The Experimental Oberon System

Andreas Pirklbauer

22.11.2017

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 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 side effect, the initial learning curve for users new to the system 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*, 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. 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.

The command *System.OpenDisplay* opens a new logical 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*
[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 scrolling modes 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 logical display on the fly *and* displays a copy of the initiating viewer *there*. The user can easily toggle between the two copies of the viewers (i.e. switch *displays*) with a single mouse click[4].

Alternatively, the user can activate the command *System.Expand*, also displayed in the title bar of every menu viewer, to expand the 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 clicking on *System.Expand* again in any of their (still visible) title bars.

3. Enhanced viewer management

The basic viewer operations *Change* and *Modify* have been generalized to include pure vertical translations (without changing the viewer's height), adjusting the top bottom line, the bottom line and the height of a viewer using a single *viewer change* operation, and dragging multiple viewers around with a single *mouse drag* operation.

Several viewer message types (e.g., *ModifyMsg*) and message identifiers (e.g., *extend, reduce)* have been eliminated, further streamlining the overall type hierarchy. The remaining 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, a number of viewer message types 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 the base module *Viewers*, resulting in fewer module dependencies. Most notably, module *TextFrames* no longer depends on module *MenuViewers*, making it now possible to recursively 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.

4. Safe module unloading

The semantics of module *unloading* has been refined as follows. If clients exist, a module or module group is never unloaded. If no clients *and* no 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 references exist, the user has the *option* to *initially* remove the module or module group 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[6]. To achieve this automatic removal of no longer referenced module data, a new command *Modules.Collect* has been added to the Oberon background task handling garbage collection[7]. It checks *all* possible combinations of module *subgroups* among the *hidden* modules for outside clients and references[8].

---

[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 (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] Removing a module from the list of loaded modules amounts to renaming it, allowing another module with the same name to be (re-)loaded again. Modules removed o n l y 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 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. 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). In this case, the command (with the module number used instead of the module name) can be activated from anywhere in the system, after first designating the viewer to be closed as operand by placing the Oberon "star" marker in it.

[6] Removing a module from memory frees up the memory area previously occupied by the module block. In Original Oberon 2013 on RISC and in Experimental Oberon, this includes the module's type descriptors. In some other Oberon implementations, such as Original Oberon on Ceres, type descriptors are not stored in the module block, but allocated dynamically 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, because module blocks are removed only from the l i s t of loaded modules, if they are still referenced by other modules. Thus, type descriptors can safely be stored in the (static) module blocks.

[7] The command Modules.Collect can also be manually activated at any time. Alternatively, one can invoke the command System.Collect which includes a call to Modules.Collect.

[8] The combinatorial scheme used is similar to Algorithm 4.3 in chapter 4.3 of Ruskey, Frank, "Combinatorial Generation" (version of Oct 1, 2003) at www.1stworks.com/ref/ruskeycombgen.pdf

Thus, 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. 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. Type descriptors also remain accessible to other modules for as long as needed[9].

If a module *group* is to be unloaded and there exist clients or references *only* within this group, the group is unloaded *as a whole*. Both *regular* and *cyclic* module imports and references *within* a module group are allowed and will *not* prevent the unloading of that group[10].

There are two *base* variants of the module *unload* command. The command *System.Unload* differs from the default command *System.Free* only in that it does *not* hide the specified modules from the list of loaded modules, if references to them exist in the remaining modules. This gives the user the *option* to *select* the desired behavior in that case.

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*[11]. 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. The additional subvariants *System.FreeRemovableImports* and *System.FreeRemovableClients* unload the largest *subset* of modules that has *no* outside clients. The corresponding variations of the command *System.Unload* are analogous.

Note that these unloading strategies amount to *heuristics* and may tend to unload rather large parts of the system, which may not be desired. Therefore, the recommended way to unload modules is to use the *base* commands *System.Free* and *System.Unload* with a *specific* set of individual modules provided as parameters. To help identify valid subsets of modules that can be unloaded from the system, the tool commands *System.ShowImports, System.ShowClients, System.ShowRemovableImports* and *System.ShowRemovableClients* are provided.

When the user attempts to unload a module or module group, clients are checked first. If clients exist among the other modules, no further action is taken and the unload command exits. If no outside clients exist, references are checked next. References that *need* to be checked prior to module unloading include *type tags* (addresses of type descriptors) in *dynamic* objects in the heap reachable by all other loaded modules, which point to descriptors of types declared in the modules to be unloaded, and *procedure variables* installed in *static or dynamic* objects of other modules referring to *static* procedures declared in the modules to be unloaded[12].

---

[9] 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.

[10] In Oberon, cyclic references (within the same module or among different modules) can be created by pointers or procedure variables, while 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 into memory, as the module loader of Experimental Oberon – adopting the approach chosen in Original Oberon on Ceres and on RISC – would simply enter an endless recursion, eventually leading to a stack overflow or out-of-memory condition – a totally acceptable solution for such an artificial case. But even if modules with cyclic module imports w e r e allowed to be loaded by the module loader, Experimental Oberon would handle them correctly upon unloading, i.e. if no outside clients or references exist, such a module group would be unloaded as a whole – as it should.

[11] The Oberon system itself (module System and all its direct and indirect imports) is of course excluded from the list of modules to be unloaded.

[12] An Oberon module can be viewed as a container of types, variables and procedures, where variables can be procedure-typed. Types can be statically declared as g l o b a l types (in which case they can be exported and therefore be referenced by name in other modules) or as types l o c a l to a procedure (in which case they cannot be exported). Variables can be declared as g l o b a l variables (allocated in the module area when a module is loaded) or as l o c a l variables (allocated on the stack when a procedure is called), or allocated in a dynamic space called the h e a p via the predefined procedure NEW. Thus, in general there can be type, variable or procedure references from static or dynamic objects of other modules to static or dynamic objects of the specified modules to be unloaded. However, only d y n a m i c t y p e and s t a t i c and d y n a m i c p r o c e d u r e references need to be checked during module unloading, for the following reasons. First, s t a t i c type and variable references from other loaded modules can only refer by n a m e to types or variables 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 again. Second, d y n a m i c variable references from global or dynamic p o i n t e r variables of other modules to d y n a m i c objects reachable by the modules to be unloaded don't need to be checked either, as such references are allowed to exist, i.e. they shouldn't prevent module unloading. Such references will be handled by the garbage collector during a future garbage collection cycle: heap records reachable by the unloaded modules a n d other (still loaded) modules will not be collected, while heap records that w e r e reachable o n l y by the just unloaded modules w i l l be collected – as they should. Thus, the handling of dynamic pointer references is entirely delegated to the garbage collector. Finally, p o i n t e r variable references to s t a t i c objects (declared as global variables) are only possible by resorting to low-level facilities and should be avoided (and, in fact, be disallowed).

