DOCUMENTATION

ON USING

INTEGRATED DEVELOPMENT ENVIRONMENT

FREE OBERON

AND PROGRAMMING LANGUAGE

OBERON

Revision 18.6.2019
for Free Oberon version 1.0.3

PUBLISHING HOUSE
OF TECHNO-THEORETICAL LITERATURE
OF FREE OBERON DEVELOPER TEAM

RIGA 2019

Table of Contents

1. Installation.

Free Oberon is a cross-platform integrated development environment. It is available for Windows in a form of a EXE setup program, as well as for GNU/Linux and other UNIX-like systems in a form of source code, that you need to compile, having preinstalled some necessary libraries on the system.

1.1. Installation under GNU/Linux.

1. Download Free Oberon source code from `freeoberon.su` in tar.gz format or from the GitHub repo. Note that the archive with the version for Windows is also suitable, because it contains the source code. Extract the archive to your home directory or to another location on the disk. (This tutorial will assume the files are extracted to the home directory.)

2. Using terminal or in any other way, install the following packages:
   ```
   libsdl2-dev             libsdl2-image-dev
   binutils                gcc
   make
   ```

   The names of the packages are given in accordance with their names in the Debian GNU/Linux operating system. They are also suitable for Ubuntu, Linux Mint, Raspbian and other. To install them, run the following command:
   ```
   apt-get install -y libsdl2-dev libsdl2-image-dev
                      binutils gcc make
   ```

   (This command must be executed with superuser privileges, that is, you must first run `su` and enter the password.)
   On OS Fedora, Red Hat, CentOS and others, the command and package names will differ (one of two packages `glibc-static` or `glibc-devel-static` might also be required):
   ```
   sudo yum install SDL2-devel SDL2_image-devel
        glibc-devel-static binutils gcc make   (untested!)
   ```

3. Go to the "`src`" subdirectory and start the compilation:
   ```
   cd ~/FreeOberon/src
   make -f Makefile_linux
   ```

4. (optional) Append the following line to the end of file "`~/.bashrc`":
   ```
   alias fo='cd ~/FreeOberon;./FreeOberon'
   ```

   This will allow you to launch Free Oberon using the "`fo`" command.

1.2. Installation under Windows.

Download the setup porgram in EXE format from `freeoberon.su` website, run it and follow the instructions.

Alternatively, you can download a version of Free Oberon in a ZIP-archive, extract it to any place on the disk and create a desktop shortcut.

Note. If you want to recompile Free Oberon under Windows from the source code yourself, refer to Appendix A of this document.

2. Checking if system works.

Run Free Oberon, press `F3` (“`File → Open`”) and open “`Book.Mod`”. Press `F9` (“`Make and Run`”). If everything works as expected, you should see a picture of a book.

Some possible issues:

1. “`Module Graph not found`”.

   An incorrect or incomplete version of VOC is used. The modified version is supplemented with `Graph` and `SDL2` modules, which are necessary for programming graphics. Make sure the files `Graph.sym`, `SDL2.sym`, `Graph.h`, `SDL2.h` and `SDL2.h0` are in place (sym-files should be in directory “`data/bin/voc/C/sym`”, and others should be located in directory “`data/bin/voc/C/include`”.
2. “`FLOOR – undefined identifier`”.

   An incorrect version of VOC is used. The modified version has a built-in operator `FLOOR(x)`, which essentially works in the same way as `SHORT(ENTIER(x))`, converting value of type `REAL` or `LONGREAL` into a value of type `INTEGER` (and not `LONGINT`).
3. Error “`.`”.

   The command “`voc`” cannot be found or one of the “`data/bin/*.sh`” files is not marked as executable (on GNU/Linux).

