The system building tools

for FPGA Oberon and Experimental Oberon

Andreas Pirklbauer

31.12.2018

The Oberon system *building tools*[1], as outlined in chapter 14 of the book *Project Oberon 2013 Edition* and described below for *FPGA Oberon 2013*[2] and *Experimental Oberon*[3], provide the necessary tools to establish the prerequisites for the *regular* Oberon startup process.

1. Overview of the Oberon system startup process

When the power to a computer is turned on or the reset button is pressed, the computer's boot *firmware* is activated. The boot firmware is a small standalone program permanently resident in the computer's read-only store, such as a read-only memory (ROM) or a programmable read-only memory (PROM). This read-only store is sometimes called the *platform flash*.

In Oberon, the boot firmware is called the *boot loader*, as its main task is to *load* a valid boot file (a pre-linked binary file containing a set of compiled Oberon modules) from a valid *boot source* into memory and then transfer control to its top module (the module that directly or indirectly imports all other modules in the boot file). Then its job is done until the next time the computer is restarted or the reset button is pressed, i.e. the boot loader is not used after booting.

There are currently two valid boot sources in Oberon: a local disk, realized using a Secure Digital (SD) card in Oberon 2013, and a communication link, realized using an RS-232 serial line. The default boot source is the local disk. It is used by the regular Oberon boot process each time the computer is powered on or the reset button is pressed. There are two valid *boot file formats* in Oberon: the "regular" boot file format used for booting from the local disk, and the "build-up" boot file format used for booting over a data link. See the appendix for a detailed description of these data formats.

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

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

[1] *http://www.github.com/andreaspirklbauer/Oberon-building-tools*
[2] *http://www.projectoberon.com*
[3] *http://www.github.com/andreaspirklbauer/Oberon-experimental*

2. Creating the Oberon system building tools

To create the Oberon system builing tools:

```
ORP.Compile Boot.Mod/s ~          # generate the boot linker/loader and system building tool
```

3. Creating a valid Oberon boot file

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 PCLink0.Mod Oberon0.Mod ~               # additional modules for the "build-up" boot file
```

The boot linker (procedure *Boot.Link*) links a set of object files together and generates a valid boot file from them. It can be used to either generate the *regular* boot file to be loaded onto the boot area[4] of a disk or the *build-up* boot file sent over a data link to a target system:

```
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, this is typically module *Modules*, the top module of the *inner core* of the Oberon system. For the *build-up* boot file, it is usually module *Oberon0*, a command interpreter mainly used for system building purposes. The boot linker automatically includes all modules that are directly or indirectly imported by the specified top module. It 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[5]. 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 boot linker 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.

4. Updating the boot area of the local disk with a new Oberon boot file

The command *Boot.Load* loads a valid *boot file*, as generated by the command *Boot.Link*, onto the boot area of the local disk, one of the two valid *boot sources* (the other is the data link):

```
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

[4] *Sectors 2-63 in FPGA Oberon 2013 and Experimental Oberon (by default)*
[5] *Location 16 in the case of FPGA Oberon 2013 and Experimental Oberon*

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.

5. Building a new Oberon system on a bare metal target system

The tools to build an entirely new Oberon system on a bare metal *target* system connected to a *host* system via a data link (e.g., an RS-232 serial line) are provided by the module pair *ORC* (for Oberon to RISC Connection) running on the host system and *Oberon0* running on the target system.

When using Oberon in an emulator, one can simulate the process of booting the target system over a data link by starting *two* emulator instances connected via Unix-style *pipes*[6]:

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

Once the command interpreter *Oberon0* is running on the target system, a new Oberon system can be built *there* by remotely executing a sequence of *ORC* commands on the host system. The commands *ORC.Send* and *ORC.Receive* transfer files to and from the target system, while the command *ORC.SR* ("send, then receive sequence of items") initiates a command on the target system and then receives the command's response, if any. A command is specified by an integer followed by parameters, which can be numbers, names, strings or characters.

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

[6] *A script that executes these instructions is provided at: https://github.com/andreaspirklbauer/Oberon-experimental/blob/master/buildtarget.sh*

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 PCLink0.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
ORP.Compile BootLoadDisk.Mod/s ~          # a boot loader for booting the target system from the local disk
ORP.Compile BootLoadLine.Mod/s ~          # a boot loader for booting the target system over the data link
ORP.Compile Boot.Mod/s ~                  # the Oberon boot linker/loader (system building tool)

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 8 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
    Oberon10.Scn.Fnt System.Tool
    Modules.bin BootLoadDisk.rsc
    RS232.rsc PCLink0.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 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 *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
```

6. 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.

