

PB's Guide to Starting with wxWidgets on Microsoft Windows with MinGW and Code::Blocks

Guide Author: PB <pbfordev@gmail.com>

Guide Version: 4.6, released September 21, 2023

Versions of Software Used in Guide

wxWidgets: 3.2 (3.2.2.1 at the time of writing)

MinGW: MSYS2 mingw-w64-ucrt-x86_64-toolchain (GCC 13.2 at the time of writing)

Code::Blocks: 20.03

The latest version of this guide is always available at <https://github.com/PBfordev/wxpbguide>.

Table of Contents

1	Introduction	4
2	Basic wxWidgets Information	5
2.1	wxWidgets Versions	5
2.2	Shared vs Static Linking	5
2.3	Multilib vs Monolithic Build	6
2.4	Debug and Release Configuration	6
2.5	32-bit vs 64-bit Build	6
2.6	To Build or Not to Build	6
3	Building wxWidgets with MinGW	7
3.1	Choosing MinGW Distribution	7
3.2	Installing MSYS2	7
3.3	Installing MSYS2 Packages for Developing C++ Applications	7
3.4	Downloading wxWidgets Sources	9
3.5	Adding System Environment Variable WXWIN	11
3.6	The Build	13
3.6.1	Starting Simple	13
3.6.2	Improving Build Commands	14
3.6.3	Making wxWidgets Build Faster	15
3.6.4	Viewing Build Results	16
3.6.5	Using the Minimal Sample to Test the Build	17
3.6.6	Cleaning the Build	19
3.6.7	Building wxWidgets Statically	20
3.6.8	More Build Options	21
4	Setting Up wxWidgets Project with Code::Blocks and MinGW	22
4.1	Installing Code::Blocks	22
4.2	Configuring the Debugger	23
4.3	Configuring the Compiler	24
4.4	Creating Global Variable wx	25
4.5	Creating a wxWidgets Project with Project Wizard	26
4.5.1	Selecting wxWidgets Project Template	27
4.5.2	Wizard Page 1: Welcome	28
4.5.3	Wizard Page 2: wxWidgets Version	29
4.5.4	Wizard Page 3: Project Title and Paths	30
4.5.5	Wizard Page 4: Project Author	31
4.5.6	Wizard Page 5: GUI Builder and Application Type	32
4.5.7	Wizard Page 6: wxWidgets Location	33
4.5.8	Wizard Page 7: Compiler and Debug/Release Configurations	34
4.5.9	Wizard Page 8: Various Configuration Options	35
4.5.10	Wizard Page 9: Advanced Options	36
4.5.11	Wizard Page 10: Additional Libraries	37
4.6	Adjusting Project Settings	38
4.6.1	Setting C++ Standard Used in the Project	39
4.6.2	Setting Preprocessor Define <i>NDEBUG</i> for the Release Build Target	40
4.6.3	Marking the Application DPI-Aware	41

4.7	<i>Finally Finished!</i>	42
4.8	<i>Extra Reading: wxWidgets Code::Blocks Project Under the Hood</i>	43
4.8.1	Compiler Include Folders	43
4.8.2	Preprocessor Defines	43
4.8.3	Linker Settings	44
4.8.4	Using Static Instead of Shared wxWidgets Build	45
	Document History	47

1 Introduction

This Microsoft Windows specific guide hopes to assist the readers with decisions regarding which wxWidgets version and configuration to use and show how to build wxWidgets using MinGW with various parameters affecting the resulting build. It also describes in detail setting-up a wxWidgets project in popular IDE Code::Blocks.

This guide is not a reference manual, it is a tutorial for new users of wxWidgets, MinGW, and Code::Blocks; supposed to be followed step by step.

This guide does not cover actual wxWidgets programming nor using Code::Blocks (besides the steps necessary to set up the IDE to be able to build a wxWidgets project).

This guide assumes that the readers did their research and decided that for their applications, the most suitable choices as the programming language and the GUI toolkit are C++ and wxWidgets; therefore the pros and cons of these two choices are not discussed here.

This guide does not intend to replace the official documentation, it hopes to complement it. Its author is not affiliated with Microsoft, wxWidgets, MSYS2, mingw-w64, or Code::Blocks.

2 Basic wxWidgets Information

2.1 wxWidgets Versions

wxWidgets has [two supported branches](#), called *stable* (currently 3.2) and *development* (currently 3.3). The stable branch version always has an even minor version number (e.g., 3.0.5 or 3.2.2) while the development branch has an odd number for its minor version (e.g., 3.1.7 or 3.3.0). The current development branch is also sometimes called “master” (e.g., in the GIT code repository) or “trunk” (e.g., in the documentation version selector).

For wxWidgets, when the branch is called stable, the API and ABI of the Major.Minor version is sealed. This means that all the versions in the stable branch are API- and binary-wise fully compatible, i.e., one can switch to newer releases of the branch (e.g., from 3.2.0 to 3.2.2) without encountering any (adverse) compile- and run-time differences compared to the older releases. No such guarantees are made for the development branch and in fact, as new features are added, the API, ABI, or the run-time behaviour change. In other words, it does not mean that the stable branch is more stable as in being less prone to crashing your application than the development one. However, it means that no new features can be added to the stable branch after its initial release. The new releases in the stable branches are for bug fixes only, but even not all bug fixes can be backported from the development to the stable branch, for example when the code for the two branches has diverged too much or just no one felt motivated enough to do that. The core developers suggest that the development branch is generally suitable for production use.

After several years of adding features to the development branch, the developers decide it is ready to become the new stable branch (e.g., 3.1 became 3.2 after 3.1.7) and a new development branch is created (e.g., 3.3 split from 3.2.0 after its release).

The number of differences in features added (and possibly bugs fixed) between the two branches increases with the time passed since the initial stable version release. At the time of writing, there has been no release in the current development branch yet (for comparison, there were eight releases in 3.1 development branch, i.e., between the release of 3.0 and 3.2). Therefore, if you want to use the development version, you need to embrace the “Live at Head” concept and use the current GIT master. If you want to do this, make sure to read and follow [the GIT instructions](#). Currently, the biggest reason for using the development version may be the experimental support for Windows dark mode, about which you can read about [here](#).

2.2 Shared vs Static Linking

wxWidgets can be built either as the static build or the shared build (the application requires wxWidgets DLL at runtime) library. Both have their pros and cons, so which one to choose depends on the use case. When the shared configuration is used, all the necessary DLLs must be shipped with the application and the install package is usually larger than when using the static libraries where only the used code is linked into the final executable. On the other hand, if one has a suite of applications using wxWidgets, having them all using the same wxWidgets DLLs is desirable. Similarly, for applications with plugins, linking to wxWidgets dynamically ensures that both the application and the plugin are using the same library.

In practice, for most cases the differences from both the user and the programmer's point of view are usually negligible, with static linking perhaps being a bit more convenient regarding the application distribution but also being slower when linking the application during its each and every build.

GCC allows linking the run-time libraries statically, so if all the libraries the application uses can be linked statically and support being built with the statically-linked compiler run-time libraries, the whole application can be contained in a single executable which does not require any DLLs at all (see chapters [3.6.7](#), [4.5.9](#), and [4.8.4](#)).

I would use the static build only if the application was small, used just wxWidgets as the compiled library and I was concerned about how much bandwidth and disk space the installer and the installed application take. Otherwise, I would prefer dynamic linking, i.e., the shared build and this build will be preferably used in this guide.

2.3 Multilib vs Monolithic Build

By default, wxWidgets is built as a [set of several libraries](#), which is called the Multilib build. wxWidgets can also be built in the so-called Monolithic build, where all the wxWidgets libraries are merged into just two DLLs (one is used only for OpenGL) instead of fifteen (see also chapter 3.6.4 and Table 4-1). I cannot think of even one significant advantage of using the Monolithic build, so I recommend sticking to the default Multilib one and will use this build in this guide.

2.4 Debug and Release Configuration

wxWidgets on Windows is supposed to be built in both Debug and Release configurations, where the Debug configuration is used for development (contains some useful runtime checks helping to find bugs) and the Release configuration for distribution to the end user.

In contrast to using Microsoft Visual C++, wxWidgets built in Debug configuration with GCC does not use debug versions of the compiler runtime libraries (e.g., with checks for invalid memory access), so the difference when using GCC is not that big. Nevertheless, I still recommend building both configurations and using the Debug one for application development. See also the [Debugging programmer guide](#) for more information.

2.5 32-bit vs 64-bit Build

These days, 64-bit Windows versions are dominant. Therefore, I believe it is best to build the application and thus the libraries it used as 64-bit. One can easily build and use wxWidgets in both 32- and 64-bit configurations, but they must each be built in its own folder (e.g., `gcc_dll-x86` vs `gcc_dll-x64`).

2.6 To Build or Not to Build

wxWidgets website provides prebuilt binaries for several popular compilers so one does not have to build wxWidgets on their own. But the compiler you use may not be amongst them. MSYS2 conveniently provides a prebuilt wxWidgets package; however, you cannot tailor the build to your needs (see chapter 3.6.8) or use the static build. Perhaps more importantly here, Code::Blocks wxWidgets project wizard unfortunately cannot use wxWidgets MSYS2 package, since it expects a different folder structure and library naming scheme.

Additionally, this guide is for new users for wxWidgets and it is strongly recommended that the new users of the library build and play with the bundled samples, which are very useful learning resource showing (a) what wxWidgets can do and (b) how to do it in the code.

Finally, you are limited to official wxWidgets releases provided by MSYS2 or the wxWidgets team, so for example, you cannot use the current GIT master which some may prefer to do.

These are basically the main reasons for building wxWidgets on our own (which is also what wxWidgets developers recommend doing on Windows). Fortunately, building wxWidgets (at least on Windows) is easy and simple (and also relatively fast), as wxWidgets does not have any external dependencies which avoids the usual major headache with compiled C++ libraries.

3 Building wxWidgets with MinGW

3.1 Choosing MinGW Distribution

MinGW is a free and open source software that can be used to create Microsoft Windows applications. There are several different “distributions” of MinGW with differences being the bundled compiler version and completeness of headers and libraries needed for Microsoft Windows.

One should pick a distribution that (a) provides all necessary headers and libraries and (b) supports required C++ features (such as C++17). wxWidgets tries hard to address possible issues with missing declarations or imports but it is still best to use a distribution which ships with the most correct header and libraries. I believe that at the time of writing, the best MinGW is mingw-w64 and this will be the toolchain used in this guide, in its variant for the 64-bit architecture.

There are several ways to obtain mingw64 but, in my opinion, the best has been using MSYS2. In addition to providing the maintained mingw64 compiler toolchain, MSYS2 also provides a large number of prebuilt libraries. If you for some reason do not want to use MSYS2, there are still other maintained (at least for now) distributions based on mingw64 such as [WinLibs](#), [niXman](#), or [nuwen.net](#). You can use those with this guide, basically the only difference would be the path to the compiler toolchain and names for compiler/debugger settings in Code::Blocks.

3.2 Installing MSYS2

Go to https://repo.msys2.org/distrib/x86_64/ and download the latest version of the installer executable. Launch the installer, leave the options at their defaults, and let it finish the installation. After the installation, the packages must be updated, so launch the MSYS environment, either from the last page of the installer or via Start Menu, where it should be listed as **MSYS2 MSYS**. On MSYS command prompt enter `pacman -Syuu` and follow the instructions to update. You can use this command to update all installed packages whenever needed.

3.3 Installing MSYS2 Packages for Developing C++ Applications

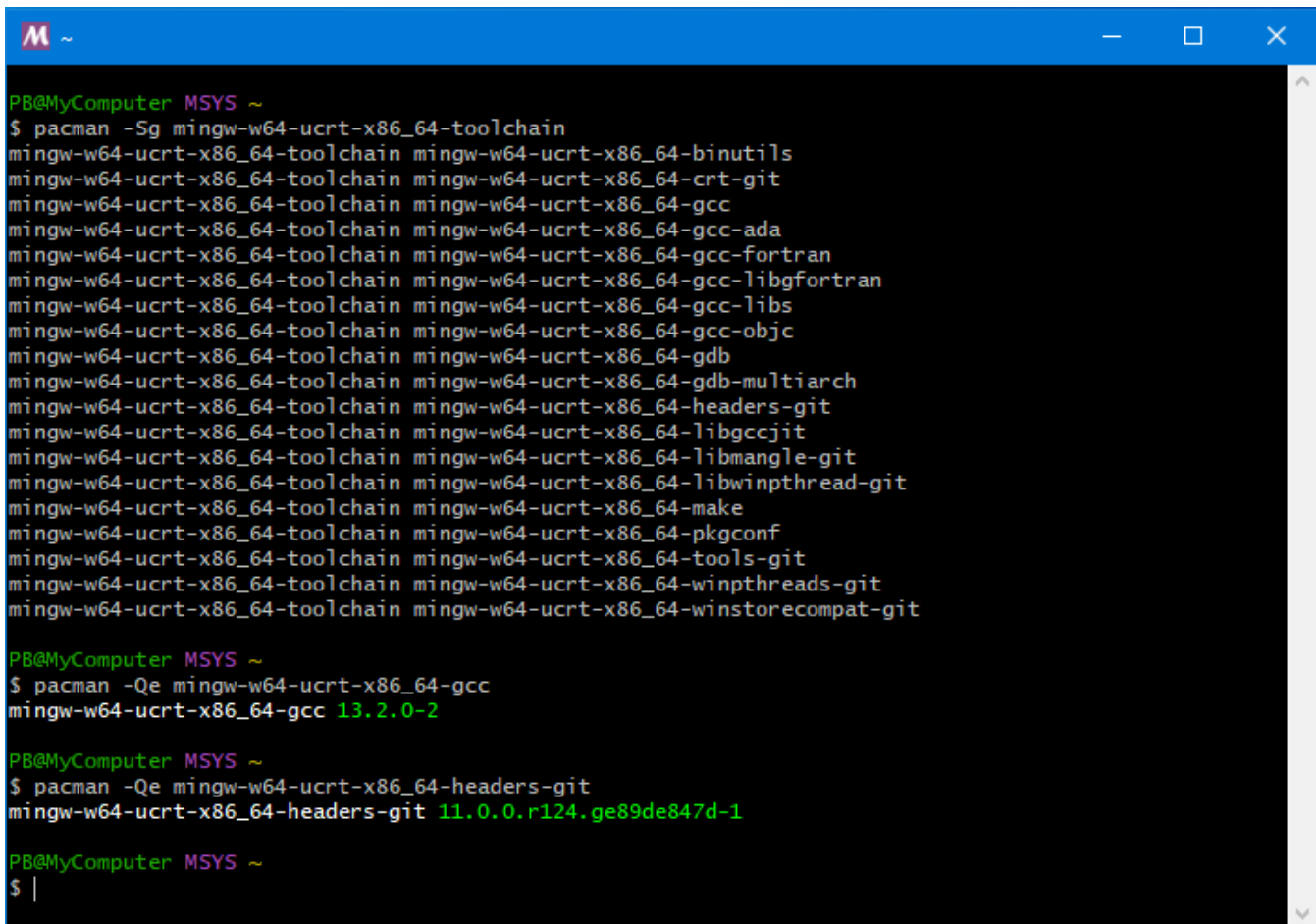
While it is possible to build wxWidgets library and applications using it also with clang (e.g., MSYS2 package *mingw-w64-clang-x86_64-toolchain*), Code::Blocks currently does not support debugging with LLDB and so we are going to use GCC in this guide. We now need to install all the tools required for the 64-bit GCC MinGW toolchain. MSYS2 provides two mingw-w64 distributions: one using the ancient MSVCRT and another using the universal CRT (UCRT). You can read more about the differences between the two [here](#) and learn more about the UCRT deployment [here](#). You need to ensure that you do not mix libraries linking to different CRTs (for prebuilt packages, install those with “ucrt” in their name, e.g., for OpenCV, use package *mingw-w64-ucrt-x86_64-opencv* and NOT *mingw-w64-x86_64-opencv*).

The MinGW toolchain is probably easiest to install as a group named *mingw-w64-ucrt-x86_64-toolchain*, so in the MSYS command prompt enter `pacman -S mingw-w64-ucrt-x86_64-toolchain`; when asked to select the packages just press **<Enter>** to select them all and then confirm that you want to continue with installation. Please notice that the toolchain executables are installed in *ucrt64\bin* subfolder of your MSYS2 root folder, e.g., *c:\msys64\ucrt64\bin*.

Do not add the folder where you installed MinGW to your *PATH* system environment variable. It may seem convenient, but it can (and sometimes does) create problems. Moreover, make sure that no other GCC-based compiler toolchain is there, as this could cause some odd issues when building the applications as well as when running them. Use the command prompt and either `set` command to inspect the content of *PATH* or better yet, use `where` command (`where $PATH: gcc.exe`) to confirm that no GCC-based compiler is installed in the folders added to *PATH* (see also *test-gcc-not-in-path.bat* provided in the GitHub repository).

You can list which packages are included in *mingw-w64-ucrt-x86_64-toolchain* group by using `pacman -Sg mingw-w64-ucrt-x86_64-toolchain` or in your web browser at https://packages.msys2.org/group/mingw-w64-ucrt-x86_64-toolchain. You can tell the version of an installed package by using `pacman -Qe <package-name>`, see Figure 3-1.