3. Writing a program.

Run Free Oberon and type in the source code of a module. A module always starts with a keyword `MODULE`, which is followed by the module's name. The name of the module should be written as one word, start with a latin letter and contain only latin letters and numbers. Then follows a semicolon. For example, “`MODULE Prog1;`”. The module's text ends with a keyword `END`, which again is followed by the name of the module and a period (“.”):

```
MODULE Prog1;
  (*the program is being put between these two lines*)
END Prog1.
```

4. Saving a file.

To save a file, press the `F2` key or click “`File → Save As`”, and you will be aked for a file name. For the module to work, the file should be named in the same way as the module, but with “`.Mod`” in the end (with a capital `M`). For example, “`Prog1.Mod`”. If the module name and the file name, in which the module is stored, are different, then it would be impossbile to run the compiled module from within Free Oberon.

Subsequent pressing of the F2 key will save the module to a file with the same name. To save the module to a file with a different name, click "File → Save As..." or press Shift+F2.

The saved files are plain text files in UTF-8 encoding, they are put in the subdirectory "Programs" and, if required, can be edited using other text editors.

5. Running a program.

To compile and run a program, click "Run → Compile & Run" or press F9. The file must be previously saved, and the file name must match the module name (see above "4. Saving File"). If the file has not been saved, the Save dialog box will be opened, in which case you should save the file, and then press F9 again.

6. Openning a file.

Press F3 and enter file name, then press Enter. If the file exists, it will be opened, and you will see the file name in the title of the opened window. If you edit file ans click "File → Save" or press the F2 key, then a file will be saved in the same place. You can copy a file by saving it using a different name, for this, click "File → Save As..." or press Shift+F2.

Menu item "File → Reload" is useful in case the file have been opened already and you want to load it from the disk again (for example, to roll back unsuccessful changes or if the file on the disk has been updated).

To create a new file, click "File → New" or press Shift+F3.

7. Navigating the text.

You can navigate the text with the mouse pointer, but it's much more convenient to do this using the keyboard.

The arrow keys allow you to move one character left and right, and one line up and down. To quickly move to the beginning of the line, press the Home key, and to move to the end of the line, you can press the End key. Using the PageUp and PageDown keys, you can move up or down by one screen, which is convenient when working with large files.

The Tab key will put one or two spaces, depending on how far from the left edge of the window the text cursor is. This can be convenient for indenting the text of the program.

When you press the Enter key, a new line is inserted at the current position, and a number of spaces are automatically added to it – the same number as the number of spaces in the beginning of the previous line (this is called "auto-indentation").

"Hanging" spaces at the end of each line are indicated by dots.

8. Copying lines.

To copy a line, you must first select it:

1. Move the text cursor to the beginning of the line you want to copy (use the Up/Down keys and the Home key, which quickly moves the cursor to the beginning of the line.
2. Hold down the `Shift` key and, without releasing it, press the down arrow key once. The cursor moves one line down, and the line to be copied becomes selected. (You can select several lines in the same way.)
3. Press `Ctrl+C` to copy the line to the clipboard.
4. Press `Ctrl+V` to make the copied line appear on the text cursor position (it will be inserted from the clipboard). The operation `Ctrl+V` can be performed several times.

To move the line instead of copying it, instead of `Ctrl+C` press `Ctrl+X`.

If you just want to delete the selected text, click "`Edit → Clear`" or press the `Delete` key or `Ctrl+Del`.

To selecte all text, press `Ctrl+A`.

All the above operations are also available from the "`Edit`" menu.

9. Moving between opened windows.

If you opened several windows with source code, you can easily move from one window to another with the `F6` key. Movement in the opposite direction is carried out using the keyboard shortcut `Shift+F6`. You can close the window with `Alt+F3`, maximize the window and normalize it with `F5`. All these actions are available in the "`Window`" menu, where you can also see the corresponding hotkeys.

You can move between windows, resize and close them using the mouse. In the lower right corner of the window there is a special handle, pulling which the window can be resized. You can move the window by moving its title.

10. A simple program.

Run Free Oberon and enter the following program:

```
MODULE MyProg;
IMPORT In, Out;
VAR a, b, c: INTEGER;
BEGIN
  Out.String('Enter  first term: '); In.Int(a);
  Out.String('Enter second term: '); In.Int(b);
  c := a + b;
  Out.String('The sum is '); Out.Int(c, 0); Out.Ln
END MyProg.
```

Save the file as "MyProg.Mod" and press F9. If everything has been entered correctly, the program will start and ask for two numbers, then display their sum and terminate.

