The Experimental Oberon System

Andreas Pirklbauer

12.12.2019

Experimental Oberon[1] is a revision of the FPGA Oberon[2] operating system and its compiler. It contains a number of enhancements, including the *Revised Oberon-2* programming language implementing as a strict superset of Revised Oberon (Oberon-07), safe module unloading, system building and maintenance tools, smooth fractional line scrolling of displayed texts with variable line spaces, multiple logical displays and other improvements. Some of these enhancements are of a purely experimental nature, while others serve the explicit purpose of exploring potential future extensions.

1. Revised Oberon-2 programming language

Revised Oberon-2 is a revision of the programming language Oberon-2[3]. The main difference to the original is that it implements a strict superset of *Revised Oberon (Oberon-07)* as defined in 2007/2016[4] rather than being based on the original language *Oberon* as defined in 1988/1990[5].

Original Oberon (1988/1990)	➔	Original Oberon-2 (1991/1993)
🡻		
Revised Oberon (2007/2016)	➔	**Revised Oberon-2 (2019/2020)**

Its principal features include type-bound procedures, a dynamic heap allocation procedure for fixed-length and open arrays, a numeric case statement, exporting and importing of string constants, and no access to intermediate objects within nested scopes.

The *Revised Oberon-2* language is described in more detail in a separate document[6].

2. Safe module unloading

In the Oberon system, there exist several possible types of references to a loaded module M:

1. *Client references* exist when other loaded modules import module M. Client modules may refer by name to exported constants, types, variables or procedures declared in module M.

2. *Type references* exist when *type tags* (addresses of type descriptors) in dynamic objects reachable by other loaded modules refer to descriptors of types declared in module M.

3. *Procedure variable references* exist when procedure variables in static or dynamic objects reachable by other loaded modules refer to procedures declared in module M.

---

[1] http://www.github.com/andreaspirklbauer/Oberon-experimental
[2] http://www.inf.ethz.ch/personal/wirth/ProjectOberon/index.html (FPGA Oberon, 2013 Edition); see also http://www.projectoberon.com
[3] Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991
[4] http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf (Revision 3.5.2016)
[5] https://inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf (Revision 1.10.1990)
[6] https://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/The-Revised-Oberon2-Programming-Language.pdf

4. *Pointer variable references to static module data* exist when pointer variables in static or dynamic objects reachable by other loaded modules refer to *static* objects declared in module M. Such references are only possible by resorting to low-level facilities and therefore should be avoided (pointer variables should point exclusively to anonymous variables allocated in the *heap* when needed during program execution). But since they *can* in theory exist, a specific implementation may opt to also check for them prior to module unloading.

Most implementations of the Oberon system only check for *client* references prior to unloading a module. Other types of references are not checked, although various approaches are typically employed to alleviate the situation where such references do exist. Not checking all references leaves a system in an *unsafe* state, which may become *unstable* the moment another module loaded later overwrites a previously released module block. In addition, unloading of *groups* of modules with (potentially cyclic) references only among themselves is usually not supported.

Experimental Oberon implements completely "safe" unloading of modules and module groups by checking *all* possible types of references from anywhere in the system. The mechanism of *safe* module unloading is described in more detail in a separate document[7].

3. System building and maintenance tools

A minimal version of the Oberon system *building tools*, as outlined in chapter 14 of the book *Project Oberon 2013 Edition*, has been added. They provide the necessary tools to establish the prerequisites for the *regular* Oberon startup process either on an existing *local* system or on a bare metal *target* system connected to a *host* via a data link (e.g., an RS-232 serial line).

The boot linker (procedure *ORL.Link*) links a set of object files together and generates a valid boot file from them. It can be used to either generate the *regular* boot file to be loaded onto the *boot area* of a disk or the *build-up* boot file sent over a data link to a target system. The command *ORL.Load* loads a valid *boot file*, as generated by the command *ORL.Link*, onto the boot area of the local disk, one of the two valid *boot sources* (the other being the data link).

The tools to build an entirely new Oberon system on a bare metal *target* system connected to a *host* system via a communication link are provided by the module pair *ORC* (for Oberon to RISC Connection) running on the host system and *Oberon0* running on the target system. One can also include an *entire* Oberon system in a single *boot file*. Sending a pre-linked binary file containing the entire Oberon system over a serial link to a target system is similar to booting a commercial operating system in a *Plug & Play* fashion over the network or from a USB stick.

There is a variety of other Oberon-0 commands that can be initiated from the host system once the Oberon-0 command interpreter is running on the target system, for example commands for system inspection, loading and unloading of modules or the (remote) execution of commands.

Finally, there are tools to modify the boot loader itself (module *BootLoad*), a small *standalone* program permanently resident in the computer's read-only store (ROM or PROM).

The system building tools for FPGA Oberon and Experimental Oberon, as outlined above, are described in more detail in a separate document[8].

---

[7] *http://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/Safe-module-unloading-in-Oberon.pdf*
[8] *https://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/The-Oberon-system-building-tools.pdf*

4. Smooth scrolling of displayed texts with variable line spaces

Experimental Oberon enables completely smooth, fractional line scrolling of displayed texts with variable lines spaces and dragging of entire viewers with continuous content refresh. Both *far* (positional) scrolling and *near* (pixel-based) scrolling are realized. The system automatically switches back and forth between the two scrolling modes based on the horizontal position of the mouse pointer. For 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 the Oberon system and its user interface is *considerably* reduced.

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

The Oberon system 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 (or *displays* for short) on the fly and to seamlessly switch between them. Thus, the extended conceptual hierarchy of the display system consists of the triplet *(display, track, viewer)* and consequently the underlying base module *Viewers* exports procedures to add and remove *displays*, open and close *tracks* within existing displays and open and close individual *viewers* within tracks. There are no restrictions on the number of displays, tracks or viewers that can be created. Focus viewers and text selections are separately defined for each display, but there is only one system-wide log.

This scheme naturally maps to systems with multiple *physical* monitors attached to the same computer. 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 name* opens a new logical display with the specified name, *System.CloseDisplay id* closes an existing one. *System.ShowDisplays* lists all open displays, *System.ThisDisplay* shows the display *id* and *name* of the current display, *System.SetDisplay id* switches to a new display, *System.SetDisplayName id name* assigns a new name to an existing display, and *System.PrevDisplay* and *System.NextDisplay* "rotate" through the open displays.

The additional commands *System.Expand, System.Spread* and *System.Clone* are displayed in the title bar of every menu viewer. *System.Expand* expands the viewer *as much as possible* by reducing *all* other viewers in its track to their minimum heights, leaving just their title bars visible. The user can switch back to any of the minimized viewers by clicking on *System.Expand* again in any of these title bars. *System.Spread* evenly redistributes all viewers vertically. This may be useful after having invoked *System.Expand*. *System.Clone* opens a new logical display on the fly *and* displays a copy of the initiating viewer *there*. The user can then toggle between the two copies of the viewer (i.e. switch logical *displays*) with a single mouse click.

* * *