The system building tools

for the Oberon operating system

Andreas Pirklbauer

4.4.2020

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

In this document, the term “FPGA Oberon” refers to a re-implementation of the original *Project Oberon* on an FPGA development board around 2013, as published at www.projectoberon.com

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 FPGA Oberon, 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 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

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

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.

2. Creating the Oberon boot linker/loader

To create the Oberon boot linker/loader:

```
ORP.Compile ORL.Mod/s ~                                    # create the Oberon boot linker/loader
```

3. Creating a valid Oberon boot file

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

FPGA Oberon:

```
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
```

Extended Oberon:

```
ORP.Compile Kernel.Mod Disk.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
```

To link a set of object files together and generate a valid Oberon boot file from them:

```
ORL.Link Modules ~          # generate a pre-linked binary file of the "regular" boot file (Modules.bin)
ORL.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 of the local disk, this is typically module *Modules*, the top module of the *inner core* of the Oberon system. For the *build-up* boot file to be sent over a data link to a target system, it is usually module *Oberon0*, a simple command interpreter mainly used for system building and maintenance 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. This information is used by the *boot loader* to determine the *number* of bytes to be transferred into memory.

The Oberon boot linker (procedure *ORL.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.

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

To load a valid Oberon boot file, as generated by the command *ORL.Link*, onto the boot area of the local disk, one of the two valid *boot sources* (the other being the data link):

```
ORL.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 (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 the *host* system via a communication 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* (a simple command interpreter) running on the target system.

In the following sections we assume an *Oberon* system as the *host* system and a *target* system, which is already configured with an Oberon *boot loader* resident in its read-only store (ROM or PROM). See section 10 for instructions how to download or modify the *boot loader* itself.

The command *ORC.Load Oberon0.bin* loads the Oberon-0 command interpreter 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 after the target system has been restarted over the data link, as its boot loader is waiting for a boot file. It 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 commands *ORC.Send* and *ORC.Receive* transfer files between the host and the target system. The command *ORC.SR* ("send, then receive sequence of items") initiates a command on the target system and 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 on the target system involves several steps:

a. *Generate* the necessary binaries on the host system for Oberon system building:

```
Edit.Open Build.Tool                    # open a tool viewer on the host system containing the commands shown below

ORP.Compile Kernel.Mod Disk.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 BootLoad.Mod/s ~                  # the regular Oberon boot loader
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 ORL.Mod/s ORX.Mod/s ~             # the Oberon boot linker/loader and boot converter
ORL.Link Modules ~                  # generate a pre-linked binary file of the "regular" boot file (Modules.bin)
ORL.Link Oberon0 ~                  # generate a pre-linked binary file of the "build-up" boot file (Oberon0.bin)
```

b. *Restart* the target system over the data link by selecting the *alternative* boot source on the target system[4] and pre[5]ssing the reset button. When using an emulator to run Oberon, one can

[4] On a Xilinx Spartan-3 FPGA board, this is achieved by setting switch 0 = 1 ("up").
[5] Module Disk is only required for building Extended Oberon

simulate the process of booting the target system over the data link by starting *two* emulator instances connected via *two* Unix-style *pipes* – one for each direction[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 pipe2 --serial-out pipe1 ob2.dsk --boot-from-serial &  # start the target system
./risc --serial-in pipe1 --serial-out pipe2 ob1.dsk &        # start the host system from the local disk
```

c. *Build* Oberon on the target system by executing 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 (root page of the) file directory on the target system

ORC.Send Modules.bin Oberon10.Scn.Fnt System.Tool
    Input.rsc Display.rsc Viewers.rsc
    Fonts.rsc Texts.rsc Oberon.rsc
    MenuViewers.rsc TextFrames.rsc System.rsc
    BootLoadDisk.rsc ORL.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
```

d. *Start* the newly built target system from its local *boot area*, either by selecting the *primary* boot source on the target system[7] and pressing the reset button, or *remotely* by executing the following command on the host system:

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

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*.