Figure 3-1 Displaying information for MSYS2 packages



```

PB@MyComputer MSYS ~
$ pacman -Sg mingw-w64-ucrt-x86_64-toolchain
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-binutils
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-crt-git
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-gcc
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-gcc-ada
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-gcc-fortran
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-gcc-libgfortran
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-gcc-libs
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-gcc-objc
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-gdb
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-gdb-multiarch
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-headers-git
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-libgccjit
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-libmangle-git
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-libwinpthread-git
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-make
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-pkgconf
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-tools-git
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-winpthread-git
mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-winstorecompat-git

PB@MyComputer MSYS ~
$ pacman -Qe mingw-w64-ucrt-x86_64-gcc
mingw-w64-ucrt-x86_64-gcc 13.2.0-2

PB@MyComputer MSYS ~
$ pacman -Qe mingw-w64-ucrt-x86_64-headers-git
mingw-w64-ucrt-x86_64-headers-git 11.0.0.r124.ge89de847d-1

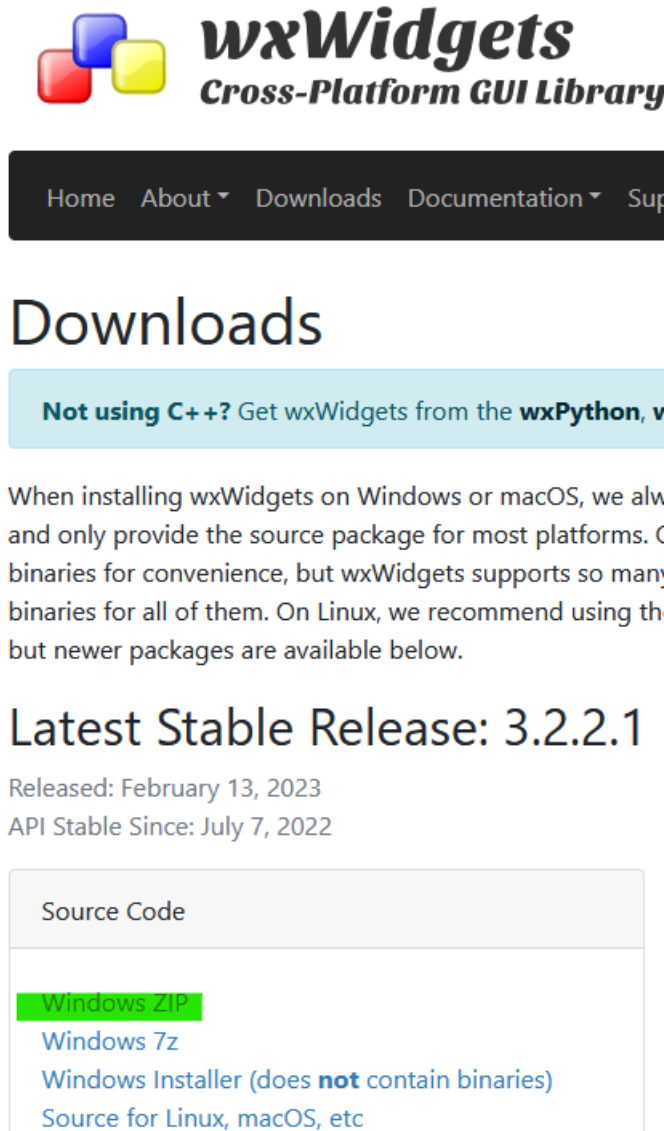
PB@MyComputer MSYS ~
$ |

```


3.4 Downloading wxWidgets Sources

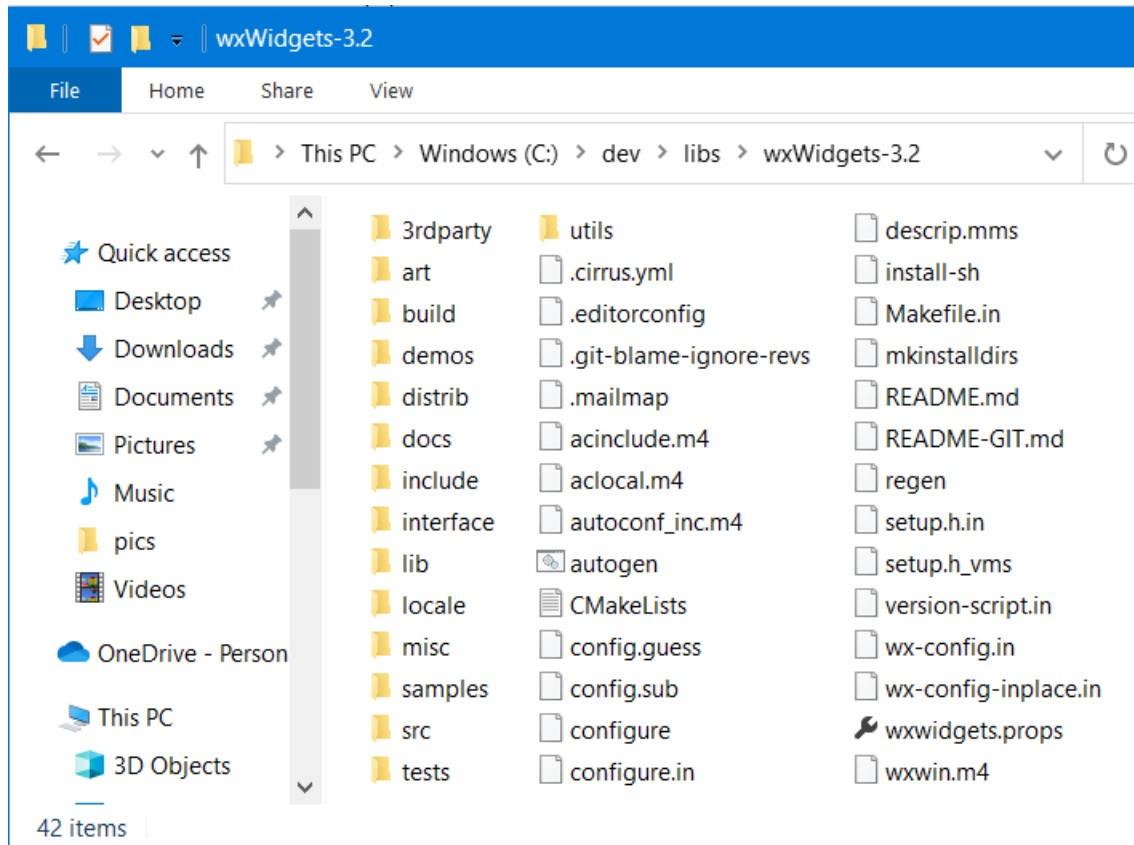
This guide shows how we can build wxWidgets by ourselves, so we will download sources and build the binaries. This guide will use the latest stable release, which at the time of writing was 3.2.2.1, but the procedure would be the same for all wxWidgets 3.2 releases. Go to <https://wxwidgets.org/downloads/> and download the **Windows ZIP** archive file, see Figure 3-2.

Figure 3-2 Downloading wxWidgets source code



Extract the archive `wxWidgets-3.2.2.1.zip` to a folder with the folder hierarchy preserved (do not use a path with spaces in it) such as `c:\dev\libs\wxWidgets-3.2`, see Figure 3-3. This wxWidgets root folder will be referred to as *WXDIR* further on in this guide. In other words, if this guide tells you to go for example to `WXDIR\samples` folder, you are expected to go to folder such as `c:\dev\libs\wxWidgets-3.2\samples`.

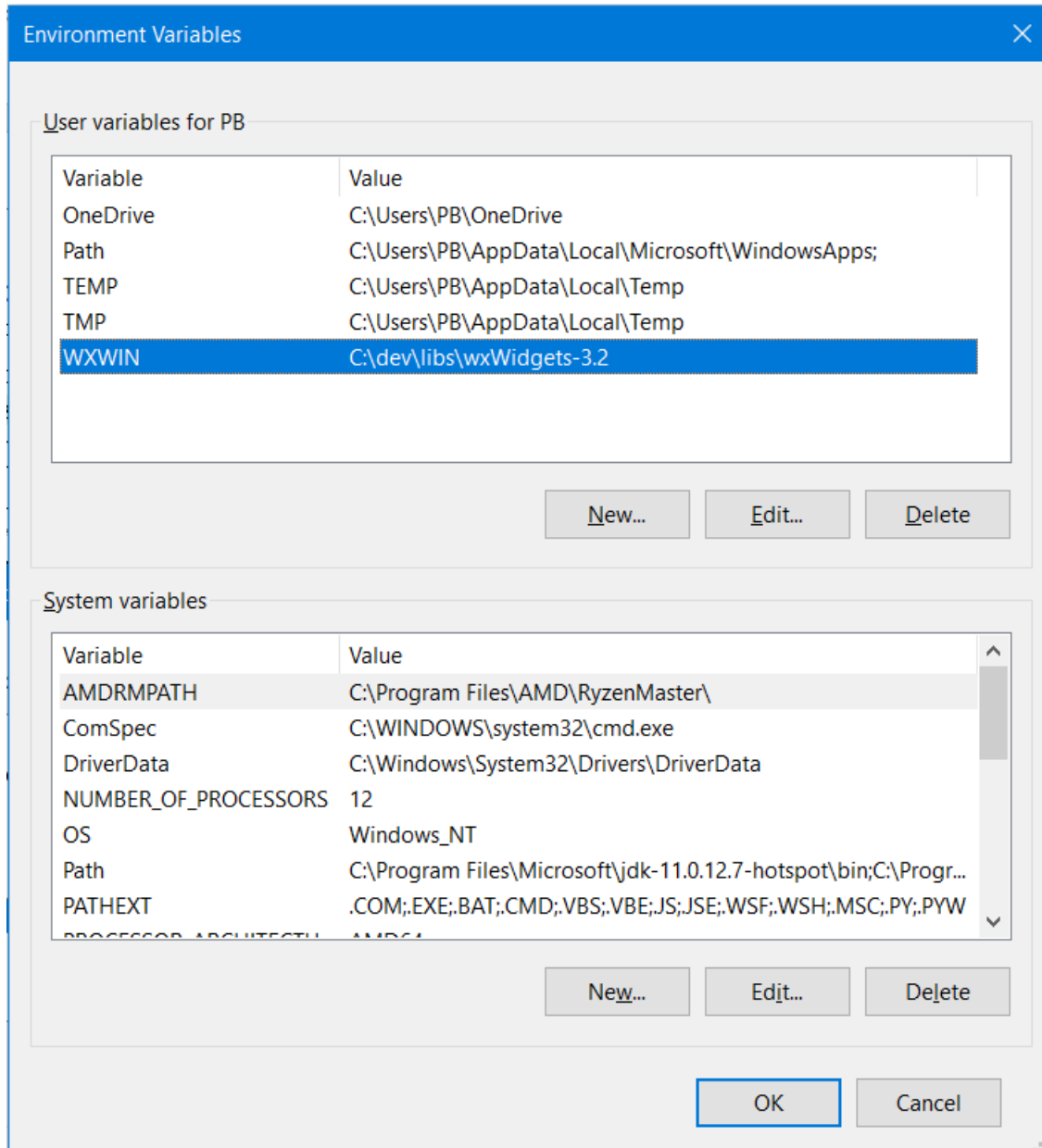
Figure 3-3 wxWidgets root folder shown in File Explorer



3.5 Adding System Environment Variable *WXWIN*

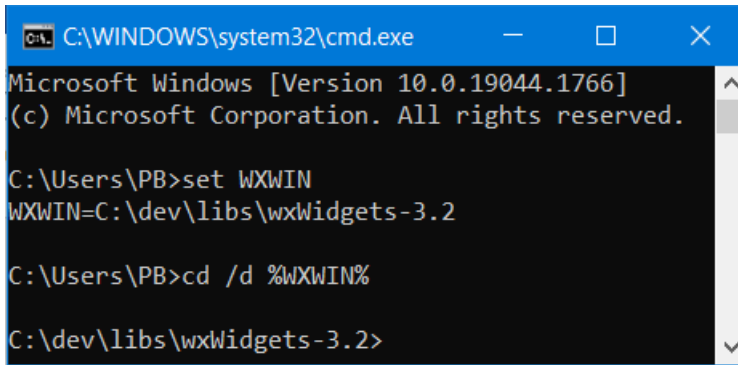
Use the internet to find out how to set a system environment variable on your system (this is quite different depending on the Windows version you are using) and create system environment variable *WXWIN* pointing to your *WXDIR* (such as *c:\dev\libs\wxWidgets-3.2*), see Figure 3-4. While it is not usually needed, it may still be best to restart the computer after that, as the environment variables of running processes are not updated, which can sometimes cause issues.

Figure 3-4 Setting *WXWIN* environment variable



Once you are done, you may launch the Command Prompt and use the command `set WXWIN` to verify that the *WXWIN* system environment variable is properly set (Figure 3-5), see also bundled *test-wxwin.bat*.

Figure 3-5 Verifying WXWIN environment variable



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19044.1766]
(c) Microsoft Corporation. All rights reserved.

C:\Users\PB>set WXWIN
WXWIN=C:\dev\libs\wxWidgets-3.2

C:\Users\PB>cd /d %WXWIN%

C:\dev\libs\wxWidgets-3.2>
```

Note. If you for some odd reason decide to not set `WXWIN` as described above and recommended by wxWidgets developers, you need to replace `%WXWIN%` with the hardcoded `WXDIR` path (e.g., `c:\dev\libs\wxWidgets-3.2`) everywhere `%WXWIN%` is used in this guide.

3.6 The Build

We now have wxWidgets sources as well as the software to build it so let's build wxWidgets. You can use several approaches to build wxWidgets with MinGW (bundled GCC makefile, CMake, configure...), in this guide I will use the bundled GCC makefile located in `WXDIR\build\msw` and Windows command prompt (not MSYS2 environment).

Note. The batch files referenced in this chapter are available in [the bats/build folder of the GIT repository](#).

3.6.1 Starting Simple

We are going to build wxWidgets in Debug and Release configurations in the shared (DLL) Multilib build using multiple commands. We need to (1) add the compiler directory to `PATH` to make it available for the following commands, (2) change the current directory to the folder with the makefile, and (3) start the shared build for Debug and Release configurations. The commands needed for these three steps are probably best run as a batch file (a text file with `.bat` extension containing commands for the command interpreter) and they could look like those in Listing 3-1.

Note. I recommend just reading and understanding the build commands in chapters 3.6.1 and 3.6.2 but actually building wxWidgets as explained in chapter 3.6.3, which should be much faster. Additionally, before building wxWidgets, I recommend reading chapter 3.6.8 about customizing the build.

Listing 3-1 Basic build commands (available as `build-library-shared-simplest.bat`)

```
set PATH=c:\msys64\ucrt64\bin;%PATH%

cd /d %WXWIN%\build\MSW

mingw32-make -f makefile.gcc SHARED=1 BUILD=debug SHELL=cmd.exe
mingw32-make -f makefile.gcc SHARED=1 BUILD=release SHELL=cmd.exe
```

The first command adds MinGW's bin folder to `PATH` (if your MinGW location is different, you need to use that one), but only for the commands in this batch file. Please note that we added the folder at the beginning of `PATH`, which means that our MinGW programs (make, compiler, linker...) will be hopefully picked if there is another GCC-based compiler toolchain in `PATH` (it should not be, see chapter 3.2, but better be safe than sorry).

The second command changes the current directory to the wxWidgets MSW build folder, because that is where `makefile.gcc` is located. The other two commands actually build wxWidgets with the make, using parameters affecting the build.

Firstly, `-f makefile.gcc` tells the make we want to use file `makefile.gcc` as the makefile. Secondly, using `1` as the value for the parameter `SHARED` tells it the build will be dynamic (for the static build we would simply omit the `SHARED` parameter as the static build is the default, or use `SHARED=0` to make our intent clearer). Thirdly, the value of parameter `BUILD` decides whether to build the Debug or the Release build, as we want both, we need to run the make twice with a different value of `BUILD`. If parameter `BUILD` is not used, the build defaults to Debug. You may notice there is no parameter asking to build the Multilib build: Multilib is built by default; we would have to ask for a monolithic build with `MONOLITHIC=1`.

The last build parameter `SHELL` may not be always needed but is necessary when GNU Bash (`sh.exe`) can be found in `PATH`, for example because you have GIT's or MSYS's bin folder there. Unless told otherwise, when MinGW make finds `sh.exe` it is going to use it as its shell. However, this does not work for the Windows command line build and leads to errors with rather confusing messages (usually containing `CreateProcess` in them), so here we tell the make to always use the Windows command interpreter `cmd.exe` as the shell.

3.6.2 Improving Build Commands

There is nothing wrong with the commands used in the previous section but there is still room for improvement. Firstly, you probably want to use all nice modern C++ features, so we will tell the compiler we want to use C++17 standard (GCC 13 defaults to GNU++17). Secondly, it is possible that the build fails, so it would be great if the command window stayed open in this case so we can see what went wrong. Listing 3-2 contains the improved version of the batch file, with the differences in blue and green. Lines starting with **REM** are comments (REMARKs) and are not executed.

Listing 3-2 Improved build commands (available as *build-library-shared.bat*)

```
set PATH=c:\msys64\ucrt64\bin;%PATH%

cd /d %WXWIN%\build\MSW

REM Build wxWidgets Shared Debug configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17" SHELL=cmd.exe
IF %ERRORLEVEL% NEQ 0 goto FAIL

REM Build wxWidgets Shared Release configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=release CXXFLAGS="-std=c++17" SHELL=cmd.exe
IF %ERRORLEVEL% NEQ 0 goto FAIL

goto SUCCESS

:FAIL
echo Build failed, see above why.
pause
goto FINISHED

:SUCCESS

:FINISHED
```

If you are not familiar with batch files, the above may appear intimidatingly complex, but it is actually very simple. You probably deduced that `CXXFLAGS="-std=c++17"` was used to tell the compiler we want it to use C++17 standard. However, the rest of new commands can look a bit cryptic. Line `IF %ERRORLEVEL% NEQ 0 goto FAIL` tells the command processor that if the build failed (make's exit code does not equal 0), the command processing should resume at label `FAIL`, where `echo` command is used to display "Build failed, see above why." and `pause` command to keep the command line window open. You can see nothing is done at label `SUCCESS`, which means that if both Debug and Release build succeed, the command line window will be closed.

3.6.3 Making wxWidgets Build Faster

MinGW make supports parallel building, where multiple files can be processed simultaneously. If your computer has multiple CPU cores, enough RAM, and its performance is not hindered by a slow hard disk drive, parallel building can significantly speed up building large code bases such as wxWidgets. There is a `-jN` parameter where `N` is the number of CPU cores MinGW make can use (e.g., `-j4` will use 4 CPU cores), using just `-j` lets the make decide how many jobs to use. However, there is a long-standing and virtually unfixable bug in wxWidgets GCC makefile when using parallel building. We need to work around this bug by building `setup_h` target first without using parallel compilation and then running make again with the same parameters but without `setup_h` and with `-jN`. Listing 3-3 contains commands for creating a shared build using parallel building with 4 jobs.

Listing 3-3 Build commands for dynamic wxWidgets build using parallel build with 4 jobs (available as *build-library-shared-parallel.bat*)