To check clients and references, the module *unload* command first *selects* the modules to be unloaded using procedure *Modules.Select*, and then calls procedure *Modules.Check*[13].

```
    PROCEDURE Check*(VAR res: INTEGER); (*whether clients or references to selected modules exist*)
 1   VAR mod, imp, m: Module; continue: BOOLEAN;
 2     pref, pvadr, r: LONGINT; p, q, impcnt, resType, resProc: INTEGER;
 3  BEGIN res := 0; mod := root;
 4   WHILE (mod # NIL) & (res = 0) DO
 5     IF (mod.name[0] # 0X) & mod.selected & (mod.refcnt > 0) THEN m := root; impcnt := 0;
 6       WHILE m # NIL DO (*count clients within selected modules*)
 7         IF (m.name[0] # 0X) & (m # mod) & m.selected THEN p := m.imp; q := m.cmd;
 8           WHILE p < q DO imp := Mem[p];
 9             IF imp = mod THEN (*m imports mod*) INC(impcnt); p := q ELSE INC(p, 4) END
10           END
11         END ;
12         m := m.next
13       END ;
14       IF mod.refcnt # impcnt THEN res := 1 END
15     END ;
16     mod := mod.next
17   END ;
18   IF res = 0 THEN mod := root;
19     WHILE mod # NIL DO (*mark dynamic records reachable by all other modules*)
20       IF (mod.name[0] # 0X) & ~mod.selected THEN Kernel.Mark(mod.ptr) END ;
21       mod := mod.next
22     END ;
23     Kernel.Scan(ChkSel, ChkSel, resType, resProc); (*check dynamic type and procedure references*)
24     IF resType > 0 THEN res := 2 ELSIF resProc > 0 THEN res := 3
25     ELSE mod := root; continue := TRUE;
26       WHILE continue & (mod # NIL) DO (*check static procedure references*)
27         IF (mod.name[0] # 0X) & ~mod.selected THEN
28           pref := mod.pvar; pvadr := Mem[pref];
29           WHILE continue & (pvadr # 0) DO r := Mem[pvadr];
30             IF ChkSel(r, continue) > 0 THEN res := 4 END ;
31             INC(pref, 4); pvadr := Mem[pref]
32           END
33         END ;
34         mod := mod.next
35       END
36     END
37   END
    END Check;
```

Clients are checked by verifying whether the number of clients of each module *within* the group of selected modules matches its *total* number of clients (lines 3-17). References from *dynamic* objects in the heap are checked using a conventional *mark-scan* scheme. During the *mark* phase (lines 18-22), heap records reachable by all *named* global pointer variables of *all other* loaded modules are marked (line 20), thereby excluding records reachable *only* by the specified modules themselves. This ensures that when a module or module group is referenced *only* by itself, it can still be unloaded. The subsequent *scan phase* (line 23), implemented as a separate procedure *Scan* in module *Kernel*[14], scans the heap sequentially, unmarks all *marked* records and checks whether the *type tags* of the marked records point to descriptors of types in the *selected* modules to be unloaded, and whether *procedure variables* declared in these records refer to global procedures declared in those same modules. The latter check is then also performed for all *static* procedure variables of all other loaded modules (lines 25-35).

[13] *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.*
[14] *The original procedure Kernel.Scan (implementing the scan phase of the Oberon garbage collector) has been renamed to Kernel.Collect, in analogy to Modules.Collect.*

In order to omit in module *Kernel* any reference to the module list rooted in module *Modules*, procedure *Kernel.Scan* is expressed as a *generic* heap scan scheme.

```
    PROCEDURE Scan*(type, proc: Handler; VAR resType, resProc: INTEGER);
 1   VAR p, r, mark, tag, size, offadr, offset: LONGINT; continue: BOOLEAN;
 2  BEGIN p := heapOrg; resType := 0; resProc := 0; continue := (type # NIL) OR (proc # NIL);
 3   REPEAT mark := Mem[p+4];
 4    IF mark < 0 THEN (*free*) size := Mem[p]
 5    ELSE (*allocated*) tag := Mem[p]; size := Mem[tag];
 6     IF mark > 0 THEN Mem[p+4] := 0; (*unmark*)
 7      IF continue THEN
 8       IF type # NIL THEN INC(resType, type(tag, continue)) END ; (*call type for type tag*)
 9       IF continue & (proc # NIL) THEN offadr := tag - 4; offset := Mem[offadr];
10        WHILE continue & (offset # -1) DO r := Mem[p+8+offset];
11         INC(resProc, proc(r, continue)); (*call proc for each procedure variable*)
12         DEC(offadr, 4); offset := Mem[offadr]
13        END
14       END
15      END
16     END
17    END ;
18    INC(p, size)
19   UNTIL p >= heapLim
    END Scan;
```

This scheme calls *parametric* handler procedures for individual elements of each *marked* heap record, instead of *directly* checking whether these records contain *type* or *procedure* references to the modules to be unloaded. Procedure *type* is called with the *type tag* of the heap record as argument (line 8), while procedure *proc* is called for each procedure variable declared in the same record with (the address of) the *procedure* itself as argument (line 11). The results of the handler calls are *separately* added up for each handler and returned in the variable parameters *resType* and *resProc*. An additional variable parameter *continue* allows the handler procedures to indicate to the caller that they are no longer to be called (lines 7, 9, 10)[15].

Procedure *Modules.Check* uses this generic *heap scan* scheme by passing its private handler procedure *ChkSel*, which merely checks whether the argument supplied by *Kernel.Scan* (either a type tag or a procedure variable) references *any* of the modules to be unloaded[16] and *stops* checking for references as soon as the *first* such reference is found, while indicating that fact to the caller. The scan process itself continues, but only to *unmark* the remaining marked records (line 6). We note that *Modules.Check* is not only called when a module is unloaded by the *user*, but also by the Oberon *background task* that removes no longer referenced *hidden* module data from memory. Thus, it must be able to handle *both* visible and hidden modules.

In order to make the outlined validation pass possible, type descriptors of *dynamic* objects[17] in the *heap* as well as the descriptors of *global* module data in *static* module blocks have been extended with a list of *procedure variable offsets*, adopting an approach employed in one of the earlier implementations of the Oberon system (MacOberon)[18], whose run-time representation of a dynamic record and its associated type descriptor is shown in Figure 1.

---

[15] *If o n e of the two handler procedures sets continue to FALSE, procedure Kernel.Scan will stop calling b o t h of them. This somewhat artificial restriction could be lifted if needed.*
[16] *This requires an extra iteration over all modules. On systems that offer metaprogramming facilities such as persistent type and procedure objects collected in libraries, additional meta-information may be present in the run-time representation of modules, such as the locations of procedures and type descriptors in modules. In such a case, a simpler solution may exist.*
[17] *See chapter 8.2, page 109, of the book Project Oberon 2013 Edition for a detailed description of an Oberon type descriptor. It contains certain information about dynamically allocated records that is shared by all allocated objects of the same record type (such as its size, information about type extensions and the offsets of all pointer fields within the record).*
[18] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*

