

COMPILING MIDIEDITOR CUSTOM ON WINDOWS

by F.MUÑOZ “ESTWALD”

About MidiEditor Custom

MidiEditor Custom is not just an unofficial version of Marcus Schwenk’s MidiEditor.

Some time ago, enhancements were added such as support for the FluidSynth library, VST plugin support, drum machine, sequencer, and many other details that go a bit beyond the original, yet the essence of the program remains.

The purpose of this document is to explain the process required to compile the program under the Qt development environment and Windows operating system — originally compiled with version 5.15.2 — in order to port it to the current 6.9.0 version, adding the FluidSynth library in the process instead of using the precompiled official one.

ABI, Shared and Static

The Application Binary Interface (ABI) defines how code modules interact at the binary level. A different ABI can prevent a library from being recognized, and compiling code with different compilers can also lead to program instability.

Libraries can be either shared or static. A shared library can save storage space if used by multiple programs and also allows it to be updated or patched without recompiling the programs, but:

- 1) If there are many shared libraries and they’re used only by one program, the space savings aren't real, and there are issues like having to include them in the distribution, with copyright concerns, etc.
- 2) If libraries from different builds are mixed, they may share common libraries with different function symbols... and that causes errors!
- 3) A static build allows an executable with everything embedded.

Up to now, MidiEditor Custom used statically compiled libraries (namely Qt 5.15.2), which also helped avoid DLL conflicts with FluidSynth. However, at least in the 32-bit version, some stability issues were observed due to differences in the compilers used.

Qt 6.9.0

To download the Qt development environment, you need to register on its official page:

<https://login.qt.io/login>

Register as an individual for open source development.

<https://www.qt.io/download-open-source>

<https://www.qt.io/download-dev>

From there you can download the installer, and once launched you'll be able to choose Qt 6.9.0 and the necessary tools to compile with shared libraries. Only the 64-bit version is provided. So if you need 32-bit for compatibility reasons, you'll have to build Qt 6.9.0 yourself for that version.

In the tools section, the provided compilers are MinGW 13.1.0 for 64-bit, with just enough to compile in the Qt environment... but don't expect much more, like compiling FluidSynth sources (which requires libraries and tools that are not provided). If you don't want to use the development environment, you can also download the Qt 6.9.0 source code without registering, and I provide a tool — as you'll see below — that handles the whole process of downloading, configuring, and building.

MSYS2

To the point: what's needed is a Windows environment that supports the necessary tools to compile Qt 6.9.0 as static libraries for both 64-bit and 32-bit, provides compilers and tools for this and for FluidSynth, and already includes precompiled static libraries to avoid having to compile too many sources (we'll only need to compile libinstpatch). MSYS2 meets all these requirements:

<https://www.msys2.org/>

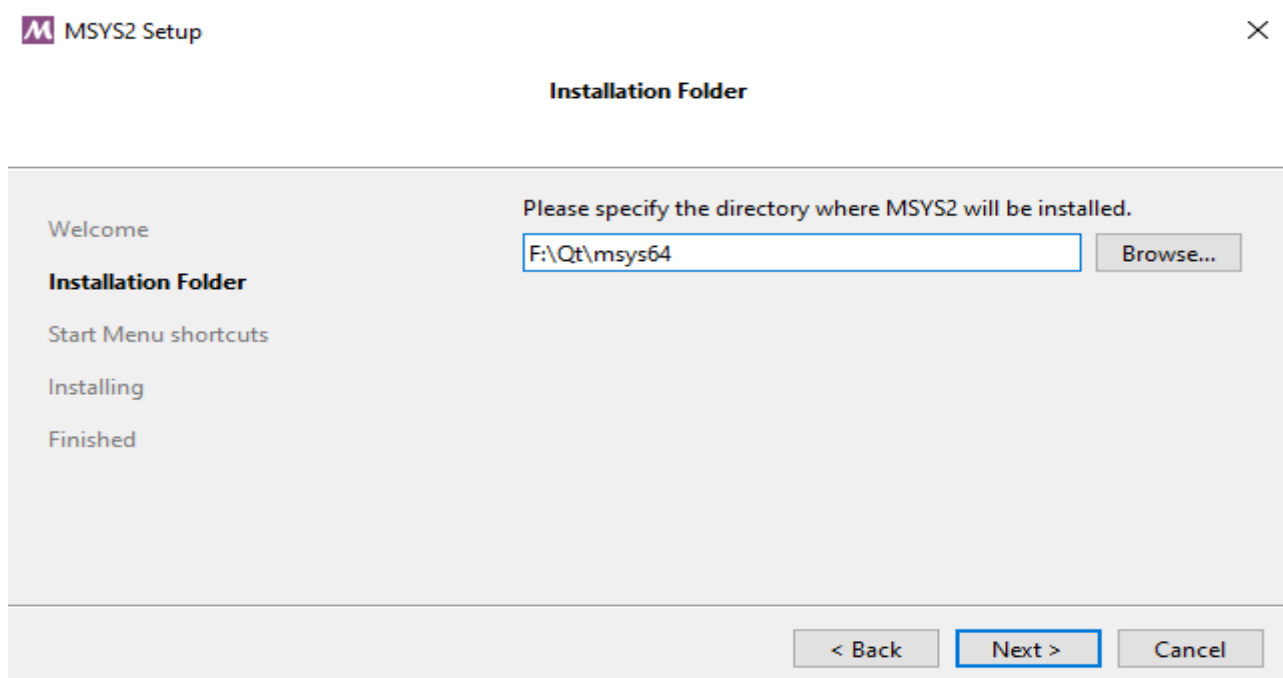
Steps:

First of all, note that you'll need about 60 GB of free disk space. The compiled Qt 6.9.0 code alone (without QtWebEngine) takes up a whopping 34 GB... and we're talking about thousands of small files, so I recommend using an SSD. Once the libraries are compiled, you can delete the built object files and also the extracted source code (the compressed tar.gz version takes up just over 1 GB, not too bad).

