there*. The user can toggle between the two copies of the viewer (switch logical displays) with a single mouse click[4].

---

[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.inf.ethz.ch/personal/wirth/ProjectOberon/index.html (Original Oberon, 2013 Edition); see also http://www.projectoberon.com*
[3] *The system automatically switches back and forth between the two types of scrolling based on the horizontal position of the mouse pointer.*
[4] *By comparison, the Original Oberon commands System.Copy and System.Grow create a copy of the original viewer in the same logical display area (System.Copy opens another viewer in the same track, while System.Grow extends the viewer's copy over the entire column or display, effectively lifting the viewer to an "overlay" in the third dimension).*

Alternatively, the user can select the command *System.Expand*, also displayed in the title bar of every menu viewer, to expand a viewer "as much as possible" by reducing all other viewers in the track to their minimum heights, and switch back to any of the "compressed" viewers by simply clicking on *System.Expand* again in any of their (still visible) title bars.

3. Unified viewer concept and enhanced viewer management

The previously separate notions of *frame* and *viewer* have been *united* into a single, unified *viewer* concept, in which the nesting properties of the original *frame* concept have been fully preserved. 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. We note that the unified viewer concept is not only simpler, but also more flexible than the original scheme, as *each* sub viewer (called a *frame* in Original Oberon) can now have its *own* state of visibility. This makes it possible to bring the concept of *hierarchical tiling*, which is already used for *top-level* viewers in Original Oberon, all the way down to the sub viewer level as well, enabling "newspaper-style" *multi-column* text layouts for example.

The 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), simultaneously modifying a viewer's bottom line, top line *and* height, and moving multiple viewers with a *single* mouse drag operation.

4. Simplified viewer message type hierarchy

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

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

5. Safe module unloading

The semantics of *unloading* a module has been refined as follows. If clients exist, a module is never unloaded. If no references to the module exist in the remaining modules and data structures, the module is unloaded and its associated memory is released. If there *are* references to the module, however, it is removed only from the *list* of loaded modules (allowing newer versions of the module to be loaded). Such *hidden* modules are later *automatically* removed from memory as soon as there are *no more* references to it. References can be in the form of *type tags* (addresses of type descriptors) in dynamic (heap) objects of other modules pointing to descriptors of types declared in the specified module, or in the form of *procedure variables* installed in static or dynamic objects of other modules referring to procedures declared in the module[5].

[5] *In general, there can be type, procedure and pointer references from static or dynamic objects of other modules to static or dynamic objects of a module to be unloaded. However, only* ***dynamic type*** *references and* ***static and dynamic procedure*** *references need to be checked. S t a t i c type references from (global variable declarations in) other modules referring to types declared in the module to be unloaded don't need to be checked, as these are handled via their import relationship (if clients exist, a module is never unloaded).*

Modules that have been removed *only* from the list of loaded modules are marked with an asterisk in the output of the command *System.ShowModules*. To achieve their automatic *final* removal from memory, the Oberon background task handling garbage collection includes a call to the command *Modules.Collect*. Thus, module data is kept in memory as long as needed and is removed from it as soon as possible.

In sum, unloading a module affects only *future* references to it, while *past* references from other modules remain completely unaffected, For example, older versions of the module's code can still be executed if they are referenced by procedure variables in other loaded modules, even if a newer version of the module has been loaded in the meantime[6].

A simple *mark-scan* scheme is used to check *dynamic* references to a module. In the *mark* phase, dynamic records reachable by *all other* loaded modules are marked. This excludes records that are reachable *only* by the module itself. The *scan* phase scans the heap element by element, unmarks marked objects and verifies whether the *type tags* of the encountered marked records point to descriptors of types declared in the module to be unloaded, or whether *procedure variables* in these records refer to procedures declared in that module. The latter check is also performed for all *static* procedure variables.

In order to make such a validation pass possible, type descriptors for *dynamic* records and descriptors of *global* module data have been extended with a list of *procedure variable offsets*, adopting an approach employed in one of the earlier implementations of Oberon[7]. These additional offsets are simply appended to the existing fields of each descriptor, i.e. their offsets are greater than those of the fields of the *pointer variables* needed for the garbage collector, in order not to impact its performance. The compiler generating these modified descriptors, the format of the Oberon object file containing them and the module loader transferring them into memory have been adjusted accordingly.

The following code excerpt (procedure *Modules.Check*) shows a possible realization:

```
PROCEDURE Check*(mod: Module; VAR res: INTEGER); (*type and procedure references*)
  VAR M: Module; pref, pvadr, r: LONGINT;
BEGIN (*mod # NIL*) M := root;
  WHILE M # NIL DO
    IF (M # mod) & (M.name[0] # 0X) THEN Kernel.Mark(M.ptr) END ;
    M := M.next
  END ;
  Kernel.Check(mod.data, mod.var, mod.code, mod.imp, res); (*dynamic refs*)
  IF res = 0 THEN M := root;
    WHILE (M # NIL) & (res = 0) DO (*static refs*)
      IF (M # mod) & (M.name[0] # 0X) THEN
        pref := M.pvar; SYSTEM.GET(pref, pvadr);
        WHILE pvadr # 0 DO (*procedure variables*) SYSTEM.GET(pvadr, r);
          IF (mod.code <= r) & (r < mod.imp) THEN res := 3 END ;
          INC(pref, 4); SYSTEM.GET(pref, pvadr)
        END
      END ;
      M := M.next
    END
  END
END Check;
```

Pointer references from static or dynamic pointer variables of other modules to dynamic objects of the module to be unloaded don't need to be checked either, as these are handled by the garbage collector. Pointer references to static objects are only possible by resorting to low-level facilities and should be avoided (and, in fact, be disallowed).

