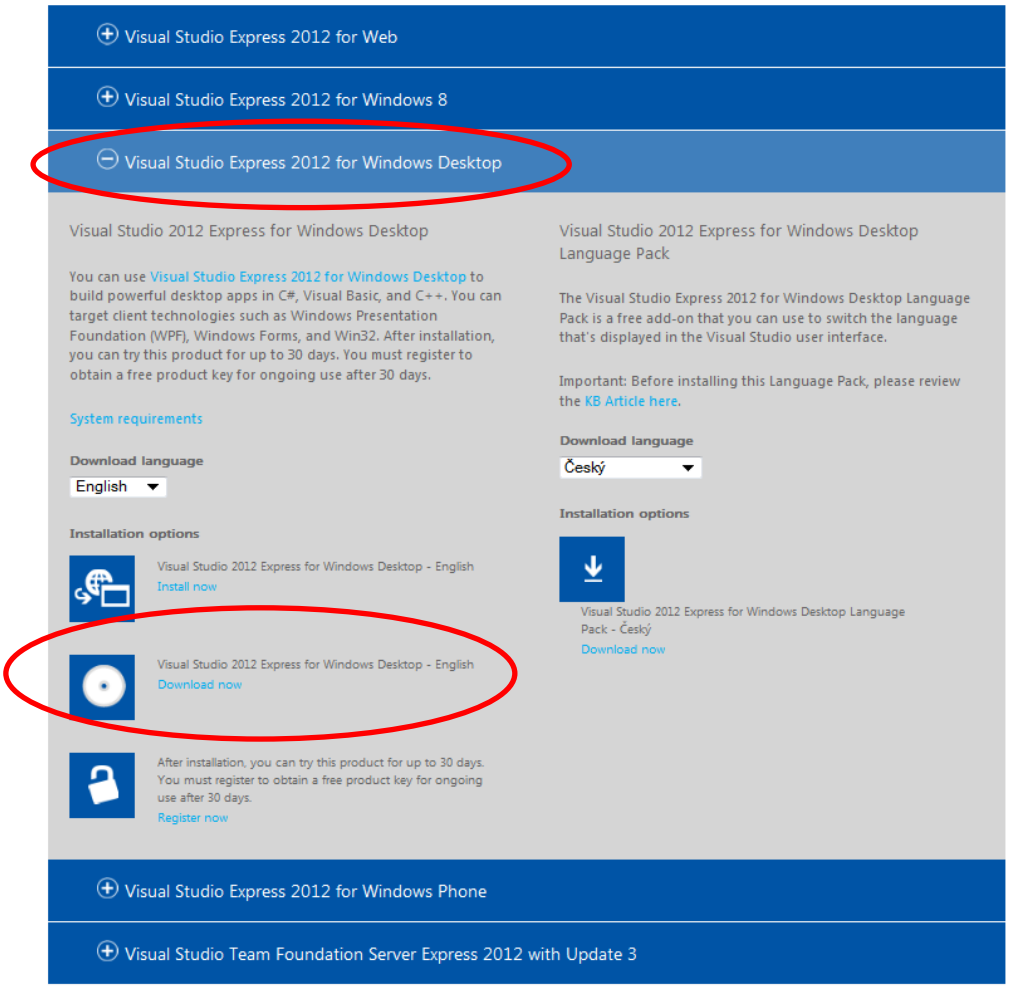
Compiling GUI on the command line

Solution

- Go to the QT website download the version of QT you want to use.
- Make sure the environment variable QTDIR is set.
- The Project file in the “eudaq\gui” folder links by default against QT 4
- The project file in the “eudaq\gui vs 2012” links against QT 5
- Download the express version of Visual Studios
  http://www.microsoft.com/visualstudio/eng/downloads#d-express-windows-desktop



  Make sure you download the version for desktop apps
- Start a visual studio Command line. This is in principle just a command line which has already all the path end environment variables you need.
- Go in the folder with the .SLN file you want to compile.
- Type: for example
  MSBUILD GUI_EUDAQ.sln /p:Configuration=Release
- Finish.

Details

MSBUILD

This is the program that processes the project (solution) files and feeds it to the compiler and linker. If you have a working project file it is more or less straight forward. It has a very simple syntax:

MSBuild MyApp.sln /t:Rebuild /p:Configuration=Release

myApp.sln is the file you want to Process. /target (short /t) tells msbuild what to do in this case rebuild. You have all the options you need like: clean, build and rebuild. You can also specify your own targets. With the “parameter property” switch you can change the properties of your Project. Let’s say you want to compile EUDAQ, you go in the folder where the solution (sln) file is and type:

MSBUILD EUDAQ.sln /p:Configuration=Release

One thing one has to keep in mind is that there are some default configurations. The default is a debug build for x86. If you want to have it different then you need to specify it in the command line. And one thing you want to have is a release build! With the /p switch you can overwrite properties like in this case the configuration. But you could also overwrite the compiler version it should use. Let’s say you want to use VS 2013 then you have to specify it by writing:

MSBUILD EUDAQ.sln /p:PlatformToolset=v120 /p:Configuration=Release

But be careful when changing the compiler settings. It is possible that some then link against an incompatible version of your external libraries.

Project Files

It’s the Visual Studio equivalent to a makefile. The Project files have a very easy syntax but a complicated mechanism behind it. Making changes to an existing file is very easy. Writing a new one from scratch is expert level. But also, in most cases, pointless because VS does it for you. Therefore usually one gets a finished Project file that was auto created by VS and one just wants to make some minor changes to it, therefore it is enough to know where one can tweak around.