- Download the installer from here:

https://github.com/msys2/msys2-installer/releases/download/2025-02-21/msys2-x86_64-20250221.exe

- Launch it and install in the appropriate Qt directory. For example, in my case: F:\Qt\msys64:



- We do this because this MSYS2 installation will be used exclusively with Qt. If you need MSYS2 for something else, create another installation, because we'll customize this one to avoid errors when compiling what we need for MidiEditor Custom.

- Once installed, it is necessary to set the environment variable **QTENV** like this: **QTENV=F:/Qt** (because in my case it is on drive F:. And it uses the forward slash /, remember).

Look up how to add environment variables in your Windows system if you don't know how to do it.

CONFIGURE MINGW32/64

The instructions work the same for both versions, as the tools I provide take care of detecting the bash from which they are launched.

- Go to the MSYS2 installation directory (in my case F:\Qt\msys64) using the file explorer.

There you will see the bash launchers mingw32.exe and mingw64.exe. Do not enter any other bash! So, let's launch mingw64.exe to create the necessary environment for 64 bits.

Well, you will see a screen like this:



We type:

\$ pacman -Syu

It will update some things; if it asks, answer 'Y' and most likely, when it finishes, the bash will close. If that happens, launch it again and run:

\$ pacman -Su

- From the file explorer, go to the directory F:\Qt\msys64\home<user> (in my case F:\Qt\msys64\home\paco_)
- Copy inside the files msystool.sh and qtmsys.sh which you will find in the Midieditor Custom sources in the directory midieditor\building\MSYS2 Tools
- Now, in the MinGW64 bash, do:

\$ **dir**

We should see them:

```
paco_@DESKTOP-NL4R0CR MINGW64 ~  
$ dir  
msystool.sh  qtmsys.sh  
  
paco_@DESKTOP-NL4R0CR MINGW64 ~$
```

msystool.sh

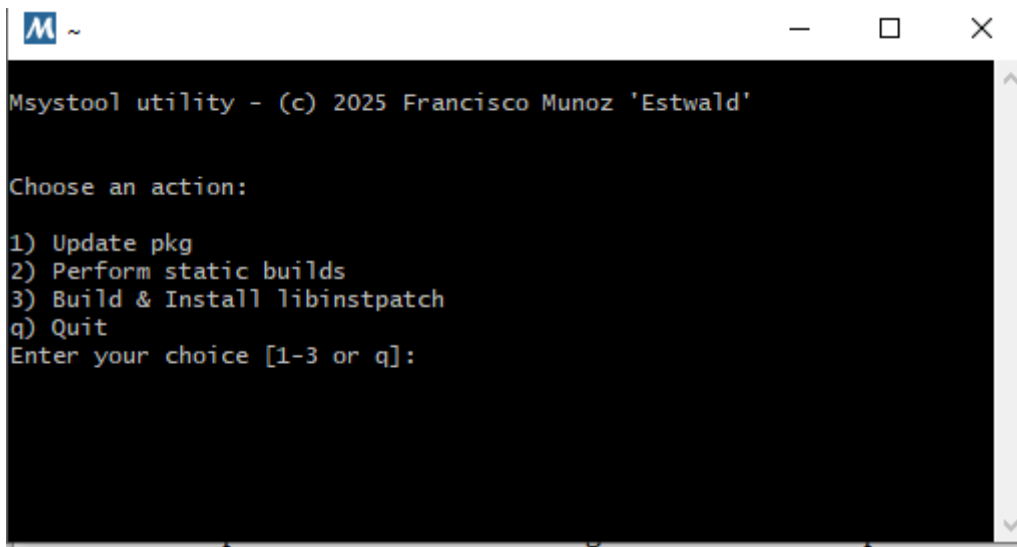
This tool consists of several phases in which it will update MinGW64 (because it is the bash we launched) with the compilers, tools, and libraries necessary to compile both Qt 6.9.0 and Fluidsynth with all the components we need. It will also apply patches and move some things to ensure that using cmake in different configurations does not end up compiling in shared mode instead of static: shared is taken by default for any minor detail.

NOTE: Before doing anything, remember it is necessary to set the environment variable QTENV like this: QTENV=F:/Qt

- We launch the tool as follows:

\$ **./msystool.sh**

and it will show us a screen like this:



You must choose all options from 1 to 3. Not all phases are automated in case unexpected errors arise or if you want to perform one of the subparts separately. It is not recommended to run each option more than once.

1) Update pkg

All necessary packages are installed (downloaded from the internet, so obviously an internet connection is required), including compilers, utilities, and necessary libraries. **Once this step is**

completed, it should not be done again: remember that any update could cause the tool to not work properly.

2) Perform static builds

The first thing done here is cloning the directory **/mingw64/lib** into **/mingw64/lib-orig** to have an unaltered copy of that directory (if needed, simply renaming the folders will give us the unaltered libraries).

Then, all shared libraries, meaning those with **.dll.a**, are located and moved to the folder **/mingw64/lib/sharedlibs**. This allows us to link them manually if needed.