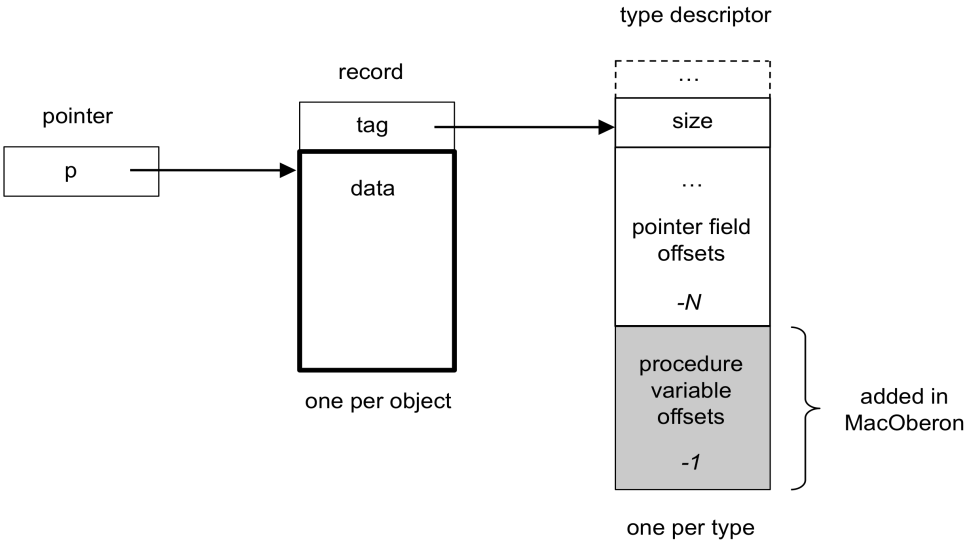

Figure 1: Run-time representation of a dynamic record and its type descriptor in MacOberon

Note, however, that the run-time layout chosen in MacOberon would require the *scan* phase of reference checking to traverse over the list of *pointer field offsets* in the type descriptor of *each* marked heap record. To avoid this extra step[19], Experimental Oberon uses a slightly different run-time layout, where *procedure variable offsets* are *prepended* (rather than appended) to the existing fields of each type descriptor, as shown in Figure 2[20].

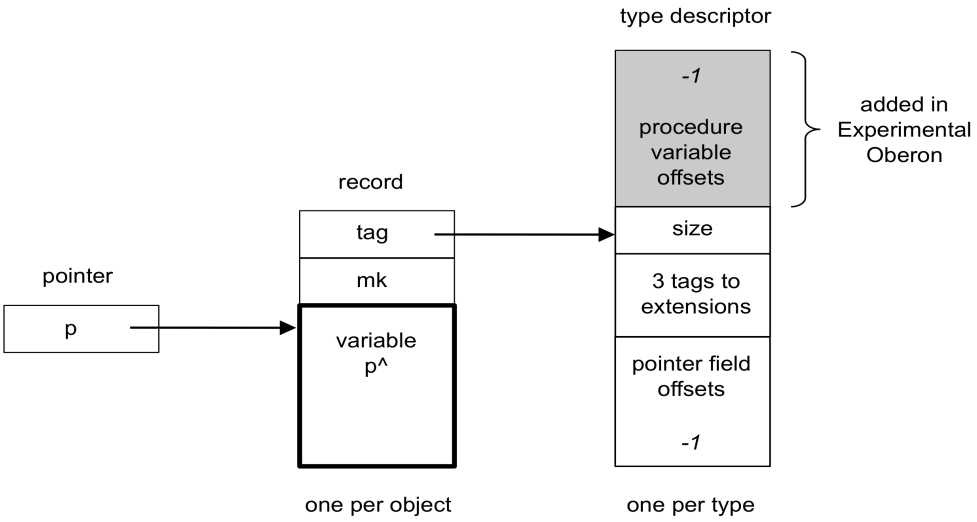

Figure 2: Run-time representation of a dynamic record and its type descriptor in Experimental Oberon

The compiler generating the modified type descriptors of dynamic objects and the descriptors of global module data, the format of the object file containing them and the module loader transferring them from object file into memory have been adjusted accordingly.

An obvious shortcoming of the reference checking scheme presented above is that it requires *additional* run-time information to be present in *all* type descriptors of *all* modules *solely* for the

[19] *The traversal 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.*
[20] *This runtime representation might come 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 completely different purpose than the list of procedure variable offsets used for reference checking in Experimental Oberon). Thus, if Oberon-2 is to be supported in Experimental Oberon, procedure variable offsets can of course also be appended to the existing fields of each type descriptor – implementing this would require only a one-line change to the compiler (procedure ORG.BuildTD) and the scan phase of reference checking (procedure Kernel.Scan). Alternatively, one could also adapt the Oberon-2 implementation, such that the method table is allocated somewhere else (e.g., in the heap at module load time).*

purpose of reference checking, which in turn is *only* needed in the relatively infrequent case of module releases. However, since there exists only one type descriptor per declared record *type* rather than one per allocated heap *record*, the additional memory requirements are negligible[21]. An additional downside is that when a module cannot be removed from memory due to existing references, the user usually does not know why the removal has failed. However, since in this case the user has the option to remove the module from the module *list*, we don't consider this issue as serious. In addition, this shortcoming can easily be remedied by providing a tool command that displays the *names* of the modules containing the offending references[22].

Although the operation of *reference checking* may appear expensive at first, in practice there is no performance issue. It is comparable in complexity to garbage collection and thus barely noticeable – at least on systems with small to medium sized dynamic spaces. This is in spite of the fact that for *each* heap record encountered during the initial *mark* phase *all* modules to be unloaded are checked for references to them during the subsequent *scan* phase. However, reference checking *stops* when the *first* reference is detected. Furthermore, module unloading is usually rare except, perhaps, during development, where however the *number* of references to a module tends to be small. We also note that modules that manage dynamic 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.

As an alternative to the outlined *conventional* mark-scan scheme – where the *mark* phase first marks *all* heap objects reachable by other loaded modules and the *scan* phase then checks for references from *marked* records to the modules to be unloaded – one might be tempted to check for references directly *during* the *mark* phase and simply *stop* marking records as soon as the first reference is found. While this has the potential to be more efficient, it would lead to a number of complications. First, we note that the pointer rotation scheme employed during the *mark* phase temporarily modifies both the *mark* field and the *pointer variable* fields of the encountered heap records. As a consequence, one cannot simply exit the *mark* phase when a reference is found, but would also need to undo *all* pointer modifications made up to that point. The easiest way to achieve this is by completing the *mark* phase all the way to its end. This in turn would undo the envisioned performance gain. Second, if one wants to omit in module *Kernel* any reference to the data structure managed by module *Modules*, one would need to express also the *mark* phase as a *generic* heap traversal scheme with parametric handler procedures for reference checking, similar to the generic heap *scan* scheme outlined above. But such a generalization would open up the possibility for an erroneous handler procedure to prematurely end the *mark* phase, leaving the heap in an potentially irreparable state. In sum, one seems well advised not to interfere with the *mark* phase. In addition, one would still need a separate *scan* phase to *unmark* the already marked portion of the heap.

Another possible variant would be to treat *procedure variables* and *type tags* like *pointers*, and *procedures* and *type descriptors* referenced by them like *records* during the *mark* phase. This could be achieved by making procedures and type descriptors *look like* records, which in turn can be accomplished by making them carry a *type tag* and a *mark field*. These additional fields

---

[21] Adding procedure variable offsets to type descriptors is, strictly speaking, not even 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. Thus, it is in principle possible to realize the reference checking scheme without additional memory requirements in type descriptors. We refrain from implementing this 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 a record's field list, only a small change needs to be made to a single procedure in the compiler (ORP.RecordType). However, a complication arises because records may recursively contain other records, each again containing procedure variables at arbitrary offsets. While it is possible to "flatten out" such recursive record structures, it would make other operations more complex. For example, assignments to subrecords would become less natural, because their fields would no longer be located in a contiguous section in memory. Second, the memory savings in the module areas holding the type descriptors would be marginal, given that there exists only one descriptor per declared record type. In addition, we believe that most applications should be programmed in the conventional programming style, where installed procedures are rare. For example, in Oberon, the object-oriented programming style is restricted to the viewer system, which provides distributed control in the form of installed handlers (of which there is typically only one per type of viewer). In sum, the benefit obtained by saving a few fields in a relatively small number of type descriptors appears negligible, and therefore the additional complexity required to implement this refinement would be hard to justify.

