# The Experimental Oberon System

Andreas Pirklbauer

31.12.2018

Experimental Oberon[1] is a revision of the FPGA Oberon[2] operating system and its Oberon-07 compiler. It contains a number of enhancements, including continuous fractional line scrolling with variable line spaces, multiple logical displays, safe module unloading, system building and maintenance tools, and an enhanced compiler with various new features, including type-bound procedures, a dynamic heap allocation procedure for fixed-length and open arrays, a numeric case statement, exporting and importing of string constants, forward references and forward declarations of procedures, no access to intermediate objects in nested scopes, and module contexts. Some of these 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 enables completely smooth 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[3]. 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.

## 2. 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. This 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.

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,

[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] *The system automatically switches back and forth between the two scrolling modes based on the horizontal position of the mouse pointer.*

*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[4].

## 3. Safe module unloading

In FPGA Oberon, there exist three possible types of references to a loaded module M[5]:

1. *Client references* exist when other loaded modules *import* 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.

In most Oberon implementations, only *client* references are checked prior to module unloading. *Type* and *procedure variable* references are usually not checked, although various approaches are typically employed to address the case where modules *with* such references are unloaded[6].

In Experimental Oberon, *all* possible types of references to a loaded module or module group are checked as follows prior to module *unloading* (see Figure 1):

- If clients exist, a module or module group is never unloaded.
- If no client, type or procedure variable references to a module or module group exist in the remaining modules or data structures, it is unloaded and its associated memory is released.
- If no clients, but type or procedure variable references exist, the module *unload* command takes no action by default and merely displays the names of the modules containing the references that caused the removal to fail. If, however, the *force* option /f is specified[7], the modules are initially removed (only) from the *list* of loaded modules, without releasing their associated memory. Such "hidden" modules are later physically removed from *memory*, as soon as there are no more references to them from anywhere in the system.

---

*[4] By comparison, the Original Oberon commands System.Copy and System.Grow create a copy of the original viewer in the same (and only) logical display area – System.Copy opens another viewer in the same track of the display, while System.Grow extends the viewer's copy over the entire column or display, lifting the viewer to an "overlay" in the third dimension.*

*[5] An Oberon module can be viewed as a container of types, variables and procedures, where variables can be procedure-typed. Types can be declared as global types (in which case they can be exported and referenced by name in clients) or as types local to a procedure (in which case they cannot be exported). Variables can be statically declared as global variables (allocated in the module area when a module is loaded) or as local variables (allocated on the stack when a procedure is called), or they can be dynamically allocated in the heap via the predefined procedure NEW. Procedures can be declared as global or local procedures. They can also be assigned to global procedure variables (of the same or a different module) or to procedure variable fields in dynamic objects – even if they are not exported. Thus, in general there can be type, variable, procedure and procedure variable references from both static and dynamic objects of other modules to static or dynamic objects of the modules to be unloaded. However, only dynamic type references and static and dynamic procedure variable references need to be checked during module unloading for the following reasons: First, static type, variable and procedure references from other loaded modules can only refer by name to types, variables or procedures declared in the modules to be unloaded. Such references are already handled via their import/export relationship during module unloading (if clients exist, a module or module group is never unloaded) and therefore don't need to be checked separately. Second, dynamic pointer variable references from global or dynamic pointer variables of other modules to dynamic objects reachable by the modules to be unloaded should not be checked, as such references should not prevent module unloading. In the Oberon system, such references will be handled by the garbage collector during a future garbage collection cycle, i.e. heap records reachable by the just unloaded modules and other still loaded modules will not be collected, whereas heap records that were reachable only by the just unloaded modules will be collected – as they should. Thus, the handling of dynamic pointer references is delegated to the garbage collector. Finally, pointer variable references to static objects declared as global variables are only possible by resorting to low-level facilities and should be avoided (and, in fact, be disallowed).*

*[6] See http://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/Historical-notes-on-module-unloading-in-Oberon.pdf*

*[7] The force option /f must be specified at the end of the list of modules to be unloaded, e.g., System.Free M1 M2 M3/f*