After that, the directory **/mingw64/lib/pkgconfig-static** is created, where the **.pc** files of libraries like **glib-2.0** or **sndfile**, required by **Fluidsynth**, will be copied and patched. This directory **/mingw64/lib/pkgconfig-static** will be added so that these patched files are found before those in **/mingw64/lib/pkgconfig**, tricking **cmake**.

3) Build & Install libinstpatch

As its name indicates, first the source code of **libinstpatch** is downloaded, then configured, compiled, and installed, proceeding to patch whatever is necessary.

q) Quit

It exits the application.

Now everything is ready to download the sources and compile the **Qt 6.9.0** libraries using the second application: **qtmsys.sh**. Remember that if you need the 32-bit libraries, you must launch the bash **mingw32.exe** and follow the same steps (the only difference is that it applies to **/mingw32** instead of **/mingw64**).

qtmsys.sh

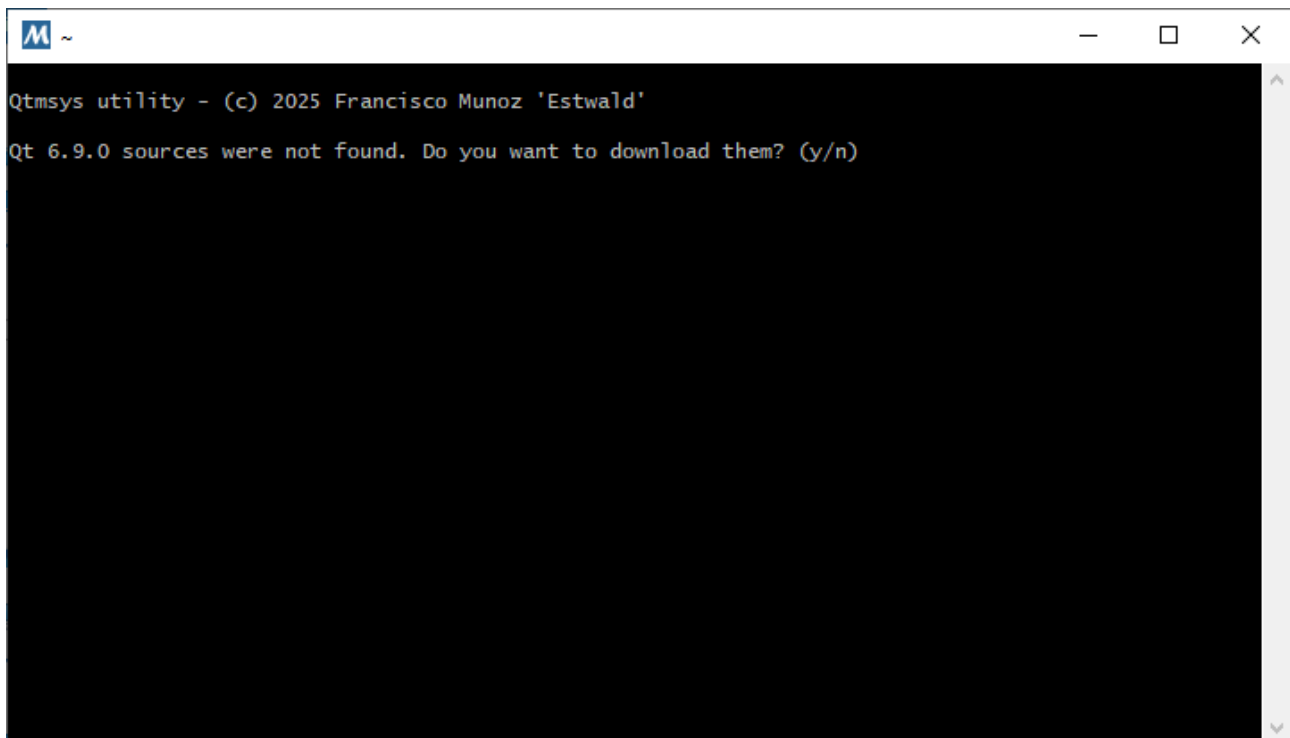
This is the tool that will download the **Qt 6.9.0** source code, patch some things, configure it, compile it, and install it so it can be used to compile **Midieditor Custom**.

First of all, remember that the **QTENV** environment variable must be set. And of course, you must have completed all the steps of **msystool.sh** beforehand.

To launch the application, run the following in the corresponding bash:

```
$ ./qtmsys.sh
```

When doing so, it will check whether the source directory exists (it will be installed in **\$QTENV/qt-everywhere-src-6.9.0**, which in my case would be **F:/Qt/qt-everywhere-src-6.9.0**). If it does not exist, the first thing we'll see is this screen to download:



We select 'y' to download. Once the download is complete, the source will be extracted to a temporary directory to prevent errors. Then, the QtWebEngine source code will be deleted (it takes up a lot of space and requires Microsoft compilers), and a few files will be patched:

qt-everywhere-src-6.9.0/qtquick3d/src/3rdparty/assimp/src/code/Common/DefaultIOStream.cpp:

#define _CRTIMP is added before the **#include** statements to allow static compilation.