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 7 ~                         # show allocation, nof sectors, switches, and timer
 ORC.SR 8 1234 ~                    # send and mirror integer s (test whether Oberon-0 is running)

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)
```

7. 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 FPGA 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*[7] 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.

```
1  PROCEDURE InitMod (name: ARRAY OF CHAR);  (*call module initialization body*)
2    VAR mod: Modules.Module; body: Modules.Command; w: INTEGER;
3  BEGIN mod := Modules.root;
4    WHILE (mod # NIL) & (name # mod.name) DO mod := mod.next END ;
5    IF mod # NIL THEN SYSTEM.GET(mod.ent, w);
6      body := SYSTEM.VAL(Modules.Command, mod.code + w); body
7    END
8  END InitMod;
```

We will make use of this procedure when including an *entire* Oberon system in a single boot file, as described in the next section.

[7] *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).*

8. Creating a pre-linked Oberon boot file containing an entire Oberon system

To include an entire Oberon system in a single boot file and send it over the serial link directly into the *memory* of the target system (note that by default it doesn't fit in the boot area of its disk), a few precautions must be taken in modules *Fonts*, *Oberon* and *System*:

- Module *Fonts* creates the default font file *Oberon10.Scn.Fnt* if needed.
- Module *Oberon* does not load module *System* or start the *Oberon loop*.
- Module *System* establishes a working file system (root page) if needed, initializes the file directory, calls the initialization bodies of all imported modules and starts the *Oberon loop*.

If the disk of the target system is already configured for a working Oberon system, no changes are made to it *during* the transfer and the sent Oberon instance will simply "run on top of it"[8].

To implement this variant, compile the following files[9]:

```
ORP.Compile RS232.Mod PCLink0.Mod Oberon0.Mod Oberon0Tool.Mod ~
ORP.Compile Fonts1.Mod Oberon1.Mod System1.Mod ~
```

Note that for the modules *Fonts*, *Oberon* and *System* only the source file names have changed, but not their module names, i.e. the object files *Fonts.rsc*, *Oberon.rsc* and *System.rsc* are generated. However, they are backward compatible with their original versions and can not only be loaded by the Oberon *boot* loader (if they are included in a boot file), but also by the *regular* module loader. In the latter case, the above precautions are of course not taken (except for creating the default font file if needed).

To generate a pre-linked binary file of a "build-up" boot file containing the entire Oberon system:

```
Boot.Link System ~          # generate a "build-up" boot file containing the entire Oberon system (System.bin)
```

To transfer the entire Oberon system as a single binary file to the target system and start it, first restart the target system over the data link, then execute the following command on the host:

```
ORC.Load System.bin ~       # load the entire Oberon system over the data link to the target system and start it
```

The Oberon system should now come up on the target system. For additional convenience, module *Oberon0* (recall that it can also be started as a *regular* module) is included in the boot file and the Oberon-0 command interpreter is automatically started. Thus, a remote host system can immediately begin transferring additional files or execute the command *ORC.SR*.

Note that even though the target system is now *running* a valid Oberon system, it is not actually *built*, i.e. its boot area is not configured yet with a valid boot file (unless there is one already).

9. Loading a pre-linked Oberon boot file containing an entire Oberon system onto the boot area of the target system

A boot file containing the *entire* Oberon system does not normally fit in the boot area of FPGA Oberon (sectors 2-63, or 62KB). To make it fit, one needs to first enlarge the boot area by adjusting procedure *Kernel.InitSecMap* to mark the additional disk sectors as allocated.

---

[8] *Once running, the Oberon instance can of course modify the file system. If true Plug & Play were realized, a new partition would be dynamically created on the fly (currently not implemented).*
[9] *If you use Experimental Oberon, these files are already installed on your system; if you use FPGA Oberon 2013, download them from http://github.com/andreaspirklbauer/Oberon-building-tools/tree/master/Sources/OriginalOberon2013.*

To implement this variant, compile the following files[10]:

```
ORP.Compile RS232.Mod PCLink0.Mod Oberon0.Mod Oberon0Tool.Mod ~
ORP.Compile Kernel1.Mod Fonts1.Mod Oberon1.Mod System1.Mod ~
```

To generate a pre-linked binary file of a "build-up" boot file containing the entire Oberon system:

```
Boot.Link System ~        # generate a new boot file containing the entire Oberon system (System.bin)
```

To transfer the entire Oberon system as a single binary file to the target system and start it, first restart the target system over the data link, then execute the following command on the host:

```
ORC.Load System.bin ~     # load the entire Oberon system over the data link to the target system and start it
```

This will automatically establish a working file system and create the default font file on the target system, if needed. If you have used an Oberon system with a boot area of a *different* size on the target system before, it is recommended to manually clear the file directory (as the command *ORC.Load* will not modify it, if one already exists):

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

To transfer the files needed to *build* Oberon on the target system and configure its boot area, execute the following command on the host system[11]:

```
ORC.Send System.bin Oberon10.Scn.Fnt System.Tool BootLoadDisk.rsc
  Edit.rsc ORP.rsc ORG.rsc
  ORB.rsc ORS.rsc ORTool.rsc ~      # send the required (plus some additional) files to the target system
```

Since the Oberon system that is currently running on the target system is configured with an *enlarged* boot area (blocks 2-159, or 158KB), these files will automatically be placed in the disk sectors starting just *after* it, i.e. starting at block 160.

Again, module *Oberon0* is included in the boot file and the Oberon-0 command interpreter is automatically started[12]. Thus, a remote host system can immediately begin transferring additional files or execute the command *ORC.SR*.

To load the enlarged boot file onto the (enlarged) boot area of the target system, execute the following command on the host system:

```
ORC.SR 100 System.bin ~   # load the enlarged "regular" boot file onto the (enlarged) boot area (sectors 2-159)
```

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

```
ORC.SR 102 BootLoadDisk.rsc ~   # reboot the target system from the local disk (initiate the "regular" boot process)
```

---

[10] *If you use Experimental Oberon, these files are already installed on your system; if you use FPGA Oberon 2013, download them from http://github.com/andreaspirklbauer/Oberon-building-tools/tree/master/Sources/OriginalOberon2013.*

[11] *Sending the default font file Oberon10.Scn.Fnt is, strictly speaking, not necessary, as it has already been created when the system was started. We transfer it anyway, to cover the case where an Oberon system with a different-sized boot area existed on the target machine before (in which case no new default font file has been created).*

[12] *Note that System1.Mod imports modules RS232, PCLink0, Oberon0 and Oberon0Tool, implying that these modules cannot be unloaded once the system is running. If you do not want to import Oberon0Tool in System1.Mod, proceed as follows: a) explicitly include Oberon0Tool and its imports in the boot file by calling the boot linker as follows: Boot.Link Oberon0Tool System ~; b) no longer import module Oberon0Tool (and therefore any of its imports) in System1.Mod; c) use the call InitMod("Oberon0Tool") instead of "Oberon0Tool.Run" in the initialization sequence of System1.Mod to initialize module Oberon0Tool; d) restart the target system over the serial line; e) send the modified boot file System.bin to the target system: ORC.Load System.bin ~. For additional details, see the initialization sequence of System1.Mod.*

10. Modifying the Oberon boot loader

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

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[13]:

```
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.

```
1  MODULE* M;
2    IMPORT SYSTEM;
3    CONST SP = 14; StkOrg = -1000H;
4  BEGIN SYSTEM.LDREG(SP, StkOrg); …
5  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:

```
1  MODULE* BootLoad;    (*asterisk indicates that the compiler will generate a standalone program*)
2    …
3  BEGIN …
4  END BootLoad.
```

To generate an Oberon object file of the boot loader as a standalone program, compile it with the *regular* Oberon compiler:

[13] *See procedure ORG.Header*

```
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:

```
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 FPGA 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
4EE90014
AFE00000
A0E00004
40000000
...
40000000
C7000000                    # line 384
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*[14] 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
```

* * *

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

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 FPGA 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, PCLink0* and *Oberon0*. These 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
```