The Experimental Oberon System

Andreas Pirklbauer

12.12.1990 / 3.4.2017

Experimental Oberon[1] is a revision of the Original Oberon[2] operating system. It contains a number of enhancements including continuous fractional line scrolling with variable line spaces, multiple logical displays, enhanced viewer management, safe module unloading and a minimal version of the Oberon system building tools. 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 has been added to the Oberon viewer system, enabling 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]. 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 the Oberon system is *considerably* reduced.

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

The Original 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. The extended conceptual hierarchy of the display system consists of the triplet *(display, track, viewer)*. Consequently, the base module *Viewers* exports routines to add and remove *displays*, to open and close *tracks* within existing 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 display, *System.CloseDisplay* closes an existing one, *System.ShowDisplays* lists all open displays, *System.ThisDisplay* shows the display *id* and *name* of the current display, *System.SetDisplay* switches between displays, and *System.SetDisplayName* assigns a new name to an existing display.

---

[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.*

The command *System.Clone*, displayed in the title bar of every menu viewer, opens a new display on the fly *and* displays a copy of the initiating viewer *there*. The user can toggle between the two copies of the viewers (i.e. switch *displays*) with a single mouse click[4].

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. Enhanced viewer management

The core viewer operations *Viewers.Change, MenuViewers.Change, MenuViewers.Modify* and *TextFrames.Modify* have been generalized to handle *arbitrary* viewer modifications, including pure vertical translations (without changing the viewer's height), adjusting the bottom line, the top line and the height of a viewer with a single *viewer change* operation, and dragging multiple viewers around with a single *mouse drag* operation.

A number of basic viewer message types (e.g., *ModifyMsg*) and message identifiers (e.g., *extend, reduce)* have been eliminated from the Oberon system, further streamlining the overall type hierarchy. 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.

In addition, message types that appeared to be generic enough to be made 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 *TextFrames* no longer depends on module *MenuViewers*, making it now possible to embed text frames into *other* types of frames or composite viewers, for example a viewer consisting of an *arbitrary* number of text, graphic or picture frames.

It is perhaps worth mentioning that as a result of these changes, the code for implementing certain viewer operations, such as in module *MenuViewers*, has actually become *simpler*.

4. Safe module unloading

The semantics of *unloading* a module or module group has been refined as follows. If clients exist among other loaded modules, a module or module group is never unloaded. If no clients *and* no references to the module or module group exist in the remaining modules and data structures, it is unloaded and its associated memory is released. If no clients, but references exist, the module or module group is initially removed only from the *list* of loaded modules, without releasing its associated memory[5]. Such *hidden* modules are later *physically* removed from *memory* as soon as there are no more references to them from other loaded modules. To achieve this automatic removal of obsolete module data, a new command *Modules.Collect* (which checks all possible combinations of module groups) has been added to the Oberon background task handling garbage collection[6]. Thus, module data is kept in memory as long as needed and is removed from it as soon as possible.

---

[4] *By comparison, the Original Oberon commands System.Copy and System.Grow create a copy of the original viewer in the s a m e 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, lifting the viewer to an "overlay" in the third dimension).*
[5] *Being removed from the module list means that another module with the same name can be (re-)loaded again. Thus, removing a module from the module list effectively amounts to renaming it (to an anonymous name). Modules removed only from the module list, but not from memory, are marked with an asterisk in the output of command System.ShowModules.*
[6] *The command Modules.Collect can also be manually activated at any time. Alternatively, the user can invoke the command System.Collect, which includes a call to Modules.Collect.*

References to a loaded module can be either in the form of *type tags* (addresses of type descriptors) in *dynamic* objects of other modules (allocated via the predefined procedure *NEW* in a dynamic memory area called the *heap*) pointing to descriptors of types declared in the module, or in the form of *procedure variables* installed in *static or dynamic* objects of other modules referring to procedures declared in the module[7].

In sum, unloading a module does not affect *past* references. For example, older versions of a module's code can still be executed if they are referenced by procedure variables in other modules, even if a newer version of the module has been reloaded in the meantime[8]. Type descriptors also remain accessible to other modules for as long as needed[9].

