The Experimental Oberon System

Andreas Pirklbauer

7.5.2018

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]. For the purist, such a feature may represent an “unnecessary embellishment” of Oberon, but it is simply indispensable if the system is to support touch display devices where a mouse is absent and viewers may not have scrollbars. In such an environment, continuous scrolling is the only acceptable way to scroll and presents a more natural user interface. As a 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*, open and close *tracks* within existing displays and open and close individual *viewers* within tracks. There are no restrictions on the number of displays, tracks or viewers that can be created. Focus viewers and text selections are separately defined for each display. This scheme naturally maps to systems with multiple *physical* monitors. It can also be used to realize fast context switching, for example in response to a swipe gesture on a touch display device.

The command *System.OpenDisplay* 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 commands *System.Expand, System.Spread* and *System.Clone* are displayed in the title bar of every menu viewer. *System.Expand* expands the viewer *as much as possible* by reducing *all* other viewers in the track to their minimum heights. The user can switch back to any of the minimized viewers by clicking on the command *System.Expand* again in any of their still visible title bars. *System.Spread* evenly redistributes all viewers vertically in the track where the viewer is located. This may be useful after having invoked *System.Expand*. *System.Clone* opens a new logical display on the fly *and* displays a copy of the initiating viewer *there*. The user can toggle between the two copies of the viewer (i.e. switch *displays*) with a single mouse click[4].

3. Enhanced viewer management

The previously separate notions of *frame* and *viewer* have been *united* into a single, *unified viewer* concept, in which the (recursive) nesting properties of the original frame concept have been fully preserved. The distinction between frames and viewers (a viewer's state of visibility) appeared to be rather marginal and hardly justified in Oberon's display system, which is based on hierarchical tiling rather than overlapping windows.

The unified viewer concept is not only simpler, but also more flexible than the original scheme, as each sub viewer (called a *frame* in Original Oberon) can now have its *own* state of visibility. This makes it possible to bring the concept of hierarchical tiling, which is already used for top-level viewers, all the way down to the sub viewer level, enabling "newspaper-style" multi-column text layouts for example. Conceptually, this constitutes a hybrid between *hierarchical tiling* and *unconstrained tiling*, but without the disadvantages of the latter (see chapter 4.1 of the book *Project Oberon 2013 Edition* for a brief overview of the various possible variants of tiling).

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 an open 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* 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 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 and *no* command option is specified in the module unload command *System.Free*, the command takes no action and merely displays the names of

[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.*

the modules containing the references that caused the removal to fail. If references exist *and* the *force* option /f is specified[5], the specified modules are initially removed only from the *list* of loaded modules, without releasing their associated memory[6]. Such “hidden” modules are later physically removed from memory as soon as there are no more references to them[7]. 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[8]. It checks all possible combinations of module *subgroups* among the hidden modules for outside clients and references[9]. The tool commands *System.ShowRefs* and *System.ShowGroupRefs* can be used to identify all modules containing references to the specified module or module group.

In sum, module unloading does not affect *past* references, as module data is kept in memory for exactly as long as necessary and removed from it as soon as possible. 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 exactly as long as needed[10].

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 to exist and will *not* prevent the unloading of that group[11].

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*[12]. 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 with *no* outside clients.

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* command *System.Free* with a *specific* set of modules provided as parameters. To help identify valid subsets of modules that can be unloaded, the additional 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

---

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

[6] *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 only from the list of loaded modules, but not from memory, are marked with an asterisk in the output of the command System.ShowModules. Commands of such “hidden” modules can be accessed by either specifying their module number or their (modified) module name, both of which are displayed by the command System.ShowModules. In both cases, the corresponding command text must be enclosed in double quotes. If a module M carries module number 14, for instance, one can activate a command M.P also by clicking on the text “14.P”. Typical use cases include hidden modules that still have 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.*

[7] *Removing a module from memory frees up the memory area 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 are dynamically allocated in the heap at module load time, in order to persist them beyond the lifetime of their associated modules. In Experimental Oberon, no such special precaution to persist type descriptors is necessary, as 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.*

[8] *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.*

[9] *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*

[10] *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.*

[11] *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 artificially constructed 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.*

