MultiOberon Compiler User Guide

1. Introduction.

Copyright © 2019-2020, by Dmitry Dagaev

Version 1.0 24-Nov-2020

Omc stands for MultiOberon Compiler.

MultiOberon is an Oberon Compiler with three different backends:

- BlackBox Native x86 code Generator (1.7, version 1.6 with partial support)
- Ofront Generated C-Language Code Translator;
- LLVM byte code Generator.

MultiOberon is Cross-Platform Compiler with supported platforms:

- Windows X86;
- Windows X64;
- Linux X86;
- Linux X64.

MultiOberon is restriction-based scaling Oberon environment with an initial condition as Component Pascal syntax. MultiOberon can be called from BlackBox environment and from command line.

MultiOberon is under GNU Lesser General Public License v3.0.

MultiOberon can be loaded from https://github.com/dvdagaev/Mob

2. __Structure.__

MultiOberon consists of binary applications and a list of program subsystems. Binary exe catalogs follow the rule Bin[os][arch].

arch\os	windows	unix (linux, …)
X86	Binwe	Binue
X64	Binwr	Binur

Binary applications consist of compilers and shells. MultiOberon compilers includes full-featured compilation support follow the om[backend]c rule. MultiOberon shells support only dynamic loading and running, the rule is om[backend]sh.

	BlackBox	Ofront	LLVM
compiler	ombc	omfc	omlc
shell	ombsh	omfsh	omlsh

Program subsystems are structured as BlackBox ones.
Backends match Om[backend] rule. The third letter of backend subsystem means as following:
- Omc is the MultiOberon compiler and console;
- Omb is the instance of MultiOberon compiler with BlackBox backend. Based on Ominc CP DevCompiler. OmbLinker is based on cross-platform solution of I.Degtarenko (Trurl);
- Omf is the instance of MultiOberon compiler with Ofront backend. Based on Josef Templ's Ofront;
- Oml is the instance of MultiOberon compiler with LLVM backend. Used the library prepared with LLVM 5.0.

Each subsystem has the Mod dir for modules sources, Docu for documents, catalogs for code and symbol files.
Code files catalogs follow the rule C[backend][os][arch].

arch\os	windows	Unix (Linux, …)
X86 BlackBox portable	Sym	Sym
X86 Omb-specific	Cbwe	Cbue
X86 OFront	Cfwe	Cfue
X64 OFront	Cfwr	Cfur
X86 LLVM	Clwe	Clue
X64 LLVM	Clwr	Clur

Symbol and usage files are resided in the symbol catalog. Symbol files catalogs follow the rule S[backend][os][arch].

arch\os	windows	Unix (Linux, …)
X86 BlackBox portable	Sym	Sym
X86 Omb-specific	Sbwe	Sbue
X86 OFront	Sfwe	Sfue
X64 OFront	Sfwr	Sfur
X86 LLVM	Slwe	Slue
X64 LLVM	Slwr	Slur

3. __Installation__.

For BlackBox 1.7 (https://blackboxframework.org/)

```
>win_toinstall.vbs <path-to-blackbox>
```
for example,
```
>win_toinstall.vbs "c:\BlackBox Component Builder 1.7"
```
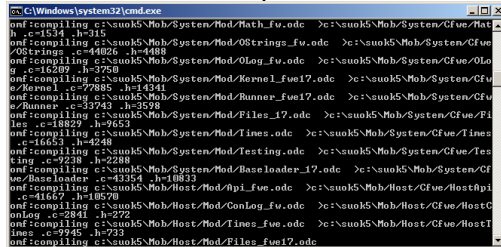for BlackBox 1.7.2 "C:\Program Files (x86)\BlackBox Component Builder 1.7.2",
```
>win_toinstall.vbs
```
For BlackBox 1.6
```
>win_toinstall.vbs <path-to-blackbox> 16
```
wait some shell scripts finished:

On Unix (this color – for Linux, FreeBSD, etc.):
For Oml - glibc 2.15 or STT_GNU_IFUNC support is needed, tinfo package required
Download and install 1.7.2 https://blackbox.oberon.org/download
```
>tclsh lin_toinstall.tcl <path-to-blackbox>
```
for example,
```
>tclsh lin_toinstall.tcl "/home/ddag/mobdev/bb"
```

It is expected, that gcc or clang is already installed and accessible from command line. Otherwise, Ofront automatic building and LLVM linking cannot work properly. Initial options are set in Omf.cfg (Oml.cfg) files in Bin[os][arch] dirs. For example in Omf.cfg:
```
gcc_opt=-O2 -fshort-wchar
gcc_lnkopt=-O2 -lm -ldl
cc=gcc
```
It means calling "gcc –O2 –fshort-wchar" for compilation and "gcc –O2 –lm -ldl" for linking.
Options in Oml.cfg are similar, but proper llc is already supported with MultiOberon.
```
llc_opt=-O0
gcc_lnkopt=-O0 -lm -ldl
lnk=gcc
```
For compilation "llc –O0" is called and "gcc –O0 –lm -ldl" for linking.

These files can be specifically tuned by user, but be careful, because next installation must rewrite them again.

In BlackBox environment Strings documents are used instead of .cfg files:
- Omb/Rsrc/Strings.odc instead of Binwe/Omb.cfg;
- Omf/Rsrc/Strings.odc instead of Binwe/Omf.cfg;
- Oml/Rsrc/Strings.odc instead of Binwe/Oml.cfg

Options in String documents are separated by tabs as following:
```
gcc_opt    -O2
gcc_lnkopt -O2
lnk   clang
clang_opt  -O2
clang_lnkopt    -O2 -luser32
```

4. Hello, World from Command Line

The simplest way is to execute OmtestHelloWorld module is running ex[ecute] command:
Bin[os][arch]/om[backend]c ex OmtestHelloWorld. In particular, for BlackBox backend the
command must be for `Windows` and `Unix` respectively:

```
>Binwe\ombc ex OmtestHelloWorld
>Binue/ombc ex OmtestHelloWorld
Hello, World
```

For 32-bit Ofront backend the command is as following:

```
>Binwe\omfc ex OmtestHelloWorld
>Binue/omfc ex OmtestHelloWorld
Hello, World
```

For 64-bit LLVM backend the command is as following:

```
>Binwr\omlc ex OmtestHelloWorld
>Binur/omlc ex OmtestHelloWorld
Hello, World
```

An execute command combines all compile, build and run. At first step, OmtestHelloWorld.mob is
compiled to Omtest/Code/HelloWorld.ocf.

```
>Binwe\ombc co OmtestHelloWorld
>Binue/ombc co OmtestHelloWorld
omb:compiling …HelloWorld.mob  >…/Omtest/Code/HelloWorld.ocf code=52
glob=0
```

The .mob extension means MultiOberon text file. For BlackBox .odc file format uses –odc option.
OmtestHelloWorld.odc is compiled to Omtest/Code/HelloWorld.ocf.

```
>Binwe\ombc co –odc OmtestHelloWorld
>Binue/ombc co –odc OmtestHelloWorld
omb:compiling …HelloWorld.mob  >…/Omtest/Code/HelloWorld.ocf code=52
glob=0
```

Compilation of 32-bit Ofront is done in the similar way, but the .c and .h files are produced in
Cfwe[Cfue] directory:

```
>Binwe\omfc co OmtestHelloWorld
>Binue/omfc co OmtestHelloWorld
omf:compiling …HelloWorld.mob  >…/Omtest/Cfwe/OmtestHelloWorld
.c=1898 .h=357
```

Compilation of 64-bit LLVM is done in the similar way, but the .ll and .bc files are produced in
Clwr[Clur] directory:

```
>Binwr\omlc co OmtestHelloWorld
>Binur/omlc co OmtestHelloWorld
oml:compiling …HelloWorld.mob  >…/Omtest/Clwr/OmtestHelloWorld
.ll=8616 .bc=3172
```