If a module *group*, say *M1*, *M2*, *M3*, is to be unloaded and there exist references *only* within this group, the module group can still be unloaded as a whole by using the new command *System.FreeGroup M1 M2 M3*. The command *System.Free M1 M2 M3* continues to unload the specified modules *individually* (i.e. one by one).

In addition to offering the ability to unload module *groups* as a whole, Experimental Oberon provides access to the commands of *hidden* modules by accepting module *numbers*[10] and the *names* of *hidden* modules when activating a user command. In both cases, the selected command must be surrounded by double quotes. If a module *M* carries module number *14*, for instance, one can activate the command *M.P* also by clicking on the text *"14.P"*. Typical usage scenarios include *hidden* modules that still have Oberon 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[11].

Dynamic references to a module or module group are checked using a simple *mark-scan* scheme. During the *mark* phase, dynamic objects reachable by *all other* loaded modules are marked, thereby excluding records reachable *only* by the specified module or module group itself. This ensures that when a module or module group is referenced only by itself, it can still be unloaded. The subsequent *scan* phase scans the heap element by element, unmarks marked objects and checks whether the *type tags* of the encountered (marked) records point to descriptors of types declared in the module or module group to be unloaded, or whether *procedure variables* in these records refer to procedures declared in those module(s). The latter check is then also performed for all *static* procedure variables.

In order to make such a validation pass possible, type descriptors for *dynamic* records[12] 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 the Oberon

---

[7] *An Oberon module can be viewed as a container of types, variables and procedures, where variables can be procedure-typed. Variables can be declared as global variables (allocated in the module area when a module is loaded), as local variables (allocated on the stack when a procedure is called) or allocated as dynamic objects in the heap (via the predefined procedure NEW). Thus, in general there can be type, variable or procedure references from static or dynamic objects of other loaded modules to static or dynamic objects of the module(s) to be unloaded. However, only dynamic type and static and dynamic procedure references need to be checked. Static type and variable references from other loaded modules referring (by name) to types or variables declared in the specified module(s) don't need to be checked, as these are already handled via their import/export relationship (if clients exist, a module or module group is never unloaded). Pointer references from global or dynamic pointer variables of other loaded modules to dynamic objects (records allocated in the heap) of the specified module(s) don't need to be checked either, as such references are handled by the garbage collector. Pointer 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).*

[8] *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.*

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

[10] *The command System.ShowModules has been extended to also display module numbers. Module numbers are assigned to each module by the module loader.*

[11] *If the command to close an open viewer of a hidden module is displayed in the menu bar of the viewer itself, the user can manually edit the text in the menu bar (by clicking within its bottom two pixel lines), replace the module name by the corresponding module number (as displayed by the command System.ShowModules) and surround the command with double quotes. Although this is somewhat clumsy, it at least allows the user to close the viewer. Alternatively, the module can provide a Close command that also accepts the marked viewer as argument (Oberon.MarkedViewer), in which case the command (with the module number instead of the module name) can be activated from anywhere in the system.*

[12] *See chapter 8.2, page 109, of the book Project Oberon (2013 Edition) for a detailed description of the fields of an Oberon type descriptor.*

system (MacOberon)[13], whose run-time representation of a dynamic heap record and its associated type descriptor is shown in Figure 1.