[12] *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.*

outside clients exist, references are checked next. 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], shown below.

```
    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 (*outside clients exist*)
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.pvr; 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 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 modules pointing to descriptors of types declared in the modules to be unloaded, and *procedure variables* installed in *static or dynamic* objects of other loaded modules referring to *static* procedures declared in the modules to be unloaded[14].

[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] *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 global types (in which case they can be exported and referenced by name in client modules) or as types local to a procedure (in which case they cannot be exported). Variables can be declared as global variables (allocated in the module area when a module is loaded) or as local variables (allocated on the stack when a procedure is called), or dynamically allocated 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 modules to static or dynamic objects of the specified modules to be unloaded. However, only dynamic type and static and dynamic procedure references need to be checked during module unloading for the following reasons. First, we note that static type and variable references from other loaded modules can only refer by name 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 separately. Second, dynamic variable references from global or dynamic pointer variables of other modules to dynamic objects reachable by the modules to be unloaded don't need to be checked either, as such references shouldn't prevent module unloading. In the Oberon operating system which features automatic storage reclamation, such references will be handled by the garbage collector during a future garbage collection cycle, i.e. heap records reachable by the just unloaded modules and other still loaded modules are not collected, while heap records that were reachable only by the just unloaded modules are collected. Thus, the handling of dynamic pointer references is effectively delegated to the garbage collector. Finally, pointer variable references to static objects declared as global variables are only possible by resorting to low-level facilities and should be avoided (and, in fact, be disallowed).*

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*[15], 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).

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, as shown below.

```
     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 rather than *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)[16].

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 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

[15] *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.*
[16] *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.*

(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.

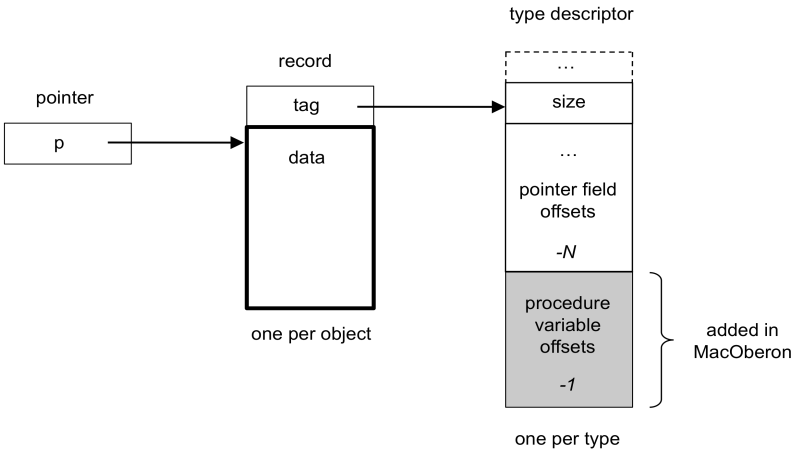

Figure 1: Run-time representation of a dynamic record and its type descriptor in MacOberon

However, this run-time layout 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[19], Experimental Oberon uses a slightly different layout, where *procedure variable offsets* are *prepended* (rather than appended) to the fields of each type descriptor, as shown in Figure 2[20].

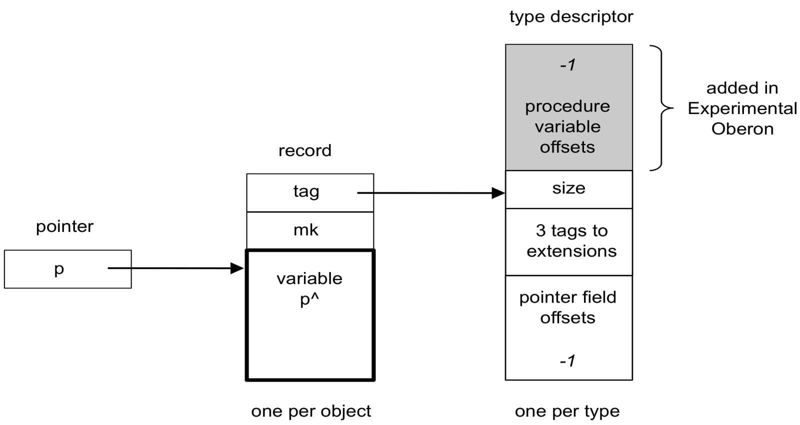

Figure 2: Run-time representation of a dynamic record and its type descriptor in Experimental Oberon

---

[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)*
[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).*

The compiler generating the modified type descriptors of dynamic objects (records) 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 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].