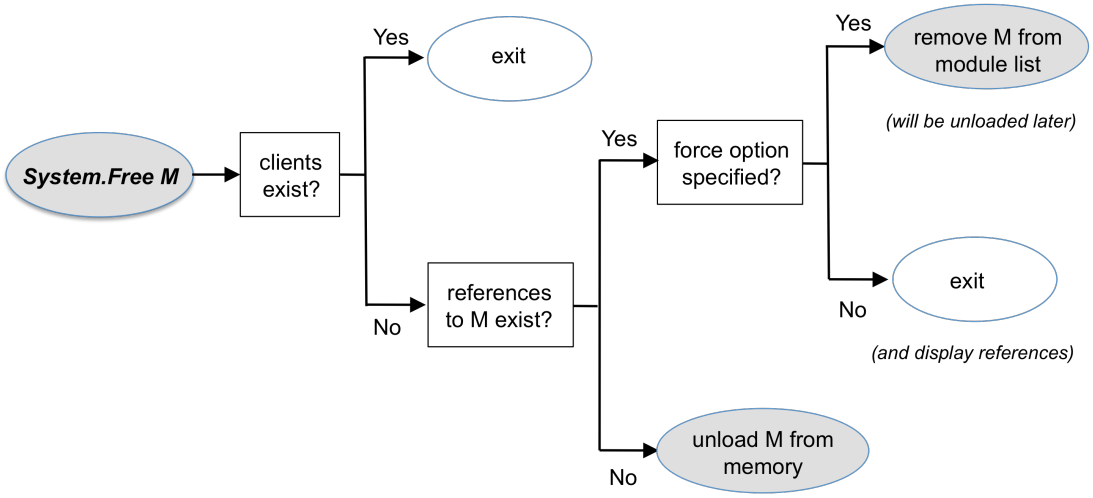


**Figure 1:** Safe module unloading in Experimental Oberon

Removing modules only from the *list* of loaded modules amounts to *renaming* them, allowing other modules with the same names to be reloaded again, without having to unload (from memory) earlier versions that are still referenced[8]. Removing a module from *memory* frees up the memory area previously occupied by the module block[9].

To make the removal of no longer referenced hidden module data *automatic*, a new command *Modules.Collect* has been included in the Oberon background task handling garbage collection. It checks all possible combinations of *k* modules chosen from *n* hidden modules for clients and references, and removes those module subgroups from memory that are no longer referenced. The command *Modules.Collect* can also be manually activated at any time. Alternatively, the user can invoke *System.Collect*, which includes a call to *Modules.Collect*.

In sum, module unloading does not affect past references, as module data is kept in memory for exactly as long as necessary and removed from it as soon as possible.

If a module *group* is to be unloaded and there exist clients or references *only* within this group, it is unloaded *as a whole*. This can be used to remove module groups with *cyclic* references[10].

It is also possible to release *entire subsystems* of modules. The command *System.FreeImports* attempts to unload the specified modules and all their direct and indirect *imports*. It may be used for conventional module stacks with a single top module, for example a compiler. By contrast, the command *System.FreeClients* unloads the specified modules and all their direct and indirect *client*s. This variant may be used for module stacks written in the object-oriented programming style with their typically inverted module hierarchy, for example a graphics editor.

---

[8] *Modules removed only from the list of loaded modules, but not from memory, are marked with an asterisk in the output of the command System.ShowModules. Commands of such "hidden" modules can be accessed by either specifying their module number or their (modified) module name, both of which are displayed by the command System.ShowModules. In both cases, the corresponding command text must be enclosed in double quotes. If a module M carries module number 14, for instance, one can activate a command M.P also by clicking on the text "14.P". Typical use cases include hidden modules that still have background tasks installed which can only be removed by activating a command of the hidden modules themselves, or hidden modules that still have open viewers. If the command to close a viewer is displayed in the viewer's menu bar, the user can manually edit the command text (by clicking within its bottom two pixel lines), replace the module name by its module number and enclose the modified command text in double quotes. Although this is somewhat clumsy, it at least enables the user to close the viewer. An alternative approach is to provide a "Close" command that also accepts the marked viewer as argument (using procedure Oberon.MarkedViewer).*

[9] *In FPGA Oberon 2013 on RISC and in Experimental Oberon, the module block includes the module's type descriptors. In some other Oberon implementations, such as Oberon on Ceres, type descriptors are not stored in the module block, but are dynamically allocated in the heap at module load time, in order to persist them beyond the lifetime of their associated modules. In Experimental Oberon, no such special precaution is necessary, as module blocks are removed only from the list of loaded modules and not from memory, if they are still referenced by other modules. Thus, type descriptors can safely be stored in the (static) module blocks.*

[10] *In Oberon, cyclic references can be created by pointers or procedure variables, whereas cyclic module imports are normally not allowed. However, through a tricky sequence of compilation and editing steps, it is in fact possible to c o n s t r u c t cyclic module imports, which cannot be detected by the compiler. But even though they can be created, such modules are not allowed to be loaded, as the module loader of Experimental Oberon – adopting the approach chosen in FPGA Oberon – would simply enter an endless recursion, eventually leading to a stack overflow or out-of-memory condition – a totally acceptable solution for such an artificially constructed case. But even if modules with cyclic module imports w e r e allowed to be loaded, Experimental Oberon would handle them correctly upon unloading, i.e. if no external clients or references exist, such a module group would simply be unloaded as a whole – as it should.*

Note that these unloading strategies amount to *heuristics* and may tend to unload rather large parts of the system, which may not be desired. Thus, the recommended way to unload modules is to use the command *System.Free* with a *specific* set of modules provided as parameters.

