The Experimental Oberon System

Andreas Pirklbauer

15.9.2018

Experimental Oberon[1] is a revision of the Original Oberon[2] operating system and the Oberon-07 compiler. It contains a number of enhancements, including continuous fractional line scrolling with variable line spaces, enhanced viewer operations, multiple logical displays, safe module unloading, system building and maintenance tools, an enhanced Oberon-07 compiler with various new features, including a numeric case statement, exporting and importing of string constants, no access to intermediate objects in nested scopes, a dynamic heap allocation procedure NEW for fixed-length and open arrays, forward references and forward declarations of procedures, and module contexts. Some of these modifications are purely of experimental nature, while others serve the explicit purpose of exploring potential future extensions, for example to add support for touch display devices.

1. Continuous fractional line scrolling with variable line spaces

Continuous fractional line scrolling 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 welcome side effect, the initial learning curve for users new to the Oberon system and its user interface is *considerably* reduced.

2. Enhanced viewer operations

The basic viewer operations *Change* and *Modify* have been generalized to include pure vertical translations (without changing the viewer’s height), adjusting the top 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.

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.

[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 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).

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.

3. 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.

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 invoking *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].

[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.*

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 the modules containing the references that caused the removal to fail. If references exist *and* the *force* option /f is specified[5], the modules to be unloaded are initially removed only from the *list* of loaded modules, without releasing their associated memory. This effectively amounts to *renaming* them, allowing other modules with the same name to be reloaded again, without having to unload (from memory) earlier versions that are still referenced[6]. Such *hidden* modules are later physically removed from memory as soon as there are no more references to them[7]. The resulting module unload process is shown in Figure 1.

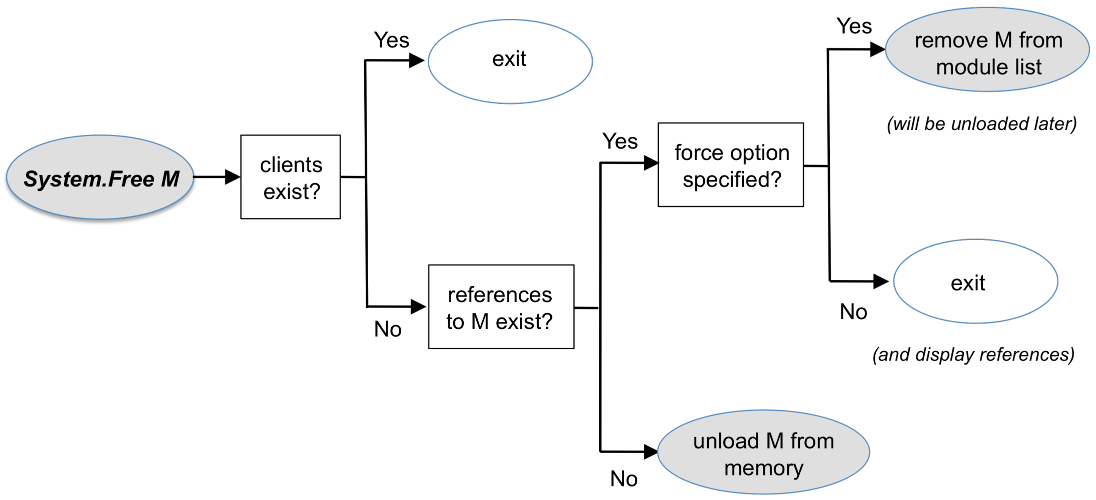


Figure 1: Safe module unloading in Experimental Oberon

To make the removal of no longer referenced (hidden) module data automatic, a new command *Modules.Collect* has been added to the Oberon background task handling garbage collection[8]. It checks all possible combinations of *k* modules chosen from *n* hidden modules for clients and references, and removes those module subgroups from memory that are no longer referenced. For added convenience, the tool commands *System.ShowRefs* and *System.ShowExternalRefs* can be used to identify all modules containing references to a given module or module group.

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.

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

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

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

[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.*

For example, older versions of a module's code can still be executed if they are referenced by static or dynamic procedure variables in other modules, even if a newer version of the module has been reloaded in the meantime[9]. Type descriptors also remain accessible to other modules for exactly as long as needed. This covers the important case where a structure rooted in a variable of base type T declared in a base module M contains elements of an extension T' defined in a client module M', which is unloaded[10]. Such elements typically contain both *type* references (type tags) and *procedure* references (installed handler procedures) referring to M'. This is common in the Oberon viewer system, for instance, where M is module *Viewers*.

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

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

When the user attempts to unload a module or module group, the unload command *selects* the modules to be unloaded using procedure *Modules.Select* and calls procedure *Modules.Check*. Clients are checked first. This is accomplished by simply verifying whether *unselected* modules import *selected* modules[12]:

```
PROCEDURE FindClients*(proc: ModHandler; VAR res: INTEGER);
  VAR mod, imp, m: Module; p, q: INTEGER; continue: BOOLEAN;
BEGIN res := 0; mod := root; continue := proc # NIL;
  WHILE continue & (mod # NIL) DO
    IF (mod.name[0] # 0X) & mod.selected & (mod.refcnt > 0) THEN m := root;
      WHILE continue & (m # NIL) DO
        IF (m.name[0] # 0X) & ~m.selected THEN p := m.imp; q := m.cmd;
          WHILE p < q DO imp := Mem[p];
            IF imp = mod THEN INC(res, proc(m, mod, continue)); p := q ELSE INC(p, 4) END
          END
        END ;
        m := m.next
      END
    END ;
    mod := mod.next
  END
END FindClients;
```

If clients exist, no further action is taken and the module unload command exits.

[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] Note that for t y p e references, it is actually possible to know at compile time, whether a module M may potentially lead to references from other modules at run time (but not whether it actually will): If M does not declare record types which are extensions T' of an imported type T, records declared in M cannot be inserted in a data structure rooted in a variable of an imported type T – precisely because they are not extensions of T (in the Oberon language, the assignment p := p' is allowed only if the type of p' is the same as the type of p or an extension of it).
[11] In Oberon, cyclic references can be created by pointers or procedure variables, whereas cyclic module imports are normally not allowed. However, through a tricky sequence of compilation and editing steps, it is in fact possible to c o n s t r u c t cyclic module imports, which cannot be detected by the compiler. But even though they can be created, such modules are not allowed to be loaded, as the module loader of Experimental Oberon – adopting the approach chosen in Original Oberon – would simply enter an endless recursion, eventually leading to a stack overflow or out-of-memory condition – a totally acceptable solution for such an artificially constructed case. But even if modules with cyclic module imports w e r e allowed to be loaded, Experimental Oberon would handle them correctly upon unloading, i.e. if no external clients or references exist, such a module group would simply be unloaded as a whole – as it should.
[12] 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.

