DOCUMENTATION

ON USING THE
INTEGRATED DEVELOPMENT ENVIRONMENT

FREE OBERON

AND THE PROGRAMMING LANGUAGE

OBERON

Revision 8.11.2023
for Free Oberon version 1.1.0

TECHNICAL-THEORETICAL LITERATURE
PUBLISHING HOUSE OF RIGA EXPERIMENTAL
DESIGN BUREAU FOR COMPILER CONSTRUCTION

R I G A   2 0 2 3

Table of Contents

1. Installation.

Free Oberon is a cross-platform integrated development environment. It is available for Windows as an EXE setup program, as well as for GNU/Linux and other UNIX-like operating 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. There are two options:

   a) Clone Free Oberon repository from GitHub. This version is the most up-to-date. For this, open terminal and type in:

   ```
   git clone --depth 1 --recurse-submodules --shallow-submodules \
     https://github.com/kekcleader/FreeOberon.git
   ```

   Note: The backslash (\) escapes the line break. You can enter the command without the "\" if the command is entered on a single line.

   b) Download Free Oberon as "tar.gz" archive from the "Download" section of the free.oberon.org website. Extract Extract the archive to your home directory or to another location on the disk.

2. Install the following software packages:

   ```
   git         gcc         libc-dev                libballegro5-dev
   liballegro-image5-dev   liballegro-audio5-dev   liballegro-acodec5-dev
   ```

   The names of the packages are given as they appear in the operating system Debian GNU/Linux. They are also suitable for Ubuntu, Linux Mint, Raspbian and other. To install the mentioned packages, run the following command in terminal:

   ```
   apt-get install -y git gcc libc-dev liballegro5-dev \
     liballegro-image5-dev liballegro-audio5-dev liballegro-acodec5-dev
   ```

   (This command must be executed with root privileges, that is, you must first run "`su`" and enter the root password, or prepend the command with the word "`sudo`".)

   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 dnf install -y git gcc glibc-devel allegro5-devel \
     allegro5-addon-image allegro5-addon-audio allegro5-addon-acodec \
     allegro5-devel allegro5-addon-image-devel allegro5-addon-audio-devel \
     allegro5-addon-acodec-devel
   ```

   For other operating systems (Arch Linux, openSUSE) output of the command "`./install.sh`" in the root folder of the extracted Free Oberon archive.

3. Run:

   ```
   ./install.sh
   ```

4. Done. If Free Oberon has been installed successfully, you can run it with “./FreeOberon”. The installation script will offer you to optionally add a line to “~/.bashrc” to allow the usage of the command “fob” (console version of Free Oberon compiler), that will look something like:

   ```
   export PATH=/home/user/FreeOberon:$PATH
   ```

   You also can add this line that will allow you to launch Free Oberon from any directory by typing the command “fo”:

   ```
   alias fo='cd ~/FreeOberon;./FreeOberon'
   ```

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” from the “Examples” subdirectory. Press F9 (“Make and Run”). If everything works as expected, you should see a picture of a book.

Note: To toggle between window mode and fullscreen, press Alt+Enter.

3. Choosing a language.

Free Oberon IDE will open in your current OS language: English or Russian. It is possible to specify the language. The command line argument is “--lang”. For example:

```
./FreeOberon --lang en      (in the Linux terminal)
FreeOberon --lang en        (in Windows Command Prompt)
```

It is also possible to add your language. See the text file in the “Data/Texts” subdirectory.

4. 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 typed in between these two lines. File name: Prog1.Mod *)
END Prog1.
```

5. Saving a file.

To save a file, press the key `F2` 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 the UTF-8 encoding, by default they are put in the subdirectory “`Programs`” and, if required, can be edited using other text editors.

6. Running a program.

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

7. Openning a file.

Press `F3` and choose a file, then press Enter. The file 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`.

The menu item “`File → Reload`” is useful in case the file has been edited and you want to reload it from the disk (for example, to roll back the changes or if the file on the disk has been changed by another program).

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

8. 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.

Copying the 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.

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.

In the `VAR` section a programmer lists variables together with their types. A variable is a named memory block in which a certain value is stored (each variable has a name). The type of a variable denotes 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 variables are of type `INTEGER` – a whole number.

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 `Out.Ln` in the program text).

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. A data type determines the set of values which variables of that type may assume, and the operators that are applicable. 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.

As well as a compound type:
`POINTER TO …` – a pointer that stores the memory address of some other value.

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. To keep the source code short, you might want to import it using the alias “`M`”:

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

Some procedures of module `Math`:
`round(x: REAL): INTEGER` – 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 Random.

Pseudorandom number generator is implemented in module `Random`. but `G.Init` also starts the generator, so there is no need to call `G.Randomize`. It is initialized using the procedure `Randomize`, but the procedure is called automatically upon importing the module, so there is no need to call `Random.Randomize` manually.

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`. A random integer in range [a; b] is obtained, in general form, using the expression `G.Random(b - a + 1) + a`.

Procedure `G.Uniform()` returns a random real 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 `Random.seed`.