The IMPORT section describes the modules used. In this example, modules In and Out are used. Modules are given as a comma-separated list and each of them can be used below, in the source code.

The VAR section lists variables and their types. A variable is a named memory block in which a certain value is stored (each variable has a name). The type of variable describes the set of values that a variable can take (be equal to). In this example, three variables are declared: a, b and c, and all three are of type INTEGER.

After the keyword BEGIN, follow the commands that the program will execute. Between each two adjacent commands there is a semicolon. At the end of the last command, a semicolon is allowed, but it is not mandatory, and therefore we will not put it (see the Out.Ln in the text of the program).

After some commands, there is a list of *parameters* in parentheses that are transferred to them. For example, after Out.String, there the text is indicated in quotation marks and in parentheses. This text is passed to the Out.String command, which outputs it to the screen.

In.Int suspends execution of the program until user enters a number and presses the Enter key, after which the entered value is stored in the variable specified in parentheses.

The command "c := a + b" is the so-called *assignment operator*. It calculates the value to the right of ":=", and the resulting value is written to the variable specified on the left. The command "c := a + b" can be read as follows: "Calculate the sum of $a + b$ and write it to the variable c".

Out.Int(c, 0) displays the number c on the screen. The second parameter 0 indicates the minimum number of characters that the displayed number should occupy on the screen. For example, Out.Int(14, 5) will display three spaces and number 14, resulting in 5 digits.

Out.Ln serves two functions. It moves the text cursor to the new line (so that the following text will be displayed on the next line) and ensures that the text previously displayed becomes visible on the screen. This means that if you do not execute the Out.Ln at the end of the program, then probably not all of the text that has been output previously will be displayed on the screen.

11. Basic data types.

Each variable must be of a certain type. The type specifies a set of possible values. The following basic types exist:

INTEGER – a whole number in range –2 147 483 648 to 2 147 483 647.
REAL – a real (fractional) number in range 10^{-38} to 10^{38}.
CHAR – a single character, i.e. one letter, digit, space etc.

`BOOLEAN` – logical (boolean) type, that can be either `TRUE` or `FALSE`.
`SET` – a set, that can include whole numbers from 0 to 31.

12. Console input and output. Modules In, Out.

Console input is usually done using the keyboard, and console output is usually being reflected on the screen. In the Oberon language, console input is done using module `In`, and the console output is done using module `Out`.

Module `In` contains the following *procedures*:

```
Int(VAR i: INTEGER)                 – input of a whole number
Real(VAR x: REAL)              – input of a real number
Char(VAR ch: CHAR)                  – input of a single character
Line(VAR str: ARRAY OF CHAR)  – input of a text string
```

Module `Out` contains the following procedures:

`Int(i, n: INTEGER)`	Prints an integer *i* so that it occupies at least *n* characters, optionally adding left blanks on the left.
`Hex(i, n: INTEGER)`	Same as `Int`, but the number is displayed in hexadecimal.
`Real(x: REAL; n: INTEGER)`	Outputs a real number (width of n characters) in scientific form.
`RealFix(x: REAL; n, k: INTEGER)`	Displays a floating-point number in decimal form (with a width of *n* characters) with *k* decimal places.
`Char(VAR ch: CHAR)`	Outputs a single character
`String(str: ARRAY OF CHAR)`	Outputs a text string
`Ln`	Goes to the new line, but before that flushes the output buffer (performs `Flush`).
`Flush`	Flushes the output buffer. As a result of this command, all previously printed text is displayed on the screen.