If no 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* heap objects reachable by all other loaded modules pointing to descriptors of types declared in the modules to be unloaded, and *procedure variables* installed in *dynamic or static* objects of other modules referring to *static* procedures declared in the modules to be unloaded[13]. References from *dynamic* objects are checked using a conventional *mark-scan* scheme:

```
 1  PROCEDURE FindDynamicRefs*(type, proc: RefHandler; VAR resType, resProc: INTEGER; all: BOOLEAN);
 2    VAR mod: Module;
 3  BEGIN mod := root;
 4    WHILE mod # NIL DO
 5      IF (mod.name[0] # 0X) & ~mod.selected THEN Kernel.Mark(mod.ptr);
 6        IF ~all THEN Kernel.Scan(type, proc, mod.name, resType, resProc) END
 7      END ;
 8      mod := mod.next
 9    END ;
10    IF all THEN Kernel.Scan(type, proc, "", resType, resProc) END
11  END FindDynamicRefs;
```

During the *mark* phase, heap records reachable by all *named* global pointer variables of *all other* loaded modules are marked (line 5), 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 *scan phase* (line 6 or 10), implemented as a separate procedure *Scan* in module *Kernel*, 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 also whether *procedure variables* declared in these heap records refer to procedures declared in those same modules. An additional boolean parameter *all* allows the caller to specify whether the *mark* phase should first mark the heap records reachable by *all* other modules before entering the *scan* phase (used for module unloading), or whether the *mark* phase should mark each of the other modules *individually* before initiating the *scan* phase (used for enumerating the references to each module *individually*).

Finally, the check for procedure references is also performed for all *static* procedure variables:

```
PROCEDURE FindStaticRefs*(proc: RefHandler; VAR res: INTEGER);
  VAR mod: Module; pref, pvadr, r: LONGINT; continue: BOOLEAN;
BEGIN res := 0; mod := root; continue := proc # NIL;
  WHILE continue & (mod # NIL) DO
    IF (mod.name[0] # 0X) & ~mod.selected THEN
      pref := mod.pvr; pvadr := Mem[pref];
      WHILE continue & (pvadr # 0) DO r := Mem[pref]
        INC(res, proc(r, mod.name, continue));
        INC(pref, 4); pvadr := Mem[pref]
      END
    END ;
    mod := mod.next
  END
END FindStaticRefs;
```

[13] *An Oberon module can be viewed as a container of types, variables and procedures, where variables can be procedure-typed. Types can be declared as global types (in which case they can be exported and referenced in client modules) or as types local to a procedure (in which case they cannot be exported). Variables can be statically declared as global variables (allocated in the module area when a module is loaded) or as local variables (allocated on the stack when a procedure is called), or they can be dynamically allocated in the heap via the predefined procedure NEW. 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 references and static and dynamic procedure references need to be checked during module unloading for the following reasons. First, 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 should not prevent module unloading. In the Oberon operating system, such references will be handled by the garbage collector during a future garbage collection cycle, i.e. heap records reachable by the just unloaded modules and other still loaded modules are not collected, while heap records that were reachable only by the just unloaded modules will be collected – as they should. Thus, the handling of dynamic pointer references is delegated to the garbage collector. Finally, pointer variable references to static objects declared as global variables are only possible by resorting to low-level facilities and should be avoided (and, in fact, be disallowed).*

Note that the procedures *FindClients*, *FindDynamicRefs* and *FindStaticRefs* are expressed as *generic* traversal schemes which accept *parametric* handler procedures that are called for each encountered object. This allows these procedures to be used for *other* purposes as well, for example to *enumerate* the clients or references to a given module or module group.

In order to omit in module *Kernel* any reference to the module list rooted in module *Modules*, procedure *Kernel.Scan*[14] is also expressed as a *generic* heap scan scheme[15]:

```
 1  PROCEDURE Scan*(type, proc: Handler; s: ARRAY OF CHAR; VAR resType, resProc: INTEGER);
 2    VAR p, r, mark, tag, size, offadr, offset: LONGINT; continue: BOOLEAN;
 3  BEGIN p := heapOrg; resType := 0; resProc := 0; continue := (type # NIL) OR (proc # NIL);
 4    REPEAT mark := Mem[p+4];
 5      IF mark < 0 THEN (*free*) size := Mem[p]
 6      ELSE (*allocated*) tag := Mem[p]; size := Mem[tag];
 7        IF mark > 0 THEN (*marked*) Mem[p+4] := 0;
 8          IF continue THEN
 9            IF type # NIL THEN INC(resType, type(tag, s, continue)) END ;
10            IF continue & (proc # NIL) THEN offadr := tag + 16; offset := Mem[offadr];
11              WHILE offset # -1 DO (*pointers*) INC(offadr, 4); offset := Mem[offadr] END ;
12              INC(offadr, 4); offset := Mem[offadr];
13              WHILE continue & (offset # -1) DO (*procedures*) r := Mem[p+8+offset];
14                INC(resProc, proc(r, s, continue));
15                INC(offadr, 4); offset := Mem[offadr]
16              END
17            END
18          END
19        END
20      END ;
21      INC(p, size)
22    UNTIL p >= heapLim
23  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 9), while procedure *proc* is called for each procedure variable declared in the same record with (the address of) the *procedure* itself as argument (line 14). 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 8, 10, 13). The scan process itself continues, but only to *unmark* the remaining marked records (line 7).

Procedure *Modules.Check* uses procedures *FindClients, FindDynamicRefs* and *FindStaticRefs* by passing its private handler procedures *HandleClient* and *HandleRef*. Procedure *HandleClient* simply sets the variable parameter *continue* to *false* in order to stop the search for external clients (during module unloading, one only needs to know *whether* such clients exist):