qt-everywhere-src-6.9.0/qtquick3d/tools/balsam/CMakeLists.txt

qt-everywhere-src-6.9.0/qtquick3d/tools/balsamui/CMakeLists.txt

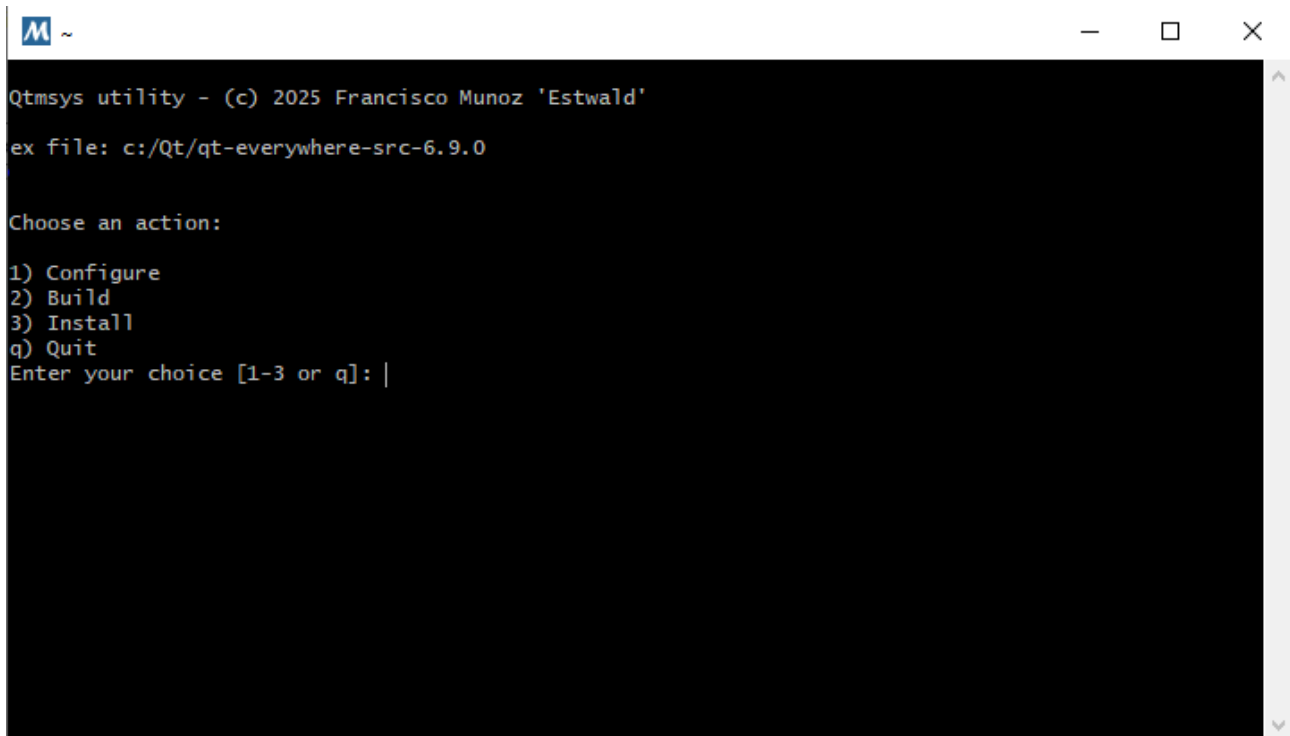
qt-everywhere-src-6.9.0/qtquick3d/tools/meshdebug/CMakeLists.txt:

The following line is added:

target_link_libraries(\${target_name} PRIVATE ucrt)

so it can locate the library.

Once these steps are completed (or detected to have been completed successfully), this menu appears: (see next page)



```
Qtmsys utility - (c) 2025 Francisco Munoz 'Estwald'
ex file: c:/Qt/qt-everywhere-src-6.9.0

Choose an action:
1) Configure
2) Build
3) Install
q) Quit
Enter your choice [1-3 or q]: |
```

1) Configure

Qt 6.9.0 is configured properly so that its libraries can be compiled. During the process, the **/mingw64/lib/cmake** directory is temporarily moved to prevent cmake, when processing the **CMakeLists.txt**, from detecting things we don't want. CMake checks the mentioned cmake and pkgconfig directories, and it also detects shared libraries (**.dll.a**), and at the slightest issue, our static build is ruined.

To keep the source code clean and free of clutter, the directory **\$QTENV/build-qt6-static** is created. Be very careful with this because it is the same directory for both the 32-bit and 64-bit builds... the configuration wipes the previous contents before starting, so there should be no problems as long as the steps are followed correctly.

2) Build

As suggested, it starts the compilation and builds all the necessary Qt 6.9.0 libraries and tools. The compilers installed in MSYS2 are currently MinGW 15.1.0, which is more up to date than the one included in the Qt tools.

3) Install

Once everything has been compiled without errors (it takes hours...), the only step left is to install it to the destination directory.

For the 32-bit version, it would be:

\$QTENV/6.9.0/mingw_msys32_static

And for the 64-bit version, it would be:

\$QTENV/6.9.0/mingw_msys64_static

With this, we now have the static libraries needed to compile with **Qt 6.9.0**.

Compiling Fluidsynth

Although my initial goal was to compile the libraries statically, for the sake of simplicity and to allow users to update more easily, the tool I created for compiling Fluidsynth supports building it either as a static or a shared library—but with a single DLL in the case of the shared version. By default, midieditor.pro is configured to use this shared library.

There are two versions: one for 32-bit and another for 64-bit, each launched via a .bat file that runs the corresponding .sh script through bash.

They are located in midieditor\building with the names **build_fluidsynth32bits.bat** and **build_fluidsynth64bits.bat**, and the subdirectory build-fluidsynth contains the corresponding **build-fluidsynth32bits.sh** and **build-fluidsynth64bits.sh** scripts, which handle downloading the source code, compiling it, and installing it for each version.