Building is not necessary for Omb, but Omf needs external tool for producing and object file.

```
>Binwe\omfc build OmtestHelloWorld
>Binue/omfc build OmtestHelloWorld
===== building OmtestHelloWorld ... done
```

In the same way, Oml needs llc llvm utility for producing and object file. However, llc for LLVM-5.0
is included in MultiOberon distribution.

```
>Binwe\omfc build OmtestHelloWorld
>Binue/omfc build OmtestHelloWorld
===== building OmtestHelloWorld ... done
```

Then we can dynamically load and run OmtestHelloWorld module.

```
>Binwe\ombc run OmtestHelloWorld
>Binue/ombc run OmtestHelloWorld
Hello, World
```

For 32-bit Ofront use commands below for dynamically load and run OmtestHelloWorld module.

```
>Binwe\omfc run OmtestHelloWorld
>Binue/omfc run OmtestHelloWorld
Hello, World
```

For 64-bit LLVM use commands below for dynamically load and run OmtestHelloWorld module.

```
>Binwr\omlc run OmtestHelloWorld
>Binur/omlc run OmtestHelloWorld
Hello, World
```

An executive can be created from ombc. Link OmtestHelloWorld executive.

```
>Binwe\ombc link -r OmtestHelloWorld
>Binue/ombc link -r OmtestHelloWorld
```

For 32-bit Ofront an executive can be created from omfc. Link OmtestHelloWorld executive.

```
>Binwe\omfc link -r OmtestHelloWorld
>Binue/omfc link -r OmtestHelloWorld
```

For 64-bit LLVM an executive can be created from omlc. Link OmtestHelloWorld executive.

```
>Binwe\omlc link -r OmtestHelloWorld
>Binue/omlc link -r OmtestHelloWorld
```

Finally we can run OmtestHelloWorld executive for BlackBox.

```
>cd Omtest/Code
>OmtestHelloWorld
Hello, World
```

We can also run of 32-bit Ofront.

```
>Omtest\Cfwe\OmtestHelloWorld
>Omtest/Cfue/OmtestHelloWorld
Hello, World
```

We can also run of 64-bit LLVM.

```
>Omtest\Clwr\OmtestHelloWorld
>Omtest/Clur/OmtestHelloWorld
Hello, World
```

64-bit Ofront modules and 32-bit LLVM modules can be done in the similar way.

5. Omb – run from Black Box

5.1   Installation
Preconditions.

Omb does not use any other services except BlackBox.

Run: `StartupBlackBox`

Open Omb/Docu/Quick-Start.odc and examples Omtest/Docu/Quick-Start.odc

5.2   Compiling the Omb
Compile the following modules from BlackBox:

1.3. Compile the following portable modules:
❗ DevCompiler.CompileThis OmcCfgfile OmcTarget OmcCRuntime OmcHooks OmcDialog OmcOPM OmcOPT OmcOPU OmcOPB OmcOPS OmcOPP OmcDump OmcTester OmcParams OmcCommandParams OmcOdcSource OmcTxtSource OmcRuntimeStd OmcDialogStd OmcDialogConsole OmcCompiler OmbOPE OmbOPH OmbOPL486 OmbOPC486 OmbOPV486 OmbInlink OmbLnkBase OmbLnkLoad OmbLnkWritePe OmbLnkWriteElf OmbLnkWriteElfStatic OmbLinkPortableProcessor OmbParams OmbBackEnd OmbCompiler OmbLinker◢

All the commands for BlackBox would be recorded in a manner below:
```
^Q DevCompiler.CompileThis OmcCfgfile OmcTarget OmcCRuntime OmcHooks
OmcDialog OmcOPM OmcOPT OmcOPU OmcOPB OmcOPS OmcOPP OmcDump OmcTester
OmcParams OmcCommandParams OmcOdcSource OmcTxtSource  OmcRuntimeStd
OmcDialogStd OmcDialogConsole OmcCompiler OmbOPE OmbOPH OmbOPL486
OmbOPC486 OmbOPV486 OmbInlink OmbLnkBase OmbLnkLoad OmbLnkWritePe
OmbLnkWriteElf OmbLnkWriteElfStatic OmbLinkPortableProcessor OmbParams
OmbBackEnd OmbCompiler OmbLinker
```

5.3   Compiling the Omtest examples
Compile the following modules:
```
^Q OmbCompiler.CompileThis OmtestHelloWorld OmtestFormats OmtestDateTime
OmtestMkTraps OmtestHeap
```

5.4   Running the Omtest examples
5.4.1 The simplest Hello, World example
```
^Q OmtestHelloWorld.MAIN
```
5.4.2 Logging with char, int and real formats
```
^Q OmtestFormats.MAIN
```

5.4.3 Shows date, time and delay
```
^Q OmtestDateTime.MAIN
```

5.4.4 Traps handling abilities of runtime
Simple Assert
```
^Q "OmtestMkTraps.RunOpt('a')"
```
Simple Halt
```
^Q "OmtestMkTraps.RunOpt('h')"
```
Zero divide
```
^Q "OmtestMkTraps.RunOpt('z')"
```
Nil pointer dereference
```
^Q "OmtestMkTraps.RunOpt('p')"
```

5.4.5 Dynamic memory and garbage collector
! The program Hangs on Omb Unix version due to not optimal Kernel in BlackBox for Unix
```
^Q OmtestHeap.MAIN
```

5.5   Creating executives
Link executives to call them from console. Option –r adds recursively all imported modules needed. Otherwise adding Kernel$+ Log Math Strings OStrings OLog HostConLog Runner explicitly must be done.

```
^Q OmbLinker.LinkExe -r -o "Omtest/Code/OmtestHelloWorld"
OmtestHelloWorld
^Q OmbLinker.LinkExe -r -o "Omtest/Code/OmtestFormats" OmtestFormats
^Q OmbLinker.LinkExe -r -o "Omtest/Code/OmtestDateTime.exe"
OmtestDateTime
^Q OmbLinker.LinkExe -r -o "Omtest/Code/OmtestMkTraps.exe" OmtestMkTraps
^Q OmbLinker.LinkExe -r -o "Omtest/Code/OmtestHeap.exe" OmtestHeap
```

Compiled executives can be ran in console:
```
>cd Omtest/Code
>OmtestHelloWorld
Hello, World
```

5.6   [Advanced] Self-Compiling Omb Compiler
Omb Compiler (OmbCoSh) is a console exe application with a full-featured compilation support.
Self-Compile portable console Oberon Compiler.

```
^Q OmbCompiler.CompileThis OmcCfgfile OmcTarget OmcCRuntime OmcHooks
OmcDialog OmcOPM OmcOPT OmcOPU OmcOPB OmcOPS OmcOPP OmcDump OmcTester
OmcParams OmcCommandParams OmcOdcSource OmcTxtSource OmcRuntimeStd
OmcDialogStd OmcDialogConsole OmcTimesDialog OmcCompiler OmcConsole
OmcDiscomp OmbOPE OmbOPH OmbOPL486 OmbOPC486 OmbOPV486 OmbInlink
OmbLnkBase OmbLnkLoad OmbLnkWritePe OmbLnkWriteElf OmbLnkWriteElfStatic
OmbLinkPortableProcessor OmbParams OmbBackEnd OmbCompiler
OmcLoaderRoutines OmcOcfLoader OmbLinker OmbCoSh
```

Windows-specific self-link console Oberon Compiler
```
^Q OmbLinker.LinkExe -r -tl 2 -o "Binwe/ombc" $+Kernel Log Math Strings
OStrings OLog HostConLog Runner Files HostFiles OmcCfgfile Dates Times
HostTimes Dialog Stores Sequencers Models Services Fonts Meta Converters
Ports Views Controllers Properties Mechanisms Containers Printers
Printing OmcTimesDialog Documents TextModels TextRulers TextSetters
TextViews OmcTarget OmcCRuntime OmcDialog OmcHooks OmcOPM OmcOPT OmcOPB
OmcOPU OmcOPS OmcOPP OmcTester OmcParams OmcOdcSource OmcDialogConsole
OmcRuntimeStd OmcConsole OmcDiscomp OmcLoaderRoutines OmcOcfLoader OmbOPE
OmbOPH OmbOPL486 OmbOPC486 OmbOPV486 OmbBackEnd OmbInlink OmbLnkBase
OmbLnkLoad OmbLnkWritePe OmbLnkWriteElf OmbLnkWriteElfStatic
OmbLinkPortableProcessor OmbCoSh
```