```
PROCEDURE HandleClient(client, mod: Module; VAR continue: BOOLEAN): INTEGER;
BEGIN continue := FALSE; RETURN 1 (*stop when the first client is found*)
END HandleClient;
```

[14] *The original procedure Kernel.Scan has been renamed to Kernel.Collect, in analogy to the newly introduced procedure Modules.Collect.*
[15] *A simplified version scanning only record blocks (allocated via NEW(p), where p is a POINTER TO RECORD) is shown. The full implementation also covers array blocks in the heap.*

Procedure *HandleRef* checks whether the argument supplied by procedures *Kernel.Scan* (either a type tag or a procedure variable within a heap record) and *FindStaticRefs* (a global procedure variable) references *any* of the selected modules to be unloaded:

```
PROCEDURE HandleRef(x: LONGINT; s: ARRAY OF CHAR; VAR continue: BOOLEAN): INTEGER;
  VAR mod: Module; i: INTEGER;
BEGIN mod := root; i := 0;
  WHILE continue & (mod # NIL) DO (*check references to selected modules*)
    IF (mod.name[0] # 0X) & mod.selected & (mod.data <= x) & (x < mod.imp) THEN
      i := 1; continue := FALSE (*stop when the first reference is found*)
    END ;
    mod := mod.next
  END ;
  RETURN i
END HandleRef;
```

We emphasize that procedure *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 written such that it can correctly handle *both* visible *and* hidden modules in the module data structure rooted in module *Modules*.

Implementation considerations and prerequisites

In order to make the outlined validation pass possible, type descriptors of *dynamic* objects[16] allocated in the *heap* as well as the descriptors of *global* module data located 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 Original Oberon system (MacOberon)[17]. The resulting run-time representation of a *dynamic* record and its associated type descriptor is shown in Figure 2.

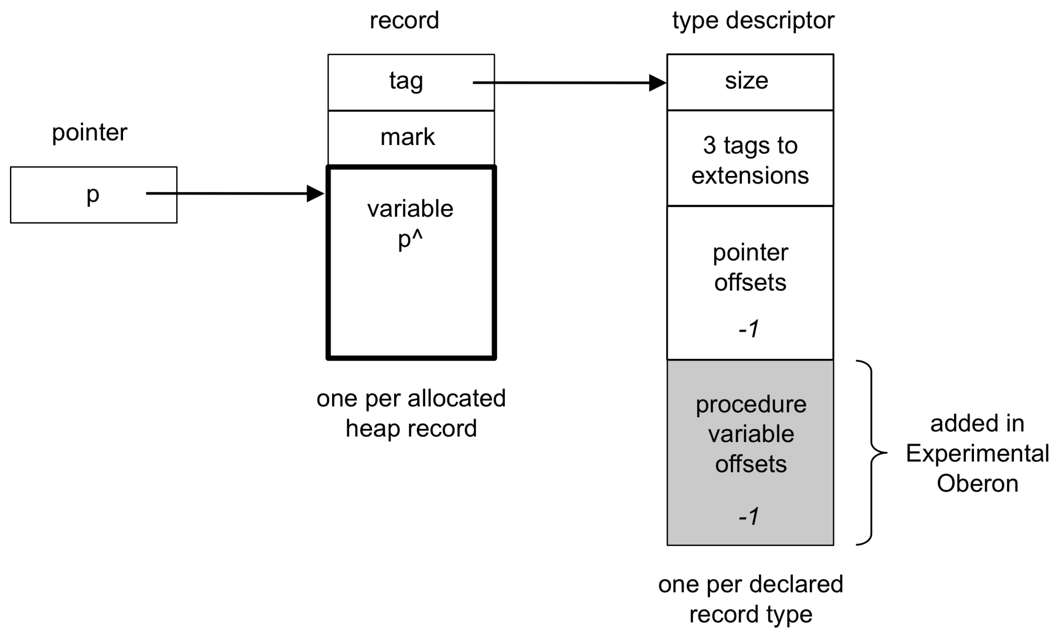

record
type descriptor
tag
size
pointer
mark
3 tags to extensions
p
variable p^
pointer offsets
-1
one per allocated heap record
procedure variable offsets
-1
added in Experimental Oberon
one per declared record type

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

[16] *See chapter 8.2, page 109, of the book Project Oberon 2013 Edition for a detailed description of an Oberon type descriptor. A type descriptor contains certain information about dynamically allocated records that is s h a r e d by all allocated objects of the same record type (such as its size, information about type extensions and the offsets of pointer fields within the record).*
[17] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*

Note that this run-time layout requires the *scan* phase of reference checking to traverse over the list of *pointer offsets* in the type descriptor of *each* marked heap record. To avoid this, one could therefore use a slightly different layout, where *procedure variable offsets* are *prepended* (rather than appended) to the fields of each type descriptor[18]. We have refrained from introducing this (minor) improvement, as the resulting 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 different purpose than the list of procedure variable offsets used for reference checking in Experimental Oberon)[19]. A comparison of the two variants showed that the overhead in our implementation is negligible.

The modified descriptors also contain the offsets of *hidden* (not exported) procedure variables, enabling the module *unload* command to check *all* possible procedure variable locations in the entire system for possible procedure references to the modules to be unloaded.

The compiler generating these descriptors, the format of the object files containing them and the module loader transferring them from object file into memory have been adjusted accordingly.

To make the offsets of hidden *procedure variables* in exported record types available to client modules, symbol files have also been adapted to include them. An importing module may, for example, declare a global variable of an imported record type, which contains *hidden* procedure variables, or declare a record type that contains or extends an imported one. We recall that in Original Oberon, hidden *pointers*, although not exported and therefore invisible in client modules, are included in symbol files because their offsets are needed for garbage collection[20]. Similarly, in Experimental Oberon, the locations of visible *and* hidden procedure variables are needed for reference checking during module unloading, as shown in the following example.

```
MODULE M0;
  TYPE Proc* = PROCEDURE; Rec* = RECORD p*, q: Proc END ;   (*q is a hidden field*)
  PROCEDURE Q*(r: Rec); BEGIN r.q END Q;
  PROCEDURE Init*(VAR r: Rec; p, q: Proc); BEGIN r.p := p; r.q := q END Init;
END M0.

MODULE M1;
  IMPORT M0;
  VAR r: M0.Rec;
  PROCEDURE Init*(p, q: M0.Proc); BEGIN M0.Init(r, p, q) END Init;
END M1.