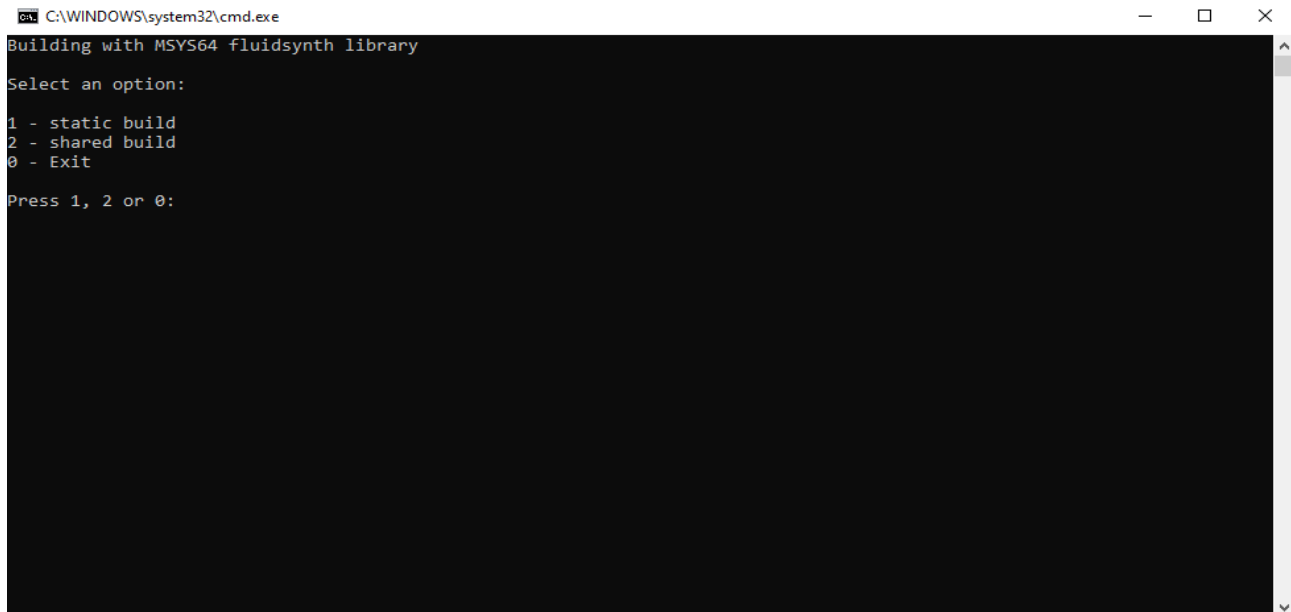
Just like with the other tools, it's necessary to set the **QTENV** environment variable. This is how the compilers are found (which will be in **\$QTENV/msys64**), where the sources downloaded with git will be installed (**\$QTENV/fluidsynth-2.4.6**), and the compiled libraries (for 64-bit, the static ones in **\$QTENV/fluidsynth-64-static** and the shared ones in **\$QTENV/fluidsynth-64**).

Integration into Midieditor must be done manually by copying the created files **libfluidsynth-3.dll** and **libfluidsynth-3.dll.a** for the shared library into **midieditor\lib64\windows**, and the static library **libfluidsynth-3.a** into **midieditor\lib64\windows_static**. It is also advisable to copy the Fluidsynth header .h files into midieditor\fluidsynth, except for fluidsynth.h, which goes into **midieditor\src\fluid**.

As long as you use Fluidsynth version 2.4.6, none of this is necessary—not even compiling—because I already provide the compiled libraries. But the tools are there in case you need them or want to update (if you open **build-fluidsynth64bits.sh** with a text editor, at the start you'll see **FSVERSION="2.4.6"**. By changing that version number, you can get the corresponding updates).

Using the build_fluidsynth64bits.bat tool

When you run **build_fluidsynth64bits.bat**, you will see a screen like this:



```
C:\WINDOWS\system32\cmd.exe
Building with MSYS64 fluidsynth library
Select an option:
1 - static build
2 - shared build
0 - Exit
Press 1, 2 or 0:
```