Linux-specific self-link console Oberon Compiler
```
^Q OmbLinker.LinkExe -o "Binue/ombc" $+Kernel Log Math Strings OStrings
OLog HostConLog Runner Testing Files HostFiles OmcCfgfile Dates Times
HostTimes Dialog Stores Sequencers Models Services Fonts Meta Converters
Ports Views Controllers Properties Mechanisms Containers Printers
Printing OmcTimesDialog Documents TextModels TextRulers TextSetters
TextViews OmcTarget OmcCRuntime OmcDialog OmcHooks OmcOPM OmcOPT OmcOPB
OmcOPU OmcOPS OmcOPP OmcTester OmcParams OmcOdcSource OmcDialogConsole
OmcRuntimeStd OmcConsole OmcDiscomp OmbOPE OmbOPH OmbOPL486 OmbOPC486
OmbOPV486 OmbBackEnd OmcLoaderRoutines OmcOcfLoader OmbInlink OmbLnkBase
OmbLnkLoad OmbLnkWritePe OmbLnkWriteElf OmbLnkWriteElfStatic
OmbLinkPortableProcessor OmbCoSh
```

Option –r is not used here, because many modules, like TextModels, are not imported directly, so the must be explicitly defined for linker.

5.7   [Advanced] Compiling minimal Shell ombsh

Omb minimal Shell (OmbShell) is a console exe application with dynamic loading and running support.

Windows-specific compiling console OmbShell

```
^Q OmbCompiler.CompileThis OmcLoaderRoutines OmcOcfLoader
OmcObjLoader_Coff OmcShell OmbShell
```

Windows-specific linking console OmbShell

```
^Q OmbLinker.LinkExe -r -tl 2 -o "Binwe/ombsh" $+Kernel Log Math Strings
OStrings OLog HostConLog Runner Files HostFiles OmcLoaderRoutines
OmcOcfLoader OmcObjLoader OmcShell OmbShell
```

Linux-specific compiling console OmbShell

```
^Q OmbCompiler.CompileThis OmcLoaderRoutines OmcOcfLoader
OmcObjLoader_Elf OmcShell OmbShell
```

Linux-specific linking console OmbShell

```
^Q OmbLinker.LinkExe -r -tl 2 -o "Binue/ombsh" $+Kernel Log Math Strings
OStrings OLog HostConLog Runner Files HostFiles OmcLoaderRoutines
OmcOcfLoader OmcObjLoader OmcShell OmbShell
```

5.8   Unloading Omb Compiler

```
^Q DevDebug.UnloadThis OmbLinker OmbCompiler OmbBackEnd OmbParams
OmbLinkPortableProcessor OmbLnkWriteElfStatic OmbLnkWriteElf
OmbLnkWritePe OmbLnkLoad OmbLnkBase OmbLinkWinProcessor OmbInlink
OmcLoader OmcCompiler OmcCommandParams OmbOPV486 OmbOPC486 OmbOPL486
OmbOPH OmbOPE  OmcTimesDialog OmcDialogStd OmcRuntimeStd  OmcOdcSource
OmcParams OmcTester OmcDump OmcOPP OmcOPS OmcOPU OmcOPB OmcOPT OmcOPM
OmcDialog OmcHooks OmcCRuntime OmcTarget OmcCfgfile Runner Testing
```

6. <u>Omb - from Command Line</u>.

6.1   Installation

Preconditions.
      Omb does not use any other services. Process all the commands below from the Mob-master root dir.

6.2   Compiling examples

A script for compiling and linking all examples.
```
bwe_tomake.bat
bue_tomake.sh
```
Cleans all the executives of example set
```
bwe_toclean
bue_toclean.sh
```

The set of commands compiles all the examples listed
```
Binwe\ombc co -odc OmtestHelloWorld OmtestFormats OmtestDateTime
OmtestMkTraps OmtestHeap
Binwe\ombc link -r OmtestHelloWorld
Binwe\ombc link -r OmtestFormats
Binwe\ombc link -r OmtestDateTime
Binwe\ombc link -r OmtestMkTraps
Binwe\ombc link -r OmtestHeap
Binue/ombc co -odc OmtestHelloWorld OmtestFormats OmtestDateTime
OmtestMkTraps OmtestHeap
mkdir -p Omtest/Cbue
Binue/ombc link -r OmtestHelloWorld
Binue/ombc link -r OmtestFormats
Binue/ombc link -r OmtestDateTime
Binue/ombc link -r OmtestMkTraps
Binue/ombc link -r OmtestHeap
```
The first command above compiles all the examples listed. The links produce a set of proper executives. If we omit –r recursive options, the link command must be as follows:
```
Binwe\ombsh li $+Kernel Log Math Strings OStrings OLog HostConLog
Runner OmtestHelloWorld
Binue/ombsh li $+Kernel Log Math Strings OStrings OLog HostConLog
Runner OmtestHelloWorld
```

6.3   Running the Examples

6.3.1 The simplest Hello, World example
```
>Omtest\Cbwe\OmtestHelloWorld
>Omtest/Cbue/OmtestHelloWorld
Hello, World
```

6.3.2 Logging with char, int and real formats
```
>Omtest\Cbwe\OmtestFormats
>Omtest/Cbue/OmtestFormats
```

6.3.3 Shows date, time and delay
```
>Omtest\Cbwe\OmtestDateTime
```

```
>Omtest/Cbue/OmtestDateTime
```

6.3.4 Traps handling abilities of runtime
Simple Assert
```
>Omtest\Cbwe\OmtestMkTraps -trap a
>Omtest/Cbue/OmtestKmTraps -trap a
```

Simple Halt
```
>Omtest\Cbwe\OmtestMkTraps -trap h
>Omtest/Cbue/OmtestKmTraps -trap h
```

Zero divide
```
>Omtest\Cbwe\OmtestMkTraps -trap z
>Omtest/Cbue/OmtestKmTraps -trap z
```

Nil pointer dereference
```
>Omtest\Cbwe\OmtestMkTraps -trap p
>Omtest/Cbue/OmtestKmTraps -trap p
```

6.3.5 Dynamic memory and garbage collector
```
>Omtest\Cbwe\OmtestHeap
>Omtest/Cbue/OmtestHeap
```

6.4   Self-compilation

A script for compiling the compiler ombc is as following:
```
bwe_compiler_tomake.bat
bue_compiler_tomake.sh
```
A script for compiling the minimal shell ombsh is as following:
```
bwe_sh_tomake.bat
bue_sh_tomake.sh
```

7. **Omf Ofront – run from Black Box**

Omf Ofront-based backend can produce program code (*.c) and header (*.h) files, which should be compiled into object files (*.o). Since object files cannot be loaded in BlackBox IDE, the Omf Ofront functionality is limited by producing files only. No dynamic loading and running can be done from BlackBox environment.

7.1   Installation

Preconditions.