[22] For a heap record with references, one could display the name of the module declaring its t y p e, which is not necessarily the module rooting the data structure containing the record. One could, however, modify the reference checking algorithm to check each module to be unloaded i n d i v i d u a l l y for references, in which case also their data structure r o o t s are known.

would be inserted as a prefix to (the code section of) *each* procedure and *each* type descriptor stored in the module block. All static *procedures* would share a common “procedure descriptor” and all *type descriptors* a common “meta-type descriptor”. These descriptors would contain no tags to extensions and no pointer field offsets. Consequently, they can be represented by one and the same global “meta descriptor”, which could be stored at a fixed location in the module area. The resulting run-time representation of *procedures* is shown in Figure 3.

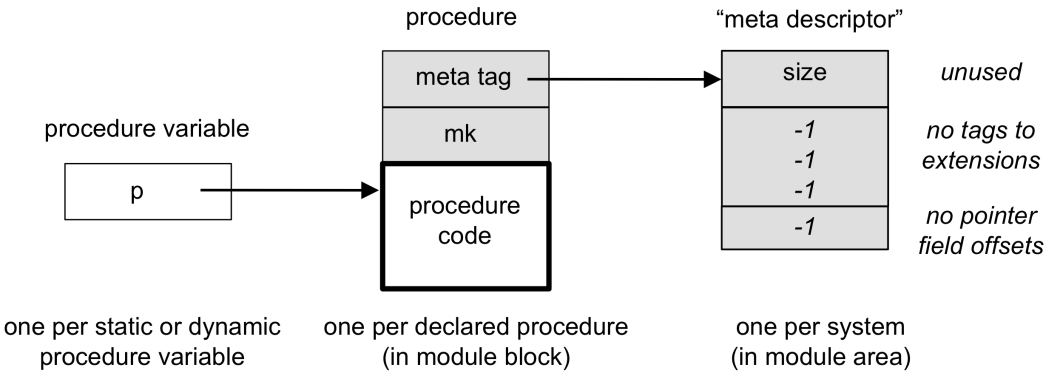

Figure 3: Procedure variable interpreted as *pointer*, and procedure interpreted as *record*

The corresponding run-time representation of *type descriptors* is shown in Figure 4.

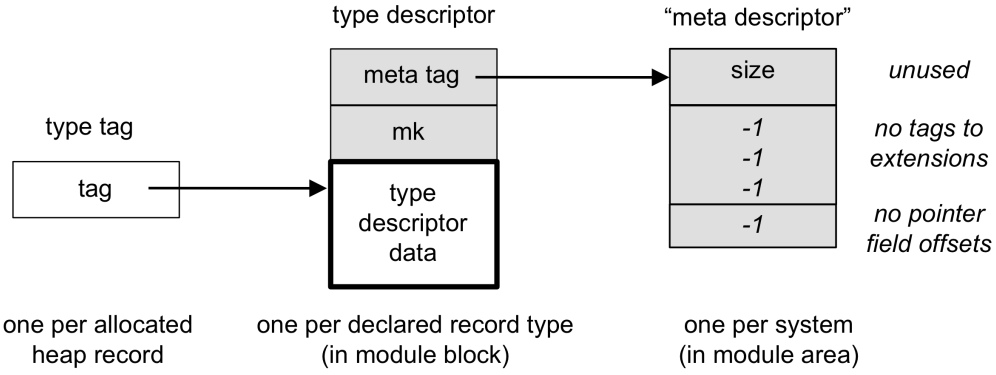

Figure 4: Type tag interpreted as *pointer*, and type descriptor interpreted as *record*

A simpler variant would be to treat procedure variables and type tags as *special cases* during the *mark* phase, eliminating the need for a *meta* tag field as well as the shared *meta* descriptor. The resulting run-time representation of *procedures* and *type descriptors* is shown in Figure 5.

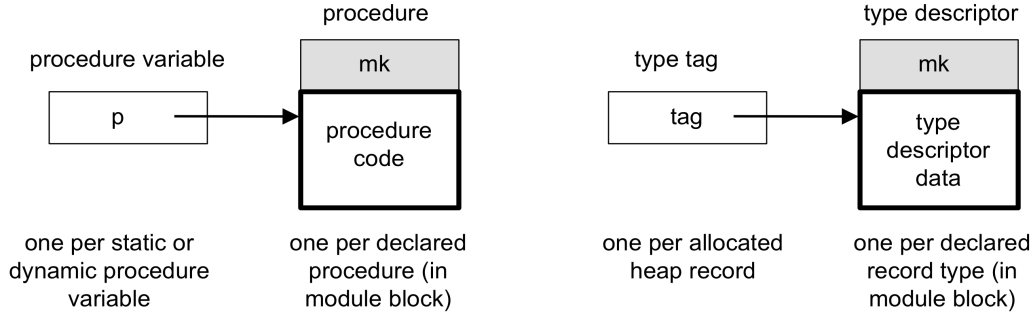

Figure 5: Procedures and type descriptors with an additional *mark* field

With these preparations, the *mark* phase of reference checking could be *extended* to include *procedure variables* and *type tags* in the list of "pointers" to be traversed for each encountered "record", which could now *also* be a declared *procedure* or a *type descriptor*. Similarly, the roots used by the *mark* phase as the starting points for the heap traversals would now include global *procedure* variables, in addition to named global *pointer* variables. In sum, the *extended* mark phase would not only mark objects in the *heap*, but also in the *module blocks*, implying that after the *mark* phase *all* referenced objects are *already* marked.

While this technique appears appealing, a few points are worth mentioning. First, the *extended* mark phase requires an extra *mark* field to be inserted as a prefix to *each* procedure and *each* type descriptor in the module block. Given that most procedures and type descriptors are never referenced, this appears to be overkill. One could therefore decide to add the *mark* field only to *module descriptors* and mark *modules* rather than individual *procedures* or *type descriptors* during the *mark* phase. While this would make the check whether any of the selected modules is referenced a trivial task, it would render the *mark* phase more complex, as it would now need to locate the module descriptor belonging to a given procedure or type descriptor[23]. Second, one would still need to mark *all* objects, for the same reason as outlined above, i.e. one cannot simply exit in the middle of the *mark* phase. And finally, a comparison of the code required to implement the various alternatives showed that our solution is *by far* the simplest: the *total* implementation cost of *all* modifications to the representation of the type descriptor, the Oberon object file format and the module loader, as described earlier, is only about a dozen of lines of source code[24], while the *reference checking* phase itself amounts to less than 75 lines of code.

5. System building 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 linker has been included from a different source[25] and has been adapted for *Experimental Oberon*'s object file format. There is also a version of the Oberon building tools described here for *Original Oberon 20*13[26].

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) typically called the *platform flash*.

In Oberon, the boot firmware is called the *boot loader*, as its main 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, i.e. the boot loader is *not* used after booting.