[6] *If an older version of a module's code accesses global variables (of itself or of other modules), it will automatically access the "right" version of such variables – as it should.*

[7] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*

where procedure *Kernel.Check* implements the *scan* phase for *dynamic* references to the module, i.e. references from objects located in the heap:

```
PROCEDURE Check*(type0, type1, code0, code1: LONGINT; VAR res: INTEGER);
  VAR p, r, mark, tag, size, offadr, offset: LONGINT;
BEGIN p := heapOrg; res := 0;
  REPEAT SYSTEM.GET(p+4, mark);
    IF mark < 0 THEN (*free*) SYSTEM.GET(p, size)
    ELSE (*allocated*) SYSTEM.GET(p, tag); SYSTEM.GET(tag, size);
      IF mark > 0 THEN SYSTEM.PUT(p+4, 0); (*unmark*)
        IF (type0 <= tag) & (tag < type1) THEN (*types*) res := 1
        ELSIF res = 0 THEN offadr := tag + 16; SYSTEM.GET(offadr, offset);
          WHILE offset # -1 DO (*pointers*) INC(offadr, 4); SYSTEM.GET(offadr, offset) END ;
          INC(offadr, 4); SYSTEM.GET(offadr, offset);
          WHILE offset # -1 DO (*procedure variables*) SYSTEM.GET(p+8+offset, r);
            IF (code0 <= r) & (r < code1) THEN res := 2 END ;
            INC(offadr, 4); SYSTEM.GET(offadr, offset)
          END
        END
      END
    END ;
    INC(p, size)
  UNTIL p >= heapLim
END Check;
```

Although the operation of reference checking, as sketched above, may appear expensive at first, in practice there is no performance issue. It is similar in complexity to garbage collection and thus barely noticeable. In addition, module unloading is usually rare.

6. System building tools

A minimal version of the Oberon system *building tools* has been added, consisting of the two modules *Linker*[8] and *Builder*. They provide the necessary mechanisms and tools to establish the prerequisites for the *regular* Oberon startup process[9].

When the power to a computer is turned on or the reset button is pressed, the computer's *boot firmware* is activated. The boot firmware is a small standalone program permanently resident in the computer's read-only store, such as a read-only memory (ROM) or a field-programmable read-only memory (PROM), which is part of the computer's hardware.

In Oberon, the computer's boot firmware is called the *boot loader*. Its main task is to load a valid Oberon *boot file* (a pre-linked binary containing a set of Oberon modules) from a valid *boot source* into memory and transfer control to its *top* module, i.e. the module that directly or indirectly imports all other modules contained in the boot file. Then its job is done until the next time the computer is restarted or the reset button is pressed.

There are currently two valid boot sources in Oberon: a local disk (realized using an SD card in Oberon 2013) and a communication channel (an RS-232 serial line). The default boot source is the local disk. It is used by the *regular* Oberon startup process each time the computer is powered on or the reset button is pressed.

[8] *The linker has been included from a different source (https://github.com/charlesap/io). It was slightly adapted and extended for Experimental Oberon's object file format.*
[9] *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 object files together and generates a valid *boot file*. The linker is almost identical to the regular module loader (*Modules.Load),* except that it writes the result to a file on disk instead of loading and linking the modules in memory.

The command *Builder.CreateBootTrack*[10] loads a boot file, as generated by the command *Linker.Link*, onto the *boot area* (sectors 2-63 in Oberon 2013) of the disk, one of the two valid boot sources. From there, the boot loader will transfer it byte for byte into memory during stage 1 of the *regular* boot process, before transferring control to its top module.

The format of the boot file is *defined* to exactly mirror the standard Oberon storage layout. In particular, location 0 in the boot file (and later in main memory once it has been loaded) contains a branch instruction to the initialization code of the top module of the boot file. Thus, the boot loader can simply transfer the boot file and then branch to location 0[11].

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
```

Note that the last command overwrites the boot area of the *running* system. A backup of the local disk is therefore recommended before experimenting with new *inner cores*[12].

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 memory by the boot loader during system restart or when the reset button is pressed.

We recall that the *boot loader* merely *transfers* the boot file byte for byte from a valid boot source into memory, but does not call the initialization bodies of the modules that are also transferred as part of the boot file (this is, in fact, 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 body of the just loaded modules from their module descriptors now present in 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 memory; here it suffices to know that it contains a pointer to 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);
```

[10] *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.*
[11] *The boot loader also deposits some additional key data in fixed main memory locations, to allow proper continuation of the boot process once control has been transferred.*
[12] *When using an Oberon emulator (e.g., https://github.com/pdewacht/oberon-risc-emu) on a host system, one can simply make a copy of the directory containing the disk image.*

```
    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 new top module of the *outer core*.

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

```
MODULE Modules; ..                      ... old top module of the inner core, now just a regular module
  IMPORT SYSTEM, Files;
  ...
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: ...);
  BEGIN ...                             ... see above (calls the initialization body of the specified module)
  END InitMod;

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(„Viewers“);
  InitMod(„Fonts“);
  InitMod(„Texts“);
  Modules.Load(„System“, Mod); ...      ... load the 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;
  IMPORT ..., MenuViewers, TextFrames;
  ...
END System.
```

We note that this module configuration reduces the number of stages in the regular *boot process* from 3 to 2, thereby simplifying it somewhat (at the expense of extending the *inner core*). If one prefers to keep the *inner core* minimal, one could also choose to extend the *outer core* instead, by including module *System* and all its direct and indirect imports. This would have the (small) disadvantage that the viewer complex would then be “locked” in the *outer core*. However, an Oberon system without a viewer manager hardly makes sense, even in closed server environments.