1 - static build

Press 1 to create the static version

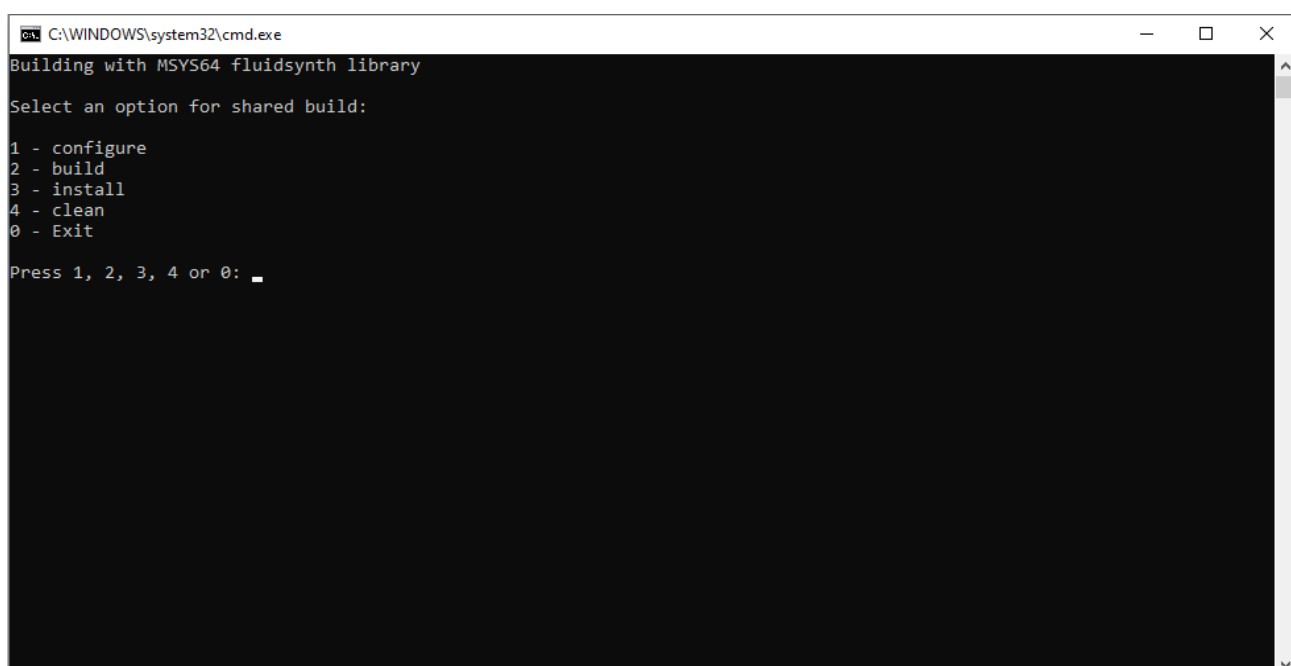
2 - shared build

Press 2 to create the shared version

0 – Exit

Press 0 to exit the application.

If we press 2, it will show us this screen:



```
C:\WINDOWS\system32\cmd.exe
Building with MSYS64 fluidsynth library
Select an option for shared build:
1 - configure
2 - build
3 - install
4 - clean
0 - Exit
Press 1, 2, 3, 4 or 0: _
```

1 – configure

Downloads and configures the Fluidsynth 2.4.6 source code

2 – build

Compiles the library and the fluidsynth.exe application (in this case, shared, so it will use a DLL)

3 – install

Installs the compilation result into the directory \$QTENV/fluidsynth-64 in this case

4 – clean

Deletes the directory \$QTENV/fluidsynth-build-64 containing the build output (once installed, it's better to delete object files that just take up space)

0 – Exit

Exits the application.

The Compilation of Midieditor Custom

Once we have everything necessary to compile **Midieditor Custom**, we need to know the steps to actually do it. The file midieditor.pro (located at midieditor/midieditor.pro) is the one used by the qmake application to configure and build the application. Arguments passed to qmake specify both the architecture and whether we want to build using static libraries, like this:

```
qmake.exe ... "DEFINES += __ARCH64__" "STATIC_BUILD = 1"... // 64-bit static
qmake.exe ... "DEFINES += __ARCH32__" "STATIC_BUILD = 1"... // 32-bit static
```

Obviously, to compile for 32-bit, for example, you must have the corresponding compilers and necessary libraries installed.

About midieditor.pro

In addition to those external arguments that configure the architecture and static compilation, there are others that internally modify how the code is built:

Midieditor Custom version:

```
MIDIEDITOR_RELEASE_DATE="2025/05"
```

```
MIDIEDITOR_RELEASE_VERSION_ID= "0"
```

```
MIDIEDITOR_RELEASE_VERSION_STRING="CUSTOM12LEs"
```

Use static/shared Fluidsynth (# is used to comment out the line and ignore it):

```
#WITH_FLUIDSYNTH=USE_FLUIDSYNTH_STATIC # static
```

```
WITH_FLUIDSYNTH=USE_FLUIDSYNTH # dynamic
```

Note: if both of those lines are commented out (by adding # at the beginning), the result will be a

build without Fluidsynth support.

Environment variables.

QTENV → (for example: **QTENV=F:/Qt**) required for tools that work using the MSYS2 environment.

Path+=F:\Qt\msys64\usr\bin → pointing to MSYS tools like sh.

INSTALLJAMMER=F:\Qt\installjammer\installjammer → (MSDOS path) to InstallJammer. This application is used to create an installer for Midieditor Custom with everything needed (DLLs, resources, etc.).

Download InstallJammer:

<https://github.com/damonicourtney/installjammer>

<https://sourceforge.net/projects/installjammer/>

For MSYS2, edit the InstallJammer file and change:

PLATFORM="" to **PLATFORM="Windows"**

Compile with: qtbuild_msys_win32bits-static.bat / qtbuild_msys_win64bits-static.bat

In the **midieditor\building** directory, apart from the tools for compiling **Fluidsynth**, we also have these two scripts to compile **Midieditor Custom** without relying on **Qt Creator**.

If we run **qtbuild_msys_win64bits-static.bat**, we'll see a screen like this:



```
C:\WINDOWS\system32\cmd.exe
QTENV=F:\Qt

Building with MSYS

Select an option:

1 - Configure
2 - Build
3 - Run
4 - Install
5 - Clean
0 - Exit

Press 1, 2, 3, 4, 5 or 0: _
```