> Omf uses previously installed C/C++ compiler gcc or clang.
> Run: `StartupBlackBox`

Open Omf/Docu/Quick-Start.odc and examples Omtest/Docu/Quick-Start.odc

7.2   Compiling the Omf

Compile modules for Omf:

```
3. Compile the following modules:
 DevCompiler.CompileThis OmcCfgfile OmcTarget OmcCRuntime OmcHooks OmcDialog OmcOPM
OmcOPT OmcOPU OmcOPB OmcOPS OmcOPP OmcDump OmcTester OmcParams OmcCommandParams
OmcOdcSource OmcTxtSource OmcOdcTextReader OmcExtSource OmcRuntimeStd OmcDialogStd
OmcDialogConsole OmcCompiler OmcTimesDialog OmcConsole OmfOPG OmfOPC OmfOPV OmfParams
OmfBackEnd OmfCompiler OmfLinker
```

All the commands for BlackBox would be recorded in a manner below:
```
^Q DevCompiler.CompileThis OmcCfgfile OmcTarget OmcCRuntime OmcHooks
OmcDialog OmcOPM OmcOPT OmcOPU OmcOPB OmcOPS OmcOPP OmcDump OmcParams
OmcCommandParams OmcOdcSource OmcTxtSource OmcOdcTextReader OmcExtSource
OmcRuntimeStd OmcDialogStd OmcDialogConsole OmcCompiler OmcTimesDialog
OmcConsole OmfOPG OmfOPC OmfOPV OmfParams OmfBackEnd OmfCompiler
OmfLinker
```

7.3   Compiling the Omtest examples

Compile the following modules (: means producing main for module):
```
^Q OmfCompiler.CompileThis :OmtestHelloWorld :OmtestFormats
:OmtestDateTime :OmtestMkTraps :OmtestHeap
```
Expected result in ~/Omtest/Cfwe/ directory (Cfue for Unix): OmtestHelloWorld.c OmtestFormats.c OmtestDateTime.c OmtestMkTraps.c OmtestHeap.c

Compile the following modules for 64 bit:
```
^Q OmfCompiler.CompileThis -64 :OmtestHelloWorld :OmtestFormats
:OmtestDateTime :OmtestMkTraps :OmtestHeap
```
Expected result in ~/Omtest/Cfwr/ directory (Cfur for Unix): OmtestHelloWorld.c OmtestFormats.c OmtestDateTime.c OmtestMkTraps.c OmtestHeap.c

Building the object file needs to call external console program like gcc. It's better to use console. But it is also availble (but time-consuming) from BlackBox.
```
^Q OmfLinker.BuildFiles -r OmtestHelloWorld
```
Linking is similar:
```
^Q OmfLinker.LinkExe -r OmtestHelloWorld
```

7.4   [Advanced] Compiling minimal Shell omfsh

Omf minimal Shell (OmfShell) is a console exe application with dynamic loading and running support.

Windows-specific compiling console OmfShell

```
^Q OmfCompiler.CompileThis OmcLoaderRoutines OmcObjLoader_Coff OmcShell
:OmfShell
^Q OmfCompiler.CompileThis -64 OmcLoaderRoutines OmcObjLoader_Coff
OmcShell :OmfShell
```

Compiling Linux-specific minimal omlsh

```
^Q OmfCompiler.CompileThis OmcLoaderRoutines OmcObjLoader_Elf OmcShell
:OmfShell
^Q OmfCompiler.CompileThis -64 OmcLoaderRoutines OmcObjLoader_Elf
OmcShell :OmfShell
```

7.5   Unloading Omf Compiler

```
^Q DevDebug.UnloadThis OmfCompiler OmfLinker OmfBackEnd OmfParams OmfOPV
OmfOPC OmfOPG OmcCoffLoader OmcLoaderRoutines OmcCompiler OmcDialogStd
OmcRuntimeStd OmcLogStd OmcOdcSource OmcCommandParams OmcParams OmcTester
OmcDump OmcOPP OmcOPS OmcOPU OmcOPB OmcOPT OmcOPM OmcDialog OmcHooks
OmcCRuntime OmcTarget Runner
```

8. Omf Ofront - from Command Line.

8.1 Installation

Preconditions.

Omf uses previously installed C/C++ compiler gcc or clang. Omf.cfg must contain external compiler names and options (see 3. - Installation). Omb may be needed also while compiling omfc. Process all the commands below from the Mob-master root dir.

8.2 Compiling examples

A script for compiling and linking all examples.
```
fwe_tomake.bat
fue_tomake.sh
```
Use fwr_/fur_ for 64-bit environments here and below in Ofront.
```
fwr_tomake.bat
fur_tomake.sh
```
Cleans all the executives of example set
```
fwe_toclean
fue_toclean.sh
```

The set of commands compiles all the examples listed
```
Binwe\omfc co -odc :OmtestHelloWorld :OmtestFormats :OmtestDateTime
:OmtestMkTraps :OmtestHeap
Binwe\omfc build -r OmtestDateTime
Binwe\omfc build OmtestHelloWorld
Binwe\omfc build OmtestFormats
Binwe\omfc build OmtestMkTraps
Binwe\omfc build OmtestHeap
Binwe\omfc link -r OmtestDateTime
Binwe\omfc link -r OmtestHelloWorld
Binwe\omfc link -r OmtestFormats
Binwe\omfc link -r OmtestMkTraps
Binwe\omfc link -r OmtestHeap
Binue/omfc co -odc :OmtestHelloWorld :OmtestFormats :OmtestDateTime
:OmtestMkTraps :OmtestHeap
Binue/omfc build -r OmtestDateTime
Binue/omfc build OmtestHelloWorld
Binue/omfc build OmtestFormats
Binue/omfc build OmtestMkTraps
Binue/omfc build OmtestHeap
Binue/omfc link -r OmtestDateTime
Binue/omfc link -r OmtestHelloWorld
Binue/omfc link -r OmtestFormats
Binue/omfc link -r OmtestMkTraps
Binue/omfc link -r OmtestHeap
```
The first command above compiles all the examples listed. The links produce a set of proper executives. We definitely use ':' for main modules and –r for recursive import.

8.3 Running the Examples

8.3.1 The simplest Hello, World example
```
>Omtest\Cfwe\OmtestHelloWorld
```

```
>Omtest/Cfue/OmtestHelloWorld
Hello, World
```

8.3.2 Logging with char, int and real formats
```
>Omtest\Cfwe\OmtestFormats
>Omtest/Cfue/OmtestFormats
```

8.3.3 Shows date, time and delay
```
>Omtest\Cfwe\OmtestDateTime
>Omtest/Cfue/OmtestDateTime
```

8.3.4 Traps handling abilities of runtime
Simple Assert
```
>Omtest\Cfwe\OmtestMkTraps -trap a
>Omtest/Cfue/OmtestKmTraps -trap a
```

Simple Halt
```
>Omtest\Cfwe\OmtestMkTraps -trap h
>Omtest/Cfue/OmtestKmTraps -trap h
```

Zero divide
```
>Omtest\Cfwe\OmtestMkTraps -trap z
>Omtest/Cfue/OmtestKmTraps -trap z
```

Nil pointer dereference
```
>Omtest\Cfwe\OmtestMkTraps -trap p
>Omtest/Cfue/OmtestKmTraps -trap p
```

8.3.5 Dynamic memory and garbage collector
```
>Omtest\Cfwe\OmtestHeap
>Omtest/Cfue/OmtestHeap
```

8.4   Self and cross compilation

A script for self-compiling the compiler omfc is as following:
```
fwe_compiler_tomake.bat
fue_compiler_tomake.sh
```
A script for cross-compiling the compiler omfc by the ombc is as following:
```
fwe_bcompiler_tomake.bat
fue_bcompiler_tomake.sh
```
A script for cross compiling the 64-bit compiler Binwr\omfc by the 32-bit Binwe\omfc is as following:
```
fwr_ecompiler_tomake.bat
fur_ecompiler_tomake.sh
```
A script for compiling the minimal shell omfsh is as following:
```
fwe_sh_tomake.bat
fue_sh_tomake.sh
```

9. <u>Oml LLVM – run from Black Box</u>

Oml LLVM-based backend can produce text code (*.ll) and and binary (*.bc) files, which should be compiled into object files (*.o). Since object files cannot be loaded in BlackBox IDE, the Oml LLVM functionality is limited by producing files only. No dynamic loading and running can be done from BlackBox environment.