```
set PATH=c:\msys64\ucrt64\bin;%PATH%

cd /d %WXWIN%\build\MSW

REM Build wxWidgets Shared Debug configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17" SHELL=cmd.exe setup_h
IF %ERRORLEVEL% NEQ 0 goto FAIL
mingw32-make -j4 -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17" SHELL=cmd.exe
IF %ERRORLEVEL% NEQ 0 goto FAIL

REM Build wxWidgets Shared Release configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=release CXXFLAGS="-std=c++17" SHELL=cmd.exe setup_h
IF %ERRORLEVEL% NEQ 0 goto FAIL
mingw32-make -j4 -f makefile.gcc SHARED=1 BUILD=release CXXFLAGS="-std=c++17" SHELL=cmd.exe
IF %ERRORLEVEL% NEQ 0 goto FAIL

goto SUCCESS

:FAIL
echo Build failed, see above why.
pause
goto FINISHED

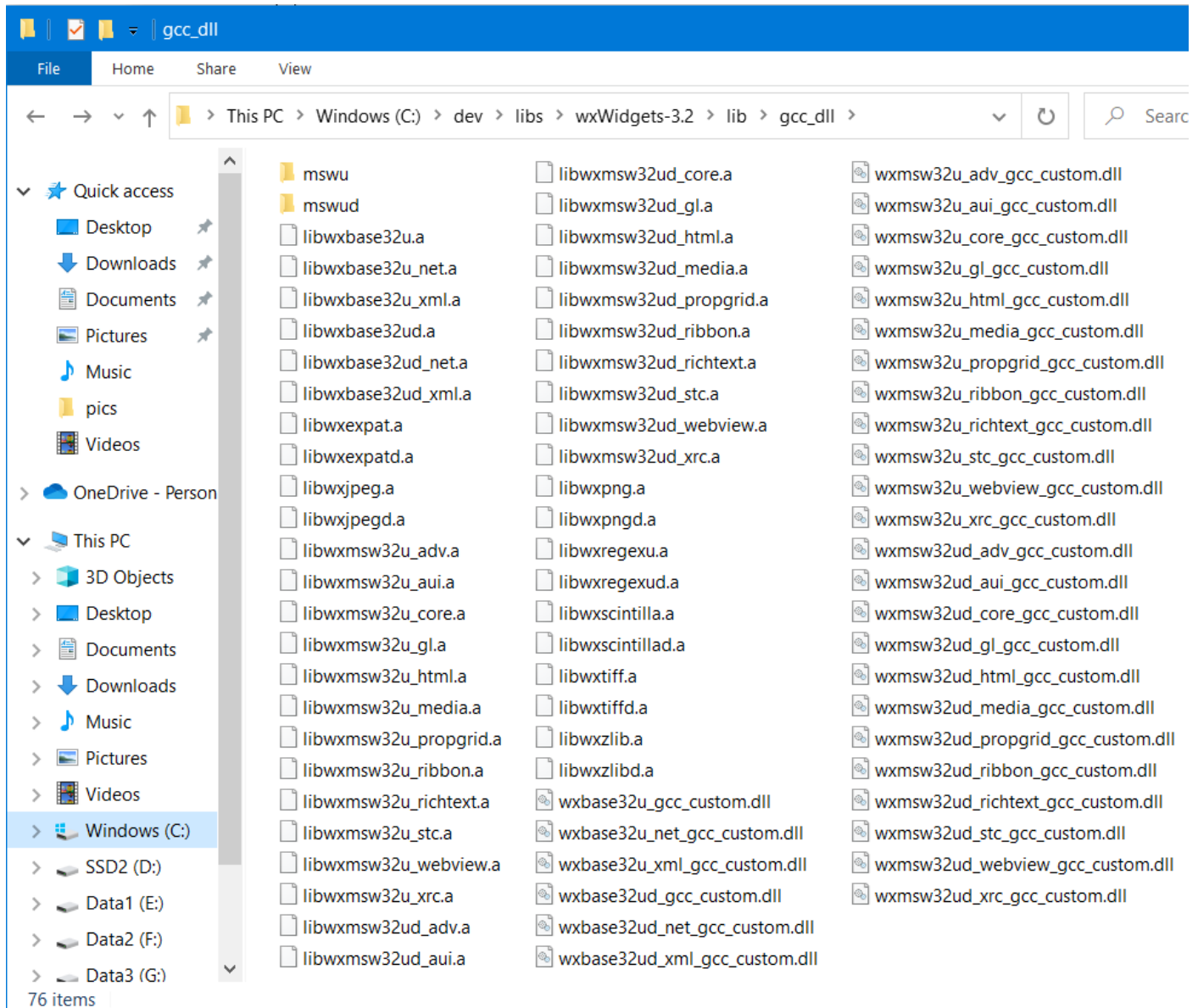
:SUCCESS

:FINISHED
```

3.6.4 Viewing Build Results

Now run the build commands from Listing 3-3 and wait till the build finishes. If it finished successfully, the command line window should be closed (if not, it will stay open with “Build failed, see above why. Press any key to continue...” message at its bottom and the actual error message above it). The libraries for the shared build will be by default generated in `WXDIR\libgcc_dll` where 2 folders and 74 files should be present, see Figure 3-6.

Figure 3-6 Contents of `WXDIR\libgcc_dll` folder



Files starting with `lib` and having an extension `.a` are import libraries used by the linker when building the application while files with an extension `.dll` are dynamic libraries needed by Windows to run the application (e.g., `libwxbase32u.a` vs `wxbase32u_gcc_custom.dll`).

You can differentiate files for the Debug and Release configurations as the Debug ones have an extra `d` in their name compared to the Release (e.g., `libwxbase32ud.a` vs `libwxbase32u.a` and `wxbase32ud_gcc_custom.dll` vs `wxbase32u_gcc_custom.dll`, or `libwxzlibd.a` vs `libwxzlib.a`).

You may have noticed that the file names for the import and dynamic libraries do not differ only in the prefix and file extension. For example, let's take a look at the base library, where the import library is named `libwxbase32u.a` and the

dynamic library `wxbase32u_gcc_custom.dll`. We can see that the file name for the dynamic library carries more information. Firstly, the compiler name is included (`gcc`). Secondly, there is string “`custom`” which is the so-called vendor part defaulting to just “`custom`”. You can change the vendor when building wxWidgets, using `VENDOR` build parameter. For example, if you used `VENDOR=PB` as one of the build parameters, the file name for the base dynamic library would be `wxbase32u_gcc_PB.dll` instead of the default `wxbase32u_gcc_custom.dll`.

You may also have noticed that not every library ending in `.a` has its `.dll` equivalent. Some of the libraries are linked into the dynamic libraries. This applies to the third-party libraries such as those for image formats or zlib. See also Table 4-1 for more information about the individual libraries.

Remember that you need to ship all required DLLs (compiler’s, wxWidgets’, and possibly others) with your application and let the installer put them in the same folder where the application executable is (but you can avoid this, see chapter 3.6.7).

The two folders are build-specific (`mswu` for the Release and `mswud` for the Debug build) and contain build-specific `setup.h`. They need to be added to the compiler include directories, in addition to the build-agnostic main include folder (`WXDIR\include`), see chapter 4.8.1 for more information.

Note 1. The names of DLLs provided in the official binaries include the compiler version, have the vendor empty, and 64-bit versions end with “`_x64`”. For example, the name of the 64-bit release DLL for the base library built with GCC 12.2.0 is `wxbase32u_gcc1220_x64.dll`.

Note 2. If you are not sure which DLLs your executable dynamically links with, you can use program [Dependencies](#) to tell you that.

Note 3. Files for the static build are by default created in `WXDIR\lib\gcc_lib`.

3.6.5 Using the Minimal Sample to Test the Build

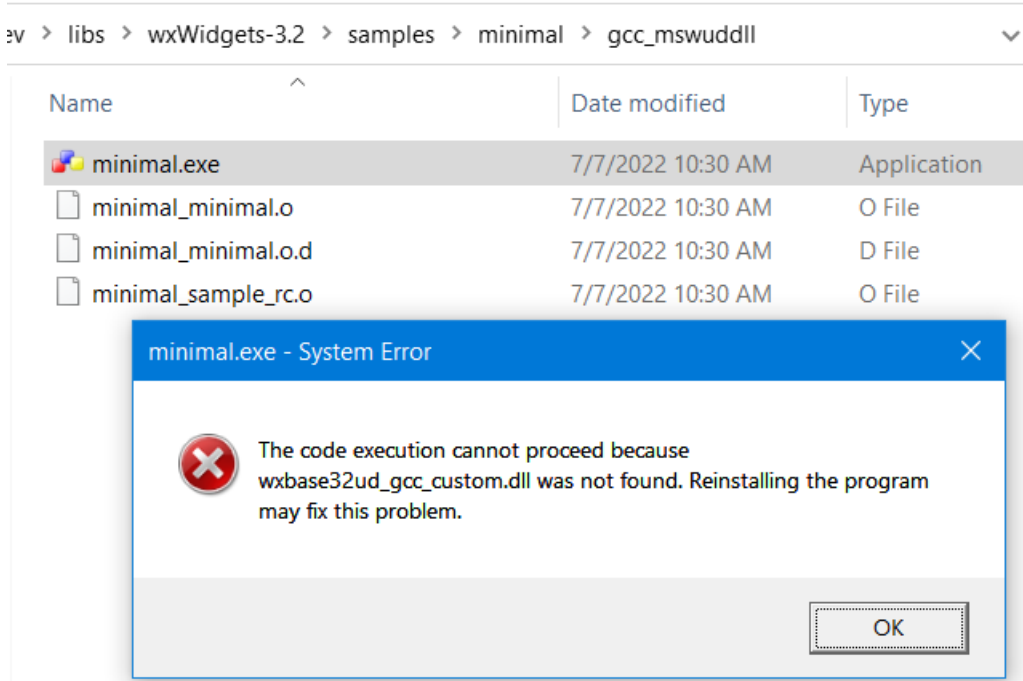
Now let us test our wxWidgets build with the bundled Minimal sample. Go to `WXDIR\samples\minimal` folder and run the commands shown in Listing 3-4.

Listing 3-4 Build commands for the minimal sample (available as `build-sample-minimal-shared-debug.bat`)

```
set PATH=c:\msys64\ucrt64\bin;%PATH%
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17" SHELL=cmd.exe
```

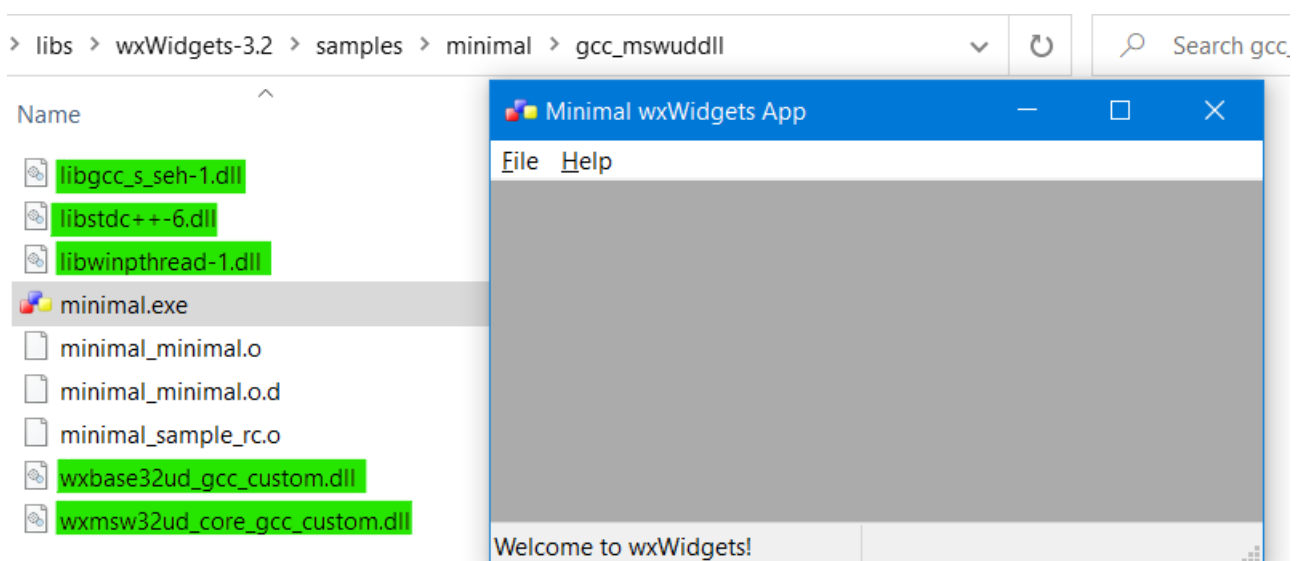
This should build the Debug configuration of the Minimal sample. Please note that the build parameters must be the same as when building wxWidgets itself. Assuming the build succeeded, you should have file `minimal.exe` in the `gcc_mswuddll` folder of the Minimal sample folder. However, if you just try to launch the sample now, instead of the expected Minimal sample window you will probably end up seeing something similar to Figure 3-7, with several error messages about missing DLLs popping one after another.

Figure 3-7 Launching minimal sample with missing DLLs



What happened? The application executable needs both GCC runtime libraries DLLs and wxWidgets DLLs. The recommended way is to put those in the same folder as the executable. So, you need to copy all the required DLLs to the folder where `minimal.exe` is. Firstly, from the MinGW bin folder such as `c:\msys64\ucrt64\bin` copy `libgcc_s_seh-1.dll`, `libstdc++-6.dll`, and `libwinpthread-1.dll`. Secondly, copy from folder `WXDIR\lib\gcc_dll` files `wxbase32ud_gcc_custom.dll` and `wxmsw32ud_core_gcc_custom.dll`. Please note that you may need other wxWidgets DLLs for applications using more wxWidgets features (and release versions of the DLLs for the release build of an application). Now run `minimal.exe` again and you should see what is in Figure 3-8.

Figure 3-8 Launching minimal sample successfully



Note. If the GCC and wxWidgets DLLs are not put in the same folder as the application executable run-time dynamic linking with them, it is possible that a DLL with the same name (e.g., *libstdc++-6.dll*) but in a different version than the application was built against (i.e., incompatible with the application) will [be picked by Windows](#) from somewhere in *PATH* or the current working directory. In this case, Windows will probably show an error when attempting to launch the application, usually about missing an export.

3.6.6 Cleaning the Build

When changing build parameters for the same configuration it is best to make clean build, to make sure there are no leftovers from the previous build. All you need is to run the same build commands as when building but with **clean** target added, see Listing 3-5.

Listing 3-5 Example of cleaning the build

```
set PATH=c:\msys64\ucrt64\bin;%PATH%
cd /d %WXWIN%\build\MSW

REM First clean and then build wxWidgets Shared Debug configuration with C++17 support
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17" SHELL=cmd.exe clean
IF %ERRORLEVEL% NEQ 0 goto FAIL
mingw32-make -f makefile.gcc SHARED=1 BUILD=debug CXXFLAGS="-std=c++17" SHELL=cmd.exe
IF %ERRORLEVEL% NEQ 0 goto FAIL

goto SUCCESS

:FAIL
echo Build failed, see above why.
pause
goto FINISHED

:SUCCESS

:FINISHED
```

However, that may not always work well, in particular when trying to clean the build after changing the some of its parameters such as switching between multilib and monolithic builds or modifying *WXDIR\include\msw\setup.h*. What I usually do is just delete the folders starting with *gcc_* in *WXDIR\build\msw* and *WXDIR\lib*. You can include the delete commands in the batch file, for example, as shown in Listing 3-6.

Listing 3-6 Cleaning all GCC builds in a batch file using rd command

```
for /d %%G in ("%WXWIN%\build\msw\gcc_*") do rd /s /q "%%~G"
for /d %%G in ("%WXWIN%\lib\gcc_*") do rd /s /q "%%~G"
```

3.6.7 Building wxWidgets Statically

It is possible to build wxWidgets so that at run-time your application does not need any wxWidgets or compiler DLLs. To create static build of wxWidgets, we need to use **SHARED=0** instead of **SHARED=1** we used for dynamic build. To link the compiler libraries statically we will use **-static** flag for the linker. Listing 3-7 contains all the commands, with the differences to the default build described in Listing 3-2 in red.

Listing 3-7 Build commands for static wxWidgets build (available as *build-library-static.bat*)

```
set PATH=c:\msys64\ucrt64\bin;%PATH%

cd /d %WXWIN%\build\MSW

REM Build wxWidgets Static Debug configuration with C++17 support and GCC libraries linked statically
mingw32-make -f makefile.gcc SHARED=0 BUILD=debug CXXFLAGS="-std=c++17" LDFLAGS="-static" SHELL=cmd.exe
IF %ERRORLEVEL% NEQ 0 goto FAIL