e. Once the target system is started, one can re-enable the ability to transfer files between the host and the target system by executing the following command 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
```

After this, one can again use the commands *ORC.Send* and *ORC.Receive* on the host system to initiate file transfers to and from the target system (in addition to serving as the top module of a *build-up* boot file to be sent over a data link to a target system, module *Oberon0* can also be loaded as a *regular* Oberon module by activating the command *Oberon0Tool.Run* – in which case its main loop is simply not started).

The Oberon-0 command interpreter also implements a generic *remote procedure call (RPC)* mechanism, i.e. a remote computer connected to the target system can execute the commands *ORC.SR 22 M.P* ("call command") or *ORC.SR 102 M.rsc* ("call program") to remotely initiate execution of the specified command or (standalone) program on the target system.

[6] *A script that executes these instructions is provided at http://github.com/andreaspirklbauer/Oberon-building-tools/blob/master/buildtarget.sh*
[7] *On a Xilinx Spartan-3 FPGA board, this is achieved by setting switch 0 = 0 (“down”).*

Note that the command *ORC.SR 22 M.P* does *not* transmit any parameters to the target system for the following reason: The parameter text of an Oberon command typically refers to objects that exist *before* command execution starts, i.e. the *state* of the system represented by its global variables. It appeared 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 remotely initiated from the host system once the Oberon-0 command interpreter is running on the target system, including:

Boot

```
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
```

System

```
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)
```

Files

```
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
```

Modules

```
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
```

Disk

```
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))
```

Memory

```
ORC.SR 1 50000 16 ~          # show memory   adr, n words (in hex) M[a], M[a+4], …, M[a+(n-1)*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))
```

Display

```
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.

If the source text of a module is not available or cannot change, an alternative approach is to extract the starting address of the initialization body of the just loaded module from its module descriptor in memory and simply call it, as shown in procedure *InitMod*[8] below. See chapter 6 of the book *Project Oberon 2013 Edition* for a description of the format of a *module descriptor* in memory. Here it suffices to know that it contains a pointer to a list of *entries* for exported entities, the first one of which points to the initialization code of the module itself.

```
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.

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 data link directly into the *memory* of the target system (by default it doesn't fit in the boot area of its disk), a few precautions must be taken in modules *Oberon* and the (new) top module *System1*:

- Module *Oberon* does not load module *System* or start the *Oberon loop*.

[8] *Procedure InitMod could be placed in modules Oberon or Modules (note: the data structure rooted in the global variable Modules.root is transferred as part of the boot file).*

- Module *System1* establishes a working file system (root page) if necessary, creates the default font file *Oberon10.Scn.Fnt* unless it already exists, calls the initialization bodies of all imported modules, starts the *Oberon0* background task 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 of the boot file and the transmitted Oberon instance will simply "run on top of it"[9]. Once running, the Oberon instance can of course modify the file system.

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

```
ORP.Compile RS232.Mod PCLink0.Mod Oberon0.Mod Oberon0Tool.Mod ~
ORP.Compile System1.Mod ~                          # on Extended Oberon
ORP.Compile Oberon1.Mod System1.Mod ~              # on FPGA Oberon[11]
```

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

```
ORL.Link System1 ~        # generate a "build-up" boot file containing the entire Oberon system (System1.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 System1.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 loaded 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* yet, i.e. its boot area is not configured with a valid boot file (unless there is one already).

9. Loading an entire Oberon system onto the boot area of the target system

A boot file containing the *entire* Oberon system will not normally fit in the boot area of a typical FPGA Oberon system (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 (sectors 64-159, or an additional 96KB) as allocated. This is the purpose of module *Kernel1.Mod* (on FPGA Oberon) or *Disk1.Mod* (on Extended Oberon).

To implement this variant, compile the following files:

```
ORP.Compile RS232.Mod PCLink0.Mod Oberon0.Mod Oberon0Tool.Mod ~
ORP.Compile Disk1.Mod System1.Mod ~                # on Extended Oberon
ORP.Compile Kernel1.Mod Oberon1.Mod System1.Mod ~  # on FPGA Oberon
```

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

```
ORL.Link System1 ~        # generate a new boot file containing the entire Oberon system (System1.bin)
```

[9] *If true Plug & Play were realized, a new partition would be dynamically created on the fly (currently not implemented).*
[10] *If you use Extended Oberon, these files are already installed on your system; if you use FPGA Oberon, download them from http://github.com/andreaspirklbauer/Oberon-building-tools/tree/master/Sources/FPGAOberon2013.*
[11] *On FGPA Oberon you will also need to re-compile ORG.Mod from http://github.com/andreaspirklbauer/Oberon-building-tools (configures a larger string area) before compiling System1.Mod.*

Optionally, you can also generate a pre-linked binary file of the *regular* boot file containing only the *inner core* of the Oberon system::

```
ORL.Link Modules ~      # (optional) generate a new "regular" boot file (Modules.bin, built using Kernel1 or Disk1)
```