9.1   Installation
Preconditions.
      Oml uses LLVM 5.0 Services in LLVMT.dll / LLVMT.so and llvm llc utility distributed with MultiOberon.
      Run: `StartupBlackBox`
Open Oml/Docu/Quick-Start.odc and examples Omtest/Docu/Quick-Start.odc

9.2   Compiling the Oml
Oml compiler modules list for BlackBox is compiled by BlackBox compiler in two steps: LLVM Services and Oml compiler for BlackBox:

```
1.3 Compile LLVM Services
  DevCompiler.CompileThis LlvmC LlvmForAArch64 LlvmForAMDGPU LlvmForARM LlvmForBPF
LlvmForHexagon LlvmForLanai LlvmForMips LlvmForMSP430 LlvmForNVPTX LlvmForPowerPC LlvmForSparc
LlvmForSystemZ LlvmForX86 LlvmForXCore LlvmNative LlvmRefs
  DevCompiler.CompileThis OmcCfgfile OmcTarget OmcCRuntime OmcHooks OmcDialog OmcOPM
OmcOPT OmcOPU OmcOPB OmcOPS OmcOPP OmcDump OmcTester OmcParams OmcCommandParams
OmcOdcSource OmcOdcTextReader OmcExtSource OmcRuntimeStd OmcDialogStd OmcDialogConsole
OmcCompiler OmcConsole OmlOPG OmlOPL OmlOPF OmlOPC OmlOPV OmlParams OmlBackEnd
OmlCompiler OmlLinker
```

All the commands for BlackBox would be recorded in a manner below:
```
^Q DevCompiler.CompileThis LlvmC LlvmForAArch64 LlvmForAMDGPU LlvmForARM
LlvmForBPF LlvmForHexagon LlvmForLanai LlvmForMips LlvmForMSP430
LlvmForNVPTX LlvmForPowerPC LlvmForSparc LlvmForSystemZ LlvmForX86
LlvmForXCore LlvmNative LlvmRefs
^Q DevCompiler.CompileThis OmcCfgfile OmcTarget OmcCRuntime OmcHooks
OmcDialog OmcOPM OmcOPT OmcOPU OmcOPB OmcOPS OmcOPP OmcDump OmcTester
OmcParams OmcCommandParams OmcOdcSource OmcOdcTextReader OmcExtSource
OmcRuntimeStd OmcDialogStd OmcDialogConsole OmcCompiler OmcConsole OmlOPG
OmlOPL OmlOPF OmlOPC OmlOPV OmlParams OmlBackEnd OmlCompiler OmlLinker
```

9.3   Compiling the examples
Compile the following modules (: means producing main for module):
```
^Q OmlCompiler.CompileThis :OmtestHelloWorld :OmtestFormats
:OmtestDateTime :OmtestMkTraps :OmtestHeap
```
Expected result in ~/Omtest/Clwe/ directory (Clue for Unix): OmtestHelloWorld.ll OmtestFormats.ll OmtestDateTime.ll OmtestMkTraps.ll OmtestHeap.ll

Compile the following modules for 64 bit:
```
^Q OmlCompiler.CompileThis -64 :OmtestHelloWorld :OmtestFormats
:OmtestDateTime :OmtestMkTraps :OmtestHeap
```
Expected result in ~/Omtest/Clwr/ directory (Clur for Unix): OmtestHelloWorld.ll OmtestFormats.ll OmtestDateTime.ll OmtestMkTraps.ll OmtestHeap.ll

Building the object file needs to call llc external console program. It's better to use console. But it is also availble (but time-consuming) from BlackBox.

```
^Q OmlLinker.BuildFiles -r OmtestHelloWorld
```

Linking is similar:

```
^Q OmlLinker.LinkExe -r OmtestHelloWorld
```

9.4 [Advanced] Compiling Oml minimal Shell from BlackBox

Oml minimal Shell (OmlShell) is a console exe application with dynamic loading and running support.

Compiling Windows-specific minimal omlsh

```
^Q OmlCompiler.CompileThis OmcLoaderRoutines OmcObjLoader_Coff OmcShell
OmlBcLoader :OmlShell
^Q OmlCompiler.CompileThis -64 OmcLoaderRoutines OmcObjLoader_Coff
OmcShell OmlBcLoader :OmlShell
```

Compiling Linux-specific minimal omlsh

```
^Q OmlCompiler.CompileThis OmcLoaderRoutines OmcObjLoader_Elf OmcShell
OmlBcLoader :OmlShell
^Q OmlCompiler.CompileThis -64 OmcLoaderRoutines OmcObjLoader_Elf
OmcShell OmlBcLoader :OmlShell
```

9.5 Unloading Oml Compiler

```
^Q DevDebug.UnloadThis OmlCompiler OmlLinker OmlBackEnd OmlParams OmlOPV
OmlOPC OmlOPF OmlOPL OmlOPG OmcCompiler OmcDialogStd OmcRuntimeStd
OmcOdcSource OmcCommandParams OmcParams OmcTester OmcDump OmcOPP OmcOPS
OmcOPU OmcOPB OmcOPT OmcOPM OmcDialog OmcHooks OmcCRuntime OmcTarget
OmcCfgfile Runner OLog HostTimes Times OStrings Testing
```

10.   Oml LLVM - from Command Line.

10.1   Installation

Preconditions.

      Oml uses LLVM 5.0 Services in LLVMT.dll / LLVMT.so and llvm llc utility distributed with MultiOberon. Oml.cfg must contain external compiler names and options (see 3. - Installation). Process all the commands below from the Mob-master root dir.

10.2   Compiling examples

A script for compiling and linking all examples.

```
lwe_tomake.bat
lue_tomake.sh
```

Use lwr_/lur_ for 64-bit environments here and below in Oml LLVM.

```
lwr_tomake.bat
lur_tomake.sh
```

Cleans all the executives of example set

```
lwe_toclean
lue_toclean.sh
```

The set of commands compiles all the examples listed

```
Binwe\omlc co -odc :OmtestHelloWorld :OmtestFormats :OmtestDateTime
:OmtestMkTraps :OmtestHeap
Binwe\omlc build -r OmtestDateTime
Binwe\omlc build OmtestHelloWorld
Binwe\omlc build OmtestFormats
Binwe\omlc build OmtestMkTraps
Binwe\omlc build OmtestHeap
Binwe\omlc link -r OmtestDateTime
Binwe\omlc link -r OmtestHelloWorld
Binwe\omlc link -r OmtestFormats
Binwe\omlc link -r OmtestMkTraps
Binwe\omlc link -r OmtestHeap
Binue/omlc co -odc :OmtestHelloWorld :OmtestFormats :OmtestDateTime
:OmtestMkTraps :OmtestHeap
Binue/omlc build -r OmtestDateTime
Binue/omlc build OmtestHelloWorld
Binue/omlc build OmtestFormats
Binue/omlc build OmtestMkTraps
Binue/omlc build OmtestHeap
Binue/omlc link -r OmtestDateTime
Binue/omlc link -r OmtestHelloWorld
Binue/omlc link -r OmtestFormats
Binue/omlc link -r OmtestMkTraps
Binue/omlc link -r OmtestHeap
```

The first command above compiles all the examples listed. The links produce a set of proper executives. We definitely use ':' for main modules and –r for recursive import.