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 link, realized using an RS-232 serial line in Oberon 2013. The default boot source is the local disk. It is used by the *regular* Oberon boot process each time the computer is powered on or the reset button is pressed. There are

[23] *On systems where additional meta-information is present in the run-time representation of modules, such as the locations of procedures and type descriptors in module blocks and/or backpointers from each object of a module to its module descriptor, the mark phase could be made simpler. However, Experimental Oberon does not offer such metaprogramming facilities.*
[24] *See procedures ORG.BuildTD (+ 1 line), ORG.Close (+ 7 lines) and Modules.Load (+ 4 lines).*
[25] *http://www.github.com/charlesap/io*
[26] *http://www.github.com/andreaspirklbauer/Oberon-building-tools (subdirectory Sources/OriginalOberon2013)*

two valid *boot file formats* in Oberon: the "regular" boot file format used for booting from the local disk, and the "build-up" boot file format used for booting over a data link. See the appendix for a detailed description of these formats.

The boot loader *transfers* the boot file byte for byte from the boot source into memory, but does not call the initialization bodies of the just transferred modules. However, the memory location which the boot loader will branch to at the end of the boot load phase will transfer control to the *top* module in the just transferred boot file, making it the only module in the boot file whose initialization body is *actually* executed. For the *regular* Oberon boot file, this is module *Modules*.

To allow proper continuation of the boot process *after* having transferred the boot file into memory, the boot loader deposits some *additional* key data in fixed memory locations before passing control to the *top* module of the boot file. Some of this data is contained in the boot file itself and is transferred into memory by virtue of reading the first block of the boot file. See chapter 14.1 of the book *Project Oberon 2013 Edition* for a description of these data elements.

Creating a valid Oberon boot file

The command *Linker.Link* links a set of Oberon object files together and generates a valid boot file from them. The linker is almost identical to the *regular* loader (procedure *Modules.Load*), except that it outputs the result in the form of a file instead of depositing the object code of the linked modules in newly allocated module blocks in memory.

To compile the modules that should become part of the boot file[27]:

```
ORP.Compile Kernel.Mod FileDir.Mod Files.Mod Modules.Mod ~     ... modules for the "regular" boot file
ORP.Compile RS232.Mod Oberon0.Mod ~                           ... additional modules for the "build-up" boot file
```

To link these object files together and generate a single boot file from them:

```
Linker.Link Modules ~      ... generate a pre-linked binary file of the "regular" boot file (Modules.bin)
Linker.Link Oberon0 ~      ... generate a pre-linked binary file of the "build-up" boot file (Oberon0.bin)
```

The name of the top module is supplied as a parameter. The linker automatically includes all modules that are directly or indirectly imported by the top module. It also places the address of the *end* of the module space in memory used by the linked modules in a fixed location within the generated binary file[28]. In the case of the *regular* boot file loaded onto the boot area of the local disk using the command *Builder.Load*, this information is used by the boot loader to determine the *number* of bytes to be transferred from the boot file into memory.

To convert a pre-linked build-up boot file to the stream format used for booting over a data link:

```
Builder.WriteStream Oberon0.bin Oberon0.stream 512 0 ~
```

The first parameter is the name of the pre-linked boot file, as generated by the command *Linker.Link* (input file). The second parameter is the name of the file containing the boot file in stream format (output file). The third parameter is the block size (the same block size is used for all blocks, and the last block is zero-filled to this size). A value of 0 means that the length of the input file is used as the block size. The fourth parameter is the destination address in

[27] *Note that in our implementation of the building tools, module PCLink1 is not needed to create the build-up boot file, as the procedures for transferring files between the host and the target system are contained in the top module Oberon0.*
[28] *location 16 in the case of Oberon 2013*

memory. We will not make use of this command, as the command *ORC.Load*, as described below, automatically performs the required conversion. It is provided in case other mechanisms are used to send a pre-linked boot file to the target system.

Updating the boot area for starting Oberon from the local disk

The command *Builder.Load* loads a valid boot file, as generated by *Linker.Link*, onto the *boot area* of the local disk (sectors 2-63 in Oberon 2013), one of the two valid boot sources.

```
Builder.Load Modules.bin ~          ... load the "regular" boot file onto the boot area of the local disk
```

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. A backup of the disk is therefore recommended before experimenting with new Oberon *boot files* (when using on Oberon emulator on a Mac or Linux computer, one can create a backup by simply making a copy of the directory containing the disk image used by the emulator). If the module interface of an *inner core* module has changed, all client modules required to successfully restart the Oberon system as well as the compiler itself must be recompiled *before* restarting the system.

The format of the boot file is *defined* to exactly mirror the standard Oberon storage layout[29]. In particular, location 0 in the boot file (and later in memory once it has been loaded by the boot loader) 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 from a valid boot source into memory and then branch to location 0 – which is precisely what it does.

Building a new Oberon system on a bare metal target system

This section assumes an Oberon system running on a "host" system connected to a "target" system via a data link (e.g., an RS-232 serial line). Before getting started, open a tool viewer on the host system containing the commands described below:

```
Edit.Open Builder.Tool
```

To generate the necessary binaries for the build-up boot process:

```
ORP.Compile Kernel.Mod FileDir.Mod Files.Mod Modules.Mod ~  ... modules for the "regular" boot file
ORP.Compile RS232.Mod Oberon0.Mod ~                        ... additional modules for the "build-up" boot file
ORP.Compile ORC.Mod/s Oberon0Tool.Mod/s ~                  ... partner program ORC and Oberon0 tool module

ORP.Compile BootLoadDisk.Mod/s ~    ... generate a boot loader for booting the target system from the local disk
ORP.Compile BootLoadLine.Mod/s ~    ... generate a boot loader for booting the target system over the data link

Linker.Link Modules ~               ... generate a pre-linked binary file of the "regular" boot file (Modules.bin)
Linker.Link Oberon0 ~               ... generate a pre-linked binary file of the "build-up" boot file (Oberon0.bin)
```

Then restart the target system with the switch on the target system set to *“serial link”.*

When using Oberon in an emulator on Unix, one can simulate the process of booting the target system over a data link by starting 2 Oberon instances connected via 2 Unix pipes, e.g.,

```
mkfifo pipe1 pipe2                  ... create 2 pipes (one for each direction) linking host and target
```

[29] *See chapter 8.1, page 103, of the book Project Oberon 2013 Edition for a detailed description of Oberon’s storage layout.*

```
cp S3RISCinstall/RISC.img ob1.dsk      ... make a copy of a valid Oberon disk image for the host system
rm -f obj2.dsk; touch obj2.dsk         ... create an "empty" disk for the target system
./risc --serial-in pipe1 --serial-out pipe2 ob1.dsk &        ... start the host system from the local disk
./risc --serial-in pipe2 --serial-out pipe1 ob2.dsk --boot-from-serial &  ... start the target system
```

To load the build-up boot file over the serial link to the target system and start it:

ORC.Load Oberon0.bin ~ *... load the Oberon-0 command interpreter over the serial link to the target system & start it*

This command must be the first *ORC* command to be run on the host system after the target system has been rebooted, as its boot loader is waiting for a boot file. If one tries to send other data to the target system before downloading the boot file, it must be restarted again. The command automatically performs the conversion of the input file to the stream format required for booting over the serial link (i.e. a format accepted by procedure *BootLoad.LoadFromLine*).