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 in the *mark* phase *all* modules to be unloaded are checked for references 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 tends to be small. We also note that modules that manage dynamic data structures shared by clients (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 *all other* loaded modules and the *scan* phase then checks for references from the *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 used during the *mark* phase temporarily modifies not only the *mark* field, but also 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. But this would undo the performance gain initially hoped for. 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 a potentially irreparable state. In sum, one seems well advised not to interfere with the *mark* phase. Finally, 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 of reference checking. This could easily be achieved by making procedures and type descriptors *look like* records, which can be accomplished by making them carry a *type tag* and a *mark field*. These additional fields would be inserted as a prefix to (the code section of) *each* declared

[21] *Adding procedure variable offsets to type descriptors 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, 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 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 always 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 placed 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 t y p e rather than one per allocated heap o b j e c t. Finally, we believe that most applications should be programmed in the conventional programming style, where installed procedures are rare. For example, in the Oberon operating system, 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 viewer t y p e. In sum, the benefit obtained by saving a few fields in a relatively small number of type descriptors appears negligible and the additional effort required to realize it hard to justify.*

procedure and (the type descriptor of) *each* declared record type 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 shared global "meta descriptor", which could be stored at a fixed location within the module area for example. The resulting run-time representation of *procedures* and *type descriptors* is shown in Figure 3.

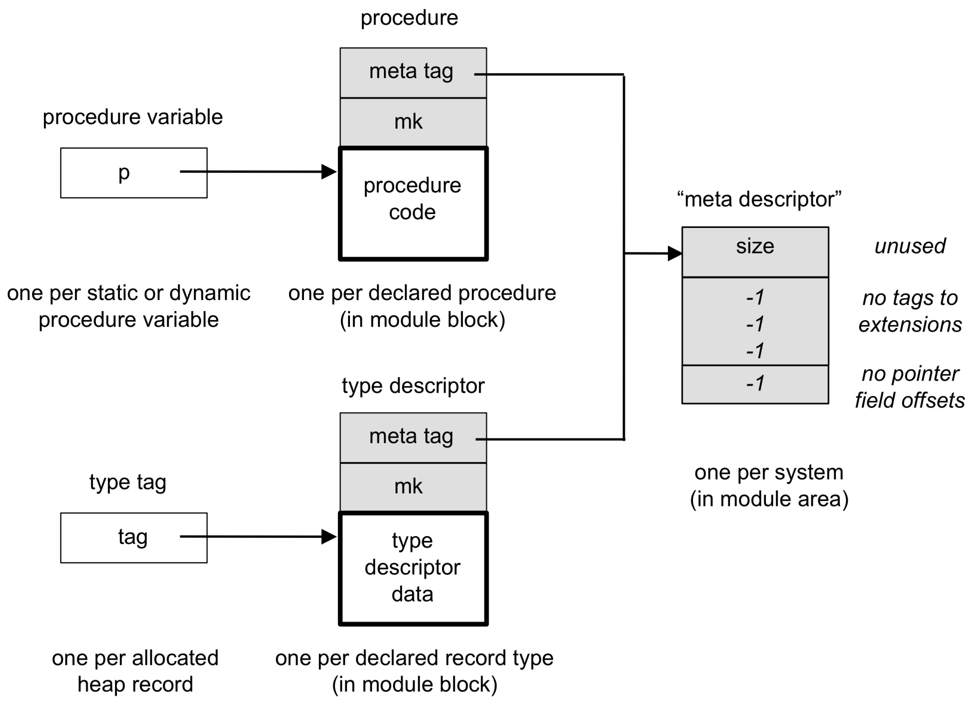

Figure 3: Procedure variable and type tag interpreted as *pointer*, and procedure and type descriptor interpreted as *record*

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 4.

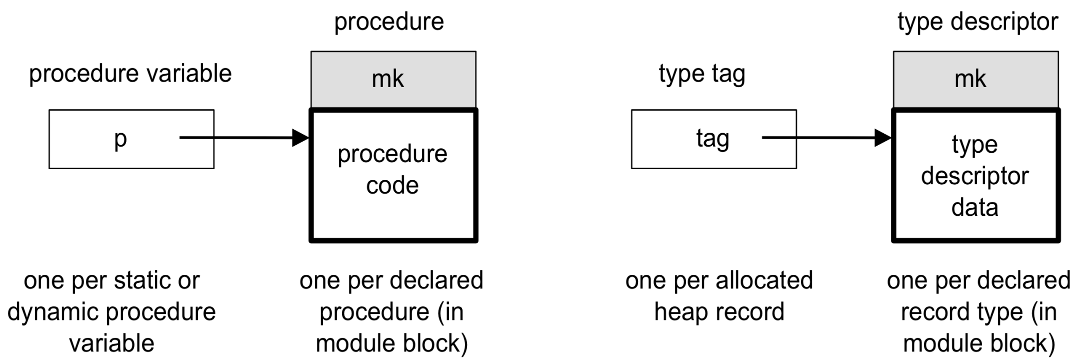

Figure 4: Procedures and type descriptors with an additional *mark* field

With these preparations, the *mark* phase of reference checking could be suitably *extended* to also include *procedure variables* in the list of "pointers" to be traversed, with the roots used as the starting points for the traversals now also including global *procedure* variables, in addition to global *pointer* variables. Thus, the *extended* mark phase would not only touch objects in the *heap*, but also in the *module blocks*, with the effect that after the *mark* phase *all* referenced dynamic *and* static objects are already marked, not just dynamic objects in the heap.

