GSL on Windows

aweatherguy

Version 1
December, 2016

Copyright © 2016, aweatherguy (email: wsdl at osengr.org)

GSL on Windows is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 2 of the License, or (at your option) any later version.

GSL on Windows is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with GSL on Windows. If not, see `<http://www.gnu.org/licenses/>`.

GSL-on-Windows is a tool set package which makes it possible to compile GSL natively on Windows using Visual Studio. This package does not include a GSL source distribution, and can build a working VS solution from several different versions of GSL source packages. The intention is for this package to work with future GSL releases, but the reality will of course depend on those future releases.

By analyzing the contents of an off-the-shelf GSL source distribution, this tool set creates a working Visual Studio build environment. Three files are added to the GSL source and several files are patched during this process. See the section on how it works for more information.

This package has been tested with the following configuration(s):

- GSL source versions between 1.8 and 2.3
- Windows 7 (8 and 10 probably work, too)
- Microsoft Visual Studio 2010 or 2012 (later versions probably work)
- ActiveState perl 5, version 14 (later versions probably okay)
- 7-zip (any version is probably okay)

All of the test programs in all versions listed above pass with two exceptions in the version 2.3 build. These failures have been reported to the GSL bugs mailing list and the response is that these are not really failures but cases where limits have been set too tight. They should disappear in the next version of GSL.

Instructions

Before proceeding you might (or might not) decide whether to link the test projects with the static or dynamic library.

- If you only plan to use either the static or DLL library, then link the tests against the same type of library.
- If you're not sure, then don't worry about it – this option can be changed at any future time.
- If you plan to use both types of library then it would be wise to build both kinds of tests. This cannot be simultaneously; the solution may be configured to test the static library first for example. After the tests have been run, the solution can be re-built to test the DLL instead.

Creating the VS Solution

Here is the sequence of steps required to get a working VS solution. But first, a couple of preliminary clarifications to prevent confusion:

- When instructed to extract an archive into its own directory below, this can be done by right-clicking on the archive and then choosing "7-Zip ⇒ Extract Here".

- From a DOS prompt, to pass command line arguments to a perl script, add them as an argument following the name of the script. For example:

  ```
  > perl script.pl arg1 arg2 ... etc
  ```

So without further adeiu, here is the process to follow:

- Run the `makePackate.bat` batch file. This will create a 7-zip archive named `GSL-on-Windows.7z`.
- Create a new, empty directory somewhere. We'll assume it is named `xyz` here just for discussion, but it can have any valid name for a directory.
- Copy the GSL source archive into `xyz` and unpack it there. Unpacking the `.tar.gz` file will yield a `.tar` file. Unpacking the `.tar` file will yield a directory named `gsl-<version>x`. For example, you might get `gsl-2.2`.
- Rename this directory to `source`.
- Copy the `GSL-on-Windows.7z` file into the `xyz` directory and extract it there. This will yield two new files: `README.txt` and `GSL-Build-Tools.7z`.
- Unpack `GSL-Build-Tools.7z` into the same directory.
- There will be five perl scripts, numbered 1-5.
  - The first one (numbered 1) can take an argument, either `-static` or `-dll`. This determines whether the test projects will link against the static library or DLL. It will default to DLL linking if no argument is given.
  - The second one (numbered 2) requires the GSL version number as a command line argument.
- Run the perl scripts in the indicated order from a Windows command prompt. The last one (5) is only required for GSL version 1.8 (or perhaps earlier).
  - As an alternative to running the five scripts, one by one, there are two batch files which will do this automatically.
  - `run-all-perl-scripts-static.bat` will generate a solution with tests statically linked.
  - `run-all-perl-scripts-dll.bat` generates a solution with dynamically linked tests.

After running the four or five perl scripts you should have fully functional Visual Studio 2010 solution.

Building the Solution

This should be self-evident to many. There's nothing tricky here, just the standard build process.

- Open the Visual Studio solution file named `gslnw.sln`.
- Select the desired build platform, x64 or Win32 (aka x86), and configuration (Debug or Release).
- Build the solution. This will create both a static library and a DLL. The outputs are in subdirectories named after the platform and configuration. For example, the 64-bit/Release outputs will be in the `x64\Release\` directory.

It can take a while for the build to create the Release DLL – it is busy generating code for all of the in-line functions. This does not occur for the Debug builds and they run faster. Two libraries will be built:

libgsl.lib is the static library. This is all that's required to link with an application if using static linking.

libgsl-dll.dll is the dynamic library. The companion lib file will also be there in the output directory (`libgsl-dll.lib`).