Let’s start easy and assume you want to change the output directory. You can do this by adding the following line to the corresponding Property group.

```
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">
        <OutDir>..\..\Windows Binaries\</OutDir>
</PropertyGroup>
```

Or lets say you want to change the compiler version. You can do this by changing the platform toolset to the version you need. You can find this option in

```
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'"
Label="Configuration">
…
<PlatformToolset>v110</PlatformToolset>
</PropertyGroup>
```

V110 stands for Visual Studio 2012. V120 stands for VS 2013 and so on.

The next interesting switches are in here:

```
<ItemDefinitionGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
   <ClCompile>
      <PrecompiledHeader></PrecompiledHeader>
      <WarningLevel>Level3</WarningLevel>
      <Optimization>Disabled</Optimization>
      <PreprocessorDefinitions>
          WIN32;
          _DEBUG;
          _CONSOLE;
          %(PreprocessorDefinitions)
      </PreprocessorDefinitions>
      <AdditionalIncludeDirectories>
          ..\..\main\include;
          ..\..\extern\pthread-win32\include;
          ..\..\tlu\include;
          ..\..\extern\ZestSC1\windows 7\Inc;
          ..\..\extern\libusb-win32-bin-1.2.6.0\include
     </AdditionalIncludeDirectories>
  </ClCompile>
  <Link>
    <SubSystem>Console</SubSystem>
    <GenerateDebugInformation>true</GenerateDebugInformation>
    <AdditionalLibraryDirectories>
        ..\..\extern\libusb-win32-bin-1.2.6.0\lib\msvc\;
        ..\..\extern\ZestSC1\windows 7\lib\x86\
   </AdditionalLibraryDirectories>
   <AdditionalDependencies>
      ZestSC1.lib;libusb.lib;kernel32.lib;user32.lib;gdi32.lib;winspool.lib;comdlg32.lib;
      advapi32.lib;shell32.lib;ole32.lib;oleaut32.lib;uuid.lib;odbc32.lib;
      odbccp32.lib;%(AdditionalDependencies)
    </AdditionalDependencies>
```

```
  </Link>
</ItemDefinitionGroup>
```

lets go through it from start to end.
An Item definition Group is the place where you define your items. One can compare Items to a struct in C++; it is an object that contains different types of information. The Condition statement works like an IF in C++.

In this article you can find all the possibilities you have: http://msdn.microsoft.com/de-de/library/7szfhaft.aspx

In the next line you are defining an item called “CLCompile” and you give it the some attributes like “PreprocessorDefinitions” or “AdditionalIncludeDirectories”. This Object contains all the information that gets sent to the compiler. That means all the compiler flags are set here. The actual files are included later in the project file. So for now you have only defined how you want to compile your files but not what files you want to compile. AdditionalIncludeDirectories does exactly what you think it does. It understands all relative paths and path with environment variables exactly as it should. Next thing is “PreprocessorDefinitions”. It also works exactly as you think it does. That means you can either define just names for your #ifdef statements in the code or you can define macros like

```
<PreprocessorDefinitions>
  SOMEVALUE=3;
  WIN32;
  _DEBUG;
  _CONSOLE;
  %(PreprocessorDefinitions)
</PreprocessorDefinitions>
```

Then you can call in your code SOMEVALUE and it will be 3. I don’t know if it is possible to define macro function like “#define x_square(x) x*x”.

Lets proceed to the “Link” item. Here you can set all the flags for the linker. Most important here are the two attributes where you can set the “AdditionalLibraryDirectories” and the “AdditionalDependencies”. Again these two attributes do exactly what you think they do. You can use all sorts of relative path and so on. For example you can say something like

```
    <AdditionalDependencies>
            $(myFancyLibPath)\*.lib;
            odbccp32.lib;%(AdditionalDependencies)
     </AdditionalDependencies>
```

And it will link against all *.lib files in this directory.

Next thing you need to know is where to put your files you want to compile. Somewhere below the ItemDefinitionGroup there is an ItemGroup which contains the Include statements. It looks like this:

```
<ItemGroup>
  <ClCompile Include="src\someFile.cc"/>
  <ClCompile Include="src\someOtherFile.cc"/>
  …
  <ClCompile Include="src\*.cpp"/>
   …
</ItemGroup>
```

Here you can either put individual files or groups of files in. But be careful that you don't include the same file twice. There is also an ItemGroup which contains the include files. This one seems to be more important for the IDE of VS so that it shows the header files in the Solution Explorer.

A typical use case is that you wrote your own "Data Converter Plugin". This file needs to be mentioned here!

What you won't find in the project file is the section that passes the files to the compiler. This part is hidden behind the following import statement:

```
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets"/>
```

It is usual not needed to go in this file and you should not make any changes there. But if you want to see it you can find it in this folder:

C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\

Don't go there in the attempt to understand how it works. It is not written to be very clear nor understandable, go to Documentation pages like:

http://msdn.microsoft.com/en-us/library/dd293626.aspx

Known Problems

- The environment variables are pulled in as properties therefore they can be overwritten in the project file or in the "vcxproj.user" file. So if for example your QT Project won't compile and keeps complaining about not finding the correct directory make sure you are not overwriting the QTDIR environment Variable with a Property.