MODULE M2;
  IMPORT M1;
  PROCEDURE Q; BEGIN END Q;
  PROCEDURE Set1*; BEGIN M1.Init(NIL, NIL) END Set1;
  PROCEDURE Set2*; BEGIN M1.Init(NIL, Q) END Set2;
END M2.

M2.Set1  System.Free M2 ~       … can unload M2
M2.Set2  System.Free M2 ~       … can't unload M2 (as M2.Q is referenced in global procedure variable M1.r.q)
```

Here the global record *r* declared in module *M1* contains a hidden field *M1.r.q* that is accessible only in the exporting module *M0*. If another module *M2* installs a procedure *M2.Q* in this hidden

[18] *Implementing this would require only a one-line change to both the compiler (procedure ORG.BuildTD) and the scan phase of reference checking (procedure Kernel.Scan).*
[19] *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).*
[20] *See chapter 12.6.2, page 41, of the book Project Oberon 2013 Edition for a description of why the offsets of hidden pointers are needed during garbage collection.*

record field, for example by using an installation procedure *Init* provided by *M0* and called in *M1*, a procedure variable reference from *M1* to *M2* is created, with the (intended) effect that module *M2* can no longer be unloaded.

Hidden *pointers* are included in symbol files without their names, and their (imported) type in the symbol table is of the form *ORB.NilTyp*. Hidden *procedure variables* are also included in symbol files without their names, and their symbol table type is of the form *ORB.NoTyp*[21].

These changes have been implemented by extending the procedures *ORB.FindHiddenPointers*, *ORG.FindPtrFlds*, *ORG.NofPointers* and *ORG.FindPointers* and appropriately renaming them to *ORB.FindHiddenFields*, *ORG.FindRefFlds*, *ORG.NofRefs* and *ORG.FindRefs*.

Assessment of the chosen approach

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. In addition, modules now also contain an *additional* section in the module block containing the offsets of global procedure variables. However, since there exists only one type descriptor per declared record *type* rather than one per allocated heap *record*, and global procedure variables tend to be rare, the additional memory requirements are negligible[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 in the *mark* phase *all* modules to be unloaded are checked for references during the *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. Thus, the presented solution, which was mainly chosen for its simplicity, appears to be amply sufficient for most practical purposes.

Alternatives considered

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 (it essentially establishes a "return path" for later steps in the mark phase). As a consequence, one

[21] *This is acceptable because for record fields, the types ORB.NilTyp and ORB.NoTyp are not used otherwise.*

[22] *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 fields in type descriptors. We refrained 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 in Oberon 2013). 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 record operations more complex. For example, assignments to subrecords would become less natural, as their fields would no longer be placed in a contiguous section in memory. Second, the memory savings in type descriptors would be marginal, given that there exists only one type descriptor per declared record t y p e rather than one per allocated heap r e c o r d. Most applications are 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 usually only one per t y p e of viewer. 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 implement this refinement would be hard to justify.*

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. Although this would be straightforward to implement, 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. In sum, this potential alternative would not improve performance signifcantly.

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 in turn could 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 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 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.

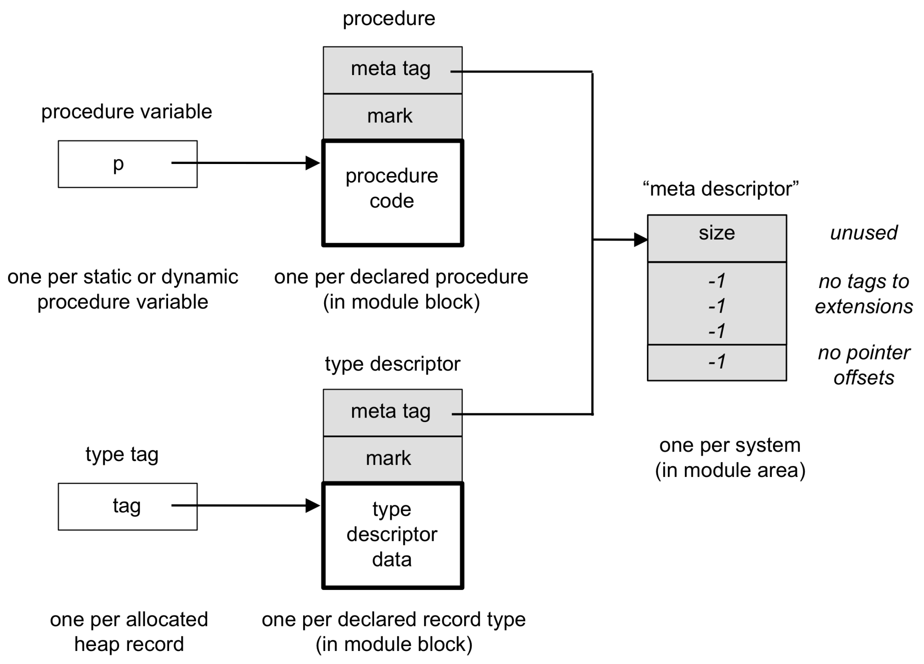
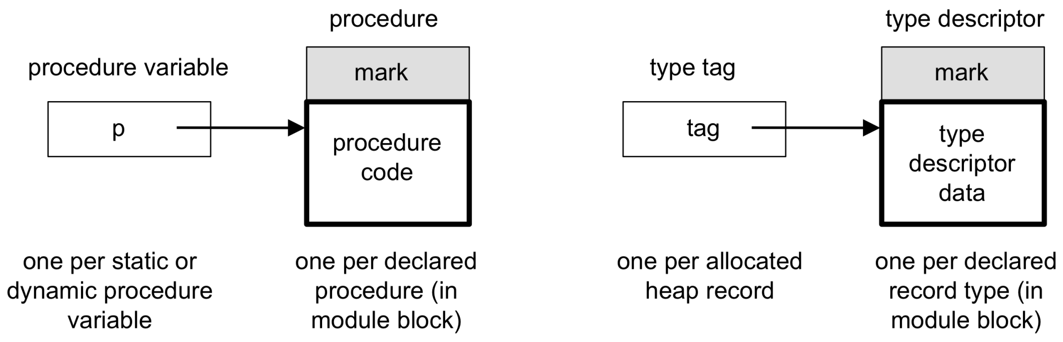

procedure
meta tag
mark
procedure variable
p
procedure code
one per static or dynamic procedure variable
one per declared procedure (in module block)
“meta descriptor”
size
unused
-1
-1
-1
no tags to extensions
-1
no pointer offsets
one per system (in module area)
type descriptor
meta tag
mark
type tag
tag
type descriptor data
one per allocated heap record
one per declared record type (in module block)

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.



procedure
type descriptor
procedure variable
mark
type tag
mark
p
procedure code
tag
type descriptor data
one per static or dynamic procedure variable
one per declared procedure (in module block)
one per allocated heap record
one per declared record type (in module block)

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. In sum, such an *extended* mark phase would not only touch dynamic objects in the *heap*, but also static objects in the *module blocks*, with the effect that after the *mark* phase *all* referenced dynamic *and* static objects are already marked, not just heap objects.

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 offer 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 without additional action.

Finally, a comparison of the code required to implement the various alternatives showed that our solution is *by far* the simplest: the combined implementation cost of *all* modifications to the runtime representation of type descriptors and descriptors of global module data, the object and symbol file formats and the module loader, is only about 15 source lines of code[23], while the *reference checking* phase itself amounts to less than 75 lines (the *total* implementation cost to realize *safe module unloading*, including the ability to unload module *groups* and the automatic collection of no longer referenced hidden modules, amounts to about 250 source lines of code).

5. Oberon system building and maintenance tools

A minimal version of the Oberon system *building tools*, as described in chapter 14 of the book *Project Oberon 2013 Edition*, has been added. They provide the necessary tools to establish

[23] *See procedures ORB.InType (+ 1 line), ORB.FindHiddenRefs (+ 1 line), ORG.BuildTD (+ 1 line), ORG.Close (+ 7 lines) and Modules.Load (+ 4 lines).*

the prerequisites for the *regular* Oberon startup process. The Oberon system building tools[24] described below exist for both *Original Oberon 2013*[25] and *Experimental Oberon*[26].

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