REM Build wxWidgets Static Release configuration with C++17 support and GCC libraries linked statically
mingw32-make -f makefile.gcc SHARED=0 BUILD=release CXXFLAGS="-std=c++17" LDFLAGS="-static" SHELL=cmd.exe
IF %ERRORLEVEL% NEQ 0 goto FAIL

goto SUCCESS

:FAIL
echo Build failed, see above why.
pause
goto FINISHED

:SUCCESS

:FINISHED
```

The static wxWidgets libraries will be by default generated in the *gcc_lib* folder of *WXDIR\lib* folder. If the build succeeded, the content of the folder should look similar to that in Figure 3-6, except there will be no DLL files and the libraries (the *.a files) will be much larger.

Note 1. Batch file for static parallel build is available as *build-library-static-parallel.bat*.

Note 2. The Code::Blocks project used in this guide uses the dynamic build by default, chapters 4.5.9 and 4.8.4 provide information about using the static build there.

3.6.8 More Build Options

There are many more wxWidgets options that affect the build. Some of them can be set in `WXDIR\include\msw\setup.h`, see the comments in that file and/or in [the official documentation](#). Others are set through arguments to make, see their description in [the official documentation](#). Discussing them in detail is beyond the scope of this guide, so I am going to mention only the select few I consider important for most users. All the following changes need to be done in file `WXDIR\include\msw\setup.h`. If you make any changes there, make sure to properly clean the build (see chapter 3.6.6) and perform full wxWidgets rebuild.

If you want to draw with `wxGraphicsContext` using the Direct2D based `wxGraphicsRenderer`, you need to make sure that `wxUSE_GRAPHICS_DIRECT2D` is set to 1. If you want to use `wxWebView`, you are probably better off with the backend using Chromium-based Microsoft Edge instead of one based on the obsolete Microsoft Internet Explorer, which means that you need to set `wxUSE_WEBVIEW_EDGE` to 1 (and make sure to follow instructions in the `wxWEBVIEW_BACKEND_EDGE` part of [the wxWebView documentation](#) and read the documentation regarding the backend limitations). Be aware that the Edge backend has some limitations compared to the old Explorer one.

You may also set `wxUSE_STD_CONTAINERS` to 1, so that wxWidgets container classes such as `wxVector` will use standard container classes shipped with your compiler instead of wxWidgets own implementations. I would also recommend setting `wxUSE_UNSAFE_WXSTRING_CONV` to 0, in particular, if you are not depending on some old third party wxWidgets-using code. This can prevent some nasty bugs when dealing with strings.

Note. Code::Blocks wxWidgets project wizard supports custom `CFG` (see chapter 4.5.9) but does not support custom `COMPILER_PREFIX`. Therefore, if you want to build both 32- and 64-bit configurations, you need to use `CFG` parameter to distinguish between the two builds (e.g., `CFG=-x86` and `CFG=-x64`, resulting into libraries being built in folders `WXDIR\lib\gcc_dll-x86` and `WXDIR\lib\gcc_dll-x64`, respectively).

4 Setting Up wxWidgets Project with Code::Blocks and MinGW

4.1 Installing Code::Blocks

This guide uses Code::Blocks v20.03, which is its latest version available at the time of writing. Go to <https://www.codeblocks.org/downloads>, click **Download the binary release** and download **codeblocks-20.03-setup.exe**. This is the installer without a compiler bundled, as we are going to use MinGW we installed in chapter 3.2. Launch the installer and install Code::Blocks with the default settings. Now launch Code::Blocks for the first time. It should display a dialog stating it could not auto-detect any (GCC-based) compilers. This is actually a good thing as it means there is no GCC-based compiler toolchain in the path or a known location where it could interfere with our own MinGW installation. Close the dialog with **OK** and continue. Select your choice in the next dialog asking whether to use Code::Blocks as the default application for C++ files and you will end up in the Code::Blocks IDE.

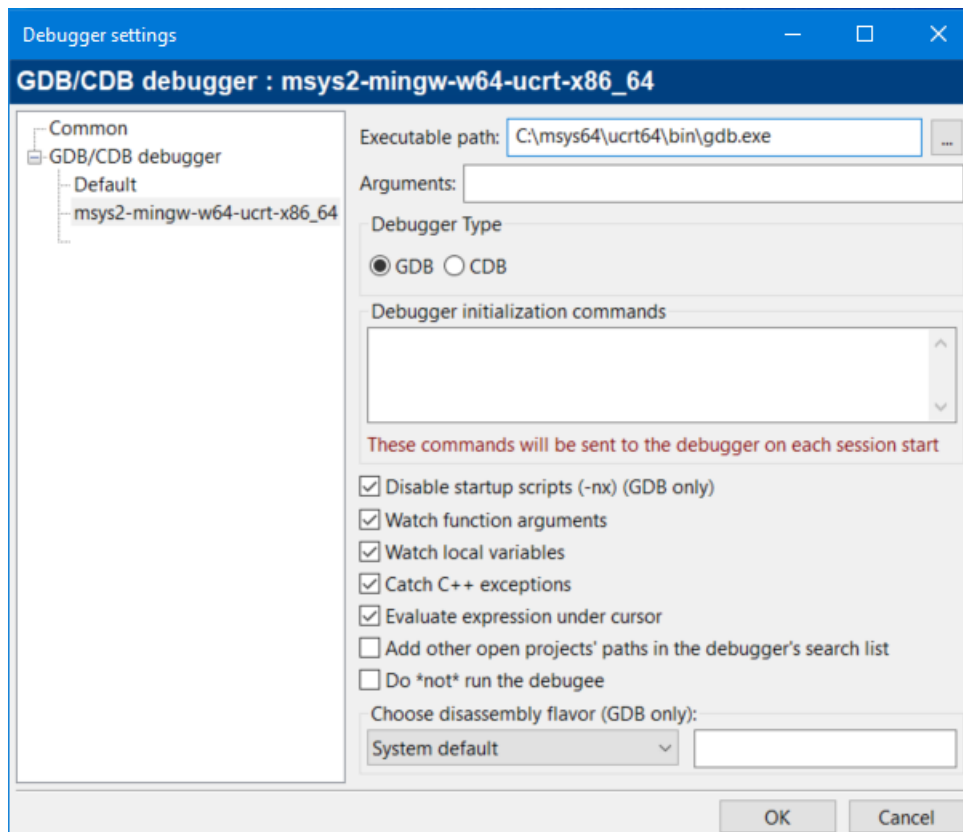
Now we have to tell Code::Blocks that it should use the MinGW compiler toolchain we installed in chapter 3.2 of this guide. Somewhat counter-intuitively, we need to set up the debugger before compiler, because we will use the new debugger configuration in the compiler settings.

Note. Code::Blocks 20.03 was released back in March of 2020 and so is rather old. However, while there was no official release since then, its development continues and if you want, you can try using the latest unreleased version (at your own risk), so-called nightly build. You can learn more about those [here](#).

4.2 Configuring the Debugger

From the main menu choose **Settings / Debugger...**, in the left part of the **Debugger settings** dialog select **GDB/CDB debugger** and click the **Create Config** button. Type **msys2-mingw-w64-ucrt-x86_64** as the name for the new debugger configuration. Select the newly created debugger configuration and in the **Executable path** set the path to GDB from the compiler toolchain we are using (e.g., `c:\msys64\ucrt64\bin\gdb.exe`), see Figure 4-1. Leave all other options at their defaults, perhaps except **Evaluate expressions under cursor**, and close the dialog with the **OK** button.

Figure 4-1 Setting the debugger in Code::Blocks



Note 1. Code::Blocks unfortunately does not display messages produced in wxWidgets applications with `wxLogDebug()` and `wxLogTrace()` (which use `OutputDebugString()` Win32 function). These messages can be quite useful for discovering bugs in the user code. If you want to see these messages during debugging you need to run the application under gdb, or during debugging run [DebugView](#) along Code::Blocks.

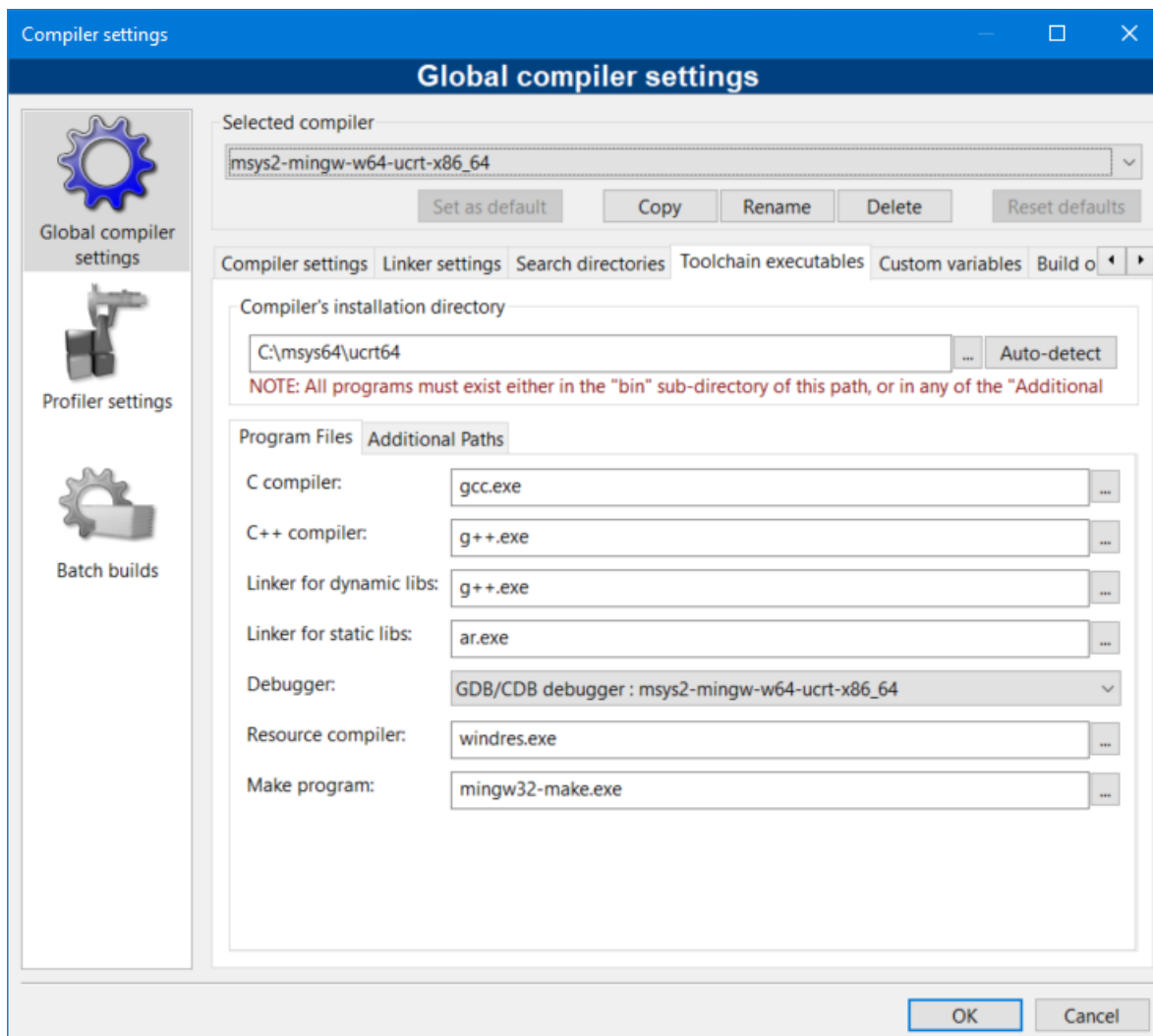
Note 2. You can find GDB pretty printer for several basic wxWidgets types in `WXDIR\misc\gdb\print.py`.

4.3 Configuring the Compiler

From the main menu choose **Settings** / **Compiler...** and in the **Compiler settings** dialog go to page **Global compiler settings**. We will not modify the existing **GNU GCC Compiler** but we will use the **Copy** button to copy **GNU GCC Compiler** to a new compiler we will name after the toolchain we are using **msys2-mingw-w64-ucrt-x86_64**. Creating a new compiler here allows us to switch between different compilers when needed.

Select the newly added compiler **msys2-mingw-w64-ucrt-x86_64**, go to the **Toolchain executables** tab, press the button with three dots next to the field for **Compiler's installation directory** and select the folder where Code::Blocks will look for the build tools. Well, it will actually not look for the tools' executables in this folder – it will use its **bin** subfolder – so for our MinGW installation we set the directory to **c:\msys64\ucrt64** (or wherever did you install MSYS2 on your PC). Make sure the file names for the individual tools match those shown in Figure 4-2 and set **Debugger:** to the configuration **msys2-mingw-w64-ucrt-x86_64** we created in chapter 4.2. Now press the **Set as Default** button, confirm the message about the new compiler being default for new projects and close the dialog with the **OK** button.