Examples:

```
Out.String('Hello '); Out.Int(123, 0); Out.Int(a, 4); Out.Ln;
Out.String('x = '); Out.RealFix(x, 0, 2); Out.Flush
```

13. Module Math.

Module `Math` contains some math functions and is usually imported under the pseudonym "`M`":

```
IMPORT M := Math;
```

Some procedures of module `Math`:

`round(x: REAL): LONGINT` – number *x*, rounded to the nearest integer,
`sqrt(x: REAL): REAL` – square root of *x*,
`exp(x: REAL): REAL` – number *e* in power *x*,

`ln(x: REAL): REAL` – natural logarithm of *x*,
`sin(x: REAL): REAL` – sine of angle *x*, expressed in radians,
`cos(x: REAL): REAL` – cosine of angle *x*, expressed in radians,
`tan(x: REAL): REAL` – tangent of angle *x*, expressed in radians,
`arcsin(x: REAL): REAL` – arcsine of number *x*, expressed in radians,
`arccos(x: REAL): REAL` – arccosine of number *x*, expressed in radians,
`arctan(x: REAL): REAL` – arctangent of number *x*, expressed in radians,
`power(base, exp: REAL): REAL` – number *base* in power *exp*,
`ipower(x: REAL; base: INTEGER): REAL` – number *x* in (whole) power *base*,
`log(x, base: REAL): REAL` – logarithm of number *x* on base *base,*
`sincos(x: REAL; VAR Sin, Cos: REAL)` – sine and cosine of angle *x* (in rad.),
`arctan2(xn, xd: REAL): REAL` – arctangent of quotient *xn/xd* (rad.),
as well as hyperbolic functions: `sinh(x), cosh(x), tanh(x), arcsinh(x), arccosh(x), arctanh(x).`

14. Module Graph.

Module `Graph` allows you to program graphics and sound, do more in-depth interaction with the keyboard and generate pseudo-random numerical sequences. Module `Graph` is usually imported under the alias "`G`":

```
IMPORT G := Graph;
```

A simple graphics example:

```
MODULE GrTest;
IMPORT G := Graph;
VAR screen: G.Bitmap;
BEGIN
  screen := G.Init();
  G.Line(screen, 20, 30, 150, 100, G.MakeCol(255, 0, 0));
  G.Flip;
  G.Pause;
  G.Close
END GrTest.
```

This program will open a (black) graphic window, draw red lines (`G.Line`) from points (20; 30) to a point (150; 100) on it, waits for the user to press a key (`G.Pause`) and end.

`G.Init` initializes (starts) the graphic, i.e. opens the graphic window and *returns* a pointer to it, this pointer is saved in a variable `screen` of type `G.Bitmap`, which is then used for drawing.

`G.Flip` displays the image on the screen. Until `G.Flip` procedure is called, the drawn image is not yet visible.

`G.Pause` pauses the execution of the program and waits until the user has pressed a key.

`G.Close` closes the graphic window.

14.1. Drawing lines.

Procedure `G.Line` draws a line. It takes six pramteres. The first one means where to draw the line (i. e. on the screen), the next four parameters are the coordinates of the beginning and the end of the line: $(x_1; y_1)$, $(x_2; y_2)$, they are being passed as four integers and are comma-separated. The coordinates are counted from the upper left corner of the screen — from the point (0; 0), and increase to the right along the axis *OX* and downwards along the axis *OY*. The last parameter is the color of the line (see «14.2. Making colors»).

If the line is horizontal, it can be drawn with `G.HLine(screen, x1, y, x2, color)`, where the value *y* is being passed only once. In the same way, the vertical line can be drawn using `G.VLine(screen, x, y1, y2, color)`.