To test whether the Oberon-0 command interpreter is now running on the target system:

ORC.SR 0 1234 ~ *... test whether the Oberon-0 command interpreter is running (send and mirror integer s)*

To build a new Oberon system on the target system:

ORC.SR 101 ~ *... clear the file directory on the target system*

ORC.Send Input.rsc Display.rsc Viewers.rsc ~
ORC.Send Fonts.rsc Texts.rsc Oberon.rsc ~
ORC.Send MenuViewers.rsc TextFrames.rsc ~
ORC.Send System.rsc System.Tool Oberon10.Scn.Fnt ~ *... send the required files to start Oberon from the local disk*
ORC.Send RS232.rsc Oberon0.rsc Oberon0Tool.rsc ~ *... send additional files to the target system (optional)*
ORC.Send Linker.rsc Builder.rsc ~ *... send additional files to the target system (optional)*
ORC.Send Edit.rsc PCLink1.rsc ~ *... send additional files to the target system (optional)*

ORC.Send Modules.bin ~ *... send the regular boot file "Modules.bin" to the target system*
ORC.SR 100 Modules.bin ~ *... load the regular boot file onto the boot area of the local disk of the target system*

The target system can now be restarted, either *manually* by setting the corresponding switch to "disk" and pressing the reset button, or *remotely* by using the partner program *ORC*:

ORC.Send BootLoadDisk.rsc ~ *... send the boot loader for booting from the local disk of the target system*
ORC.SR 102 BootLoadDisk.rsc ~ *... reboot the target system from the local disk (initiates the "regular" boot process)*

Alternatively, one can simply load module *Oberon* on the target system via module *ORC*:

ORC.SR 20 Oberon ~ *... load module "Oberon" on the target system (will also load module "System"*)

The Oberon system should now come up on the target system. The entire process from initial booting over the serial link to a fully functional Oberon system on the target system only takes a few seconds. Once the full Oberon system is running on the target system, one can re-enable the ability to transfer files between the host and the target system by running the command

PCLink1.Run *... start the PCLink1 Oberon background task*

on the *target* system. After this step, which one can again use the commands *ORC.Send* and *ORC.Receive* on the host system.

Alternatively, we note that even though the main use of module *Oberon0* is to serve as the top module of a *build-up* boot file loaded over a data link to a target system, it can *also* be loaded as a *regular* module. In the latter case, its main loop is not started, as it would block the system.

Instead, a user can install the Oberon-0 command interpreter as a regular background task:

```
PCLink1.Stop        ... stop the PCLink1 background task if it is running (uses the same RS232 queue as Oberon0)
Oberon0Tool.Run     ... start the Oberon-0 command interpreter as an Oberon background task
```

This effectively implements a *remote procedure call (RPC)* mechanism, i.e. a remote computer connected via a data link can use the commands *ORC.SR 22 M.P* ("call command") or *ORC.SR 102 M.rsc* ("call standalone program") to initiate execution of the specified command or program on the computer where the Oberon-0 command interpreter is running. To stop the Oberon-0 command interpreter background task, execute the command *Oberon0Tool.Stop*.

Note that the command *ORC.SR 22 M.P* does *not* transmit parameters to the target system. We recall that in Oberon the parameter text of a command typically refers to objects that exist before command execution starts, i.e. the "state" of the system represented by its global variables. Even though it would be easy to realize, it appears unnatural to allow denoting a state from a *different* (remote) system. Indeed, an experimental implementation showed that it tends to be confusing to users. If a user wants to execute a command *with* parameters, he/she can always execute it directly on the target system.

Other available Oberon-0 commands

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. These commands are listed in chapter 14.2 of the book *Project Oberon 2013 Edition*. Below are some examples of how to use them:

```
ORC.Send Modules.bin ~              ... send the regular boot file "Modules.bin" to the target system
ORC.SR 100 Modules.bin ~            ... load the regular boot file onto the boot area of the target system's local disk

ORC.Send BootLoadDisk.rsc ~         ... send the boot loader for booting from the local disk of the target system
ORC.SR 102 BootLoadDisk.rsc ~       ... reboot from the boot area of the local disk ("regular" boot process)

ORC.Send BootLoadLine.rsc ~         ... send the boot loader for booting the target system over the serial link
ORC.SR 102 BootLoadLine.rsc ~       ... reboot the target system over the serial link ("build-up" boot process)
ORC.Load Oberon0.bin ~              ... after booting over the data link, one needs to run ORC.Load Oberon0.bin again

ORC.SR 0 1234 ~                     ... send and mirror integer s (test whether Oberon-0 is running)
ORC.SR 7 ~                          ... show allocation, nof sectors, switches, and timer

ORC.Send Draw.Tool ~                ... send a file to the target system
ORC.Receive Draw.Tool ~             ... receive a file from the target system
ORC.SR 13 Draw.Tool ~               ... delete a file on the target system

ORC.SR 12 "*.rsc" ~                 ... list files matching the specified prefix
ORC.SR 12 "*.Mod!" ~                ... list files matching the specified prefix and the directory option set
ORC.SR 4 Builder.Tool ~             ... show the contents of the specified file

ORC.SR 10 ~                         ... list modules on the target system
ORC.SR 11 Kernel ~                  ... list commands of a module on the target system
ORC.SR 22 M.P ~                     ... call command on the target system
```

```
ORC.SR 20 Oberon ~          ... load module on the target system
ORC.SR 21 Edit ~            ... unload module on the target system

ORC.SR 3 123 ~              ... show sector   secno
ORC.SR 52 123 3 10 20 30 ~  ... write sector  secno, n, list of n values (words)
ORC.SR 53 123 3 ~           ... clear sector  secno, n (n words))

ORC.SR 1 50000 16 ~         ... show memory   adr, n words (in hex) M[a], M[a+4],...,M[a+n*4]
ORC.SR 50 50000 3 10 20 30 ~ ... write memory  adr, n, list of n values (words)
ORC.SR 51 50000 32 ~        ... clear memory  adr, n (n words))

ORC.SR 2 0 ~                ... fill display with words w (0 = black)
ORC.SR 2 4294967295 ~       ... fill display with words w (4294967295 = white)
```

Adding modules to an Oberon boot file

When *adding* modules to a boot file, the need to call their 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 reset. 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 module initialization sequences of the just transferred modules (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 a boot file is to move its initialization code to an exported procedure *Init* and call it from the top module 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 in memory and simply call them, as shown in procedure *InitMod*[30] below. See chapter 6 of the book *Project Oberon* for a description of the format of an Oberon *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); (*call module initialization body*)
1     VAR mod: Modules.Module; body: Modules.Command; w: INTEGER;
2   BEGIN mod := Modules.root;
3     WHILE (mod # NIL) & (name # mod.name) DO mod := mod.next END ;
4     IF mod # NIL THEN SYSTEM.GET(mod.ent, w);
5       body := SYSTEM.VAL(Modules.Command, mod.code + w); body
6     END
    END InitMod;
```

In the following example, module *Oberon* is chosen as the new top module of the *boot file*.