Figure 4-2 Setting the compiler in Code::Blocks



We will not set any other global compiler settings; if needed, we will specify them in the project settings instead. This may be a bit more work per project, but it allows flexibility not possible with setting them globally. You certainly do not want to add any library specific stuff here (such as search directories for a compiler or linker), these undoubtedly belong to the project settings. If you added something there earlier, please remove it, as it could interfere with our build later.

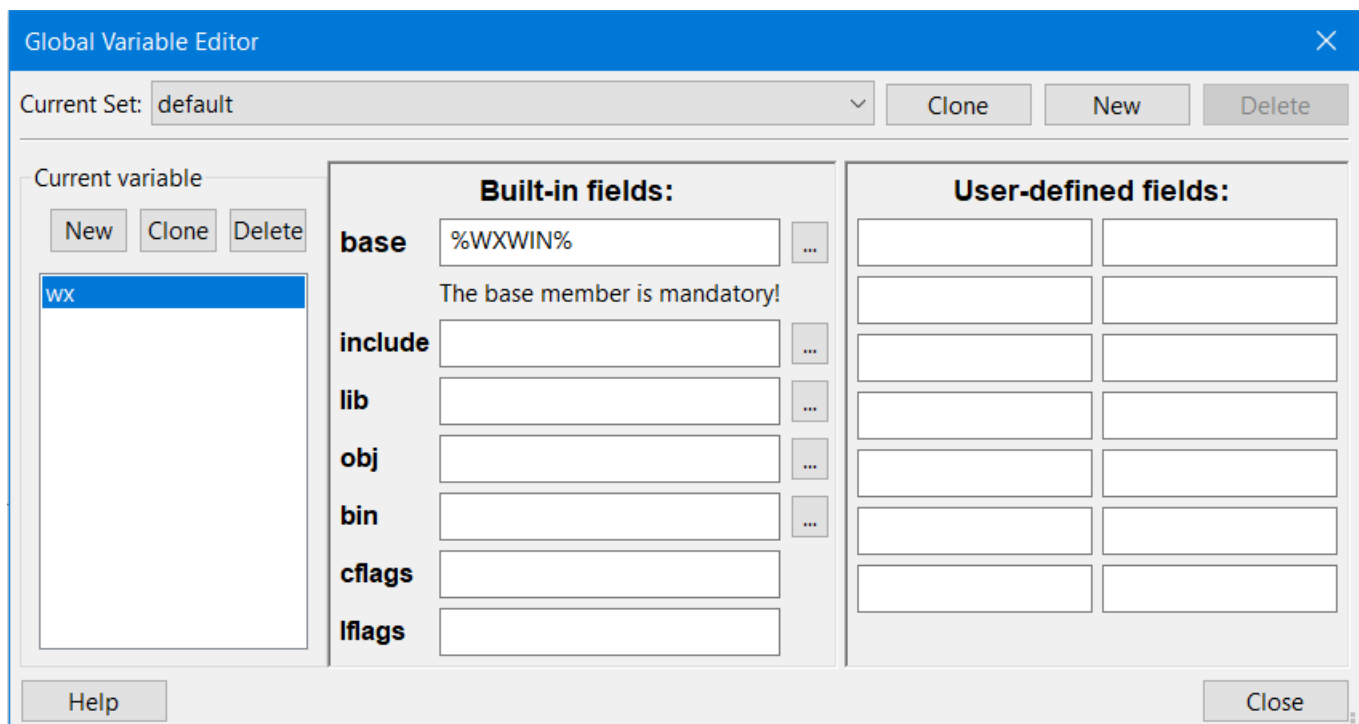
Note. If Code::Blocks cannot find a tool it needs, it complains about it, but just shows a pop-up in the bottom right corner of the screen, which could be easy to miss on a bigger display. Make sure there is no such pop-up, if there is, make sure the **Compiler's installation directory** is correct and the file names for the individual tools are the same as in Figure 4-2. However, it seems that sometimes Code::Blocks gets confused when creating a new compiler and complains even when everything is set correctly, either after closing the Compiler Settings dialog or after restarting it. It seems to fix itself after closing and restarting Code::Blocks again.

4.4 Creating Global Variable wx

As we do not want to use hard-coded paths in our projects, which would make difficult to transfer the project between computers with different folder hierarchies, we will not use a hard-coded path to wxWidgets. We will use a Code::Blocks global variable named `wx`. This variable will use system environment variable `WXWIN` we created in chapter 3.5.

From the main menu choose **Settings / Global variables...** and in the **Global Variable Editor** dialog use the **New** button (below **Current variable**) to create a variable named `wx`. In the **Built-in-fields** set **base** to `%WXWIN%` (Figure 4-3). Please note that there are percent signs around `WXWIN`, which indicates that this is not a filesystem path but a system environment variable that will be expanded to an actual path. Leave all other fields empty and use the **Close** button to close the dialog.

Figure 4-3 Creating global variable `wx`



4.5 Creating a wxWidgets Project with Project Wizard

Now let us create our first wxWidgets project, using a Code::Blocks template wizard. The procedure shown in this chapter assumes you built both Debug and Release Shared builds of wxWidgets (see chapter 3.6) and installed and configured Code::Blocks as shown in chapters 4.3, and 4.4.

wxWidgets project wizard template shipped with Code::Blocks v20.03 is quite outdated. Therefore, this guide (starting with v4.0) uses the wizard I updated to better support modern wxWidgets. You can download it at <https://github.com/PBfordev/wxpbguide/tree/master/cb/wizard>. Among else, unlike the wizard shipped with Code::Blocks v20.03, the updated wizard allows linking with the libraries introduced in wxWidgets 2.9 (PropertyGrid, Ribbon, STC, and WebView). It also allows linking with Winsock2 instead of ancient original Winsock library, Winsock2 is by default used by wxWidgets since v3.1.6.

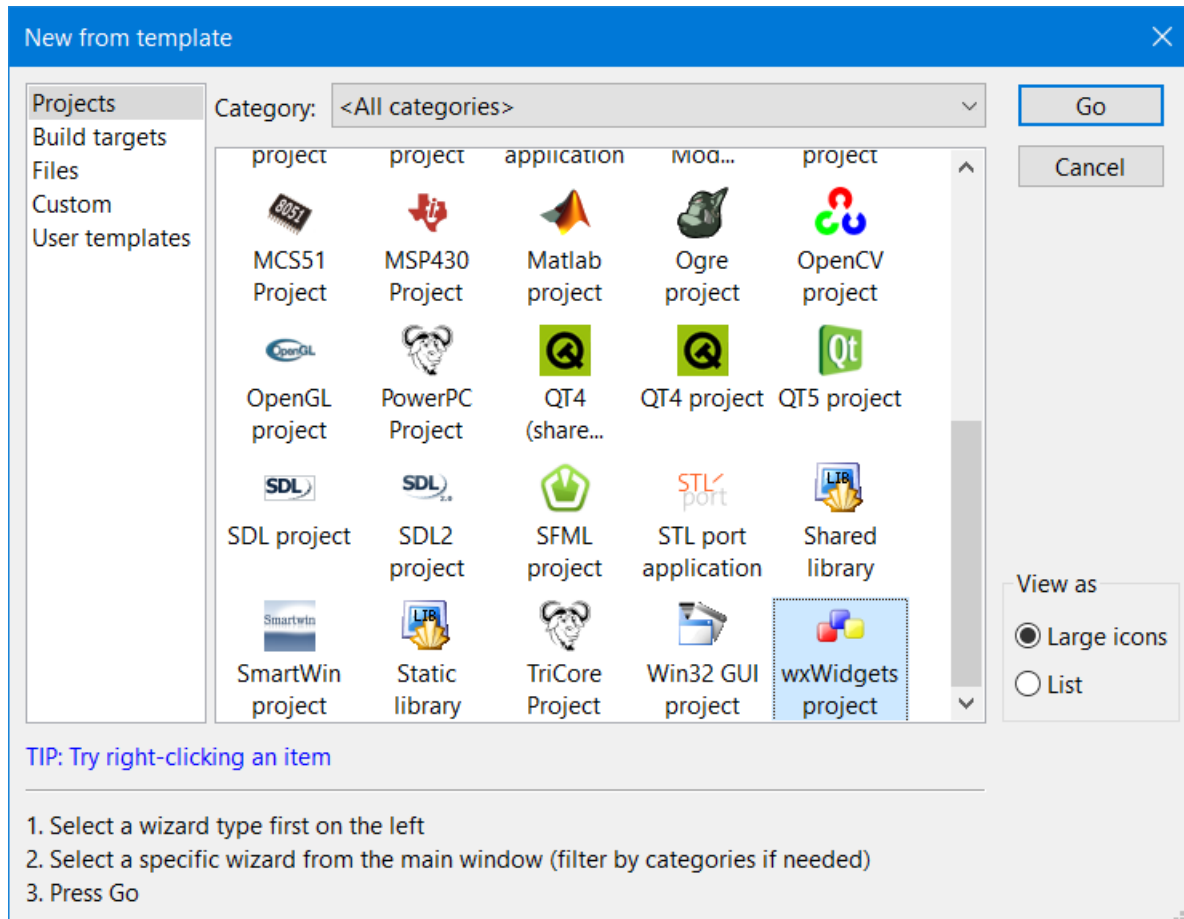
To use the updated wizard, you need to download files `wizard.xrc` and `wizard.script` and put them into folder `%APPDATA%\CodeBlocks\share\codeblocks\templates\wizard\wxwidgets` (e.g., if the username is `PB`, then the path would be `c:\users\PB\AppData\Roaming\CodeBlocks\share\codeblocks\templates\wizard\wxwidgets`). Or instead of copying the files manually, you can use batch file `install.bat` provided in the repository to do it for you. Either way, make sure the wizard is not running in Code::Blocks before you update it.

4.5.1 Selecting wxWidgets Project Template

The wizard has quite few steps, but each individual step is easy and simple. As noted before, the settings used are for a project linking to wxWidgets dynamically. However, possible changes needed when creating a project linking to wxWidgets statically are described in a note for a wizard page.

To launch the wizard, from the main menu go to **File / New / Project...** and in the **New from template** dialog scroll down to the bottom of the project list, select **wxWidgets project** (see Figure 4-4) and press the **Go** button.

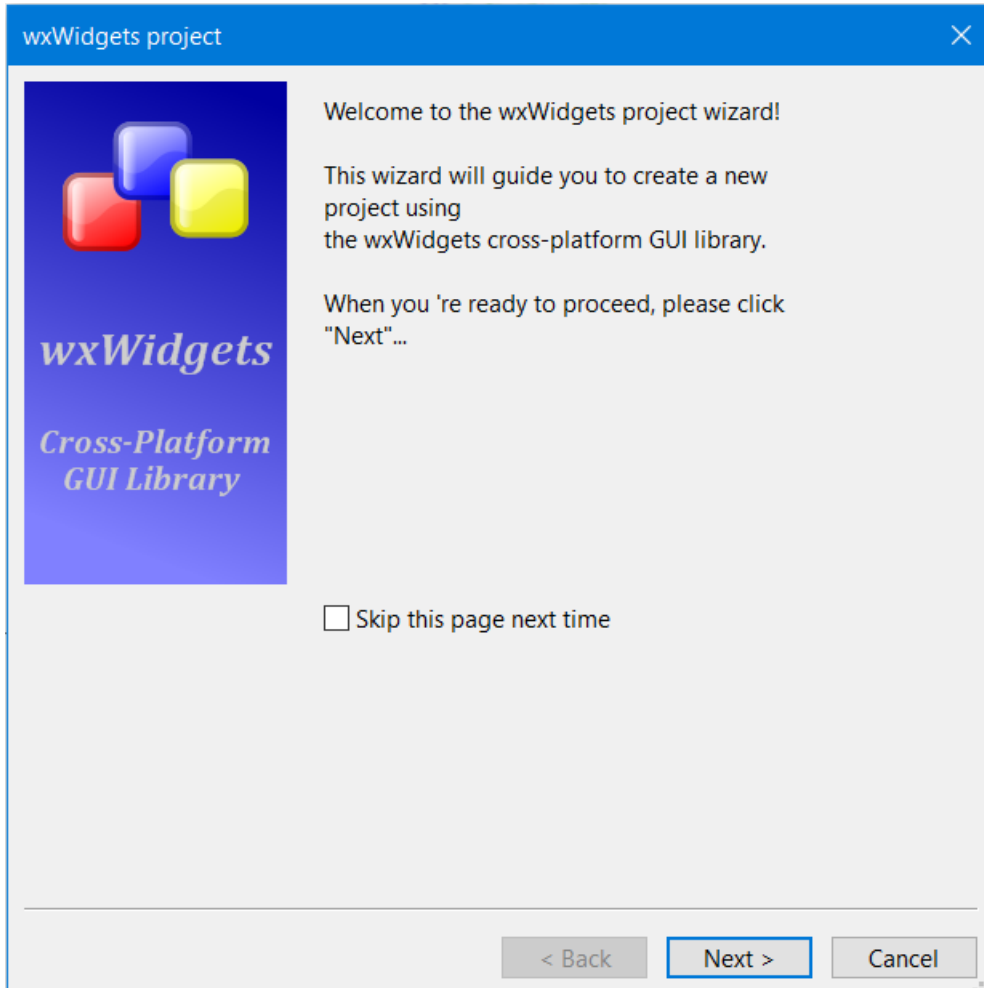
Figure 4-4 Selecting wxWidgets project template



4.5.2 Wizard Page 1: Welcome

The first wizard page (Figure 4-5) only welcomes you and has no settings so just press **Next**.

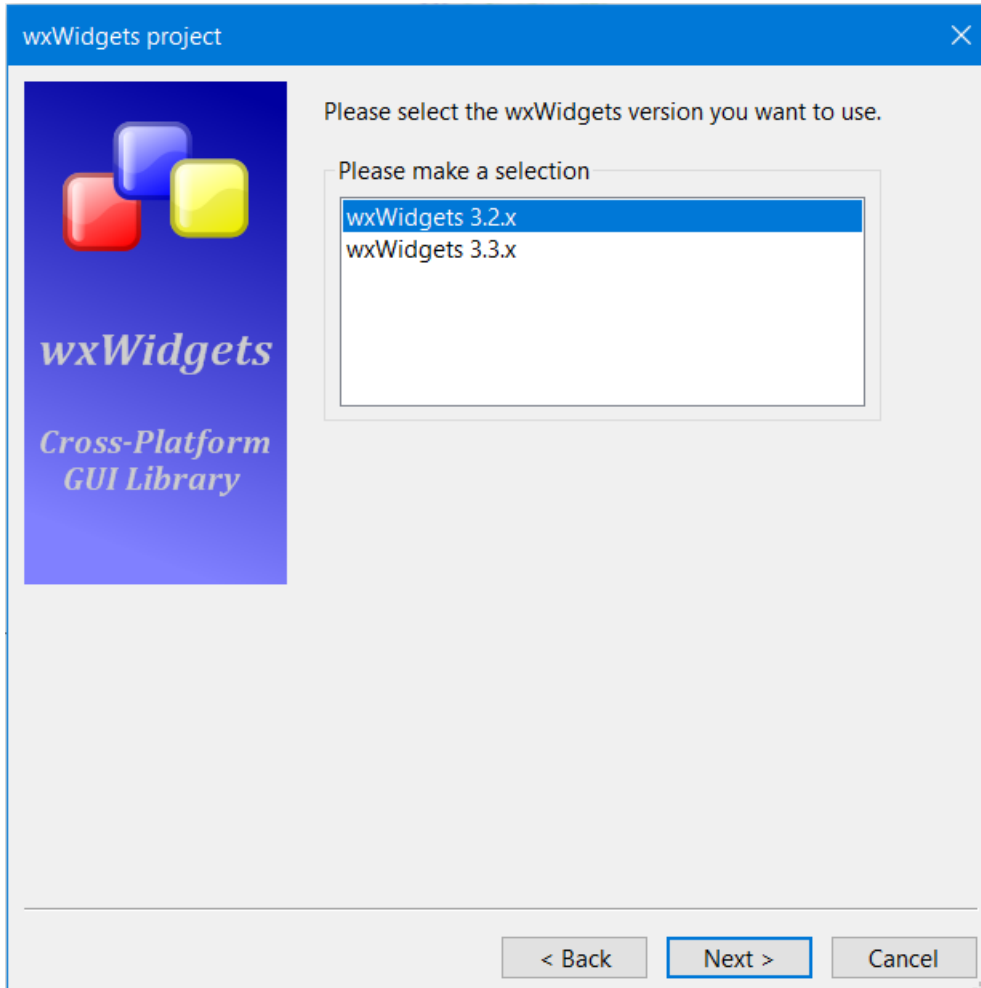
Figure 4-5 Welcome page



4.5.3 Wizard Page 2: wxWidgets Version

In this step we need to select wxWidgets version we are using: Select **wxWidgets 3.2.x** (Figure 4-6) and press **Next**.

Figure 4-6 Selecting wxWidgets version



4.5.4 Wizard Page 3: Project Title and Paths

Fill the project title and the folder where to create the project (Figure 4-7), I strongly recommend against using a path with spaces in it, and press **Next**.

Figure 4-7 Setting project title and paths

wxWidgets project

Please select the folder where you want the new project to be created as well as its title.

Project title:
Test

Folder to create project in:
C:\dev\myapps\

Project filename:
Test.cbp

Resulting filename:
C:\dev\myapps\Test\Test.cbp

< Back Next > Cancel

4.5.5 Wizard Page 4: Project Author

Fill the information about the project's author (Figure 4-8) or leave it empty and press **Next** again.

Figure 4-8 Setting Project Author

wxWidgets project

Please Enter Project Details. Details will be used in labelling source code of project.

Author:

Author's email:

Author's website:

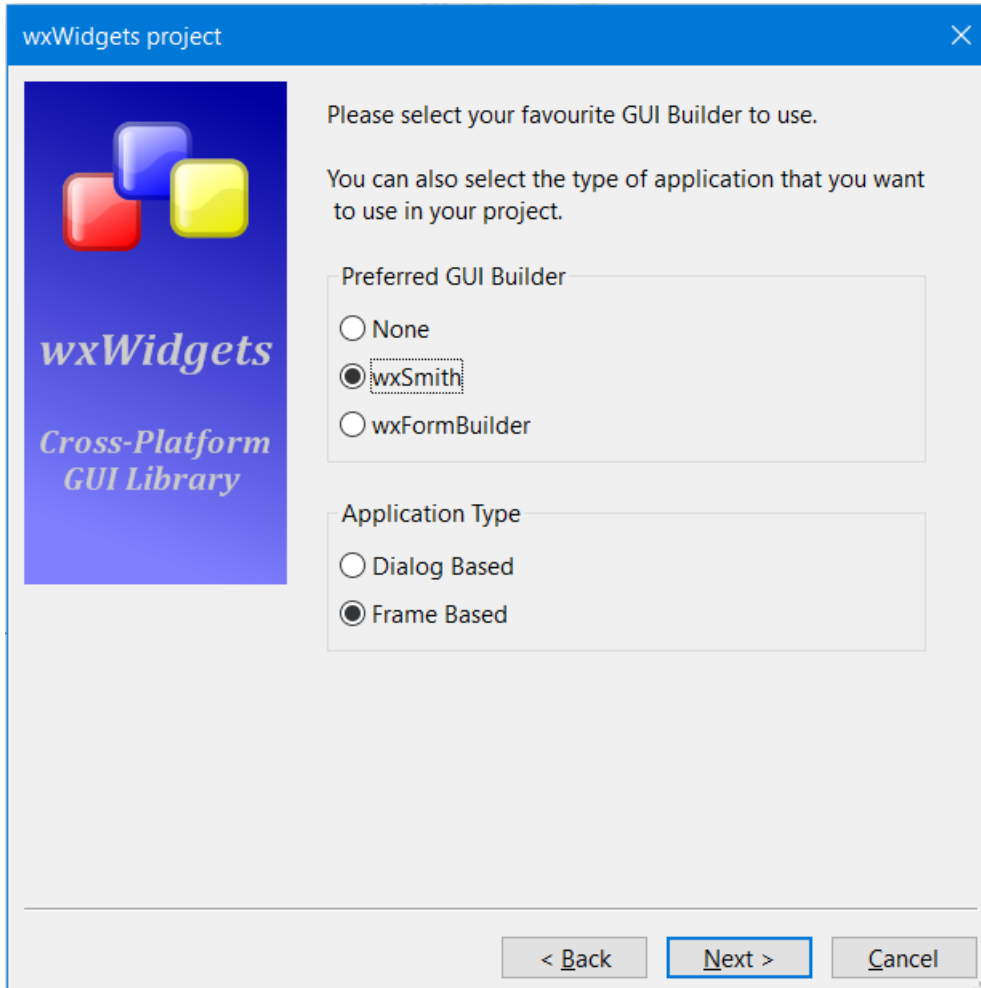
wxWidgets
Cross-Platform
GUI Library

< Back Next > Cancel

4.5.6 Wizard Page 5: GUI Builder and Application Type

Select your preferred GUI builder and application type (Figure 4-9), I chose **wxSmith** and **Frame Based**, and press **Next**.

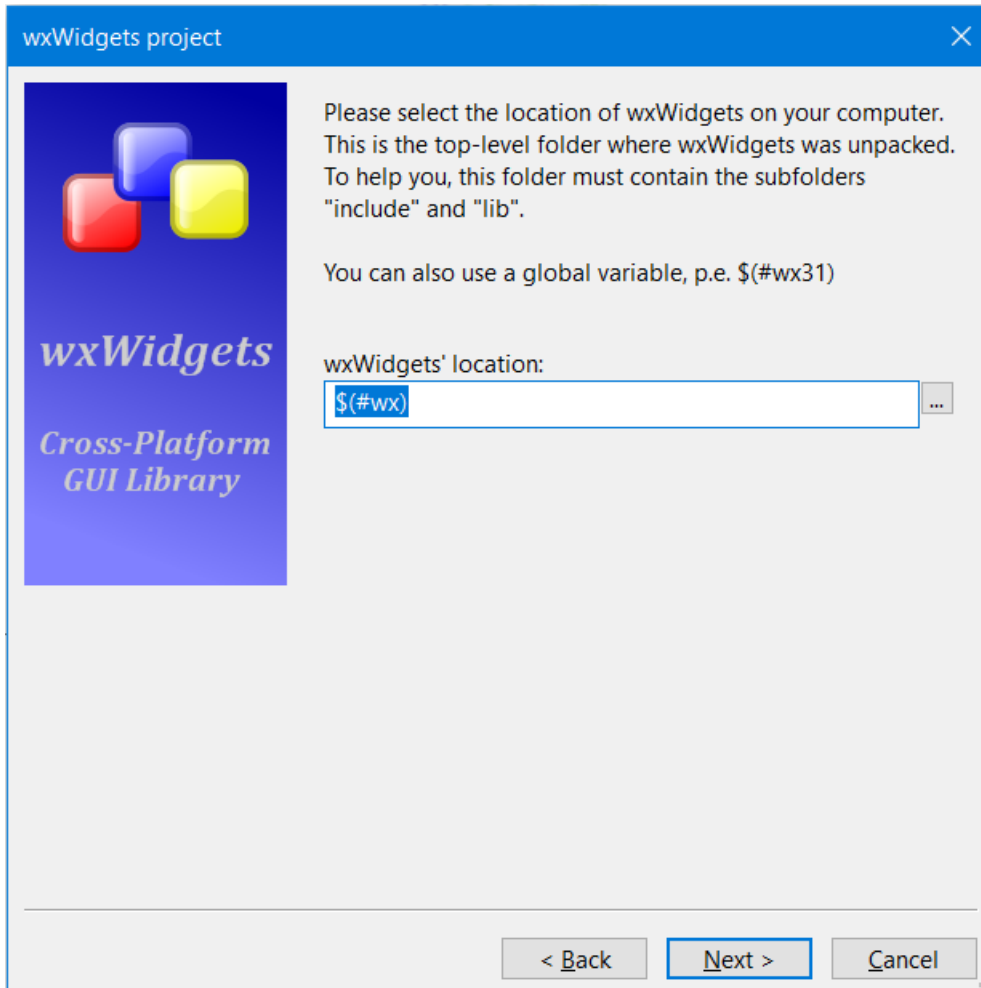
Figure 4-9 Selecting GUI Builder and Application Type



4.5.7 Wizard Page 6: wxWidgets Location

Now we are asked to enter wxWidgets location (Figure 4-10). Instead of using a hard-coded path such as `c:\dev\libs\wxWidgets-3.2`, we will set it to global variable `wx` we created in chapter 4.4. Set **wxWidgets' location** to `$(#wx)` and press **Next**.

Figure 4-10 Setting wxWidgets location



4.5.8 Wizard Page 7: Compiler and Debug/Release Configurations

On this page, make sure that **Compiler** contains **msys2-mingw-w64-ucrt-x86_64** (see chapter 4.3) and that you have checked both **Create “Debug” configuration** and **Create “Release” configuration** (Figure 4-11). Leave the rest of fields at their defaults and press **Next**.

Figure 4-11 Selecting Compiler and setting Debug/Release configurations

wxWidgets project

Please select the compiler to use and which configurations you want enabled in your project.

Compiler: msys2-mingw-w64-ucrt-x86_64

☒ Create "Debug" configuration: Debug

"Debug" options

Output dir.: bin\Debug\

Objects output dir.: obj\Debug\

☒ Create "Release" configuration: Release

"Release" options

Output dir.: bin\Release\

Objects output dir.: obj\Release\

< Back Next > Cancel

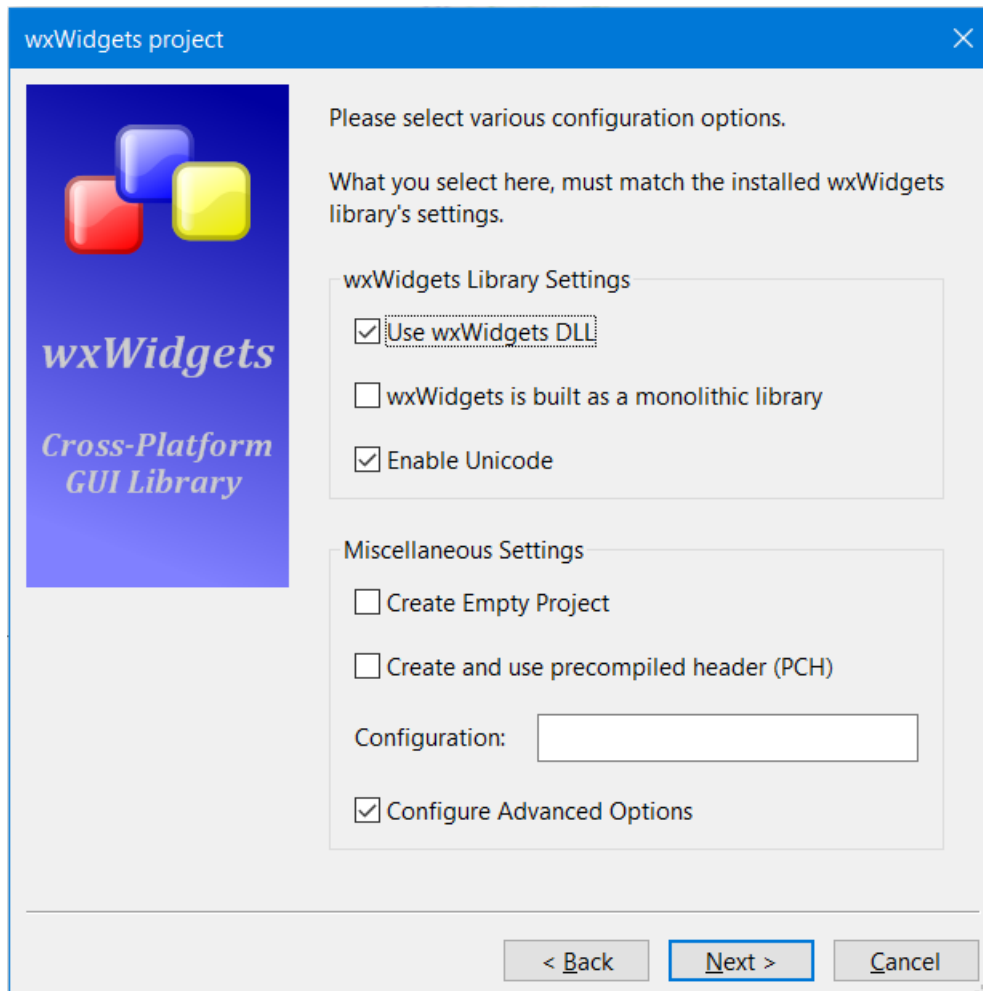
4.5.9 Wizard Page 8: Various Configuration Options