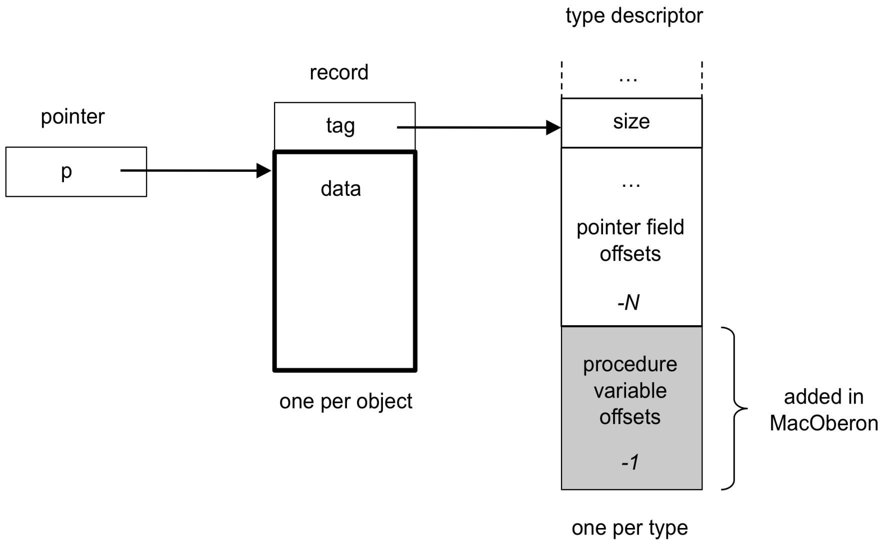

Figure 1: Run-time representation of a dynamic record and its type descriptor in MacOberon

To avoid skipping over the list of *pointer field offsets* in the type descriptor for *each* record encountered in the heap during the *scan* phase[14], Experimental Oberon uses a slightly different run-time representation of type descriptors, where the *procedure variable offsets* are *prepended* (rather than *appended*) to the existing fields of each descriptor, i.e. they are allocated in the *opposite* direction of the fields holding the *pointer variable offsets* needed for the garbage collector, as shown in Figure 2[15].

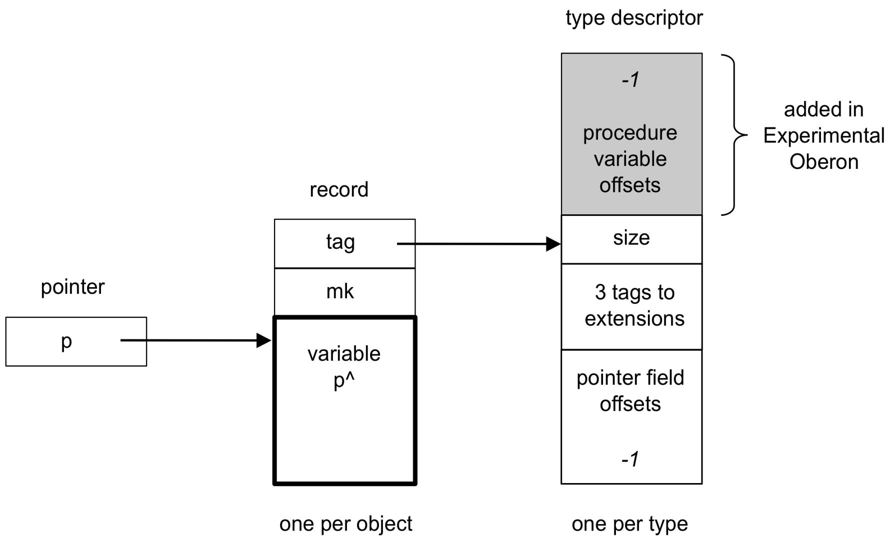

Figure 2: Run-time representation of a dynamic record and its type descriptor in Experimental Oberon

---

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

[14] *This could also be avoided in other ways (e.g., by storing the number of pointer field offsets at a fixed location inside the type descriptor), but we opted for a simpler solution.*

[15] *However, this runtime representation may get into conflict with some implementations of the Oberon-2 programming language, which typically also prepend additional run-time information to the fields in each type descriptor (namely a "method table" associated with an Oberon-2 type descriptor, which however serves a different purpose than the list of procedure variable offsets used in Experimental Oberon). Thus, if the Oberon-2 language needs to be supported in Experimental Oberon, procedure variable offsets can also be appended to the existing fields of each type descriptor (as in MacOberon). This was, in fact, realized in an earlier version of Experimental Oberon. Reverting back to that version would require only small changes to the compiler (procedure ORG.BuildTD) and to the scan phase of reference checking (procedure Kernel.Scan). Alternatively, the Oberon-2 implementation could be adapted such that the method table is allocated somewhere else (e.g., in the heap, as in Object Oberon).*

The compiler generating these modified type descriptors for dynamic records (procedure *ORG.BuildTD*) and the descriptors for global module data (procedure *ORG.Close*), the format of the Oberon object file containing them and the module loader transferring them from object file into memory (procedure *Modules.Load*) have been adjusted accordingly.