14.2. Making colors.

A color can be defined using procedure `G.MakeCol(r, g, b)`, which takes three integers for red, green and blue components. Each number lies in the interval from 0 to 255. Below you can see some examples:

```
G.MakeCol(  0,   0,   0)          — black
G.MakeCol(255, 255, 255)          — white
G.MakeCol( 80,  80,  80)          — light gray
G.MakeCol(  0,   0, 255)          — blue
G.MakeCol(  0, 255, 255)          — cyan
G.MakeCol(120,  60,   0)          — brown
G.MakeCol(255, 229, 180)          — peach
```

The resulting color also is of the type `INTEGER`, so it can be written in a variable, and then used in place of `G.MakeCol(...)`:

```
VAR orange: INTEGER;
BEGIN
  orange := G.MakeCol(255, 128, 0);
  G.Line(screen, 100, 100, 200, 100, orange)
```

To split the color into its components, there is a procedure `ColorToRGB`:

```
VAR color, r, g, b: INTEGER;
BEGIN
  color := G.MakeColor(0, 128, 255);
  G.ColorToRGB(color, r, g, b)
```

14.3. Getting the size of the screen.

After graphics has been initialized with `screen := G.Init()`, two values become valid: `screen.w` and `screen.h`, where the first one means the width of the screen in pixels and the second one is the height. For example, to cross the screen with two lines, do this:

```
G.Line(screen, 0, 0, screen.w - 1, screen.h - 1, G.MakeCol(255, 0, 0));
G.Line(screen, screen.w - 1, 0, 0, screen.h - 1, G.MakeCol(0, 255, 0))
```

It should be noted that the rightmost pixel that is visible on the screen has abscissa of `screen.w - 1`, because the abscissa of the leftmost one is 0. The same holds for `screen.h`.

14.4. Manipulating the color of individual pixels.

A color of an individual pixel on the screen can be changed like so:

```
G.PutPixel(screen, 100, 60, G.MakeCol(0, 0, 255))
```

To get the color of a pixel, use `GetPixel`.

```
VAR color: INTEGER;
BEGIN
  color := G.GetPixel(screen, 100, 60)
```

If you want to quickly change the color of individual pixels, it's better to use the `PutPixelFast` procedure. To do this, first call `LockBitmap`, and at the end call `UnlockBitmap`:

```
LockBitmap(screen);
PutPixelFast(screen, x, y, color);
UnlockBitmap(screen)
```

14.5. Clearing the screen.

To paint the whole screen in black, run procedure `G.ClearScreen`. To paint it with another color, run procedure `G.ClearScreenToColor`:

```
G.ClearScreenToColor(G.MakeCol(0, 0, 80))
```

14.6. Drawing of some figures.

Use `G.Rect` procedure to draw a non-filled rectangle:

```
G.Rect(screen, 50, 100, 200, 150, G.MakeCol(255, 0, 0))
```

The second and third parameters are the coordinates of the upper-left corner, and the fourth and fifth are the coordinates of the lower right corner. In the example, a rectangle with a width of 151 and a height of 51 pixels will be drawn.

The `G.RectFill` procedure draws a filled rectangle.

14.7. Customizing the graphics window.

Before calling `G.Init`, you can set some parameters of the graphics window by calling the `G.Settings(w, h, flags)` procedure. The first two parameters are the desired width and height of the screen in pixels, `flags` is a set in which you can include the following constants:

`G.fullscreen` — turn on fullscreen mode
`G.spread` — enlarge screen size if proportions allow to do that
`G.sharpPixels` — use an integer multiple size of a virtual pixel (1, 2, 3, 4...)
`G.software` — turn off hardware rendering
`G.initMouse` — initialize automatic mouse control

Example:

```
G.Settings(320, 200, {G.fullscreen, G.spread, G.sharpPixels});
screen := G.Init()
```

Default screen size is 640x400 pixels, but in practice the size can be larger, because the default options are `G.fullscreen` and `G.spread` (as well as `G.sharpPixels`). To simply take use the whole screen, set width or height to zero and turn on `G.fullscreen`.

After the graphics window has been already initialized, you can switch to window mode using `G.SwitchToWindowed`, and switch back to fullscreen using `G.SwitchToFullscreen`. Procedure `G.ToggleFullscreen` toggles the graphics mode back an forth.