1 – Configure → sets up the build for a 64-bit application with static libraries

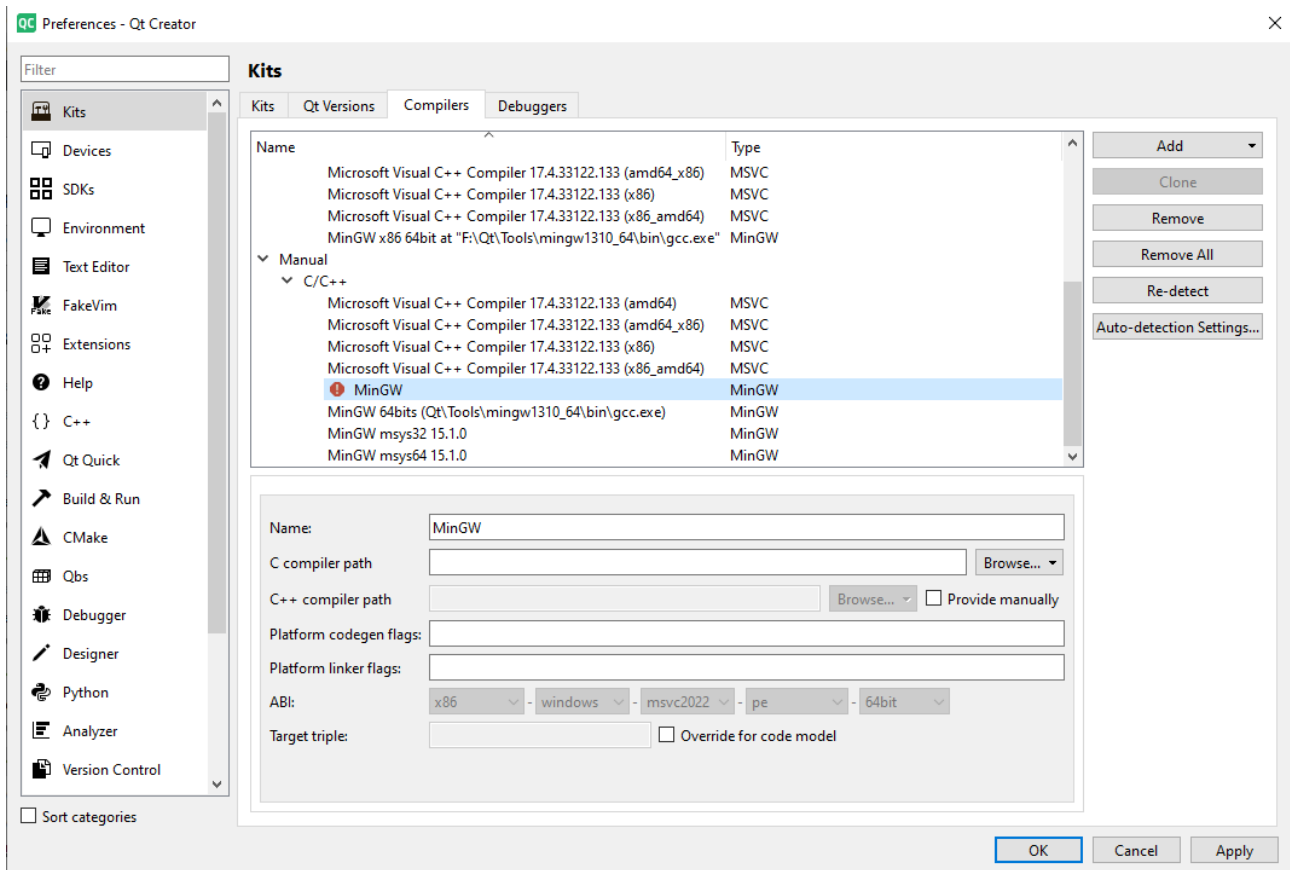
- 2 – Build → compiles Midieditor
- 3 – Run → runs Midieditor (once it has been compiled, of course)
- 4 – Install → packages and creates an installer for Midieditor (midieditor.exe must be built)
- 5 – Clean → deletes the built code for a fresh compilation
- 0 – Exit → exit

The compilation is done in **midieditor\build\build-Midieditor-64bits-static** and inside that directory you will find the executable in **\bin**. **InstallJammer** will work on **\MidiEditor-win64** and will leave the packaged installer in **\install**.

Compiling from Qt Creator

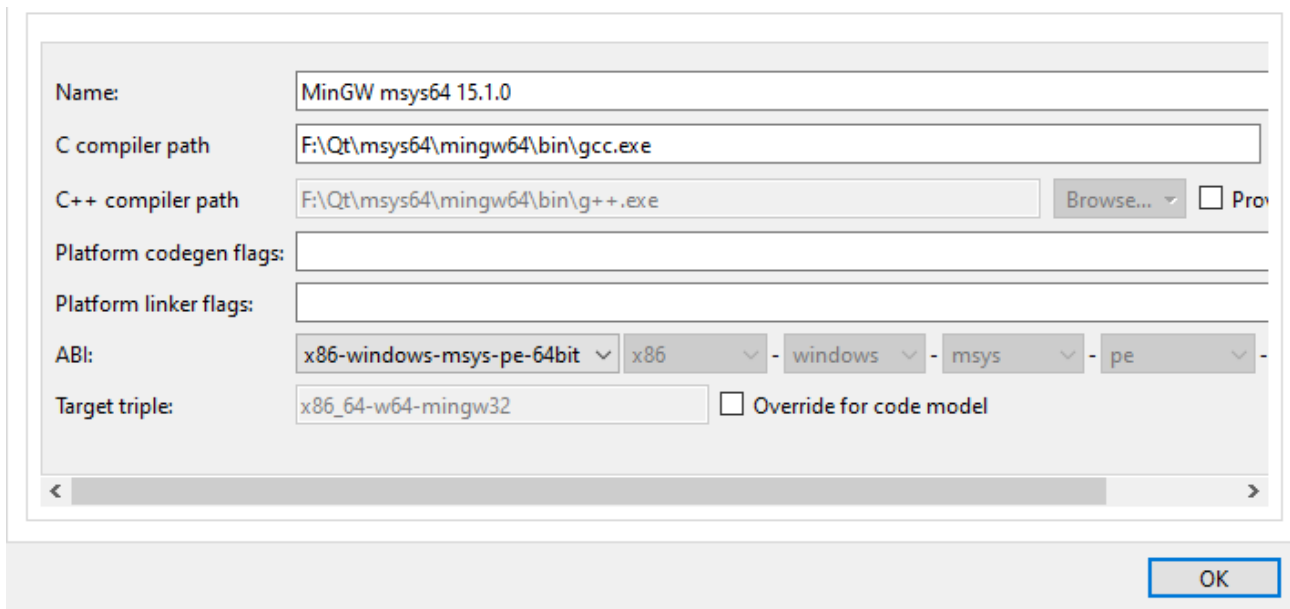
If Qt Creator is installed, double-clicking on midieditor.pro will open the project. The first thing to do here is to create the kit we are going to use to compile. To do this, in the sidebar, click on the “**Projects**” icon and then on “**Manage Kits.**”