The following code excerpt shows a possible realization of the outlined validation pass for static and dynamic references to a module *group* (procedure *Modules.Check*)[16]. Before invoking it, one must *select* the modules to be checked against using *Modules.Select*.

```
PROCEDURE Check*(VAR res: INTEGER); (*references to selected modules*)
  VAR mod: Module; pref, pvadr, r: LONGINT; res0, res1: INTEGER; continue: BOOLEAN;
BEGIN mod := root;
  WHILE mod # NIL DO (*mark dynamic records reachable by all other loaded modules*)
    IF ~mod.selected & (mod.name[0] # 0X) THEN Kernel.Mark(mod.ptr) END ;
    mod := mod.next
  END ;
  Kernel.Scan(chksel, chksel, res0, res1); (*check dynamic references*)
  IF res0 > 0 THEN res := 1 ELSIF res1 > 0 THEN res := 2
  ELSE res := 0; mod := root; continue := TRUE;
    WHILE continue & (mod # NIL) DO
      IF ~mod.selected & (mod.name[0] # 0X) THEN
        pref := mod.pvar; pvadr := Mem[pref];
        WHILE continue & (pvadr # 0) DO r := Mem[pvadr];
          IF chksel(r, continue) > 0 THEN res := 3 END ; (*check static procedure references*)
          INC(pref, 4); pvadr := Mem[pref]
        END
      END ;
      mod := mod.next
    END
  END
END Check;
```

where procedure *Kernel.Scan* implements a generic heap *scan* algorithm[17]:

```
PROCEDURE Scan*(type, proc: Handler; VAR res0, res1: INTEGER); (*res0, res1 = sum of handler results*)
  VAR p, r, mark, tag, size, offadr, offset: LONGINT; continue: BOOLEAN;
BEGIN p := heapOrg; res0 := 0; res1 := 0; continue := TRUE;
  REPEAT mark := Mem[p+4];
    IF mark < 0 THEN (*free*) size := Mem[p]
    ELSE (*allocated*) tag := Mem[p]; size := Mem[tag];
      IF mark > 0 THEN Mem[p+4] := 0; (*unmark*)
        IF continue THEN
          IF type # NIL THEN INC(res0, type(tag, continue)) END ; (*call type for type tag*)
          IF continue & (proc # NIL) THEN offadr := tag - 4; offset := Mem[offadr];
            WHILE continue & (offset # -1) DO r := Mem[p+8+offset];
              INC(res1, proc(r, continue)); (*call proc for each procedure variable*)
              DEC(offadr, 4); offset := Mem[offadr]
            END
          END
        END
      END
    END ;
    INC(p, size)
  UNTIL p >= heapLim
END Scan;
```

[16] *Mem stands for the entire memory and assignments involving Mem are expressed as SYSTEM.GET(a, x) for x := Mem[a] and SYSTEM.PUT(a, x) for Mem[a] := x.*
[17] *The original procedure Kernel.Scan (implementing the scan phase of the garbage collector) has been renamed to Kernel.Collect, in analogy to procedure Modules.Collect.*

Procedure *Kernel.Scan* scans the heap element by element, unmarks marked elements and calls the two parametric handler procedures *type* and *proc* for (individual elements of) each encountered marked record. Procedure *type* is called with the *type tag* of the record as argument, while procedure *proc* is called for each procedure variable declared in the record with the *procedure variable* itself as argument. Procedure *Modules.Check* uses this generic heap scan mechanism to pass a (private) procedure *chksel*, which checks whether the argument supplied by *Kernel.Scan* (a type tag or a procedure variable) refers to *any* of the selected modules. Results are separately added up for the two handler procedures. An additional parameter *continue* allows to stop calling (both of) them.