```
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, invoke the Oberon *boot linker* as follows:

```
Boot.Link Modules ~          … generate a pre-linked binary file of the "regular" boot file (Modules.bin)
Boot.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. For the *regular* boot file loaded onto the boot area[27] 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[28]. In the case of the *regular* boot file, this information is used by the Oberon *boot loader (BootLoad.Mod)* – 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 format of the boot file is *defined* to exactly mirror the standard Oberon storage layout. 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 body 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.

To load 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):

```
Boot.Load Modules.bin ~      … load the "regular" boot file onto the boot area of the local disk (sectors 2-63)
```

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 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.

[24] *http://www.github.com/andreaspirklbauer/Oberon-building-tools*
[25] *http://www.projectoberon.com*
[26] *http://www.github.com/andreaspirklbauer/Oberon-experimental*
[27] *Sectors 2-63 in Original Oberon 2013 and Experimental Oberon (by default)*
[28] *Location 16 in the case of Original Oberon 2013 and Experimental Oberon*

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 last step corresponds to starting the target system with the switch set to *“serial link”.*

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 target system’s file directory. The command *ORC.SR* (“send, then receive sequence of items”) remotely initiates a command on the target system, then receives the command’s response, if any. A command is specified by an integer followed by parameters, which can be numbers, names, strings or characters.

The command *ORC.Send* transfers files from the host to the target system. It is typically used to transfer the files required to start Oberon on the target system. The command *ORC.SR 100 Modules.bin* loads the just transferred *regular* boot file *Modules.bin* onto the boot area of the target system. The command *ORC.Receive* receives files from the target system.

To open a tool viewer on the host system containing the commands described below:

```
Edit.Open Build.Tool
```

To *generate* the necessary binaries (on the host system) 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

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

To *build* a new Oberon system on the target system from scratch, restart it over the data link, then execute the following commands on the *host* system:

```
ORC.Load Oberon0.bin ~       ... load the Oberon-0 command interpreter over the data link to the target system and start it

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

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

ORC.Send Input.rsc Display.rsc Viewers.rsc
    Fonts.rsc Texts.rsc Oberon.rsc
    MenuViewers.rsc TextFrames.rsc
    System.rsc System.Tool
    Oberon10.Scn.Fnt
    Modules.bin
    BootLoadDisk.rsc
    Oberon0.rsc Oberon0Tool.rsc
    Edit.rsc PCLink1.rsc
    ORP.rsc ORG.rsc
    ORB.rsc ORS.rsc ORTool.rsc ~       … send the required (and some additional) files 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
```

To *start* the newly built Oberon operating system on the target system, either *manually* set the corresponding switch *on* the target system to "disk" and press the reset button, or execute the following command on the host system:

```
ORC.SR 102 BootLoadDisk.rsc ~      ... reboot the target system from the local disk (initiate 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 (this 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 system running on the target system only takes a few seconds.

To re-enable the ability to transfer files, execute the following command on the *target* system:

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

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

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 to a target system, it can also be loaded as a *regular* Oberon module by activating the command *Oberon0Tool.Run*. In the latter case, its main loop is simply not started (if it were started, it would block the system).

To start the Oberon-0 command interpreter as a background task on the target system:

```
PCLink1.Stop        ... stop the PCLink1 background task if it is running (as it 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 execute 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.

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 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:

Oberon0Tool.Stop	*... stop the Oberon-0 command interpreter background task*

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 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*[29] below. See chapter 6 of the book *Project Oberon 2013 Edition* 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*)
  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[30].

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

[29] *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).*
[30] *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, which would reduce disk access time even further. However, this would increase the complexity of the boot loader considerably. Such a "code-generating boot loader" would translate the slim binary boot file "on the fly" into the native object code of the target system during the boot load phase. However, the boot loader is an intentionally tiny program permanently resident in the computer's read-only store and should not be abused for such complex tasks with questionable benefit. Indeed, experiments have shown that with the advent of fast secondary storage devices, e.g. solid-state drives (2010s), the speedup has become negligible, compared with systems that use a rotating disk or floppy disk as the boot device (1990s). Also note that the boot process is already instantaneous.*

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.

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 processor register holding the *stack pointer* SP (R14) is also initialized to a fixed value in the initialization sequence of a standalone program[31]:

```
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.

If a standalone program wants to use a *different* memory area as its *stack* space, it can adjust the register holding the *stack pointer SP* (R14) using the low-level procedure SYSTEM.LDREG.

```
MODULE* M;
  IMPORT SYSTEM;
  CONST SP = 14; StkOrg = -1000H;
BEGIN SYSTEM.LDREG(SP, StkOrg); …
END M.
```

Note also that a standalone program uses the memory area starting at memory location 0 as the *variable* space for global variables. 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*).