Running Test Programs

The test executables are found in the `tests` directory under platform and configuration sub-directories. There is a `test-all.bat` file in each executable directory which will run all the tests in succession. Beware that some of the tests will spew huge amounts of data to the console when run in the Debug configuration.

Installation

The files comprising the built package are:

- The header files found in the `source\gsl` subdirectory.
- Library files from the Release build of your choice, either 32 or 64-bit (or both). This would be one or both of

  – `libgsl.lib`
  – `libgsl-dll.lib` and `libgsl-dll.dll`

  depending on your choice of static and/or dynamic libraries.

There is no one way to "install" this package. It can be located in any local directory, or for a perhaps more secure setup it could be placed in a protected `Program Files` subdirectory. Wherever located, the header files should be in a subdirectory named `gsl` and the compiler's include search path should include the parent directory of the `gsl` directory.

There is even less restriction on the location of the library files. When running a DLL-linked application, a copy of the DLL must either be in the application's directory or a directory that is part of the system search path.

One logical setup would be to create a directory named after the release (e.g. `gsl-2.2`). Then create subdirectories named `include\gsl` for the header files, and `bin` (lib and DLL files). Add the `bin` directory to the system's `PATH` environment variable. This directory could then be placed in the `Program Files` or `Program Files (x86)` directories as appropriate.

Another option would be to create the same directory structure but place it in a local user folder. Hopefully this discussion has provided some sense of what's involved in installing or using GSL on Windows.

Compiling Applications

There are very few things to setup before building an application with GSL on Windows:

- The sub-directory `source/gsl` contains all of the header files your application will need to compile. You can place a copy of this directory anywhere, somewhere local or in a program files folder. Configure the VS compiler options to search this directory for header files in addition to the usual suspects.
- If linking against the DLL, and your application references any of the global variables exported from GSL, then you must define the macro `GSL_DLL` when compiling. This can be done either in the application source files (before any GSL headers are included), or on the compiler command line – in the project's properties page.
- The build configuration will be different for linking with static and dynamic libraries. In both cases, the linker's search path must include the directory where these libraries are located.

  Static Specify `libgsl.lib` as an input to the linker. Nothing else is required.

  DLL Specify `libgsl-dll.lib` as an input to the linker.

Linking Applications to GSL

I know this is repetitive, but sometimes, repetitive is good! Linking against the static library is easiest. Just select the `libgsl.lib` file as a linker input. That's it.

Linking against the DLL requires a little more care. First of all, the macro `GSL_DLL` must be defined when compiling your application if you access any of the global variables exported from the DLL. Without this macro there will be undefined symbol errors from the linker. The linker input must be set to `libgsl-dll.lib`. Do *not* point the linker at the DLL file. Then, when running the application, a copy of the DLL (`libgsl-dll.dll`) must either be in the same directory as the application, or somewhere in the system library search path.

How it Works

This will start with a description of the actions performed by the five perl scripts. Following that there is a description of the template-based approach used by this tool set.

Script File: `1-create-vs-sln.pl`

Most of the heavy lifting happens here.

- Generates a list of library sub-projects based on the subdirectories found in `source`
- Creates `vcxproj` files to build the static and dynamic libraries based on the list of library sub-projects. This is done by adding a list of sub-libraries to template files.
- For each library sub-project found:
  - Creates a `vcxproj` file based on the source file list found in `Makefile.am` in the sub-project directory. This step uses a template file.
  - Creates a `vcxproj` file for testing that sub-project, again based on the list of test source files found in `Makefile.am`. This also uses a template file.
  - Copies local header files into the main `gsl` include directory – again based on the header files listed in `Makefile.am`.
- Appends Windows-specific macros to the header files `gsl_math.h` and `gsl_test.h`. These macros take care of issues like disabling a few compiler warnings and defining the `PCTZ` macro based on the build platform (x86 vs x64).
- Creates a Visual Studio solution file from scratch. This includes all of the library sub-projects, test projects, static and dynamic library projects. Project dependencies are also generated so that everything builds in the proper order.
- Generates the `test-all.bat` batch script to run all test projects serially. This is copied into each of the four executable output directories in the `tests` directory.

Perl Script: `2-create-def.pl`

Symbols to be exported from the DLL are defined in a module `.def` file. This script analyzes each of the header files in the `source/gsl` directory to build a valid `.def` file.