Note that procedure *Modules.Check* is not only called when a module is unloaded by the user (via the commands *Modules.Free* or *Modules.FreeGroup*), but also by the background process that automatically removes no longer referenced *hidden* module data from memory (procedure *Modules.Collect* which in turn calls procedure *Modules.Check*). Thus, it must be implemented such that it can handle *both* visible and hidden modules.

Although the operation of *reference checking*, as sketched above, may appear expensive at first, in practice, there is no performance issue. It is comparable in complexity to garbage collection and thus barely noticeable. In addition, module unloading is usually rare (except, perhaps, during development, where however the number of references tends to be small), while modules that manage data structures shared by client modules, such as a viewer manager, are often never unloaded at all. Thus, the presented solution, which was mainly chosen for its simplicity, appears to be amply sufficient for most practical purposes.

Its main shortcoming then is that it requires *additional* run-time information to be present in type descriptors *solely* for the purpose of reference checking, which in turn is *only* needed in the relatively infrequent case of module releases[18].

An additional downside is that when a module cannot be removed from memory due to pre-existing references, the user usually does not know why the removal has failed. However, since in this case the module is removed only from the *list* of loaded modules and not from *memory*, we don't consider this issue as serious. In addition, this shortcoming can be easily alleviated by displaying the *names* of the modules containing the references[19].

5. System building tools

A minimal version of the Oberon system *building tools* has been added, consisting of the two modules *Linker*[20] and *Builder*. They provide the necessary mechanisms and tools to establish the prerequisites for the *regular* Oberon startup process[21].

---

[18] *Adding procedure variable offsets to type descriptors of dynamic records is, strictly speaking, not necessary, as the compiler could always rearrange the memory layout of a record such that all procedure variable fields are placed in a contiguous section at the beginning of it, thereby making their offsets implicit. Even the number of procedure variable fields could be encoded in one of the existing fields of its associated type descriptor, for example in the sentinel at the end of the pointer offset section. Thus, it is, in principle, possible to realize the reference checking scheme without any additional memory requirements in type descriptors. However, we refrain from implementing this additional refinement for two main reasons. First, it would increase the complexity of the compiler. At first glance, it might seem that in order to rearrange the field list of a given record, only a small change needs to be made to a single procedure in the compiler (procedure ORP.RecordType). However, a complication arises because declared records may recursively contain other records, each again containing procedure variables at arbitrary offsets. While it is always possible to "flatten" such recursive record structures, it would make other record operations more complex. For example, assignments to subrecords would become less natural, because their fields would no longer be placed in a contiguous section in memory. Thus, we would no longer obtain a straightforward, uniform implementation covering both dynamic records and global module data. And second, the memory savings in the module areas holding type descriptors would be rather marginal, given the fact that there is only one descriptor per type rather than one per allocated object. In addition, we believe that most applications should be programmed in the conventional programming style. Therefore, installed procedures should be rare, while in the few places where they do exist, there are typically only a few of them. For example, in the Oberon system, the object-oriented programming style is restricted to the viewer system, which provides distributed control in the form of installed handler procedures – of which there is typically only a single one per type of viewer. On balance, the benefit obtained by saving a few fields in a small number of type descriptors appears negligible and therefore the additional complexity required to implement it would be hard to justify. Indeed, in the entire Oberon system the total memory savings would amount only to a few dozen bytes.*

[19] *In the case of dynamic records, one can display the name of the module declaring its t y p e, which is not necessarily the module rooting the data structure holding the record. One could however modify the reference checking algorithm such that each module is checked individually for dynamic references, in which case also the data structure roots are known.*

[20] *The linker has been included from a different source (https://github.com/charlesap/io). It was adapted for Experimental Oberon's object file format (which includes a modified module descriptor, modified type descriptors for dynamically allocated records and an additional 'pvar' section in the module block holding global procedure variable references).*

[21] *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).*

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 boot firmware is called the *boot loader*, as its primary task is to *load* a valid *boot file* (a pre-linked binary containing a set of compiled Oberon modules) from a valid *boot source* into memory and then transfer control to its *top* module (the module that directly or indirectly imports all other modules in the boot file). Then its job is done until the next time the computer is restarted or the reset button is pressed. In general, there is no need to modify the boot loader (*BootLoad.Mod*). If one really has to, one typically has to resort to proprietary tools to load the boot loader onto the specific hardware platform used.