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 System1.bin ~      # load the entire Oberon system over the data link to the target system and start it
```

This will automatically configure the target system with an *enlarged* boot area (sectors 2-159, or 158 KB). If you have used a boot area of a *different* size before, it is recommended to manually clear the file directory (as *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 required for booting the target system with the *enlarged* boot file containing the *entire* Oberon system, execute the following command on the host system[12]:

```
ORC.Send System1.bin Oberon10.Scn.Fnt System.Tool
   BootLoadDisk.rsc ORL.rsc
   RS232.rsc PCLink0.rsc
   Oberon0.rsc Oberon0Tool.rsc
   Edit.rsc PCLink1.rsc
   ORP.rsc ORG.rsc
   ORB.rsc ORS.rsc ORTool.rsc ~      # Group A: send the files for booting with the enlarged boot file
```

Optionally, you can also send the files required for booting the target system with the *regular* boot file (this will allow you to re-configure the system later to use *any* boot file you want):

```
ORC.Send Modules.bin
   Input.rsc Display.rsc Viewers.rsc
   Fonts.rsc Texts.rsc Oberon.rsc
   MenuViewers.rsc TextFrames.rsc
   System.rsc ~                    # Group B (optional): send the files for booting with the regular boot file
```

Since the Oberon system that is currently running on the target system is now 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. Thus, a remote host system can immediately begin transferring additional files or execute the command *ORC.SR*.

Now decide *which* boot file to load onto the enlarged boot area of the disk of the target system.

To load the boot file containing the *entire* Oberon system onto the boot area of the target system, execute the following command on the host system:

```
ORC.SR 100 System1.bin ~      # load the enlarged boot file onto the boot area (sectors 2-159)
```

[12] *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).*

To load the boot file containing only the *inner core* of the Oberon system onto the boot area of the target system, execute the following command on the host system:

```
ORC.SR 100 Modules.bin ~        # alternatively, load the regular boot file onto the boot area (sectors 2-159)
```

To start the newly built target system from its local *boot area*, either select the *primary* boot source on the target system and press the reset button[13], 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)
```

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[14]:

```
MOV SP -64          # set the stack pointer SP register (R14) to memory location -64 (= 0FFFC0H in FPGA Oberon)
```

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 absolute address 8 as the *variable* space for global variables[15]. 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

[13] *On a Xilinx Spartan-3 FPGA board, this is achieved by setting switch 0 = 0 ("down").*
[14] *See procedure ORG.Header*
[15] *See ORP.Module, which sets the data counter ORP.dc to start at absolute memory address 8. If the standalone program itself is loaded at absolute memory address 0 (which is not necessarily the case), the 6 words from code[2] to code[7] (i.e. a total of 24 bytes) can be used for the absolute variables of the standalone program (see ORG.Open).*

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:

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

To create the Oberon boot converter:

```
ORP.Compile ORX.Mod/s ~             # create the Oberon boot converter
```

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

```
ORX.WriteFile BootLoad.rsc 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 name of the PROM file to be generated (output file).

In the case of FPGA Oberon 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. The output is zero-filled to the next higher multiple of 512 lines.

```
E7000151                    # line 1
00000000
...
00000000
4EE90014
AFE00000
A0E00004
40000000
...
40000000
C7000000                    # line 384
00000000
...
00000000                    # line 512
```

Note that the command *ORX.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 *ORX.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 *ORX.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.

```
ORX.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 a newly created Oberon boot loader to the permanent read-only store of the target hardware typically requires the use of proprietary (or third-party) tools, such as *data2mem* or *fpgaprog*[16], which are usually not available on an FPGA Oberon system, but are executed on a commercially available operating system (e.g., Windows, macOS, Linux).

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*.

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

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

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

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

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

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

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

Appendix 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):

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 FPGA Oberon), while the destination address is usually a fixed memory location (location 0 in FPGA Oberon). 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* (in Extended Oberon, module *Kernel* has been split into two modules *Kernel* and *Disk*). These 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 *ORL.Link* and loaded as a sequence of *bytes* onto the boot area of the local disk, using the command *ORL.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} zero StartAdr
block    = size adr {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 FPGA Oberon – the address field of the last block may not actually be sent, in which case the format *effectively* becomes:

```
BootFile = {block} zero
block    = size adr {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* (in Extended Oberon, module *Kernel* has been split into two modules *Kernel* and *Disk*). These modules constitute the 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 *ORL.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* (which automatically performs the conversion of the input file to the stream format used for booting a system over a data link). 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
```