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, compile it with the *regular* Oberon compiler:

```
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:

[31] *See procedure ORG.Header*

```
Boot.WriteFile BootLoad.rsc 512 BootLoad.mem ~     ... extract the code section from the object file in PROM format
```

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, one number per line. If the *actual* code size is *less* than 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
...
A0100000
40000000
C7000000                    ... line 384
00000000
00000000
...
00000000                    ... line 512
```

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 extensions, type tests or type guards.

Note further 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.

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 *Boot.WriteFile*, 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 *Boot.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.

```
Boot.WriteCode BootLoad.rsc BootLoad.code ~     … extract the code section from the object file in binary format
```

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*[32] can be used. For further details, the reader is referred to the pertinent documentation available online, 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
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 *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
```

[32] *See Appendix B for the syntax of the fpgaprog command*

6. Enhanced Oberon-07 compiler

The official Oberon-07 compiler has been enhanced with various new features, including

- Numeric case statement
- Exporting and importing of string constants
- No access to intermediate objects within nested scopes
- Dynamic heap allocation procedure NEW for fixed-length and open arrays
- Forward references and forward declarations of procedures
- Module contexts

The combined *total* implementation cost of these new features in source lines of code (*sloc*)[33], broken down by Oberon-07 compiler module, is shown below:

Compiler module	**Original Oberon**	**Experimental Oberon**	**Difference**	**Percent**
ORS (scanner)	293	293	0	0 %
ORB (base)	394	390	-4	- 1.0 %
ORG (generator)	984	1043	59	+ 6.0 %
ORP (parser)	949	1044	95	+ 10.0 %
Total	**2620**	**2770**	**150**	**+ 5.7 %**

Numeric case statement

The official Oberon-07 language report[34] allows numeric CASE statements, which are however not implemented in the official release[35]. The modified Oberon-07 compiler brings the compiler in line with the language report, i.e. it also allows *numeric* CASE statements *(CASE integer OF, CASE char OF)* in addition to *type* CASE statements *(CASE pointer OF, CASE record OF)*[36].

Implemented syntax:

```
CaseStatement  =  CASE expression OF case {"|" case} [ELSE StatementSequence] END.
case           =  CaseLabelList ":" StatementSequence.
CaseLabelList  =  LabelRange {"," LabelRange}.
LabelRange     =  label [".." label].
label          =  integer | string | qualident.
```

Example:

```
CASE i OF
      2..5:  k := 1                 (*lower case label limit = 2*)
    | 8..10:  k := 2
    | 13..15:  k := 3
    | 28..30, 18..22:  k := 4
    | 33..36, 24:  k := 5           (*higher case label limit = 36*)
END
```

The essential property of the *numeric* CASE statement is that it represents a *single*, indexed branch, which selects a statement sequence from a set of cases according to an index value. This is in contrast to a cascaded conditional statement which contains multiple branches. Case statements are recommended only if the set of selectable statements is reasonably large.

[33] *Not counting empty lines and about 100 additional lines of code in modules Kernel, Modules and System to complement the implementations of procedure NEW(p, len) and module contexts.*
[34] *http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf (as of 3.5.2016)*
[35] *http://www.projectoberon.com*
[36] *http://github.com/andreaspirklbauer/Oberon-numeric-case-statement*

The compiler constructs a *jump table* of branch statements (containing the branch distances as operands) to the various component statements, leading to a *constant* number of instructions needed for any selection in a numeric CASE statement. The selection ("switch") is generated by procedure *ORG.CaseHead*, using a jump table generated by procedure *ORG.CaseTail.*

Jump tables are located in the *code* section of a module and are addressed relative to the *static base* (SB) of the module. The offset to the static base is restricted to a maximum of 64KB (one could also have placed a table of jump table offsets in the module's area for constants, which is sometimes used if the processor features an indexed branch instruction, as is typically the case in CISC processors, but not in the RISC processor used by FPGA Oberon).

The following rules and restrictions apply:

- Case labels must have values between 0 and 255. This makes them ideally suitable for use with CHAR selectors, without imposing any serious restrictions on integer selectors.
- Case expression values that are between 0 and the highest case label limit, but do not correspond to any case label value in the source text, do *not* cause an error termination, i.e. we treat such events as "empty" actions, unless an ELSE clause is present.
- Case expression values that are *higher* than the highest case label limit *do* cause an error termination (trap 1), regardless of whether an ELSE clause is present. This additional rule has been introduced in order to keep the size of the generated jump table reasonably small, which helps optimize the common case where fewer than 256 entries are actually needed.
- Case statements do *not* allow for the empty case, unlike in the official Oberon-07 language, where a *case* is defined using the following grammatical production *with* brackets [and]

  ```
  case = [CaseLabelList ":" StatementSequence].
  ```

  The *inclusion* of the empty case would allow the insertion of superfluous bars similar to the insertion of superfluous semicolons between statements, which we view as .. superfluous ☺

Note that in our implementation, the ELSE clause is also implemented for both the *numeric* and the *type* CASE statement, even though it is not part of the Oberon-07 language definition.

At first glance, this may seem to be somewhat in contradiction with conventional wisdom in programming language design, which suggests that the ELSE clause in a language construct should normally be reserved for the *exceptional* cases only, i.e. those that are neither numerous among the possible cases in the source text nor do occur frequently at run time.

In addition, the presence of an ELSE clause in the source text of a program might potentially obfuscate the thinking of the programmer, for example if program execution falls through to the ELSE clause unintentionally. In general, language constructs that allow program execution to continue or “fall through” to the next case or a “default” case are not recommended.

We have nevertheless opted to re-introduce the ELSE clause for the following reason. In our implementation, case expression values *higher* than the *highest* case label limit cause an error termination. This additional rule effectively limits the *total* range of possible case label values, making it easier to align it with the actual use case. Consequently, if a CASE statement is well designed, the ELSE clause is much more likely to *actually* be used for the *exceptional* cases – as it should. In addition, there exist genuine examples where an ELSE clause appears useful.

Exporting and importing of string constants

The official Oberon-07 language report allows exporting and importing of string constants, but the compiler does not support it. The modified Oberon-07 compiler implements this feature[37].

Example:

```
MODULE M;
  CONST s* = "This is a sample string";            (*exported string constant*)
END M.