10.3   Running the Examples

10.3.1 The simplest Hello, World example

```
>Omtest\Clwe\OmtestHelloWorld
```

```
>Omtest/Clue/OmtestHelloWorld
Hello, World
```

10.3.2 Logging with char, int and real formats
```
>Omtest\Clwe\OmtestFormats
>Omtest/Clue/OmtestFormats
```

10.3.3 Shows date, time and delay
```
>Omtest\Clwe\OmtestDateTime
>Omtest/Clue/OmtestDateTime
```

10.3.4 Traps handling abilities of runtime
Simple Assert
```
>Omtest\Clwe\OmtestMkTraps -trap a
>Omtest/Clue/OmtestKmTraps -trap a
```

Simple Halt
```
>Omtest\Clwe\OmtestMkTraps -trap h
>Omtest/Clue/OmtestKmTraps -trap h
```

Zero divide
```
>Omtest\Clwe\OmtestMkTraps -trap z
>Omtest/Clue/OmtestKmTraps -trap z
```

Nil pointer dereference
```
>Omtest\Clwe\OmtestMkTraps -trap p
>Omtest/Clue/OmtestKmTraps -trap p
```

10.3.5 Dynamic memory and garbage collector
```
>Omtest\Clwe\OmtestHeap
>Omtest/Clue/OmtestHeap
```

10.4   Self-compilation

A script for self-compiling the 32-bit compiler omlc is as following:
```
lwe_compiler_tomake.bat
lue_compiler_tomake.sh
```
A script for self-compiling the 64-bit compiler omlc is as following:
```
lwr_compiler_tomake.bat
lur_compiler_tomake.sh
```
A script for compiling the minimal 32-bit shell omlsh is as following:
```
lwe_sh_tomake.bat
lue_sh_tomake.sh
```
A script for compiling the minimal 64-bit shell omlsh is as following:
```
lwr_sh_tomake.bat
lur_sh_tomake.sh
```

11.   Usage and Traverse

MultiOberon adds Usage files *.ouf, which contains imports lists and restriction info. These files are located in Sym or S[backend][os][arch] catalogs. Usage is a binary file of the format (item, value)* :

- item – integer number of option;
- value – string constant as option value.

The following options are used for now:

16	sNAME	Module name
50	sIMPNAME	Import Module name
51	sIMALIAS	Import Module alias
52	sLIBCODE	Library name with code
53	sLIBNOCODE	Library name with no code

Usage files afford to traverse through all imported modules tree. Let us look at example:

```
>Binwe\ombc trav OmtestHelloWorld
>Binue/ombc trav OmtestHelloWorld
OmtestHelloWorld
  Runner
    SYSTEM
    ?c:\suok5\dsu/System/Slwe/SYSTEM.ouf
    Kernel
      Api
      -Api [libcmt]
      OLog
        OStrings
        -OStrings
      -OLog
    -Kernel
  -Runner
-OmtestHelloWorld
```

Runner is imported from OmtestHelloWorld, SYSTEM and Kernel are imported from Runner, Api and OLog is imported from Kernel. SYSTEM was never compiled, so there is no SYSTEM.ouf file. Api is a library named 'libcmt'.

It must be mentioned that import information is backend-specific; it can be placed in .ocf file for BlackBox and .c file for Ofront. In order to avoid specific time-consuming parsing when getting module imports, it was decided to add explicit .ouf files.

The traverse algorithms are widely used in recursive builds and links when –r option is enabled.

12.   Dynamic Modules

MultiOberon implements dynamic loading for all backends listed above. It can be done in the following ways:

```
>Binwe\ombc run OmtestHelloWorld
>Binwe\ombsh OmtestHelloWorld
>Binwe\omlsh OmtestHelloWorld –ext bc
>Binue/ombc run OmtestHelloWorld
>Binue/ombsh OmtestHelloWorld
>Binue/omlsh OmtestHelloWorld –ext bc
```

The first and second can be implemented in all platforms for all backends, the last is in all platforms for Oml LLVM only. The dynamically loaded module must be prepared in binary form according to table below.

	Windows	Unix
Omb	.ocf BlackBox	.ocf BlackBox
Omf	.o COFF-format	.o ELF-format
Oml	.o COFF-format	.o ELF-format
Oml –ext bc	.bc LLVM	.bc LLVM

The .ocf and .bc files are created during the compilation.
The .o object files are created during build process.

Usage of .bc LLVM files activates LLVM JIT-compiler, which is much heavier than normal object loading. Therefore, JIT usage seems impractical and too time-consuming.

In order to implement all the features above, the Baseloader module was implemented in System. Specific modules as OmcOcfLoader, OmcObjLoader_Coff, OmcObjLoader_Elf, and OmlBcLoader implement specific functionality of ocf, coff, elf and bc loading.

13.   Runner module

MultiOberon uses special Runner module to be used in different environments. The Runner.SetRun procedure registers function MAIN to be called after loading the shell.

```
MODULE OmtestMkTraps;
      IMPORT Runner, OLog, SYSTEM;
      PROCEDURE MAIN*;
            VAR str: Runner.SName;
      BEGIN
            IF ~Runner.StringOpt("-trap", str) THEN
                  OLog.String("usage: "); OLog.SString(Runner.argV0);
                  OLog.String(" -trap"); OLog.Ln;
                  OLog.Tab; OLog.String("where -trap is as following:"); OLog.Ln;
                  OLog.Tab; OLog.String("a - assert"); OLog.Ln;
                  OLog.Tab; OLog.String("h - halt"); OLog.Ln;
                  OLog.Tab; OLog.String("z - zero divide"); OLog.Ln;
                  OLog.Tab; OLog.String("p - nil pointer dereference"); OLog.Ln;
            ELSE
                  RunOpt(str);
            END;
      END MAIN;

BEGIN
      Runner.SetRun(MAIN)
END OmtestMkTraps.
```

Runner also has command line option parsing routines as Runner.StringOpt(), Runner.IntOpt().

Runner constants determine platform-specific values as following:
Runner.RUN_TIME = "OMB" | "OMF" | "OML" for BlackBox, Ofront, LLVM environments.
Runner.OS_NAME = "Windows" | "Unix"
Runner.BIN_BITS = 64 | 32
Runner.KERNEL_VERSION = 16 | 17 | 18 for BlackBox Kernel

14.   Testing abilities

Testing abilities cover the separate modules and compiler as a whole. For example, we are testing OmtestSimple module with a PROCEDURE Sum(x, y: INTEGER): INTEGER. We need a [module_name]Test module – OmtestSimpleTest.

```
MODULE OmtestSimpleTest;
      IMPORT T := Testing, OmtestSimple;
      PROCEDURE Test0Basic* (VAR rec: T.Rec);
      BEGIN
            CASE rec.n_test OF
            | 0:
                  rec.res_type := T.RES_INT;
                  rec.msg := 'Sum x+x';
                  rec.i_req := 6;
                  rec.i_res := OmtestSimple.Sum(3, 0);
            | 1:
                  rec.res_type := T.RES_INT;
                  rec.msg := 'Sum x+y';
                  rec.i_req := 5;
                  rec.i_res := OmtestSimple.Sum(3, 2);
            | 2:
                  rec.finish := TRUE;
            ELSE
            END;
      END Test0Basic;
END OmtestSimpleTest.
```

The OmtestSimpleTest module implements set of routines Test*, called by testing environment. Test0*, Test1* naming convention in used for printing in sorted up-down order.

Each test sets the fields of rec: Testing.Rec. Result type res_type is set first to show what fields are specified. Message msg shown outputs a record for user. For each type, result is compared with required value. In our case i_res is integer result, i_req is integer required value.
Testing environment clears each test before call; only set number and test number (n_test) are specified. In case of any mismatch error in printed by testing system.