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 (on systems where additional meta-information is present in the run-time representation of loaded modules, such as the locations of procedures and type descriptors in module blocks or *back pointers* from each object of a module to its module descriptor, the mark phase would be simpler; however, neither Original Oberon 2013 nor Experimental Oberon offers such *metaprogramming* facilities). 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 object file format and the module loader, as described earlier, is only about a dozen of lines of source code[22], 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 Oberon system building tools described below exist for both *Original Oberon 2013* and *Experimental Oberon*[23].

The command *Boot.Link* links a set of Oberon object files together and generates a valid *boot file* from them. The name of the top module is supplied as a parameter. For the *regular* boot file loaded onto the boot area[24] of a disk, this is typically module *Modules*, the top module of the *inner core* of the Oberon system. For the *build-up* boot file sent over a data link to a target system, it is usually module *Oberon0*, a command interpreter mainly used for system building purposes. The boot linker automatically includes all modules that are directly or indirectly imported by the top module. It also places the address of the *end* of the module space used by the linked modules in a fixed location within the generated binary file[25]. In the case of the *regular* boot file, this information is used by the Oberon *boot loader* (a small program permanently resident in the computer's read-only store which loads a *boot file* into memory when the system is started) to determine the *number* of bytes to be transferred. The Oberon boot linker (procedure *Boot.Link*) is almost identical to the *regular* module loader (procedure *Modules.Load*), except that it outputs the result in the form of a file on disk instead of depositing the object code of the linked modules in newly allocated module blocks in memory.

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

[22] *See procedures ORG.BuildTD (+ 1 line), ORG.Close (+ 7 lines) and Modules.Load (+ 4 lines).*
[23] *http://www.github.com/andreaspirklbauer/Oberon-building-tools*
[24] *Sectors 2-63 in Original Oberon 2013 and Experimental Oberon*
[25] *Location 16 in the case of Original Oberon 2013 and Experimental Oberon*

is therefore recommended before experimenting with new Oberon *boot files* (when using an Oberon emulator, one can create a backup by making a copy of the directory containing the Oberon disk image). If the module interface of a module contained in the *boot file* has changed, all client modules required to restart the Oberon system and the Oberon compiler itself must also be recompiled before restarting the system.

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). When using Oberon in an emulator, one can simulate the process of booting the target system over a data link by starting two Oberon emulator instances connected via two Unix *pipes*, one for each direction:

```
mkfifo pipe1 pipe2                    ... create two pipes (one for each direction) linking host and target system
rm -f ob1.dsk ob2.dsk                 ... delete any old disk images for the host and the target system (optional)
cp S3RISCinstall/RISC.img ob1.dsk     ... make a copy of a valid Oberon disk image for the host system
touch ob2.dsk                         ... create an "empty" disk image for the target system (will be “filled” later)
./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
```

The tools to build a new Oberon system on a bare metal machine are provided by the module pair *ORC* (for Oberon to RISC Connection) running on the host system and *Oberon0* running on the target system connected to the host via the data link (see the tool text *Build.Tool*).

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

The command *ORC.SR 101* clears the root page of the file directory on the target system.

The command *ORC.Send* transfers files from the host to the target system. It is typically used to transfer the object files required to start Oberon on the target system (module *System* and its imports), the *System.Tool* tool text, the font file *Oberon10.Scn.Fnt*, the boot file *Modules.bin* and the standalone program *BootLoadDisc.rsc* used to initiate the *regular* boot process.

The command *ORC.SR 100 Modules.bin* loads the just transferred *regular* boot file *Modules.bin* onto the boot area of the target system. After this, the target system can be restarted either *manually* by setting the corresponding switch *on* the target system to "disk" and pressing the reset button, or *remotely* by executing the command *ORC.SR 102 BootLoadDisk.rsc* on the host system. The entire process from the initial transfer of the *build-up* boot file to a fully functional Oberon system on the target system only takes 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* on the *target* system. Alternatively, note that even though the primary use of module *Oberon0* is to serve as the top module of a *build-up* boot file loaded over a data link, it can *also* be loaded as a *regular* Oberon module by activating the command *Oberon0Tool.Run* on the *target* system[26].