This page lets you set other configuration related options (Figure 4-12), check **Use wxWidgets DLL**, **Enable unicode**, and **Configure Advanced Options**. Make sure that **wxWidgets DLL is built as a monolithic library** is **not** checked (unless you have a monolithic wxWidgets build) and press **Next**.

If you see error message such as “A matching Debug configuration cannot be found”, it means you did something wrong. You either did not build the configurations you asked for (Debug and/or Release), did not build it in requested configuration (e.g., you checked on **Use wxWidgets DLL** here but you built only the static wxWidgets build) or you did not set the `wx` variable so that it points to your wxWidgets folder (see chapter 4.4). If you built wxWidgets using a custom configuration (i.e., with `CFG=SomeString`, see the note in chapter 3.6.8), you need to enter that *SomeString* into **Configuration:**, so that Code::Blocks can find your build in a folder with a non-default name. Please check which it is and correct the issue.

Note. If you want to use the static build of wxWidgets, do not check **Use wxWidgets DLL** (see also chapters 3.6.7 and 4.8.4).

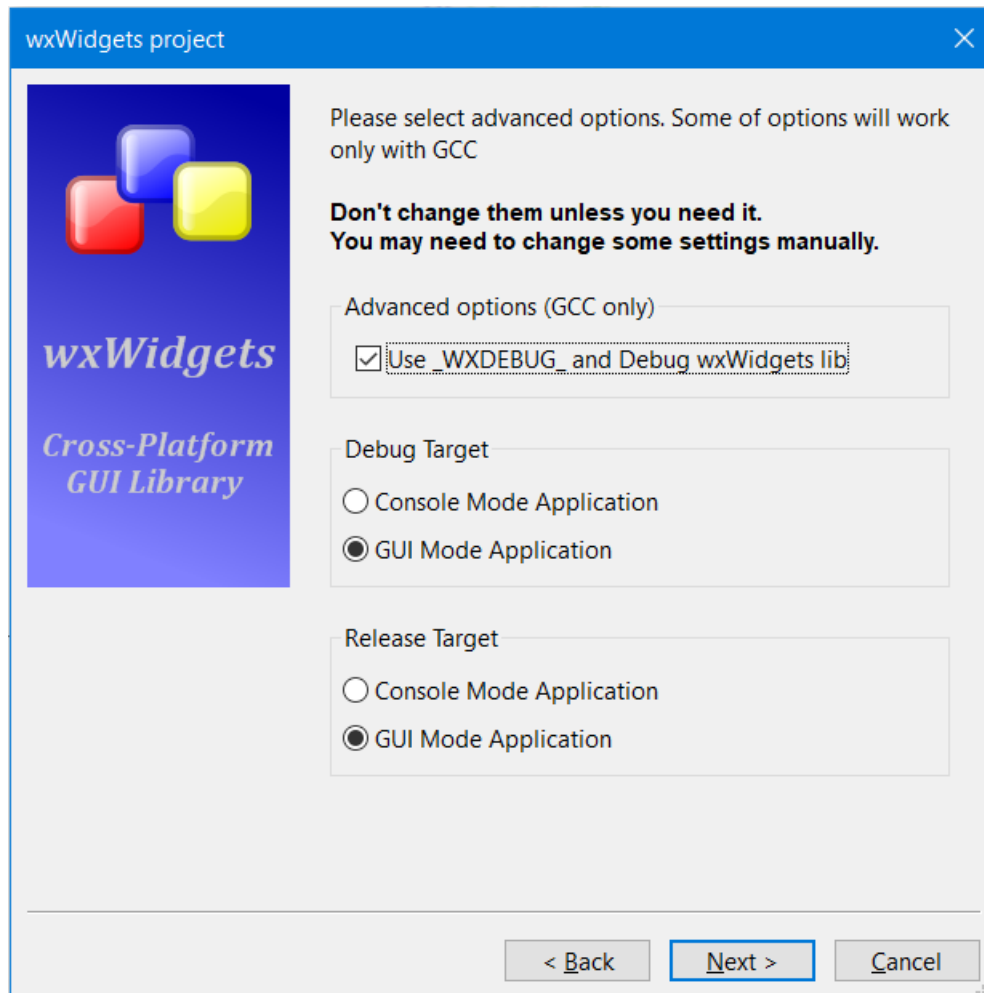
Figure 4-12 Setting various configuration options



4.5.10 Wizard Page 9: Advanced Options

On this page make sure that **Use _WXDEBUG_ and Debug wxWidgets lib** is checked, **GUI Mode Application** for both **Debug Target** and **Release Target** selected (Figure 4-13) and press **Next**.

Figure 4-13 Setting advanced options



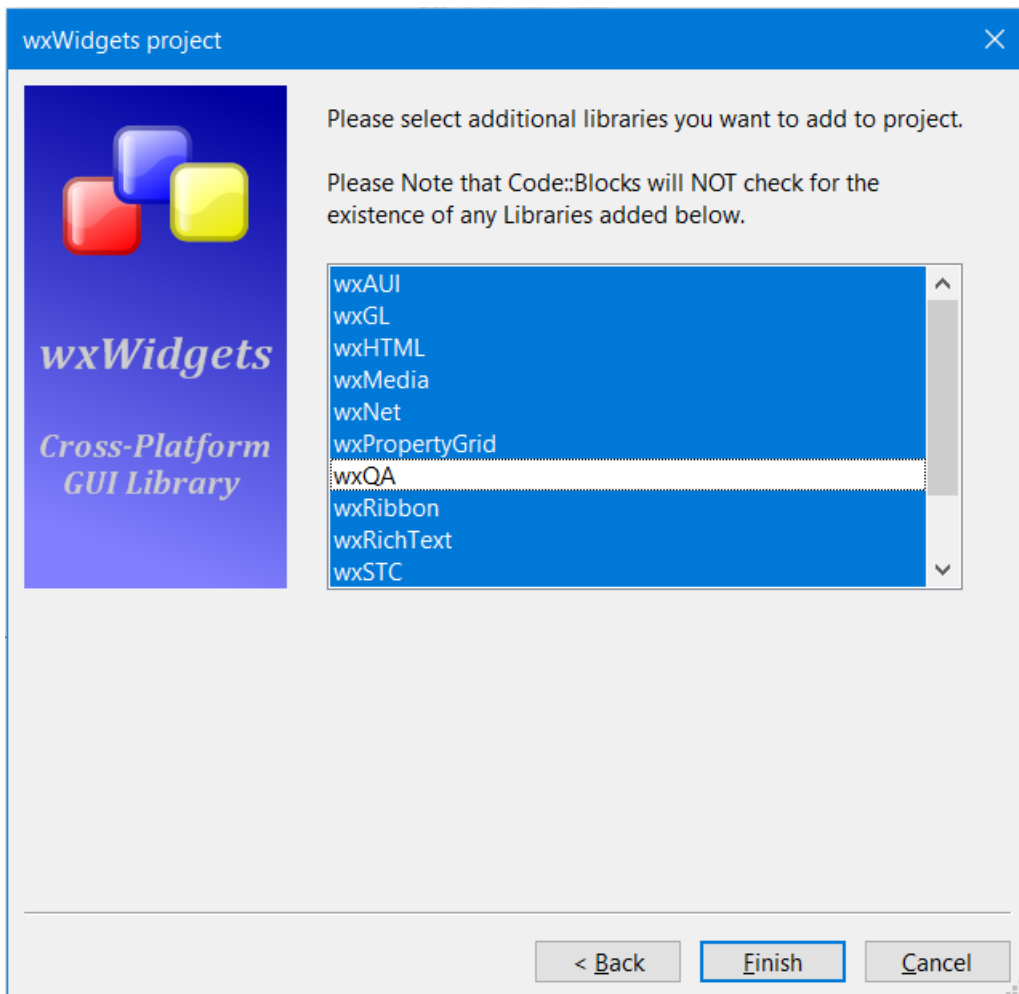
4.5.11 Wizard Page 10: Additional Libraries

We are on the last page of the project wizard and all that is left is to select additional wxWidgets libraries. For simplicity's sake select them all now (Figure 4-14, including those not shown in the figure) except for **wxQA** (which is only for Microsoft Visual C++).

You are probably not going to need at least some of these additional libraries in your actual project. If you want to know more about using wxWidgets libraries in a Code::Blocks project, see chapter 4.8.3.

Press **Finish** to complete the project wizard.

Figure 4-14 Selecting additional libraries



4.6 Adjusting Project Settings

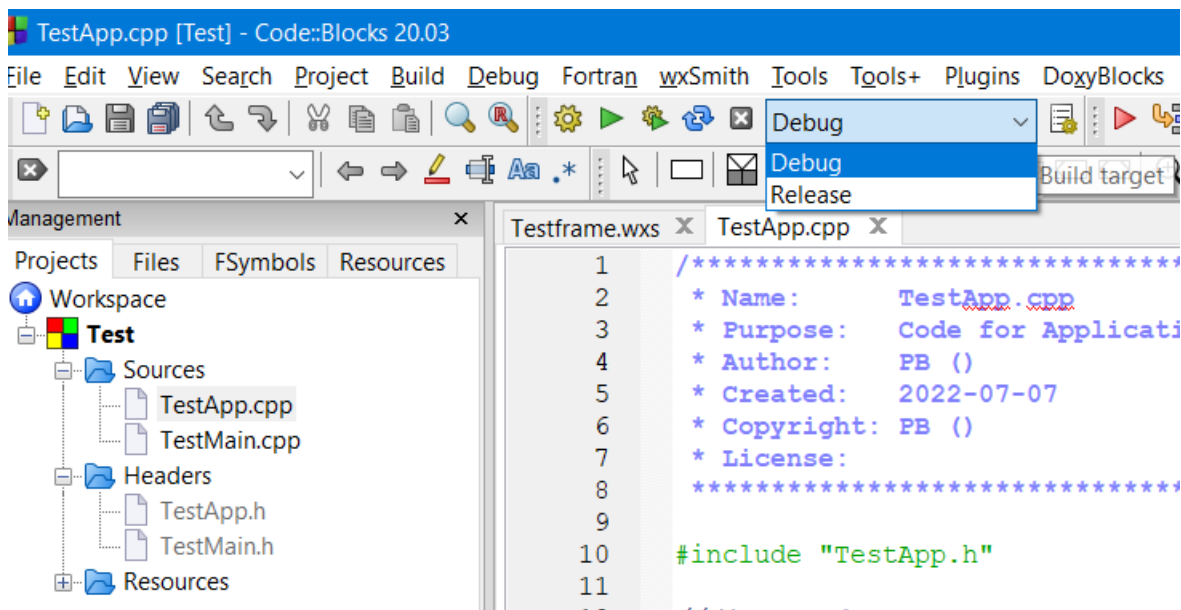
After finishing the project wizard, we now have a basic Code::Blocks project. There is some boilerplate C++ code, basic resource file, the paths for C++ and resource compilers as well as for linker libraries are set, and the required wxWidgets and Windows libraries are added.

Our project was created with two build targets: Debug and Release (see chapter 4.5.8). The Debug target serves for the developer, using debug versions of the libraries. The debug libraries may be slower and take more memory than release ones, but they contain very useful run-time checks that can save enormous amount of time, effort, and frustration when dealing with bugs. The executable produced in the Release target is the one to be distributed to the end users. If you want to know more, see chapter 4.8 for more details about the project settings.

You can switch between the targets using the toolbar (Figure 4-15) or menu **Build / Select target**. The build commands (**Build**, **Run**, **Build and run**, **Start/Continue** ...) apply to the currently selected build target. You will probably mostly use Debug target during development, but when you want to build the Release one, you need to switch to it. Unlike for example Microsoft Visual Studio, Code::Blocks does not have “Batch build” command that allows to build multiple targets at once.

Some project settings are common for all build targets, some are specific for a build target. Before we build our first application, we need to adjust some of them.

Figure 4-15 Switching the build target using toolbar

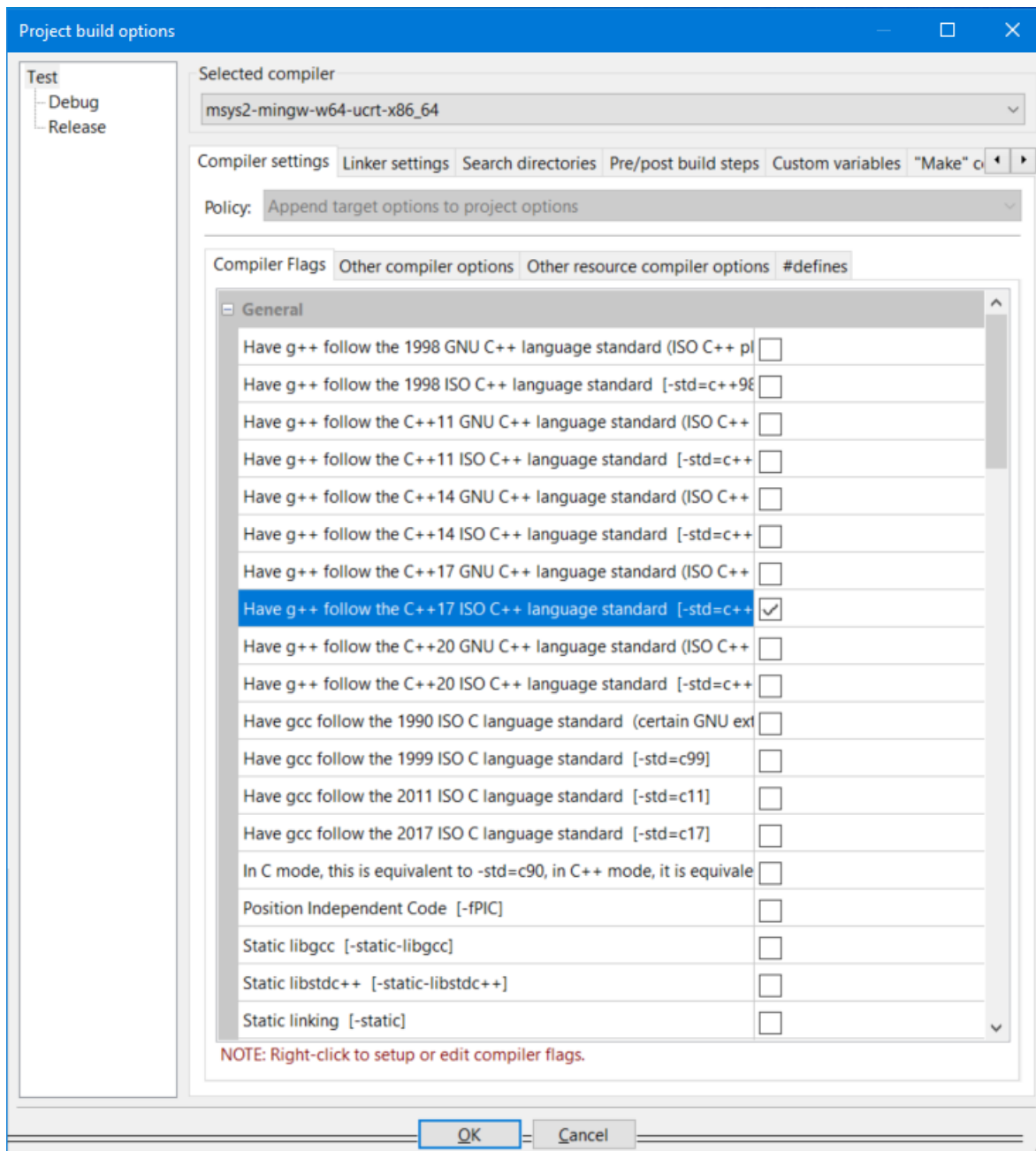


4.6.1 Setting C++ Standard Used in the Project

From the main menu choose **Project / Build Options** which will show the Project build options dialog. Please notice that in the left part of the dialog is a tree control, where the root item is the project name (**Test** in our project) and its children are individual build targets (**Debug** and **Release**). Options set when the root item is selected apply to all build targets, but they are not shown in the individual target settings.

We built wxWidgets with C++17 standard, so we must tell the compiler that our project also uses it. Some people set this in the Global Compiler Settings, but I prefer to keep that setting per project which allows to use different C++ standards between projects, when needed. Make sure that **Test** is selected, go to **Compiler settings** tab, there to **Compiler flags** tab and check **Have g++ follow the C++17 ISO C++ language standard (-std=c++17)**, see Figure 4-16. Press the **OK** button to close the dialog.

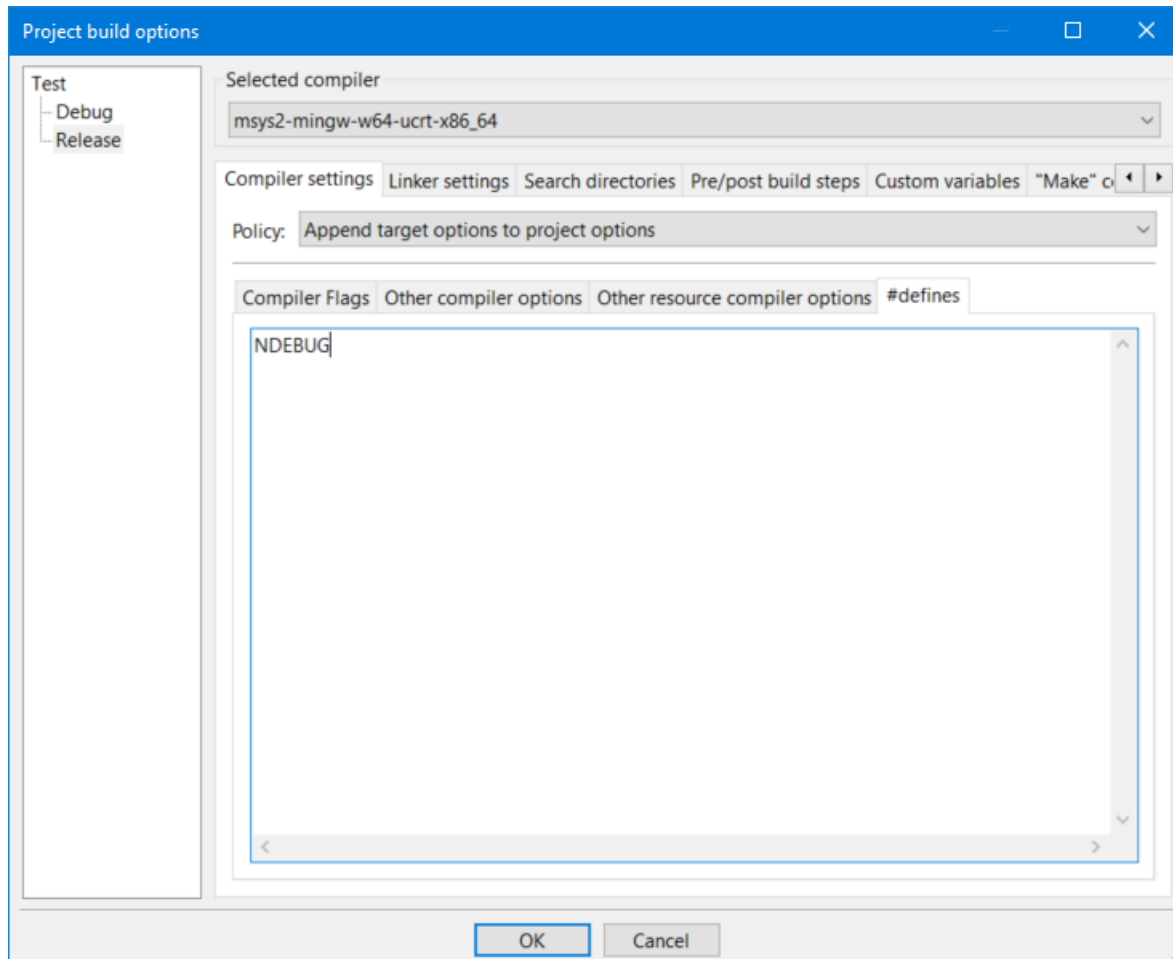
Figure 4-16 Setting C++ standard for all build targets



4.6.2 Setting Preprocessor Define *NDEBUG* for the Release Build Target

Now we want to set *NDEBUG* preprocessor define in the Release target to disable showing possible assert messages to the end user. From the main menu choose **Project / Build Options** which will show the Project Build options dialog. In the top left part of the dialog select the **Release** target. Go to **Compiler settings** tab, there to **#defines** tab and type *NDEBUG* there, see Figure 4-17. Press the **OK** button to close the dialog.

Figure 4-17 Setting preprocessor define *NDEBUG* for the Release build target



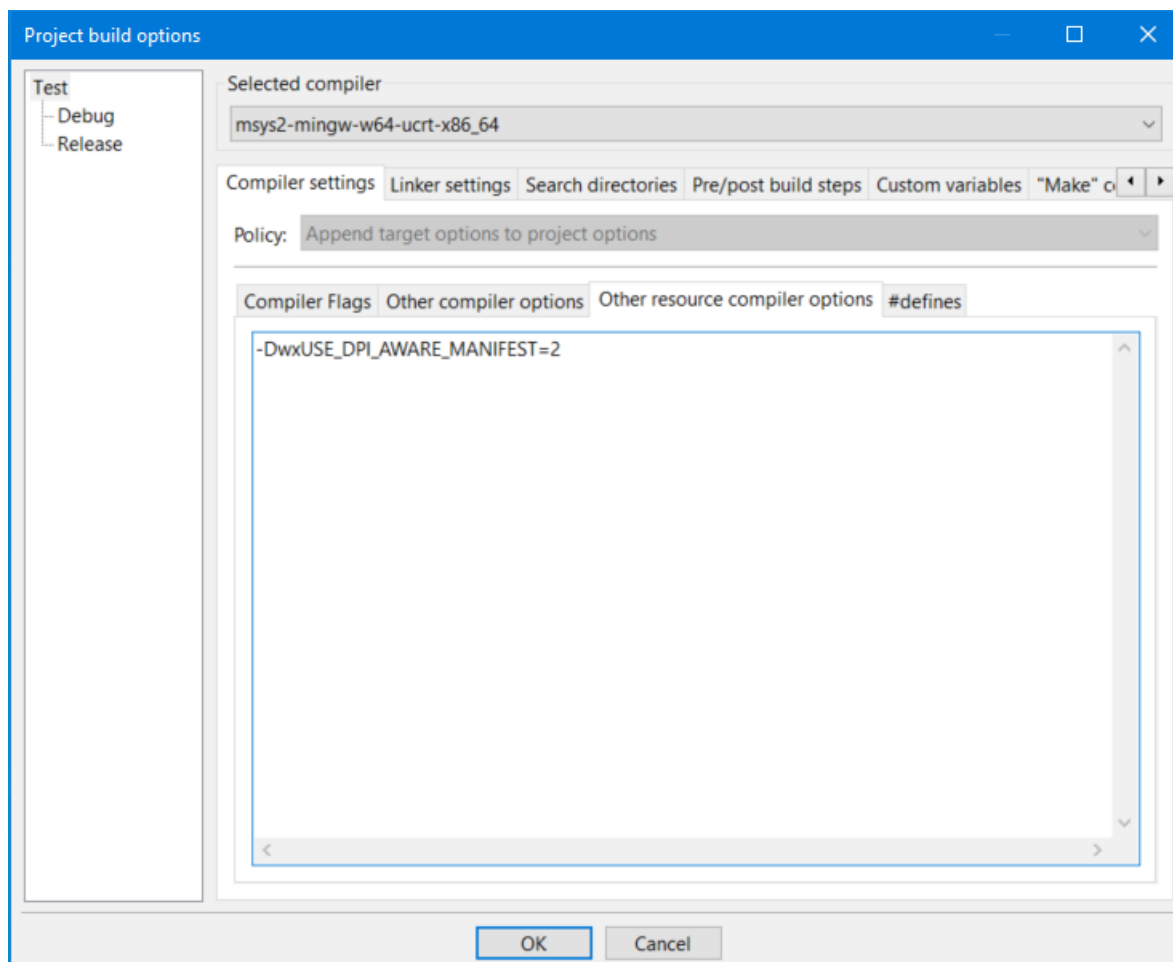
4.6.3 Marking the Application DPI-Aware

High DPI monitors are widely used, so every application is expected to work properly with non-default display scaling. Discussing DPI-awareness is beyond the scope of this guide, if you need more information you should read the [Microsoft Windows High DPI Guide](#) and [wxWidgets High DPI Overview](#).

In this guide, we will use the simplest way to mark our application DPI-aware, using the manifest options provided in wxWidgets resource file, by setting a value of `wxUSE_DPI_AWARE_MANIFEST` define. If we set the value to `1`, *System DPI Awareness* is used; if to `2`, *Per Monitor V2 DPI Awareness* is used. Please be aware that some wxWidgets controls may still have issues with per monitor awareness.

It seems that Code::Blocks does not pass defines set in compiler settings to the resource compiler, so we need to modify the resource compiler options directly. From the main menu choose **Project / Build Options** and in the top left part of the Project Build options dialog select the root item (**Test** in our project), as we want the option to apply to all build targets. Now go to **Compiler settings** tab, there to **Other resource compiler options** tab and type `-DwxUSE_DPI_AWARE_MANIFEST=2` there, see Figure 4-18. Press the **OK** button to close the dialog.

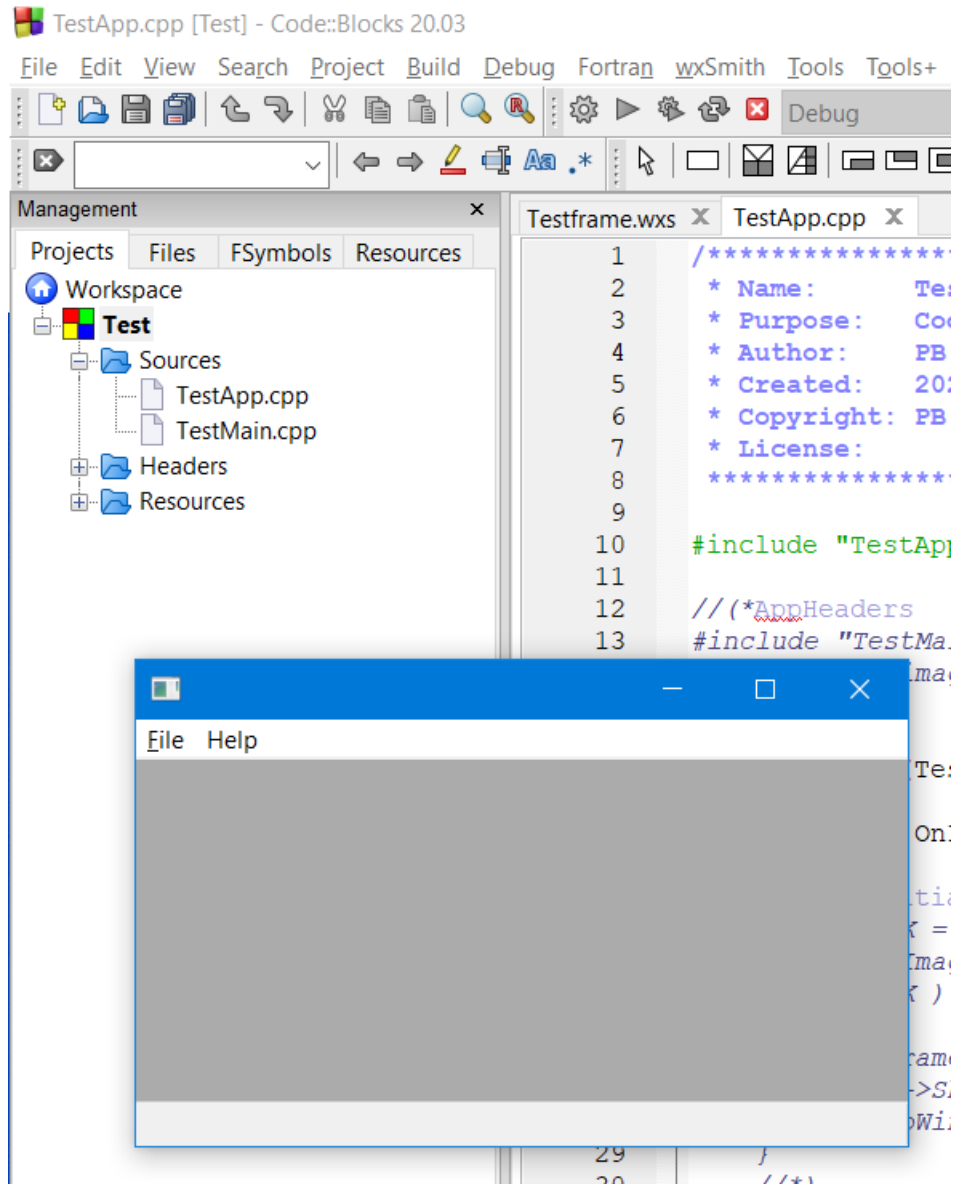
Figure 4-18 Setting resource compiler options



4.7 Finally Finished!

We are done: From the main menu choose **Build / Build and run** (or just press <F9>) to build and run our test application. If you followed this guide carefully, you should see the build succeed and the test application launched (Figure 4-19).