The compiler shells tests can be loaded and ran dynamically. So modules must be prepared for dynamic loading before:

```
>Binwe\ombc co -odc OmtestSimple OmtestSimpleTest
>Binue/ombc co -odc OmtestSimple OmtestSimpleTest
```

After compilation use test command in compiler shell:

```
>Binwe\ombc test OmtestSimple
>Binue/ombc test OmtestSimple
[ALL] ========== Total 2 tests, 0 bad, result= 100.0%
```

Print-level option –pl provides more information:

```
>Binwe\ombc test -pl 3 OmtestSimple
>Binue/ombc test -pl 3 OmtestSimple
[OmtestSimpleTest.Test0Basic.000] INT i_res= 6 i_req= 6 :Sum x+x
[OmtestSimpleTest.Test0Basic.001] INT i_res= 5 i_req= 5 :Sum x+y
[OmtestSimpleTest.] ---------- In module 1 sets, 2 tests, 0 bad
[ALL] ========== Total 2 tests, 0 bad, result= 100.0%
```

Special tests for compiler were also provided:

```
bwe_tests_tomake.bat
bue_tests_tomake.sh
```

The following tests compiled (OmtestOmc*) are not connected to the specific module:

```
Binwe\ombc co -odc OmtestOmcSimpleTest OmtestOmcStringsTest
OmtestOmcSystemTest OmtestOmcImportsTest OmtestOmcExtensionsTest
OmtestOmcBoundTest OmtestOmcAdvancedTest
```

Special tests for compiler were also provided. Running shown console output like this:

```
bwe_tests_run.bat
bue_tests_run.sh
```

```
c:\suok5\dsu>Binwe\ombc test -pl 2 OmtestOmcSimpleTest
OmtestOmcStringsTest OmtestOmcSystemTest OmtestOmcImportsTest
OmtestOmcExtensionsTest OmtestOmcBoundTest OmtestOmcAdvancedTest
[OmtestOmcSimpleTest.] ---------- In module 9 sets, 104 tests, 0 bad
[OmtestOmcStringsTest.] ---------- In module 5 sets, 64 tests, 0 bad
[OmtestOmcSystemTest.Test1Addr.028] ?LONG li_res= 12 li_req= 16 :Proper
position of ptr after long
[OmtestOmcSystemTest.Test1Addr.029] ?LONG li_res= 16 li_req= 24 :SIZE of
Rec with LONGINT aligned by 8
[OmtestOmcSystemTest.Test1Addr.030] ?LONG li_res= 12 li_req= 16 :Proper
position of ptr after Rec with long
[OmtestOmcSystemTest.Test1Addr.031] ?LONG li_res= 12 li_req= 16 :SIZE of
Rec with LONGINT and char
[OmtestOmcSystemTest.Test1Addr.032] ?LONG li_res= 16 li_req= 24 :SIZE of
Rec with Rec with LONGINT aligned by 8
[OmtestOmcSystemTest.] ---------- In module 3 sets, 49 tests, 5 bad
[OmtestOmcImportsTest.] ---------- In module 6 sets, 59 tests, 0 bad
[OmtestOmcExtensionsTest.] ---------- In module 10 sets, 80 tests, 0 bad
[OmtestOmcBoundTest.] ---------- In module 6 sets, 95 tests, 0 bad
[OmtestOmcAdvancedTest.] ---------- In module 6 sets, 42 tests, 0 bad
[ALL] ========== Total 493 tests, 5 bad, result= 98.98580121703854%
```

JIT-compiled tests for Oml LLVM can be run via scripts:

```
lwe_tests_jit_run.bat
lue_tests_jit_run.sh
```

The compiler tests provided are also linked in an executive OmtestOmcCompiler, like:

```
Omtest\Cfwe\OmtestOmcCompiler -pl 2
Omtest/Cfue/OmtestOmcCompiler -pl 2
```

The list of failed tests demonstrates the quality of specific backend. In is used for further compiler development.

15.   Benchmarking abilities

Benchmarking abilities are used for time measurements of particular routines. We need a [module_name]Test module – for example, OmtestBenchRoutinesTest to measure OmtestBenchRoutines.

```
MODULE OmtestBenchRoutinesTest;
     IMPORT T := Testing, OmtestBenchRoutines;
     PROCEDURE BenchmarkPalindrome* (IN bench: T.Bench; VAR num_done: INTEGER);
          VAR count: INTEGER;
     BEGIN
          num_done := 0; count := 0;
          WHILE num_done < bench.num DO
               IF OmtestBenchRoutines.IsPalindrome(
               "A man, a plan, a canal: Panama") THEN
                    INC(count)
               END;
               INC(num_done)
          END
     END BenchmarkPalindrome;
END OmtestBenchRoutinesTest.
```

The OmtestBenchRoutinesTest module implements set of routines Benchmark*, called by testing environment. Each routine has an input structure Testing.Bench. Parameter bench.num sets the required number of iterations. Output parameter num_done sets the real number of iterations, to ensure that required number of repetitions was not optimized.

The compiler shells bench tests can be loaded and ran dynamically. So modules must be prepared for dynamic loading before:

```
>Binwe\ombc co -odc OmtestBenchRoutines OmtestBenchRoutinesTest
>Binue/ombc co -odc OmtestBenchRoutines OmtestBenchRoutinesTest
```

After compilation use bench command in compiler shell:

```
>Binwe\ombc bench OmtestBenchRoutines
>Binue/ombc bench OmtestBenchRoutines
[OmtestBenchRoutinesTest.BenchmarkPalindrome] 1000000
00:00:00.282000 282.0 ns/op
```

The testing environment reports that 1000000 iterations were done in 282 milliseconds. The average time is 282.0 nanosec/operation:

16.   Developing platform-specific modules

MultiOberon is for multi-platform development. Some modules procedures do not depend on platform-specific features, other procedures do. These features include operating system external procedures, different data structures for 32 and 64 bit, Kernel-specific interfaces.
Each module source can have specific versions, marked as [module]_[specific]. Modules names in MultiOberon cannot have underscore symbols inside, error "string expected" must occur if any. However, file names can have underscores and specific extensions for module. Moreover, the underscore in file name means platform-specific implementation of module code. For example, there are two quite different versions of OmcObjLoader:
- OmcObjLoader_Coff – for Windows COFF object file format;
- OmcObjLoader_Elf – for Unix ELF object file format.

So OmcObjLoader_Coff is compiled to OmcObjLoader.o in Windows script and OmcObjLoader_Elf is compiled to OmcObjLoader.o in Unix script.

Platform-specific modules obviously have different interfaces. Those interfaces differ from platform. Compiling platform-independent module:

```
>Binwe\ombc co -odc OmcDiscomp
omb:compiling c:\suok5\dsu/Omc/Mod/Discomp.odc
new symbol file >c:\suok5\dsu/Omc/Code/Discomp.ocf code=3652 glob=8716
```
Platform-independent module interface for BlackBox is in Omc/Sym and code is in Omc/Code.

Compiling OmcObjLoader_Coff means platform-specific module. It is compiled as:

```
>Binwe\ombc co -odc OmcObjLoader_Coff
omb:compiling c:\suok5\dsu/Omc/Mod/ObjLoader_Coff.odc
new symbol file >c:\suok5\dsu/Omc/Cbwe/ObjLoader.ocf code=18480 glob=32
```
Platform-dependent module interface for BlackBox is in Omc/Sbwe and code is in Omc/Cbwe.

Even Platform-independent modules for Ofront and LLVM are compiled into separated directories:

```
>Binwe\omfc co -odc OmcDiscomp
omf:compiling c:\suok5\dsu/Omc/Mod/Discomp.odc
>c:\suok5\dsu/Omc/Cfwe/OmcDiscomp .c=26490 .h=3555
```
Platform-independent module interface for Ofront is in Omc/Sfwe and code is in Omc/Cfwe.

The most part of platform-specific modules is located in System and Host Directory. They are compiled during the installation phase separated from user's development process.
Naming conventions used some letters in MultiOberon:
- b-BlackBox, f-Ofront, l-LLVM;
- w-Windows, u-Unix;
- e-X86, r-X64;
- 16-BlackBox 1.6, 17-BlackBox1.7.

The Runner module needs 15 platform-specific code modules.

	Omb	Omf32	Omf64	Oml32	Oml64
BlackBox17Win	Runner_bwe17	Runner_fwe17	Runner_fwr17	Runner_lwe17	Runner_lwr17
BlackBox16Win	Runner_bwe16	Runner_fwe16	Runner_fwr16	Runner_lwe16	Runner_lwr16
BlackBox17Unix	Runner_bue17	Runner_fue17	Runner_fur17	Runner_lue17	Runner_lur17

It is too inconvenient to develop all 15 platform-specific modules separately, so common tools are used in the section below.

17.   Generic modules Preprocessing

Generic tools are not included in MultiOberon compiler. Generic tools are not planned to be included in future. However, the simple .odc processing utility is included in Omc system. OmcPrep is a text preprocessor to convert from generic to specific modules.