14.8. Animation.

Animation implies fast change of frames, therefore a loop is required in which the program will draw frames, then display it on the screen with `G.Flip` and perhaps make a small delay using procedure `G.Delay`. The cycle can be interrupted by pressing any key – `G.KeyPressed()` or using some other condition. Example:

```
MODULE FlyingDot;
IMPORT G := Graph;
VAR s: G.Bitmap;
  x, y, vy: INTEGER;
BEGIN
  s := G.Init();
  x := 0; y := 10; vy := 0;
  REPEAT
    G.PutPixel(s, x, y, G.MakeCol(255, 255, 255));
    INC(x, 2); INC(y, vy); INC(vy);
    IF vy > 15 THEN vy := -13 END;
    G.Flip;
    G.Delay(20)
  UNTIL G.KeyPressed();
  G.Close
END FlyingDot.
```

Procedure `G.Delay` takes an integer – the number of milliseconds to wait (there are 1000 milliseconds in one second). Parentheses are placed at the end of the call to `G.KeyPressed()`, it returns a value of type `BOOLEAN`. If a key has been pressed, it will return `TRUE` and the `REPEAT` loop will end itself.

14.9. Random number generator.

Pseudorandom number generator is often necessary for graphics programming. It is started using the `G.Randomize` procedure, but `G.Init` also starts the generator, so there is no need to call `G.Randomize`.

Procedure `G.Random(n)` returns a random integer from 0 to (n − 1) (inclusive). If you need a number from 1 to n, use the expression `G.Random(n) + 1`.

Procedure `G.Uniform()` returns a random number in range [0; 1). If it is then multiplied by n, we get the range [0; n).

Example:

```
VAR a: INTEGER;
  x: REAL;
```

```
BEGIN
  a := G.Random(10);      (* integer in range [0; 9]     *)
  x := G.Uniform() * 9;   (* real number in range [0; 9) *)
```

The pseudo-random sequence is based on a random seed, which is set at the beginning of the program when `G.Randomize` is called, and changes each time the next random number is generated. This seed can also be specified explicitly using procedure `G.PutSeed(n)`. This technique can be used to re-generate the same pseudo-random sequences. Random seeds are of type `INTEGER` and are readable by `G.randomSeed`.

14.10. Working with images.

Procedure `G.LoadBitmap(filename)` allows you to load an image from a file in `BMP`, `PNG` or `JPG` format. The procedure returns a pointer of type `G.Bitmap`. Then, this image can be drawn on the screen or on another image using procedures `G.Blit`, `G.BlitWhole` and `G.StretchBlit`. Example:

```
MODULE BlitBmp;
IMPORT G := Graph;
VAR s, b: G.Bitmap;
BEGIN
  s := G.Init();
  b := G.LoadBitmap('data/examples/rocket.png');
  G.BlitWhole(b, s, 100, 60);
  G.Flip; G.Pause; G.Close
END BlitBmp.
```

Procedure `BlitWhole` draws the image in the given coordinates, `Blit` can draw a part of the image, and `StretchBlit` can stretch and compress the image when drawing.

```
BlitWhole(src, dest: Bitmap; x, y: INTEGER)
Blit(src, dest: Bitmap; sx, sy, sw, sh, dx, dy: INTEGER)
StretchBlit(src, dest: Bitmap; sx, sy, sw, sh, dx, dy, dw, dh: INTEGER)
```

Parameter `src` (this means “source”) tells where the image is taken *from*, and parameter `dest` (destination) — *where* should it be drawn. Parameters `sx`, `sy`, `sw` and `sh` are related to `src`, and `dx`, `dy`, `dw` and `dh` are related to `dest`.

Example:

```
MODULE StretchBlitBmp;
IMPORT G := Graph;
VAR s, b: G.Bitmap;
BEGIN
  s := G.Init();
  b := G.LoadBitmap('data/examples/rocket.png');
  G.StretchBlit(b, s, 0, 0, b.w, b.h, 0, 0, s.w, s.h);
  G.Flip; G.Pause; G.Close
END StretchBlitBmp.
```

The program will disproportionately stretch the image of a rocket on the whole screen.

Appendix A. Recompilation of Free Oberon under Windows.