15. Module Graph.

Module `Graph` allows you to program graphics and interact with the keyboard and mouse. Module `Graph` is usually imported under the alias "`G`":

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

A simple graphics example:

```
MODULE GrTest;
IMPORT G := Graph;
VAR c: G.Color;
BEGIN
  G.Init;
  G.MakeCol(c, 255, 0, 0);
  G.Line(20, 30, 150, 100, c);
  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.

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

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

`G.Close` closes the graphic window.

15.1. Drawing lines.

Procedure `G.Line` draws a line. It takes six pramteres. The first four parameters are the coordinates of the beginning and the end of the line in the following order: (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 *OX* axis and downwards along the *OY* axis. The last parameter is the color of the line (see “15.2. Making colors”).

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

15.2. 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 (inclusive). Below you can see examples of how to make some of the colors:

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

Here `c` is of type `G.Color` – the variable, where the resulting color is stored.

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

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

15.3. Getting screen size.

After graphics has been initialized with `G.Init`, you can call `G.GetScreenSize(W, H)`, where `W` and `H` are variables of type `INTEGER`. The width of the screen (in pixels) will be stored in the first variable, and the height will be stored in the second.

One can cross the screen with two lines like so:

```
MODULE SizeTest;
IMPORT G := Graph;
VAR c: G.Color;
  W, H: INTEGER;
BEGIN
  G.Init;
  G.GetScreenSize(W, H);
  G.MakeCol(c, 0, 255, 0);
  G.Line(0, 0, W - 1, H - 1, c);
  G.Line(0, H - 1, W - 1, 0, c);
  G.Flip; G.Pause; G.Close
END SizeTest.
```

It should be noted that the rightmost pixel that is visible on the screen has abscissa of `W - 1`, because the abscissa of the leftmost pixel is 0. The bottom pixel is at `H - 1`.

15.4. Manipulating the color of individual pixels.

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

```
G.PutPixel(100, 60, color)         (* X, Y and color *)
```

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

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

If you need to repeatedly use the `PutPixel` and `GetPixel` procedures, you can significantly speed up their operation by first locking the bitmap and then unlocking it at the end. To do this, you should first call the `LockBitmap` procedure and, at the end, call `UnlockBitmap`.

```
G.LockBitmap(bmp);
FOR x := 0 TO 100 DO
  G.GetPixel(bmp, x, y, c);
  (*...*)
END;
G.UnlockBitmap(bmp)
```

15.5. Clearing the screen.

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

```
G.ClearToColor(c)
```

15.6. Drawing some shapes.

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

```
G.Rect(50, 100, 200, 150, c)
```

The first and second parameters are the coordinates of a rectangle’s corner, and the third and fourth are the coordinates of its opposite 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.

15.7. 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.window`	— open graphics in a window (turn off fullscreen mode)
`G.center`	— open window in center (for window mode only)
`G.resizable`	— make window resizable (for window mode only)
`G.exact`	— do not enlarge the screen size automatically
`G.smooth`	— do not ensure the sharpness of the virtual pixel
`G.software`	— turn off hardware rendering
`G.noMouse`	— do not initialize the mouse
`G.maximized`	— open window as maximized (for window mode only)
`G.minimized`	— open window as minimized (for window mode only)
`G.frameless`	— open window without an OS frame (for window mode only)
`G.nobuffer`	— disable double buffering

Example:

```
G.Settings(320, 200, {G.exact, G.smooth, G.noMouse});
G.Init;
```

Default screen size is 640x400 pixels, but in practice, the size can be larger, because, by default, it adjusts towards increasing to occupy the entire screen and ensure the virtual pixel size is whole number (1, 2, 3 etc.). If width or height is zero, it will simply use the whole screen, and the virtual pixels will match the real screen pixels.

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

15.8. 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 c: G.Color;
  x, y, vy: INTEGER;
BEGIN
  G.Init;
  G.MakeCol(c, 255, 255, 255);
  x := 0; y := 10; vy := 0;
  REPEAT
    G.PutPixel(x, y, c);
    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.

15.9. 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.Draw`, `G.DrawPart`, `G.DrawTintedPart`, `G.DrawFlip`, `G.DrawPartFlip`, `G.DrawRotated`, `G.DrawScaledRotated` and `G.DrawEx`.

Example:

```
MODULE Rocket1;
IMPORT G := Graph, Out;
VAR b: G.Bitmap;
BEGIN
  G.Init();
  b := G.LoadBitmap('Data/rocket.png');
  IF b = NIL THEN Out.String('Could not load rocket.png'); Out.Ln
  ELSE
    G.Draw(b, 100, 60);
    G.Flip; G.Pause; G.Close
  END
END Rocket1.
```

Procedure `Draw` draws the image in the given coordinates, `DrawPart` can draw a part of the image, and `DrawEx` can stretch and compress the image when drawing.

```
Draw(bmp: Bitmap; x, y: INTEGER);
DrawPart(bmp: Bitmap; sx, sy, sw, sh, dx, dy: INTEGER);
DrawEx(bmp: Bitmap; sx, sy, sw, sh, dx, dy, dw, dh: INTEGER; flip: SET);
```

Values *sx, sy, sw, sh* correspond to the source image (what is being drawn), and *dx, dy, dw, dh* correspond to the destination image (where it is being drawn). The `flip` set may be empty (`{}`) or contain one or both elements from the set `{flipHorz, flipVert}` for horizontal and vertical flipping of the source image part.

```
G.DrawEx(b, 0, 0, b.w, b.h, 50, 50, b.w, b.h, {G.flipVert})
```

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 Allegro5 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. Download the development package for Allegro5. Copy the contents of the subdirectory «`i686-w64-mingw32`» of the archive to «`C:\mingw-w64\mingw32`».

    Copy the file «`allegro-5.2.dll`» from the archive into the directory «`C:\FreeOberon`».

12. Do the same for «`allegro-primitives-5.2.dll`» and «`allegro-image-5.2.dll`».

13. Go to the source directory and start the compilation:

    ```
    cd C:\FreeOberon\src
    make.bat
    ```

    As a result, Free Oberon will be compiled together with its libraries, and also Free Oberon console compiler will be compiled – *foc.exe*.