This is the trickiest, twitchiest operation in the process. It also has to do with somewhat buggy configuration information in GSL release 2.2. There is one global variable declared for export in 2.2 which does not actually exist. The tool would blindly add this symbol to the def file and a linker failure would occur on the undefined symbol. That's why this script requires the version number as an argument – so it will know whether to delete that one name from the export list.

Perl Script: `3-patch-fp-c.pl`

Setting up the system's floating point control word is accomplished by files included from `source/ieee-utils/fp.c`. There is no real C-code in this file, just macros. This file effectively does a switch on macros in `config.h` and includes then include a C-language file specific to the target machine and OS. This file (e.g. `fp-gnux86.c`) contains the real, machine/OS-specific code.

The standard GSL source distribution does not have any switch setting for a Windows OS or machine, so this perl script patches `fp.c`, adding another switch on the macro

```
HAVE_WINX64_IEEE_INTERFACE
```

And of course, this macro is defined in the `config.h` file supplied with the tool set. Also supplied (and stuffed into the `ieee-utils` directory) is the file `fp-winx64.c` having Windows-specific code to manipulate the floating point control word.

Perl Script: `4-patch-percent-z.pl`

Many of the GSL test programs (and one of the sub-libraries) use the `%z` format specifier with system calls such as `sscanf`, `fprintf` and so on. This expects an argument of type `size_t` which will either be a 32 or 64-bit integer depending on the machine and OS. With Windows, this changes between 32-bit (x86) and 64-bit (x64) build platforms.

Problem is, the Windows libraries don't support the `%z` format specifier. The way around this problem is to replace occurances of this specifier with a macro – this tool set uses the name `PCTZ`. This has to be done carefully so that replacements in quoted strings work correctly, and it results in some odd looking quoted strings.

Finally, we get to the meat of it: This perl script searches all of the source distribution, replacing all `%z` specifiers with the `PCTZ` macro. This macro will evaluate to either `%ll` or `%l` depending on whether a 32 or 64-bit build is being executed.

Perl Script: `5-patch-test-sf.pl`

This is only required with GSL version 1.8 (and maybe earlier). There are a couple of conditional macros in that file which look like this:

```
#if RELEASED
```

The macro `RELEASED` is defined (or, usually not) in `config.h`. It would seem that when there is no such macro defined, `gcc`'s pre-processor thinks this is just fine, but the Windows compiler barfs. Later GSL releases have changed this macro to:

```
#ifdef RELEASED
```

which works just fine on Windows too. This perl script merely makes this change in the `source/test_sf.c` C-language file.

Template Based Processing

The first perl script creates a VS solution file from scratch, but all `vcxproj` project files are created from one of five different template files. The templates are nearly complete project files which are only missing a list of objects required to build the project. The perl script adds the list of objects to generate a valid project file which is added to the directory tree. In the case of sub-libraries and test programs, the missing objects are C-language source files. The two main library templates (static and DLL) are missing the list of `.lib` files to be input to the link editor.

These are simple text files and may be edited to modify properties of the final project files that are built (by those who what they are doing, of course). If a global change is desired to some project property, it will be easiest to modify the template file(s) and then re-run the first perl-script. For example, to remove the `DEBUG` macro from all Debug configuration builds of the test programs, it will be easier to edit the `test-vcxproj-static-template.txt` and `test-vcxproj-dll-template.txt` files instead of laboriously editing over 100 individual project settings in Visual Studio.

What follows is a brief description of each of the five template files used by the first perl script.

- `vcxproj-template.txt`
  This is the basis for all of the sub-library projects. The output of each of these projects is a single static library file, which will then be combined with all sub-library static library files into the main static and dynamic output libraries.

- `test-vcxproj-static-template.txt`
  All test programs which are statically linked to the main library output are generated from this template.

- `test-vcxproj-dll-template.txt`
  Linking of test programs to the dynamic (DLL) main library output is accomplished by using this template. The choice of this template or the preceding one by the first perl script is the purpose of its `-static` and `-dll` command line arguments.

- `libgsl-vcxproj-template.txt`
  The main static library output, `libgsl.lib` is generated by the project. The first perl script adds a list of sub-library static library files to the linker input section to create the working project file.

- `libgsl-dll-vcxproj-template.txt`
  Generating the main dynamic library (DLL) output is accomplished through this template. Similar to the static library template, a list of static sub-library files is added to this template to complete it. The template also contains a reference to a `.def` file – this is where the list of symbols to be exported by the DLL is specified. The def file is generated by the second perl script.