Despite Free Oberon is shipped as an EXE file with all the necessary add-ons, in some cases it makes sense to recompile it, for example, after making changes to the Free Oberon source code or for self-education. To do this, you will need to install some software.

First of all, you will need a C compiler “MinGW-w64” and Unix-like environment “MSYS2”, then the Oberon to C translator “Vishap Oberon Compiler” and finally the graphics library SDL2 with add-ons.

1. Follow the link sourceforge.net/projects/mingw-w64 and download MinGW-w64 – a GCC compiler for Windows (despite the «64» in its name, the program runs in 32-bit mode).
2. Install the MinGW-w64 to the directory «`C:\mingw-w64`». This path will be used later.

   Note. If during the installation you changed the drive from `C:` to something else and MinGW-w64 failed to install, try using drive `C:` instead.
3. Follow the link `msys2.org` and download a version of MSYS2 for the i686 processor architecture (the downloaded file name will probably have the form of «`msys2-i686-*.exe`»). If you are using Windows XP, the latest version of MSYS2 will not work, and you should download an older version, for example, «`msys2-i686-20150916.exe`».
   It can be downloaded from:
   `sourceforge.net/projects/msys2/files/Base/i686`.

   If this does not help, download the archive in «`*.tar.xz`» format and extract it using 7-Zip (7zip.org).
4. Install MSYS2 in the directory «`C:\msys32`» and run MSYS console. It can be run using the file «C:\msys32\msys32_shell.bat» or the icon at installed menu.
5. At the MSYS console run the following command:
   `pacman -Sy pacman`

   This command updates MSYS package database, and renews pacman itself. (When the program asks if you want to start the installation, answer yes «`y`»).
6. Close MSYS console, and open it again (see step 4).
7. Update the rest of the system by running this command in MSYS console:
   `pacman -Su`
8. Close the MSYS console again and open it with administrator rights. (Right-click on `msys32_shell.bat` or the icon in the application menu and select «Run as administrator».)
9. Install the program with `make` и `diffutils`:
   `pacman -S make diffutils`

10. Set the environment variables with the following commands:
```
export PATH=$PATH:/c/mingw-w64/mingw32/bin
export CC=gcc
```

    During compilation it is necessary to use the MinGW compiler. If you specified a different path in step 2, then also use it here (in `PATH` above).

    You can append these two EXPORT lines «`~/.bashrc`» file, in which case they will be automatically triggered each time MSYS2 console is started.

11. Follow the link libsdl.org/download-2.0.php, download the file «`SDL2-devel-2.*.*-mingw.tar.gz`», unzip it and copy the contents of the directory «`i686-w64-mingw32`» to «`C:\mingw-w64\mingw32`».

    Copy the file «`bin\SDL2.dll`» from the unpacked archive into the directory «`C:\FreeOberon`».

12. Follow the link libsdl.org/projects/SDL_image and download the archive «`SDL2_image-devel-2.*.*-mingw.tar.gz`», unzip it and copy the contents of the directory «`i686-w64-mingw32`» to «`C:\mingw-w64\mingw32`». Optionally, you can install other add-ons to SDL2 library in the same way.

    Also go to the subdirectory «`i686-w64-mingw32\bin`» of the unpacked SDL2 image archive and copy the following DLL-files: `libjpeg-9.dll`, `libpng16-16.dll`, `SDL2_image.dll` и `zlib1.dll` to the directory «`C:\FreeOberon`».

    Note. If you want to run *graphic* programs from «`C:\FreeOberon\bin`» manually, you will need to copy the DLL files there too.

    Recompilation complete.

13. Go to the source directory and start the compilation:
```
cd /c/FreeOberon/src
make -f Makefile_win32
```

    As a result, VOC compiler will be compiled together with its libraries and then Free Oberon will be compiled.

    To remove unnecessary files, run:
```
make -f Makefile_win32 clean
```

14. To rebuild everything again, run this in MSYS (but don't forget about the environment variables if you open MSYS again, see step 10):
```
cd /c/FreeOberon/src
make -f Makefile_win32
```