Figure 4-19 Main window of the running test application shown on top of Code::Blocks



You may ask how it is possible that the application is running: We used the dynamic linking but did not copy any DLLs anywhere? It is because Code::Blocks is helpful and when it runs the application it adds the compiler path and the linker libraries paths to the application's executable environment, so it can find its DLLs. However, if you want to run the application outside Code::Blocks, you still need to copy the required DLLs to the same folder where the executable is (see chapter 3.6.5, and also chapters 3.6.7 and 4.8.4).

4.8 Extra Reading: wxWidgets Code::Blocks Project Under the Hood

When creating a project with template wizard (see chapter 4.5), Code::Blocks will set all necessary project settings according to our choices. However, sometimes it is good or even necessary to know what setting they are. When using a compiled library such as wxWidgets, in a project you usually need to set at least:

1. compiler include folder(s),
2. preprocessor defines,
3. linker settings: names of libraries and where to find them.

The following chapters describe how to set wxWidgets related information in a Code::Blocks project and what are the differences in settings needed when using a static build of wxWidgets. When referring to wxWidgets root folder, we will use Code::Blocks variable `wx` we created in chapter 4.4 and used in the wizard in chapter 4.5. In other words, when `$(#wx)` is used, it translates to system environment variable `WXWIN` we created in chapter 3.5, which in turn points to the actual folder, such as `c:\dev\libs\wxWidgets-3.2`.

4.8.1 Compiler Include Folders

wxWidgets uses two include folders. One is the main and is set for the whole project, it should be `$(#wx)/include`. The other is build-specific and must be set differently for each build target, should be set to `$(#wx)/lib/gcc_dll/mswud` for the Debug build target and to `$(#wx)/lib/gcc_dll/mswu` for the Release build target. These include folders must be set for the C++ compiler as well as the resource compiler.

Note. The build-specific include folder not being set for the C++ compiler is the cause of a common build error with error message such as “`wx/platform.h:136:22: fatal error: wx/setup.h: No such file or directory`”.

4.8.2 Preprocessor Defines

The only necessary project-wide defines for C++ compiler are `__WXMSW__` and `wxUSE_UNICODE`. If you are using, as default in this guide, a shared (DLL) build of wxWidgets, `WXUSINGDLL` must be defined as well.

I also strongly recommend defining `NDEBUG` for the Release build, see chapter 4.6.2. If you did not build wxWidgets with `wxUSE_UNSAFE_WXSTRING_CONV` set to 1 (see chapter 3.6.8), you may consider defining `wxNO_UNSAFE_WXSTRING_CONV`. On a similar note, you may also look into using `wxNO_IMPLICIT_WXSTRING_ENCODING`, both these `wxString`-related defines are described in [the documentation](#) and should be set for the whole project.

4.8.3 Linker Settings

The project wizard should have set the linker folder and added necessary libraries. However, you may sometime need to change some of these settings. Once again, we assume that you did not add set library related stuff (e.g., include or linker folders) in the global compiler settings, and all of this is set only at the project level.

The folder with libraries is set in **Project / Build Options** in **Search directories** tab in **Linker**, where there should be `$(#wx)/lib/gcc_dll` for the shared build; and is same for both Debug and Release build target.

The individual libraries are added in **Project / Build Options** in **Linker Settings** tab in **Link libraries**. wxWidgets libraries need to be set separately for the Debug and Release build target. Debug versions of wxWidgets libraries have an extra `d` in their name, see chapter 3.6.4 where the file naming pattern for wxWidgets libraries is explained.

Table 4-1 lists all wxWidgets libraries (built as the default multilib) for the Debug and Release build targets, in the order they must be listed (see the Note below).

Table 4-1 List of wxWidgets libraries

<i>For Debug Build Target</i>	<i>For Release Build Target</i>	<i>Purpose</i>
libwxmsw32ud_webview.a	libwxmsw32u_webview.a	wxWebView and related
libwxmsw32ud_stc.a	libwxmsw32u_stc.a	wxStyledTextCtrl
libwxmsw32ud_propgrid.a	libwxmsw32u_propgrid.a	wxPropertyGrid
libwxmsw32ud_ribbon.a	libwxmsw32u_ribbon.a	wxRibbonBar
libwxmsw32ud_richtext.a	libwxmsw32u_richtext.a	wxRichTextCtrl
libwxmsw32ud_xrc.a	libwxmsw32u_xrc.a	wxXmlResource
libwxmsw32ud_aui.a	libwxmsw32u_aui.a	Aui classes
libwxmsw32ud_media.a	libwxmsw32u_media.a	wxMediaCtrl
libwxbase32ud_net.a	libwxbase32u_net.a	network-related classes
libwxmsw32ud_gl.a	libwxmsw32u_gl.a	wxGLCanvas and wxGLContext
libwxbase32ud_xml.a	libwxbase32u_xml.a	wxXmlDocument
libwxmsw32ud_html.a	libwxmsw32u_html.a	HTML classes
libwxmsw32ud_core.a	libwxmsw32u_core.a	contains most of GUI and drawing classes
libwxbase32ud.a	libwxbase32u.a	contains the most basic classes such as wxString
libwxpngd.a ¹	libwxpng.a ¹	PNG image
libwxjpegd.a ¹	libwxjpeg.a ¹	JPEG image
libwxtiffd.a ¹	libwxtiff.a ¹	TIFF image
libwxzlibd.a ¹	libwxzlib.a ¹	zip compression
libwxregexud.a	libwxregexu.a	regular expressions
libwxexpatd.a	libwxexpat.a	XML parsing
libwxscintillad.a ²	libwxscintilla.a ²	Scintilla (needed by the stc library)

¹These libraries are needed for an application using static build of wxWidgets.

²This library is needed for an application using *wxStyledTextCtrl* and static build of wxWidgets.

You can remove libraries you do not use; many applications will not use at least some of them. The libraries in bold are the most basic ones even the simplest GUI application must link with. See [wxWidgets library list](#) to see which classes require which libraries and for inter-library dependencies.

Note. Please note that with GCC (unlike with MSVC), the library order matters. When *libraryA* requires *libraryB*, *libraryA* must be listed **before** *libraryB*. For example, *libwxpng.a* must be listed before *libwxzlib.a* on which it depends.

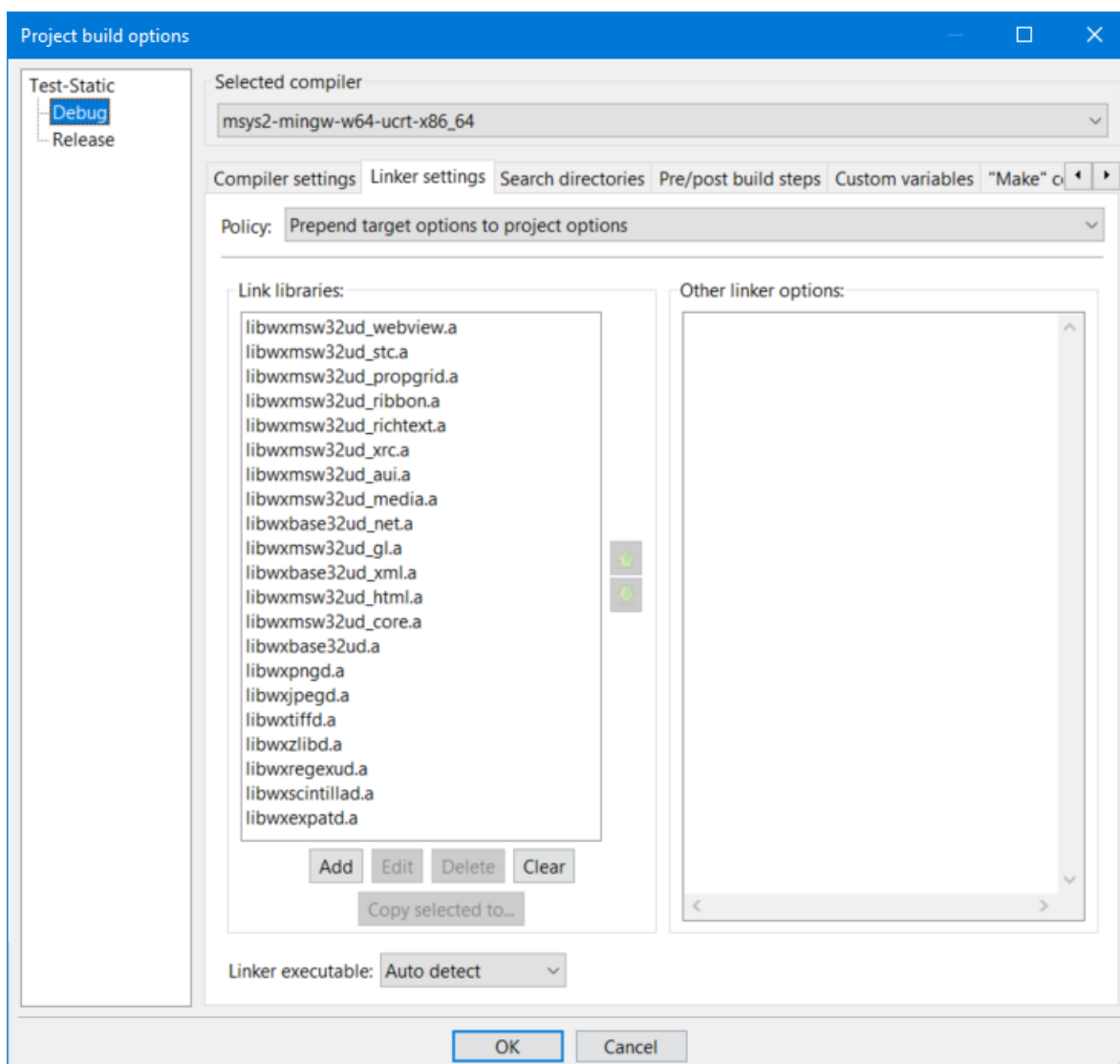
4.8.4 Using Static Instead of Shared wxWidgets Build

There are several key differences when using the static wxWidgets build compared to the shared one. For starters, you must build wxWidgets statically, see chapter 3.6.7. If you linked compiler libraries statically as well, as described in that chapter, you need to check **Static linking [-static]** in **Compiler flags** tab in **Compiler settings** tab (it is the last compiler flag visible in Figure 4-16).

In the names of include and linker folders, wherever there is `gcc_dll` for the shared build, there must be `gcc_lib` for the static build. In **Project / Build Options** do **not** define `WXUSINGDLL` in **Compiler settings** tab in **#defines**.

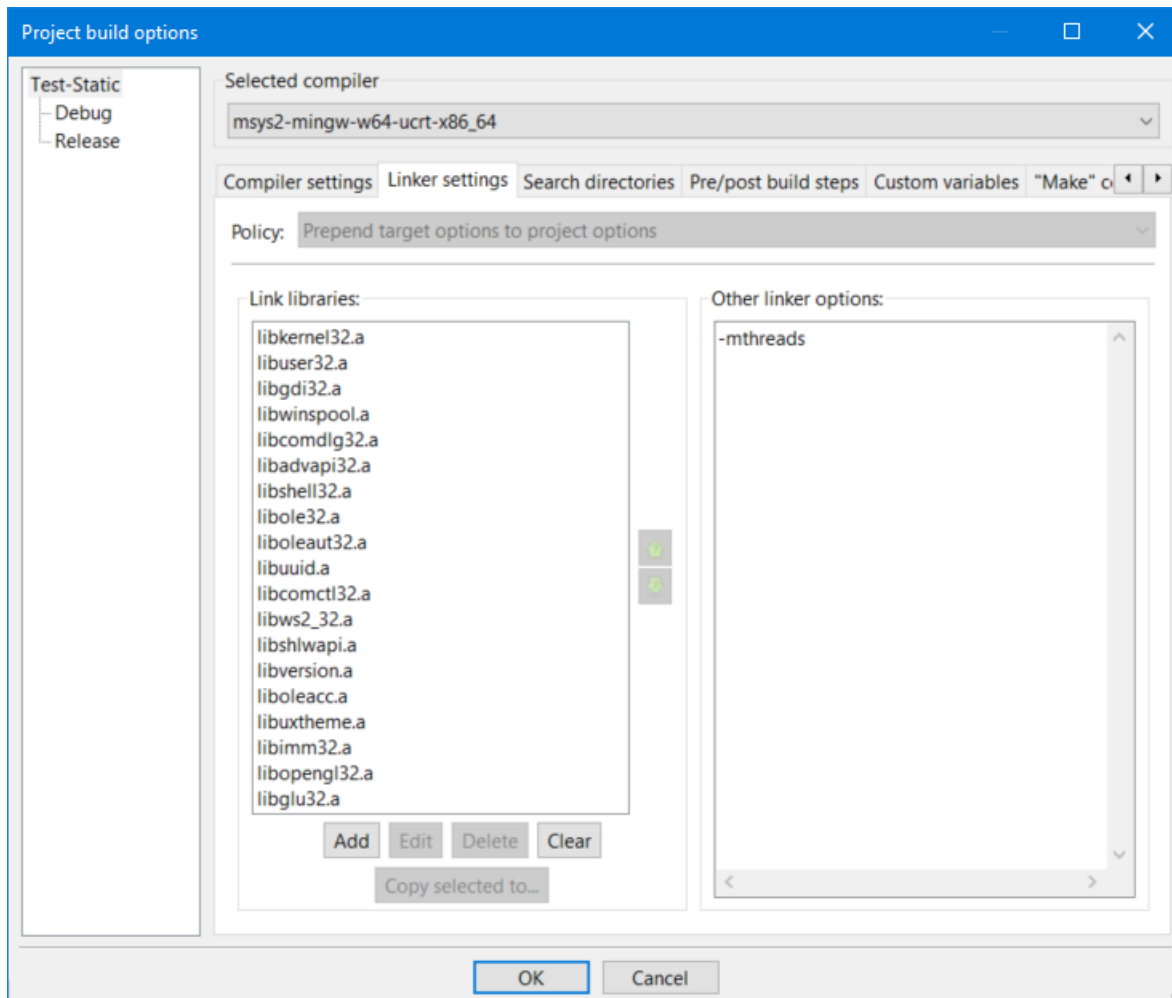
You will probably need to add more wxWidgets libraries to the linker settings, see Table 4-1 for the libraries list and Figure 4-20 for the actual linker settings for the Debug target containing basically all wxWidgets libraries. As already stated, please keep in mind that the library order matters.