```
    MODULE Modules;                  ... old top module of the boot file, now just a regular module
1     IMPORT SYSTEM, Files;
2     ...
3   BEGIN Init                       ... no longer loads module Oberon (as in Original Oberon)
    END Modules.
```

[30] *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).*

```
    MODULE Oberon;                          ... new top module (and therefore part) of the boot file
 1   IMPORT SYSTEM, Kernel, Files, Modules,
 2     Input, Display, Viewers, Fonts, Texts;
 3   …
 4   BEGIN                                  ... boot loader will branch to here after transferring the boot file
 5   Modules.Init;                          ... must be called first (establishes a working file system)
 6   InitMod(„Input“);
 7   InitMod(„Display“);
 8   InitMod(„Viewers“);
 9   InitMod(„Fonts“);
10   InitMod(„Texts“);
11   …
12   Modules.Load(„System“, Mod);           ... load module System using the regular Oberon loader
13   Loop                                   ... transfer control to the Oberon central loop
    END Oberon.
```

The following commands build the modified *boot file* and load it onto the local disk's boot area:

```
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 boot file
Linker.Link Oberon ~                             ... create a new regular boot file (Oberon.bin)
Builder.Load Oberon.bin ~                        ... load the new boot file onto the local disk's boot area
```

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 *boot file*. If one prefers to keep the *boot file* minimal (as one should), 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 disadvantage that the viewer complex and the system tools are "locked" into the *outer core*. However, an Oberon system without a viewer manager hardly makes sense, even in closed server environments. As an advantage of the latter approach, we note that such an extended *outer core* can more easily be replaced on the fly (by unloading and reloading a suitable *subset* of modules), as module *Oberon* no longer invokes the module loader itself.

One can also include an *entire* Oberon system in an Oberon *boot file* and load it over the serial link to a target system (this is similar to booting a commercial operating system in a *Plug & Play* fashion directly from a USB stick) or load it onto the boot area of the local disk (this would allow the *entire* Oberon system to be transferred *en bloc* from the boot area directly into memory, without accessing *any* additional files, and without even requiring a file system to *exist* yet)[31].

Modifying the Oberon boot loader

In general, there is no need to modify the Oberon boot loader (*BootLoad.Mod*), which is resident in the computer's read-only store. Notable exceptions include situations with special requirements, for example when there is a justified need to add network code allowing one to boot the system over an IP-based network.

If one needs to modify the Oberon boot loader, first note that it is an example of a *standalone* program. Such programs are able to run on the bare metal. Immediately after a system restart or reset, the Oberon boot loader is in fact the *only* program present in memory.

[31] *See http://www.github.com/andreaspirklbauer/Oberon-building-tools for a description of how this can be accomplished. The latter variant would truly be the fastest possible way to start the Oberon system. The only (potentially) faster way to start Oberon is if the boot file were stored in some compressed format on disk, such as a "slim binary" encoding scheme, thereby reducing disk access time even further. However, this would require a modification of the boot loader to become a "code-generating loader" which translates the slim binary "on the fly" into binary code during the boot load phase, making it more complex (the boot loader is an intentionally tiny program resident in the computer's read-only store and should not be "abused" for such exercises). Experiments have shown that with the advent of fast secondary storage devices (e.g., solid-state drives), the speedup has become negligible or even non-existent, as compared with systems using a rotating disk or floppy disk as the boot device (1990s). Note that the boot file is a rather small file anyway (less than 100KB) and booting already is practically instantaneous in Oberon.*

As compared with regular Oberon modules, standalone programs have different starting and ending sequences. The *first* location contains an implicit branch instruction to the program's initialization code, and the *last* instruction is a branch instruction to memory location 0. The two processor registers holding the *static base* SB (R13) and the *stack pointer* SP (R14) are also initialized in the initialization sequence of a standalone program.

These modified starting and ending sequences can be generated by compiling a program with the "RISC-0" option of the regular Oberon compiler. This is accomplished by marking the source code of the program with an asterisk immediately after the symbol MODULE before compiling it. One can also create other small standalone programs using this method.

To generate a new Oberon boot loader, first mark its source code with an asterisk immediately after the symbol MODULE:

```
MODULE* BootLoad;   (*asterisk indicates that the compiler will generate a standalone program*)
  ...
BEGIN
  ...
END BootLoad.
```

To generate an Oberon object file of the boot loader as a standalone program:

```
ORP.Compile BootLoad.Mod ~     ... generate the object file of the boot loader (BootLoad.rsc)
```

To extract the code section from the object file *and* convert it to a PROM file compatible with the specific hardware used:

```
Builder.WritePROM BootLoad.rsc 512 ins.mem ~
```

The first parameter is the name of the object file of the boot loader (input file). The second parameter is the size of the PROM code to be generated in words (in the above example, the number of bytes in the PROM code is 512*4 = 2KB). The third parameter is the name of the PROM file to be generated (output file).

In the case of Original Oberon 2013 implemented on a field-programmable gate array (FPGA) development board from Xilinx, Inc., the format of the PROM file is a text file containing the opcodes of the boot loader. Each 4-byte opcode of the object code is written as an 8-digit hex number. If the *actual* code size is less *then* the specified code size, the code is zero-filled to the specified size.

```
E7000151    (line 1)
00000000
00000000
00000000
00000000
00000000
00000000
00000000
4EE90014
AFE00000
A0E00004
40000000
A0E00008
```

```
...
A0100000
40000000
C7000000    (line 384)
00000000
00000000
...
00000000    (line 512)
```

Note that the command *Builder.WritePROM* transfers only the *code* section of the specified object file, but not the *data* section (containing type descriptors and string constants) or the *meta* data section (containing information about imports, commands, exports and pointer and procedure variable offsets). Note also that the *module loader* and the *garbage collector* are not assumed to be the present when a standalone program is executed.

This implies that *standalone programs* cannot ...

- import other modules (except the pseudo module SYSTEM)
- define or use string constants
- define or use type extensions
- employ type tests or type guards
- allocate dynamic storage using the predefined procedure NEW

Since a standalone program does not import other modules, no access to external global variables, procedures or type descriptors data can occur. This has a number of consequences:

- First, there is no need to "fix up" any instructions in the program code once it has been transferred to a memory location on the target system (as the *regular* loader would do).
- Second, the *static base* never changes during the execution of a standalone program. We recall that memory instructions compute the address as the *sum* of a register (base) and an offset constant. Global variables use the static base SB register (R13) pointing to the *current* module as base, local variables the stack pointer SP (R14).
- And third, as a consequence of a permanently fixed static base, the global module table referenced by the processor's MT register (R12) is also never accessed to obtain the static base of another module (or the standalone program itself). Therefore, neither the global module table nor the MT register are needed in standalone programs.

This makes the code section of a standalone program a completely self-contained, relocatable instruction stream for inclusion directly in the hardware, where the static base SB register (R13) marks the beginning of the global *variable* space and the stack pointer SP (R14) the beginning of the procedure activation stack.

These registers are initialized to fixed values in the *starting* sequence of the standalone program (as generated by the compiler in procedure *ORG.Header*):

```
MOV SB 0          # set the static base SB register (R13) to memory location 0
MOV SP -64        # set the stack pointer SP register (R14) to memory location 0FFFC0H
```

Thus, by convention a standalone program uses the memory area starting at memory location 0 as the *variable* space for global variables. If the standalone program overwrites that same memory area, for example by using the low-level procedure SYSTEM.PUT (as the Oberon *boot*

loader does), it should be aware of the fact that assignments to global variables will also affect the *same* region. In such a case, it's best to simply not declare any global variables.

If a standalone program wants to use a *different* memory area as its global variable space, it can adjust the static base register using the low-level procedure SYSTEM.LDREG at the beginning of the program. It can also adjust the stack pointer using this method, if needed.