According to MultiOberon convention, generic modules are located in GMod directory; specific modules are located in Mod. System/Docu/Quick-Start.odc demonstrates how specific modules were generated. For example, the command for generating Runner_fwr17 from GMod/Runner is as following:

```
^Q OmcPrep.ToOdcFileList('OMF WIN V64 BB17', 'System/Mod')"
@Omc/Mod/Defs.odc System/GMod/Runner.odc:Runner_fwr17
```

First environment constants are set: OMF, WIN, V64, and BB17. Next the Defs file is loaded with specific values, like @ADDR=LONGINT (it's 64-bit, V64 is defined). Then GMod/Runner is transformed to Runner_fwr17.

```
#IF @BB
          SP = 4; DLT_STACK = 256;
          RUN_TIME* = "OMB";
#ELSIF @OMF
          RUN_TIME* = "OMF";
#ELSIF @OML
          RUN_TIME* = "OML";
#END
          SysTrapProc = PROCEDURE (n: INTEGER; stpa: @ADDR);
```
The result of the transformation in Runner_fwr17 should be:
```
          RUN_TIME* = "OMF";
          SysTrapProc = PROCEDURE (n: INTEGER; stpa: LONGINT);
```

The Defs file contains specific setting for all required platforms.
The macro commands are quite similar, only conditions and name substitutions are supported.

18.   Restrictions

MultiOberon is restriction-based scaling Oberon environment with an initial condition as Component Pascal syntax. RESTRICT operator is used for enabling/disabling major language features. The Restrict system holds a set of special profiles. RestrictAdrint module simply enables ADRINT as integer of address size:

```
RESTRICT +ADRINT*;
```

The '*' symbol means that the restriction is exported for modules with IMPORT RestrictAdrint. RestrictOberon07 profile uses more complex rules, like:

```
RESTRICT -CLOSE*, -EXIT*, -LOOP*, -OUT*, -WITH*,
     -PROCEDURE(PROCEDURE)*,    (* Recursion *)
     -BEGIN(PROCEDURE)*,   (* Nested Procedures *)
     -RETURN(PROCEDURE)*;  (* Single Return in the End only *)
```

OmtestOmcRestrictTest demonstrates ADRINT usage. ADRINT var can hold address as integer value.

```
IMPORT Api, RestrictAdrint;

PROCEDURE Xxx;

    VAR ai: ADRINT;

BEGIN

    ai := SYSTEM.VAL(ADRINT, Api.TestGetAdr());

END Xxx;
```

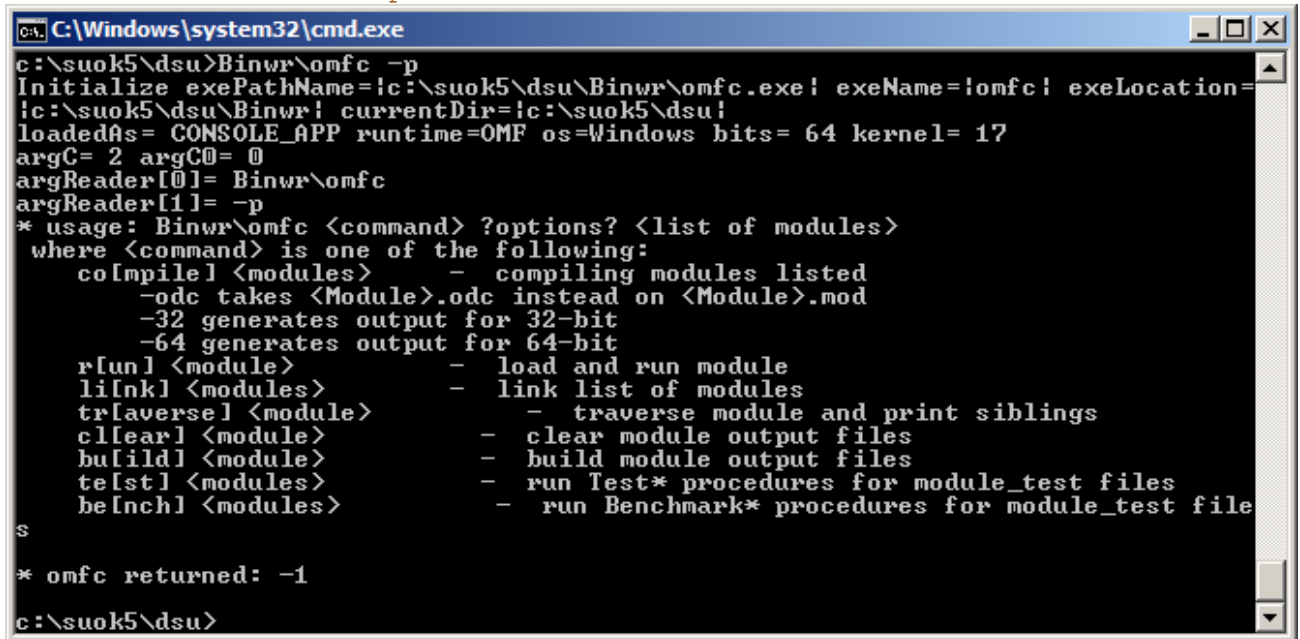OmtestOmcRestrictTest can be compiled and executed:

```
>Binwe\ombc co -odc OmtestOmcRestrictTest
>Binue/ombc co -odc OmtestOmcRestrictTest
>Binwe\ombc test -pl 2 OmtestOmcRestrictTest
>Binue/ombc test -pl 2 OmtestOmcRestrictTest
[ALL] ========== Total 14 tests, 0 bad, result= 100.0%
```

Additional functionality RESTRICT+ is implemented for ADRINT only.

19.   Commands and command line options

MultiOberon compiler om[backend]sh with no arguments shows usage print. Option –p prints
Runner constants and globals, usage prints commands available.
> Binwr\omfc co –p

```
c:\suok5\dsu>Binwr\omfc -p
Initialize exePathName=!c:\suok5\dsu\Binwr\omfc.exe! exeName=!omfc! exeLocation=
!c:\suok5\dsu\Binwr! currentDir=!c:\suok5\dsu!
loadedAs= CONSOLE_APP runtime=OMF os=Windows bits= 64 kernel= 17
argC= 2  argC0= 0
argReader[0]= Binwr\omfc
argReader[1]= -p
* usage: Binwr\omfc <command> ?options? <list of modules>
 where <command> is one of the following:
     co[mpile] <modules>    -   compiling modules listed
          -odc takes <Module>.odc instead on <Module>.mod
          -32 generates output for 32-bit
          -64 generates output for 64-bit
     r[un] <module>             -   load and run module
     li[nk] <modules>          -   link list of modules
     tr[averse] <module>        -   traverse module and print siblings
     cl[ear] <module>          -   clear module output files
     bu[ild] <module>          -   build module output files
     te[st] <modules>          -   run Test* procedures for module_test files
     be[nch] <modules>          -   run Benchmark* procedures for module_test file
s

* omfc returned: -1

c:\suok5\dsu>
```

The command line options are:

-odc – use .odc file instead of .mob;

-32 – 32-bit mode;

-64 – 64-bit mode;

-os – Windows|Linux used for cross compiling;

-r – recursive imports traversal;

-h – no HostConLog import;

-n – recompile only new files;

-tl – trace_level (0-4);

-pl – print_level (0-3);

-ht – handler type: 1-dlink, 2-frame pointer, 3-stack analysis;

-wsd – write to System Dir.

These options can be extended by dynamic modules specific options. The latter are processed in
loaded modules, not in compiler.

20.   Change Log

may 2019 original MultiOberon pre-version 0.8 released
nov 2019 MultiOberon pre-version 0.9 released
jun 2020 MultiOberon pre-version 0.95 released
nov 2020 MultiOberon version 1.0 released

Use it and enjoy! - ŸЪsalos y disfrъtalos! - Bonne utilisation - Приятного использования - Powodzenia - Viel SpaЯ

Dmitry V. Dagaev
dvdagaev@oberon.org