Figure 4-20 List of all wxWidgets libraries which may be needed for the Debug build target when linking to wxWidgets statically



You will also need to add MS Windows import libraries on which wxWidgets depends, see Figure 4-21 which lists all Windows libraries wxWidgets may need. These libraries are not added to Debug and Release targets separately, they are added for the whole project, in other words, to the root item of the build targets (**Test-Static** in the figure).

Figure 4-21 List of all MS Windows libraries which may be needed when linking to wxWidgets statically



If your project is simple enough, instead of making all these changes, you may be better off just using the wizard to create a new project or build target where you do not check **Use wxWidgets DLL** (see chapter 4.5.9).

Document History

Version 1.0, released June 4, 2020

Initial release

Version 1.1, released June 7, 2020

- Add information that wxAdvanced library is no longer needed.
- Improve formatting and wording.

Version 1.2, released June 22, 2020

- Add missing “\lib” in the include and linker paths in chapters 4.7.1 and 4.7.3.
- Remove incorrect statement about the pre-built binaries being available only in the Release configuration.
- Suggest that using the shared wxWidgets build is preferable to using the static one.
- Make minor improvements in writing style and text formatting.

Version 1.3, released June 30, 2020

- Correct Figure 4.16 where a preprocessor define was wrongly entered in the “Other compiler options” field.
- Add a note about not setting WXWIN system environment variable.
- Mention that mingw-w64 is used to build the official MS Windows binaries.
- In chapter 3.5 link to the bat files for building wxWidgets available in the GIT repository.
- Fix typos and improve wording.

Version 2.0, released August 13, 2020

- Update for wxWidgets 3.1.4.
- Switch compiler toolchain from project mingw-w64 to MSYS2.
- Include information about setting up the debugger in Code::Blocks.
- Minor text improvements.

Version 2.1, released August 14, 2020

- Fix invalid hyperlinks to some chapters and TOC formatting.
- In chapter 2.3 fix the mistake about monolithic build being used in this guide.
- Update the information about dependency of the base library on the regex one.

Version 2.2, released August 26, 2020

- Add more information about which wxWidgets libraries are needed for a project in chapter 4.8.3.
- State that this guide is not a reference manual but a tutorial for new users.

Version 2.3, released September 8, 2020

- Make some language corrections.
- Add information about using a custom configuration with Code::Blocks’ wxWidgets project wizard.
- Clarify that the build commands are for Windows command prompt, not MSYS environment.

- Move chapter Making wxWidgets Build Faster before chapter Viewing Build Results.

Version 2.4, released September 22, 2020

- In chapter Viewing Build Results refer to the listing using the parallel build.
- Add a note regarding lack of Code::Blocks wxWidgets project wizard support for a custom compiler prefix for a wxWidgets build.
- Make some language corrections.

Version 2.5, released October 28, 2020

- Add and discuss SHELL build parameter.
- Show how to mark an application as DPI-aware.
- Move the regex library into a correct place in the library order.
- Mention that the default GCC 10 C++ dialect is *gnu++14*.
- Fix a typo in chapter 2.1 (diverted -> diverged).
- Expand the note in chapter 3.3.
- Modify the build failed message in the batch files.
- Improve wording.
- Remove mention of Windows 10X.

Version 2.6, released November 21, 2020

- Mention that one may need to add more libraries to the linker settings in Code::Blocks project when using the static wxWidgets build compared to the dynamic one.
- Expand the chapter about cleaning builds.
- Expand the chapter about libraries naming scheme.
- Further clarify information about which wxWidgets libraries are needed for a project.
- Fix typos, improve wording and formatting.

Version 2.7, released December 17, 2020

- Fix page heading for chapter 4.
- Fix typos (most importantly “making” -> “marking” in DPI awareness chapter), improve wording and formatting.
- Add information about application Dependencies providing information about DLLs an executable links with.

Version 3.0, released May 5, 2021

- Upgrade for wxWidgets 3.1.5
- Expand the chapter about wxWidgets build options.
- Significantly expand the chapter about customizing a wxWidgets Code::Blocks project, in particular chapter concerning the static wxWidgets build.
- Mention that not all wxWidgets libraries are available in the list offered by the project wizard.
- Improve chapter Cleaning the Build.
- Mention GDB pretty printer for wxWidgets types.
- Fix typos and improve wording.

Version 3.1, released June 29, 2021

- Mention the compiler package using the universal CRT.
- Add information about the current impossibility of monolithic build with wxWebViewEdge.
- Provide updated wxWidgets project wizard.
- Minor text and formatting changes.

Version 3.2, released June 29, 2021

- Fix the GitHub link to the updated wizard.
- Change Code::Blocks version mentioned in the updated wizard information from incorrect 20.04 to 20.03.

Version 4.0, released April 20, 2022

- Upgrade for wxWidgets 3.1.6.
- Switch compiler from 32- to 64-bit.
- Use the updated project wizard instead of one bundled with Code::Blocks.
- Mention mingw64-based alternatives to MSYS2 mingw-w64 toolchain.
- Various other minor changes.

Version 4.1, released July 7, 2022

- Update the guide for wxWidgets 3.2.
- Update Code::Blocks wxWidgets project wizard bundled with the guide for wxWidgets 3.2 and 3.3.
- Remove information about the impossibility of monolithic build with wxWebViewEdge (fixed in 3.1.7).
- Mention a way to view messages output by wxLog{Debug|Trace}() when debugging an application.
- Minor changes in format and wording.

Version 4.2, released December 27, 2022

- Update Code::Blocks wxWidgets project wizard bundled with the guide: Remove support for wxWidgets 3.0 which simplifies it (no wxAdvanced library, always link to Winsock2) and when linking with wxGL, always link also with opengl32 and glu32 libraries.
- Update for wxWidgets 3.2.1 and GCC 12.2 (no real changes).

Version 4.3, released July 7, 2023

- Switch the compiler toolchain from the MSVCRT- to UCRT-based one.
- Update for wxWidgets 3.2.2.1 (no real changes).
- Mention the main reason for possibly using the GIT master instead of the latest stable version.
- Mention the existence of nightly builds of Code::Blocks.
- Minor changes in format and wording.

Version 4.4, released July 23, 2023

- Correct Figure 4-17 (NDEBUG was input in a wrong tab).
- Minor changes in wording.

Version 4.5, released August 21, 2023

- Revise the part about wxWidgets versions.
- Add a bit more information about Debug and Release configurations.
- Fix MSYS2 command for listing packages in a group (-S -> -Sg).
- Fix an incorrect word in chapter 3.6.4 (static -> dynamic).
- Link directly to Make Parameters in chapter 3.6.8.
- Upgrade for GCC 13.2 (no real change).
- Improve wording.

Version 4.6, released September 21, 2023

- Update chapter To Build or Not to Build.