MODULE N;
  IMPORT Texts, Oberon, M;
  VAR W: Texts.Writer;

  PROCEDURE P*;
  BEGIN Texts.WriteString(W, M.s); Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END P;

BEGIN Texts.OpenWriter(W)
END N.
```

Exported *string constants* are treated like (pre-initialized) exported *variables*. The symbol file contains the string's *export number* and *length*, but not the string itself. The object file contains the actual string, together with its *location* in the exporting module's area for *data*.

No access to intermediate objects within nested scopes

The official Oberon-07 language report disallows access to *all* intermediate objects from within nested scopes. The modified compiler brings the compiler in line with the language report, i.e. it also disallows access to intermediate *constants* and *types* within nested scopes, not just access to intermediate *variables*[38].

Like the official compiler, the modified compiler adopts the convention to implement *shadowing through scope*. This means when two objects share the same name, the one declared at the narrower scope hides, or shadows, the one declared at the wider scope. In such a situation, the shadowed element is not available in the narrower scope.

The official Oberon-07 compiler already issues an error message, if intermediate *variables* are accessed within nested scopes, *regardless* of whether a global variable with the same name exists or not (e.g. in line 23 of the program below). With the modified compiler, the same error message is now *also* issued for intermediate *constants* (line 19) and *types* (lines 14, 16).

Example:

```
 1  MODULE Test;
 2    CONST C = 10;              (*global constant C - shadowed in Q*)
 3    TYPE G = REAL;             (*global type G - NOT shadowed in Q*)
 4        T = REAL;              (*global type T - shadowed in Q*)
 5    VAR A,                     (*global variable A - NOT shadowed in Q*)
 6        B: INTEGER;            (*global variable B - shadowed in Q*)
 7
```

[37] http://github.com/andreaspirklbauer/Oberon-importing-string-constants
[38] http://github.com/andreaspirklbauer/Oberon-no-access-to-intermediate-objects

```
  PROCEDURE P;                 (*global procedure P*)
    PROCEDURE Q;               (*intermediate procedure Q*)
      CONST C = 20;            (*intermediate constant C - shadows global constant C*)
      TYPE T = INTEGER;        (*intermediate type T - shadows global type T*)
      VAR B: INTEGER;          (*intermediate variable B - shadows global variable B*)

      PROCEDURE R(x: T): T;    (*access to int. type T allowed in original, NOT allowed in modified compiler*)
        VAR i: INTEGER;
          q: T;                (*access to int. type T allowed in original, NOT allowed in modified compiler*)
          g: G;                (*access to global type G (not shadowed) allowed in BOTH compilers*)
      BEGIN (*R*)
        i := C;                (*access to int. constant C allowed in original, NOT allowed in modified compiler*)
        P;                     (*access to global procedure P allowed in BOTH compilers*)
        Q;                     (*access to intermediate procedure Q allowed in BOTH compilers*)
        i := A;                (*access to global variable A (not shadowed) allowed in BOTH compilers*)
        i := B;                (*access to intermediate variable B NOT allowed in either compiler*)
        RETURN i
      END R;
    END Q;
  END P;

END Test.
```

Dynamic heap allocation procedure NEW for fixed-length and open arrays

If *p* is a variable of type *P = POINTER TO T*, a call of the predefined procedure *NEW* allocates a variable of type *T* in free storage at run time. The type *T* can be a record type *or* an array type.

If T is a record type or an array type with *fixed* length, the allocation has to be done with

NEW(p)

If T is an *open* array type, the allocation has to be done with

NEW(p, len)

where *T* is allocated with the length given by the expression *len*, which must be an integer type.

In either case, a pointer to the allocated variable is assigned to *p*. This pointer *p* is of type *P*, while the referenced variable *p^* (pronounced *p-referenced*) is of type *T*.

If *T* is a record type, a field *f* of an allocated record *p^* can be accessed as *p^.f* or as *p.f*. If *T* is an array type, the elements of an allocated array *p^* can be accessed as *p^[0]* to *p^[len-1]* or as *p[0]* to *p[len-1]*, i.e. record and array selectors imply dereferencing.

If *T* is an array type, its element type can be a *record*, *pointer*, *procedure* or a *basic* type (BYTE, BOOLEAN, CHAR, INTEGER, REAL, SET), but not an *array* type (no multi-dimensional arrays).

Example:

```
MODULE Test;
  TYPE R = RECORD x, y: INTEGER END ;

    A = ARRAY OF R;                (*open array*)
    B = ARRAY 20 OF INTEGER;       (*fixed-length array*)

    P = POINTER TO A;                    (*pointer to open array*)
    Q = POINTER TO B;                    (*pointer to fixed-length array*)

  VAR a: P;
    b: Q;

  PROCEDURE New1*;
  BEGIN NEW(a, 100);  a[53].x := 1
  END New1;

  PROCEDURE New2*;
  BEGIN NEW(b);  b[3] := 2
  END New2;

END Test.
```

If the variable passed as parameter *p* of *NEW(p)* or *NEW(p, len)* is a *named* global pointer variable pointing to a record or an array type, the object referenced by *p* will be marked during the *mark* phase of the garbage collector and collected during the *scan* phase, if it is no longer referenced, i.e. both record *and* array blocks are garbage-collected.

Allocating dynamic arrays requires a modified version of module *Kernel*, which introduces a new *kind* of heap block (*arrayblk* in addition to *recordblk*)[39]. The implementation of garbage collection on open arrays is similar to implementations in earlier versions of the Original Oberon system[40].

The following rules and restrictions apply:

- Bounds checks on *fixed-length* arrays are performed at *compile* time.
- Bounds checks on *open* arrays are performed at *run* time.
- If *P* is of type *P = POINTER TO T*, the type *T* must be a *named* record or array type[41].

Forward references and forward declarations of procedures

The modified Oberon compiler implements forward references of procedures for 2 use cases[42]:

Use case A: To make references among nested procedures more efficient:

If a procedure Q which is local to another procedure P refers to the enclosing procedure P, as in

```
PROCEDURE P;
  PROCEDURE Q;
  BEGIN (*body of Q*) P            (*forward reference from Q to P, as the body of P is not compiled yet *)
  END Q;
BEGIN (*body of P*) …
END P;
```

then the official Oberon-07 compiler generates the following code:

```
20   P'    BL  10          … forward branch to line 31 (the body of P)
21   Q     body of Q
```

[39] A call to NEW(p, n), where p is a pointer to an array of BYTE, is equivalent to a call to the low-level procedure SYSTEM.NEW(p, n) provided in some implementations of the Original Oberon system to allocate a storage block of n bytes ("sysblk"). In our implementation, no such additional low-level procedure exported by the pseudo-module SYSTEM is necessary.
[40] See, for example, "Oberon Technical Notes: Garbage collection on open arrays", J. Templ, ETH technical report, March 1991.
[41] Restricting pointers to n a m e d arrays is consistent with the official Oberon-07 compiler, which only allows pointers to n a m e d records.
[42] http://github.com/andreaspirklbauer/Oberon-forward-references-of-procedures

```
     ...                    … any calls from Q to P are BACKWARD jumps to line 20 and from there forward to line 31
31   P      body of P
```

whereas the modified compiler generates the following, more efficient, code:

```
20   Q      body of Q
     ...                    … any calls from Q to P are FORWARD jumps to line 30, fixed up when P is compiled
30   P      body of P
```

i.e. it does not generate an extra forward jump in line 20 around Q to the body of P and backward jumps from Q to line 20. With the official compiler, the extra BL instruction in line 20 is generated, so that Q can call P (as the body of Q is compiled before the body of P).

Use case B: To implement forward declarations of procedures:

Forward declarations of procedures have been eliminated in Oberon-07, as they can always be eliminated from any program by an appropriate nesting or by introducing procedure variables[43].

Whether forward declarations of procedures *should* be re-introduced into the Oberon language, can of course be debated. Here, we have re-introduced them for three main reasons:

- Direct procedure calls are more efficient than using procedure variables.
- Legacy programs that contain forward references of procedures are now accepted again.
- Introducing forward declarations of procedures added only about 10 lines of source code.

Forward declarations of procedures are implemented in exactly the same way as in the original implementation before the Oberon-07 language revision, i.e.,

- They are explicitly specified by ^ following the symbol PROCEDURE in the source text.
- The compiler processes the heading in the normal way, assuming its body to be missing. The newly generated object in the symbol table is marked as a forward declaration.
- When later in the source text the full declaration is encountered, the symbol table is first searched. If the given identifier is found and denotes a procedure, the full declaration is associated with the already existing entry in the symbol table and the parameter lists are compared. Otherwise a multiple definition of the same identifier is present.

Note that our implementation *both* global and local procedures can be declared forward.

Example:

```
MODULE M;
  PROCEDURE^ P(x, y: INTEGER; z: REAL);          (*forward declaration of P*)

  PROCEDURE Q*;
  BEGIN P(1, 2, 3.0)                             (*Q calls P which is declared forward*)
  END Q;

  PROCEDURE P(x, y: INTEGER; z: REAL);           (*procedure body of P*)
  BEGIN …
  END P;

END M.
```

[43] *See section 2 of www.inf.ethz.ch/personal/wirth/Oberon/PortingOberon.pdf*

Module contexts

The modified Oberon-07 compiler introduces *module contexts*, originally introduced for the A2 operating system[44]. A module context acts as a single-level name space for modules. It allows modules with the same name to co-exist within different contexts. The syntax is defined as:

```
Module   =  MODULE ident [IN ident] “;”
Import   =  IMPORT ident [“:=” ident ] [IN ident] “;”
```

In the first line, the optional identifier specifies the name of the context the module belongs to. In the second line, it tells the compiler in which context to look for when importing modules.

Module contexts are implemented as follows:

- Module contexts are specified within the *source* text of a module, as an optional feature. If a context is specified, the name of the source file itself typically (but not necessarily) contains a prefix indicating its module context, for example *Oberon.Texts.Mod* or *EO.Texts.Mod*.
- If a module context is specified in the source text, the compiler will automatically generate the output files *contextname.modulename.smb* and *contextname.modulename.rsc*, i.e. the module contexts of symbol and object files is encoded in their file names.
- If no module context is specified in the source text, the output files *modulename.rsc* and *modulename.smb* are generated.
- In Experimental Oberon, the module context "EO" is implicitly specified at run time. Thus, the module loader will first look for the file *EO.modulename.rsc*, then for *modulename.rsc*.
- A module belonging to a context can only import modules belonging to the same context, and vice versa (implementation restriction).

Example:

```
MODULE Test1 IN EO;  (*Experimental Oberon*)
  IMPORT Texts, Oberon;
  VAR W: Texts.Writer;

  PROCEDURE Go1*;
  BEGIN Texts.WriteString(W, "Hello from module Test1 in context EO (Experimental Oberon)");
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END Go1;

BEGIN Texts.OpenWriter(W)
END Test1.


MODULE Test2 IN EO;  (*Experimental Oberon*)
  IMPORT Texts, Oberon, Test1 IN EO;
  VAR W: Texts.Writer;

  PROCEDURE Go2*;
  BEGIN Texts.WriteString(W, "Hello from module Test2 in context EO (Experimental Oberon)");
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END Go2;

BEGIN Texts.OpenWriter(W)
END Test2.
```

[44] *http://www.ocp.inf.ethz.ch/wiki/Documentation/Language?action=download&upname=contexts.pdf*

Appendix A: 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* (or remotely using the command *ORC.SR 100* during the initial system building phase). 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*.

Appendix B: Syntax of the *fpgaprog* command

```
Usage: fpgaprog [-h] [-v] [-j] [-d] [-f <bitfile>] [-b <bitfile>]
           [-s e|v|p|a] [-c] [-C] [-r]]
           [-a <addr>:<binfile>]
           [-A <addr>:<binfile>]

  -h                    print this help
  -v                    verbose output
  -j                    Detect JTAG chain, nothing else
  -d                    FTDI device name
  -f <bitfile>          Main bit file
  -b <bitfile>          bscan_spi bit file (enables spi access via JTAG)
  -s [e|v|p|a]          SPI Flash options: e=Erase Only, v=Verify Only,
                        p=Program  Only or a=ALL (Default)
  -c                    Display current status of FPGA
  -C                    Display STAT Register of FPGA
  -r                    Trigger a reconfiguration of FPGA
  -a <addr>:<binfile>   Append binary file at addr (in hex)
  -A <addr>:<binfile>   Append binary file at addr, bit reversed
```