```
    MODULE* M;
1    IMPORT SYSTEM;
2    CONST SB = 13; SP = 14; VarOrg = 2000H; StkOrg = -1000H;
3    VAR x, y, z: INTEGER;
4   BEGIN SYSTEM.LDREG(SB, VarOrg); SYSTEM.LDREG(SP, StkOrg);     (*x at 2000H, y at 2004H,...*)
    END M.
```

Alternatively, a special version of the compiler may offer the ability to specify directly in the source code the values to which the static base (SB) register and the stack pointer (SP) will be set in the starting sequence of the standalone program.

```
MODULE* M[SB:8192, SP:-4096]; (*set SB to 8192 = 2000H and SP to -4096 = -1000H*)
  ...
END M.
```

Instead of extracting the code section from the object file and then converting it to a PROM format compatible with the specific hardware used using the command *Builder.WritePROM*, one can also extract the code section of the Oberon boot loader and write it in *binary* format to an output file using the command *Builder.WriteCode*, as shown below. This variant may be useful if the tools used to transfer the boot loader to the specific target hardware used allows one to directly include binary code in the transferred data.

```
Builder.WriteCode BootLoad.rsc BootLoad.code ~
```

The first parameter is the name of the object file of the boot loader (input file). The second parameter is the name of the output file containing the extracted code section.

Transferring the Oberon boot loader to the permanent read-only store of the target hardware typically requires the use of proprietary (or third-party) tools. For Oberon 2013 on an FPGA, tools such as *data2mem* or *fpgaprog* can be used. For further details, the reader is referred to the pertinent documentation, e.g.,

- *www.xilinx.com/support/documentation/sw_manuals/xilinx11/data2mem.pdf*
- *www.xilinx.com/Attachment/Xilinx_Answer_46945_Data2Mem_Usage_and_Debugging_Guide.pdf*
- *www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ise_tutorial_ug695.pdf*
- *http://www.saanlima.com/download/fpgaprog.pdf*

To create a **Block RAM Memory Map (BMM)** file *prom.bmm* (a BMM file describes how individual block RAMs make up a contiguous logical data space), either use the proprietary tools to do so (such as the command *data2mem*) or manually create it using a text editor, e.g.,

```
ADDRESS_SPACE prom RAMB16 [0x00000000:0x000007FF]
  BUS_BLOCK
    riscx_PM/Mram_mem [31:0] PLACED = X0Y22;
  END_BUS_BLOCK;
END_ADDRESS_SPACE;
```

To synthesize the **Verilog** source files defining the RISC processor into a **RISC configuration** file *RISC5Top.bit*, use the proprietary tools to do so. The necessary Verilog source files can be found at www.projectoberon.com.

To create a **BitStream (BIT)** file *RISC5.bit* (a bit stream file contains a bit image to be downloaded to an FPGA, consisting of the BMM file *prom.bmm*, the RISC configuration *RISC5Top.bit* and the boot loader *ins.mem*):

```
data2mem -bm prom.bmm -bt ISE/RISC5Top.bit -bd prom.ins -o b RISC5.bit
```

To transfer (not flash) a bit file to the FPGA hardware:

```
fpgaprog -v -f RISC5.bit
```

To flash a bit file to the FPGA hardware, one needs to enable the SPI port to the *flash chip* through the JTAG port:

```
fpgaprog -v -f RISC5.bit -b path/to/bscan_spi_lx45_csg324.bit -sa -r
```

* * *

APPENDIX: Oberon boot file formats

There are currently two valid *boot file formats* in Oberon: the "regular" boot file format used for booting from the local disk, and the "build-up" boot file format used for booting over a data link.

Regular boot file format (used for booting the system from the local disk)

The "regular" boot file is a sequence of *bytes* read from the *boot area* of the local disk (sectors 2-63 in Oberon 2013):

BootFile = {byte}

The number of bytes to be read from the boot area is extracted from a fixed location within the boot area itself (location 16 in Oberon 2013). The destination address is usually a fixed memory location (location 0 in Oberon 2013). The boot loader typically simply overwrites the memory area reserved for the operating system.

The pre-linked binary file for the regular boot file (*Modules.bin*) contains the modules *Kernel, FileDir, Files*, and *Modules*. These four modules are said to constitute the *inner core* of the Oberon system. The top module in this module hierarchy is module *Modules*, and the first instruction in the binary file is a branch instruction to the initialization code of that module.

This binary file needs to be created using the command *Linker.Link* and loaded as a sequence of *bytes* onto the boot area of the local disk once, using the command *Builder.Load*. From there, it will be loaded into memory by the Oberon *boot loader*, when the Oberon system is started from the local disk. This is called the "regular" Oberon startup process.

The format of the *regular* Oberon boot file is *defined* to *exactly* mirror the standard Oberon storage layout. Thus, the Oberon boot loader can simply transfer the boot file byte for byte into memory and then branch to its starting location in memory (typically location 0) to transfer control to the just loaded top module – which is precisely what it does.

Build-up boot file format (used for booting the system over a data link)

The "build-up" boot file is a sequence of *blocks* fetched from a *host* system over a data link:

```
BootFile = {Block}
Block    = size address {byte}          # size >= 0
```

Each block in the boot file is preceded by its size and its destination address in memory. The address of the last block, distinguished by *size = 0*, is interpreted as the address the boot loader will branch to *after* having transferred the boot file.

In a specific implementation – such as in Oberon 2013 on RISC – the address field of the last block may not actually be sent, in which case the format effectively becomes:

```
BootFile = {Block} 0
Block    = size address {byte}          # size > 0
```

The pre-linked binary file for the *build-up* boot file (*Oberon0.bin*) contains the modules *Kernel, FileDir, Files, Modules, RS232* and *Oberon0*. These six modules constitute the four modules of

the Oberon inner core *plus* additional facilities for communication. The top module in this module hierarchy is module *Oberon0*, and the first instruction in the binary file is a branch instruction to the initialization code of that module.

This binary file needs to be created using the command *Linker.Link* and be made available as a sequence of *blocks* on a host computer connected to the target system via a data link, using the command *ORC.Load*. From there, it will be fetched by the Oberon boot loader on the target system and loaded into memory, when the Oberon system is started over the link. This is called the "build-up boot" or "system build" process. It can also be used for diagnostic or maintenance purposes.

After having transferred the *build-up* boot file over the data link into memory, the boot loader terminates with a branch to location 0, which in turn transfers control to the just loaded top module *Oberon0*. Note that this implies that the module initialization bodies of *all* other modules contained in the build-up boot file are never executed, including module *Modules*. This is the intended effect, as module *Modules* depends on a working file system – a condition typically not yet satisfied when the build-up boot file is loaded over the data link for the very first time.

Once the Oberon boot loader has loaded the *build-up* boot file into memory and has initiated its execution, the now running top module *Oberon0* (a command interpreter accepting commands over a communication link) is ready to communicate with a partner program running on a "host" computer. The partner program, for example *ORC* (for Oberon to RISC Connection), sends commands over the data link to module *Oberon0* running on the target Oberon system, which will execute them *there* on behalf of and send the results, if any, back to the partner program.

The Oberon-0 command interpreter offers a variety of commands for system building and inspection purposes. For example, there are commands for establishing the prerequisites for the regular Oberon startup process (e.g., creating a file system on the local disk or transferring the modules of the inner and outer core and other needed files from the host system to the local disk of the Oberon system) and commands for file system, memory and disk inspection. A list of available Oberon-0 commands is provided in chapter 14.2 of the book *Project Oberon 2013 Edition*.