In the “**Compilers**” tab, click the “**Add**” button and select “**MinGW.**” You will see something like this:



“Name” should have a sufficiently descriptive name, and what you need to do now is.

In “C compiler path,” locate gcc.exe. An example from my setup for the 64-bit compiler is:



Next, click the “Qt Versions” tab and then the “Add” button. A window will open to locate the qmake.exe file, which if you have built the static **Qt libraries/tools** using the instructions here, should be located at (assuming F:\Qt as QTENV)

F:\Qt\6.9.0\mingw_msys32_static\bin\qmake.exe for 32-bit, and
 F:\Qt\6.9.0\mingw_msys64_static\bin\qmake.exe for 64-bit.

Here's a detail example for 64-bit:

The screenshot shows a Qt configuration dialog. The 'Name' field is set to 'Qt %{Qt:Version} (mingw_msys64_static)'. The 'qmake path' is 'F:\Qt\6.9.0\mingw_msys64_static\bin\qmake.exe'. The 'Qt version' is '6.9.0 for Desktop'. The 'Register documentation' dropdown is set to 'Highest Version Only'. The 'OK' button is highlighted.

Once we have the compilers and the Qt tools/libraries ready, the next step is to create the kit. To do this, click on the “Kits” tab and then on the “Add” button. As a reference, here is the configuration of my kit for 64-bit static:

- **Name** → Desktop Qt %{Qt:Version} MinGW 15.1.0 64-bit static
- **Compiler** → MinGW msys64 15.1.0
- **Qt Version** → Qt 6.9.0 (mingw_msys64_static)

and “OK”

The screenshot shows the Qt Kit configuration dialog. The 'Name' field is 'Desktop Qt %{Qt:Version} MinGW 15.1.0 64-bit static'. The 'File system name' is empty. The 'Build device' and 'Run device' are both set to 'Desktop' with 'Local PC (default for Desktop)' as the device. The 'Compiler' is 'C/C++: MinGW msys64 15.1.0'. The 'Environment' section has 'Force UTF-8 MSVC output' unchecked and buttons for 'Edit Build Environment...' and 'Edit Run Environment...'. The 'Debugger' is 'Auto-detected CDB at C:\Program Files\Windows Kits\10\Debuggers\x64\cdb.exe'. The 'Sysroot' is empty. The 'Qt version' is 'Qt 6.9.0 (mingw_msys64_static)' and 'Mkspec' is empty. The 'Qbs Profile Additions' is empty. The 'CMake Tool' is 'CMake 3.30.5 (Qt) (Default)'. The 'CMake generator' is 'Ninja'. The 'CMake Configuration' is '-DQT_QMAKE_EXECUTABLE:FILEPATH=%{Qt:qmakeExecutable} -DCMAKE_PREFIX_PATH:PATH=%{Qt:QT_INSTALL_P...'. The 'OK' button is highlighted.

Note: When adding the Qt Version, with other builds I encountered an error related to the absence of **qt.conf** (in the **\bin** directory). That file contains the following content:

```
[Paths]
Documentation=../../Docs/Qt-6.9.0
Examples=../../Examples/Qt-6.9.0
Prefix=..
```

Now that we have our kit created, we need to add it to Midieditor. To do this, go to “**Projects**” → “**Build & Run**” and add our kit, which will appear translucent with a “+” sign to add it. Once added, click “**Build**” and at the top right, you will see “**Build Settings**”. Remove “**Debug**” and “**Profile**” from “**Edit build configuration**” by clicking the “**Remove**” button (only keep “**Release**”).

Then, under “**Build Steps**”, click “**Details**” and add the following in “**Additional arguments**”:

```
"DEFINES += __ARCH64__" "STATIC_BUILD = 1"
```

And in “**Build Environment**”, click “**Details**”, look for “**Path**”, click on it, then select “**Append Path**” and browse to the **msys64\usr\bin** directory in **MSYS2**. Alternatively, you can add it directly to the right: **Path+=F:\Qt\msys64\usr\bin** (in my case, since I have it installed there) so that it can find **sh.exe**, **cmake**, etc.

Here you can also add environment variables without needing to define them system-wide. For example, the **INSTALLJAMMER** environment variable.

Now click on “**Run**” on the left and you’ll see “**Run Settings**” at the top right. In “**Deployment**” → “**Method:**”, click on “**Add**” and select “**Deploy Configuration**”. Then click on “**Deploy Configuration**” and choose “**Deploy Configuration2**”. Now click on the “**Rename...**” button and rename it to “**Deploy Install**”.

Next, click on “**Add Deploy Step**” and select “**Make**”. In “**Make Arguments:**”, add: **packing**

And that’s it — everything is ready to compile from Qt Creator.