For added convenience, the tool commands *System.ShowRefs* and *System.ShowExternalRefs* can be used to identify all modules containing references to a given module or module group.

The mechanism of *safe module unloading* is described in more detail in a separate document[11].

## 4. Oberon system building and maintenance tools

A minimal version of the Oberon system *building tools*, as described 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.

The boot linker (procedure *Boot.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[12] of a disk or the *build-up* boot file sent over a data link to a target system. The name of the top module is supplied as a parameter. For the *regular* boot file, this is typically module *Modules*, the top module of the *inner core* of the Oberon system. For the *build-up* boot file, it is usually module *Oberon0*, a command interpreter mainly used for system building purposes. The boot linker automatically includes all modules that are directly or indirectly imported by the specified top module. It is almost identical to the *regular* module loader (procedure *Modules.Load*), except that it outputs the result in the form of a file on disk instead of depositing the object code of the linked modules in newly allocated module blocks in memory.

The command *Boot.Load* loads a valid *boot file*, as generated by the command *Boot.Link*, onto the boot area of the local disk, one of the two valid *boot sources* (the other is the data link). This command can be used if the user already has a running Oberon system. It is executed *on* the system to be modified and overwrites the boot area of the *running* system.

The tools to build an entirely new Oberon system on a bare metal *target* system connected to a *host* system via a data link (e.g., an RS-232 serial line) are provided by the module pair *ORC* (for Oberon to RISC Connection) running on the host system and *Oberon0* running on the target system. When using Oberon in an emulator, one can simulate the process of booting the target system over a data link by starting *two* emulator instances connected via two Unix-style *pipes*, one for each direction.

The command *ORC.Load Oberon0.bin* loads a valid Oberon *build-up* boot file, as generated by the command *Boot.Link Oberon0*, over the data link to the target system and starts it there. It must be the first *ORC* command to be run on the host system after the target system has been restarted over the serial line, as *its* boot loader is waiting for a valid boot file to be sent to it. The command automatically performs the conversion of the input file to the stream format used for booting over a data link (i.e. a format accepted by procedure *BootLoad.LoadFromLine*).

Once the command interpreter *Oberon0* is running on the target system, a new Oberon system can be built *there* by executing *ORC* commands on the host. The two commands *ORC.Send* and *ORC.Receive* can be used to transfer files between the host and the target system, while the command *ORC.SR* (“send, then receive sequence of items”) remotely initiates a command

[11] *http://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/Safe-module-unloading-in-FPGA-Oberon.pdf*
[12] *Sectors 2-63 in FPGA Oberon 2013 and Experimental Oberon (by default)*

on the target system, then receives the command's response (if any). A command is specified by an integer followed by parameters, which can be numbers, names, strings or characters.

Building a new Oberon system proceeds in multiple steps. First, the command *ORC.SR 101* is called to clear the root page of the target system's file directory. Second, the files required to start the Oberon system on the target system (module *System* and its imports, the default font file, the *regular* boot file to be loaded onto the boot area of the target system, a tool to remotely initiate the *regular* boot process on the target system and optionally some additional files) are transferred using the command *ORC.Send*. Third, the command *ORC.SR 100 Modules.bin* is invoked to load the just transferred *regular* boot file *Modules.bin* onto the boot area of the target system. Finally, the newly built Oberon system on the target system is *started* – either *manually* by setting the corresponding switch *on* the target system to "disk" and pressing the reset button, or *remotely* by executing the command *ORC.SR 102 BootLoadDisk.rsc* on the host 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. These commands are listed in chapter 14.2 of the book *Project Oberon 2013 Edition*.

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) – although there generally is no need to do so. Such standalone programs can be created with the regular Oberon compiler. Once compiled, the tool command *Boot.WriteFile* can be used to extract the code section from the boot loader's object file and to convert it to a PROM file compatible with the specific hardware used[13]. Transferring the boot loader to the permanent read-only store of the target hardware typically requires the use of proprietary (or third-party) tools.

The Oberon system building tools are described in more detail in a separate document[14].

## 6. Enhanced Oberon-07 compiler

The official FPGA Oberon-07 compiler[15] has been enhanced with various new features, including

- Type-bound procedures (Oberon-2 style)[16]
- Dynamic heap allocation procedure for fixed-length and open arrays (Oberon-2 style)
- Numeric case statement
- Exporting and importing of string constants
- Forward references and forward declarations of procedures
- No access to intermediate objects within nested scopes
- Module contexts

The enhanced FPGA Oberon compiler is described in more detail in a separate document[17].

[13] *A Xilinx field-programmable gate array (FPGA) contained on the development board Spartan in the case of FPGA Oberon 2013*
[14] *https://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/The-Oberon-system-building-tools.pdf*
[15] *http://www.projectoberon.com*
[16] *Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991*
[17] *https://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/Enhanced-FPGA-Oberon07-compiler.pdf*