There are currently two valid boot sources in Oberon: a local disk, realized using a Secure Digital (SD) card in Oberon 2013, and a communication channel, realized using 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.

The command *Linker.Link* links a set of Oberon object files together and generates a valid Oberon boot file from them. The linker is almost identical to the regular module loader (procedure *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.Load* loads a valid Oberon boot file, as generated by the command *Linker.Link*, onto the *boot area* (sectors 2-63 in Oberon 2013) of the local disk – one of the two valid Oberon boot sources.

Note that the format of the Oberon boot file is *defined* to exactly mirror the standard Oberon storage layout[22]. In particular, location 0 in the boot file (and later in memory once it has been loaded) contains a branch instruction to the initialization sequence of the top module of the boot file. Thus, the boot loader can simply transfer the boot file byte for byte into memory and then branch to location 0 – which is precisely what it does[23].

In sum, to generate a new regular Oberon boot file and load it onto the local disk's boot area, one can simply execute the following commands *on* the system that 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.Load Modules ~                                       ... load boot file onto the disk's boot area
```

Note that the last command overwrites the disk's boot area of the *running* system. A backup of the disk is therefore recommended before experimenting with new *inner cores*[24].

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 Oberon boot loader merely *transfers* the boot file byte for byte from a valid boot source into memory, but does not call the module initialization sequences of the

[22] *See chapter 8.1, page 103, of the book Project Oberon (2013 Edition) for a detailed description of Oberon's storage layout.*
[23] *After transferring the boot file, the boot loader also deposits some additional key data in fixed memory locations, to allow proper continuation of the boot process.*
[24] *When using an Oberon emulator (https://github.com/pdewacht/oberon-risc-emu) on a host system, one can simply make a copy of the directory containing the disk image.*

transferred modules[25] (this is, in fact, why the *inner core* modules *Kernel*, *FileDir* and *Files* don't *have* module 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 initialization bodies of the just loaded modules from their module descriptors now present in memory and simply call them, as shown in procedure *InitMod*[26] 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);
    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*.

```
MODULE Modules;                 ... old top module of the inner core, now just a regular module
  IMPORT SYSTEM, Files;
  ...
BEGIN Init                      ... 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
  …
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.
```

To build a modified *inner core* for this new module configuration and load it onto the disk's *boot area*, one can execute the following commands:

```
ORP.Compile Kernel.Mod FileDir.Mod Files.Mod Modules.Mod ~
ORP.Compile Input.Mod Display.Mod Viewers.Mod ~
ORP.Compile Fonts.Mod Texts.Mod Oberon.Mod ~    ... compile the modules of the modified inner core
Linker.Link Oberon ~                            ... create a new regular boot file (Oberon.bin)
Builder.Load Oberon ~                           ... load the boot file onto the local disk's boot area
```

[25] *Except for the top module, whose module initialization body is however executed a f t e r the boot file has been transferred into memory in its entirety.*
[26] *Procedure InitMod could be placed in modules Oberon or Modules (note: the data structure rooted in the global variable Modules.root is transferred as part of the boot file).*

We note that this module configuration reduces the number of stages in the regular Oberon *boot process* from 3 to 2, thereby streamlining 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, for example by including module *System* and all its imports. This in turn would have the (small) disadvantage that the viewer complex and the system tools are now “locked” into the *outer core*. However, an Oberon system without a viewer manager hardly makes sense, even in closed server environments. As an advantage we note that an *outer core* that also includes module *System* can more easily be replaced on the fly, namely by unloading some or all of its modules as a group and then reloading module *System*.