---

[26] *Before executing Oberon0Tool.Run, one must execute PCLink1.Stop (as modules PCLink1 and Oberon0 use the same RS-232 queue).*

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

Note that the command *ORC.SR 22 M.P* does *not* transmit any parameters from the host to the target system. 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 implement a generic parameter transfer mechanism, it appears unnatural to allow denoting a system state from a *different* (remote) system. Indeed, an experimental implementation showed that it tends to confuse users. If one really wants to execute a command *with* parameters, one can execute it directly on the target system. To stop the Oberon-0 command interpreter background task, execute the command *Oberon0Tool.Stop* on the target system.

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 usage examples:

ORC.Send Modules.bin ~	*... send the regular boot file 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*
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*
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 System.Tool ~	*... send a file to the target system*
ORC.Receive System.Tool ~	*... receive a file from the target system*
ORC.SR 13 System.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 System.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 (4’294’967’295 = FFFFFFFFH = white)*

Adding modules to an Oberon boot file

When *adding* modules to an Oberon 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* of the Oberon system.

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*[27] below. See chapter 6 of the book *Project Oberon 2013 Edition* for a description of the format of an Oberon *module descriptor*. 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*)
  VAR mod: Modules.Module; body: 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);
    body := SYSTEM.VAL(Modules.Command, mod.code + w); body
  END
END InitMod;
```

One can include individual modules or an *entire* Oberon system in a *boot file*. Sending a pre-linked binary file containing the entire Oberon system over a serial link to a target system is similar to booting a commercial operating system in a *Plug & Play* fashion over the network or from a USB stick. If such an "all-compassing" boot file is loaded onto the boot area of the local disk on the target system (after having increased its size), the resulting system would come up *instantly*, as the *entire* system is transferred *en bloc* from the boot area into memory, without accessing any additional files and without even requiring a file system to exist yet[28].

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 (ROM or PROM). 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.

---

[27] *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).*

[28] *See http://www.github.com/andreaspirklbauer/Oberon-building-tools for a description of how this can be done. This variant would truly be the fastest possible way to start the Oberon system. The only potentially faster way to start the Oberon system is if the boot file were stored in some compressed format on disk, such as a "slim binary" encoding scheme, 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 considerably more complex. The boot loader is an intentionally tiny program resident in the computer's read-only store and should not be used for such sophisticated purposes with rather questionable benefit: experiments have shown that with the advent of fast secondary storage devices such as solid-state drives (2010s), the speedup has become completely negligible, as compared with systems that use a rotating disk or floppy disk as their boot device (1990s).*

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.

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. In addition, the two processor registers holding the *static base* SB (R13) and the *stack pointer* SP (R14) are also initialized to fixed values in the initialization sequence of a standalone program[29]:

```
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 -64 (= 0FFFC0H in Oberon 2013)
```

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.

Since global variables use the static base SB register (R13) pointing to the *current* module as base, and local variables the stack pointer SP (R14), a standalone program uses the memory area starting at memory location 0 as the *variable* space for global variables by default. This implies that if the standalone program overwrites that 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 affect the *same* memory region. In such a case, it's best to simply not declare any global variables (as exemplified in the Oberon *boot loader*).

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

The command *Boot.WriteFile BootLoad.rsc 512 BootLoad.mem* extracts the code section from the object file of the Oberon boot loader *and* converts it to a PROM file compatible with the specific hardware used. The first parameter is the name of the object file (input file). The second parameter is the size of the PROM code to be generated (number of opcodes converted). 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* than the *specified* code size, the code is zero-filled to the specified size.

Note that the command *Boot.WriteFile* 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). This implies that standalone programs cannot use string constants, type

[29] *See procedure ORG.Header*

extensions, type tests or type guards. Note also that neither the *module loader* nor the *garbage collector* are assumed to be present when a standalone program is executed. This implies that standalone programs cannot import other modules (except the pseudo module SYSTEM) or allocate dynamic storage using the predefined procedure NEW.

Since a standalone program does not import other modules, no access to external variables, procedures or type descriptors can occur. Thus, there is no need to *fix up* any instructions in the program code once it has been transferred to a particular memory location on the target system (as the *regular* module loader would do). It also means that the *static base* never changes during execution of a standalone program, i.e. neither the global module table nor the MT register are needed. 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*.

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 available online.

For example, 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 *BootLoad.mem*), use the command

```
data2mem -bm prom.bmm -bt ISE/RISC5Top.bit -bd BootLoad.mem -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 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 Original 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 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 *Boot.Link* and loaded as a sequence of *bytes* onto the boot area of the local disk, using the command *Boot.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 to which the boot loader will branch *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 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 *Boot.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 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 the partner program and send the results back.

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 files from the host system to the target 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*.