LLVM-MinGW

A free C/C++ toolchain for targeting Windows on i386, x86_64, ARM and ARM64

https://github.com/mstorsjo/llvm-mingw

Martin Storsjö, martin@martin.st, LLVM Distributors Conference, September 16, 2021

What

- “MinGW” is normally a combination of
  - GCC, binutils
  - libgcc, libstdc++
  - A MinGW “SDK” (mingw-w64)
    - A freely redistributable SDK, contrary to the official SDK
    - GCC compatible platform headers
    - Import libraries for system DLLs
    - No extra runtime (includes a minimal amount of statically linked compat helpers)

What

- Lots of preexisting MinGW toolchain distributions
  - Linux distributions provide cross compiler packages
  - Standalone native Windows compiler packages
  - MSYS2 package manager/repository

Origins (2014-2017)

- Originally no free toolchain for targeting Windows on ARM
  - Some open source projects prefer not to use MSVC
    - Both for philosophical and practical reasons (hard to cross compile, non-redistributable tools and SDK)
  - GCC never got support for Windows on ARM
  - LLVM happened to have fairly complete support for Windows on ARM
  - Clang/LLVM also had fully functioning support for targeting MinGW

Origins (2017)

- For x86 targets, it’s easy to just swap in Clang instead of GCC in an existing toolchain
- For targets not supported by GCC/binutils, need to bootstrap the whole toolchain using LLVM tools
- Picking up an effort started by Martell Malone - main missing pieces: LLD, llvm-dlltool
  - llvm-dlltool takes a symbol listing (.def) and produces an import library for it
  - Initial tools merged upstream in late 2017
- Could maybe use libgcc and libstdc++, but unclear how to build them standalone from the rest of GCC
  - Therefore, source all components from LLVM: compiler-rt, libunwind and libc++

Windows on ARM64

- Rumors of Windows 10 Desktop on ARM64 in 2017
- A handful of public ARM64 .exe files in WinSDK
  - Wine had got initial support for running them on Linux (thanks to André Zwing)
  - Main missing feature: Emulating Windows va_args on Linux, needs compiler support
- Bootstrapped initial support for Windows on ARM64 in LLVM, Clang and LLD in 2017 (started by Mandeep Singh Grang)
  - Matured support for ARM in mingw-w64, added support for ARM64
  - Reference environment: Running built binaries in Wine
- Originally no access to MSVC targeting ARM64 for reference, but that got published when the LLVM support was pretty far along
- The toolchain was fairly complete when got access to a prototype device
  - Only a couple of bugs to resolve on the real OS
    - Missing stack probing for large stack frames - Wine on Linux doesn't need it
    - Windows refuses to run binaries linked without `-dynamicbase` on ARM64 (simple flag change in LLD)

Windows on ARM64

- https://youtu.be/vdYIaUeZnqc?t=1345 - Windows 10 on ARM for developers : Build 2018
  - Demoing VLC on ARM64, when Windows 10 on ARM64 was launched
- Presentation tells the entirely wrong story though:
  - Had a toolchain far in progress to build VLC when I asked VLC contacts to reach out to MS for prototype devices
  - No early MSVC tool access, only used public tool releases for reference
  - Presentation shows it off as an example of easily rebuilding existing apps with MSVC for a new target, with zero lines of code changed
    - Actually built an entirely new toolchain instead of using MSVC (VLC is built with autotools)
      - VLC actually had out-of-tree patches for building with MSVC, but wanted to get away from it
    - Originally somewhere around 60-100 patches on VLC to make it build, all later upstreamed or made unnecesasry

Toolchain contents

- Entirely unpatched build of LLVM
  - Is meant to be usable with Clang/LLD from a different distribution (but never gotten around to setting that up)
    - (Using a distribution-provided Clang could be tricky as one has to install new files in `<clang>/../lib/clang/<version>/lib/windows`)
- 4 mingw-w64 sysroots (i686, x86_64, armv7, aarch64)
  - One shared include dir, separate lib directories
- compiler-rt builtins, libunwind, libc++, sanitizers built for all 4 targets
- GCC-style triple-prefixed toolchain frontends; any build with `./configure --host=<arch>-w64-mingw32` is meant to work
  - `i686-w64-mingw32-clang`
  - `x86_64-w64-mingw32-clang++`
  - `armv7-w64-mingw32-windres`
  - `aarch64-w64-mingw32-gcc`
- Providing `<triple>-gcc/g++` as aliases to clang, to work with build systems that only check for those tool names

Wrappers

- Clang target wrappers
  - Wrappers like `aarch64-w64-mingw32-clang` invoking `clang -target aarch64-w64-mingw32 -fuse-ld=lld -stdlib=libc++ -rtlib=compiler-rt`
    - Wrappers written in shell script for hackability on Unix, built as native executables on Windows
    - Unfortunately need to use `-Qunused-arguments` to silence warnings about some of the options that are unused
      - Users of the toolchain won't get the regular warnings they'd get elsewhere
  - Considerations/tradeoffs:
    - Could set hardcoded defaults in the built Clang, e.g. `-DCLANG_DEFAULT_RTLIB=compiler-rt`, `-DCLANG_DEFAULT_CXX_STDLIB=libc++`, `-DCLANG_DEFAULT_LINKER=lld`, instead
      - Setting such defaults in a cross compilation setup makes the Clang binary unsuitable for the host system (not building any runtimes for the host system)
      - Would make it harder to use an externally provided Clang binary
      - Wrapper scripts allow setting other defaults, like `-f<type>-exceptions` for building with a nondefault exception handling mechanism (have switched to non-default exception handling mechanisms along the years to avoid bugs until they are fixed)
    - Could use clang configuration files with settings for each target
      - Avoids the warnings about unused arguments
      - Haven't retried this approach lately to see if there's any blockers

Historical hacks

- Back in 2018, llvm-objcopy didn’t support COFF, thus no working strip tool
  - objcopy and strip don’t really need to touch anything architecture specific
  - Wrapper script around GNU objcopy
    - Modify PE/COFF header, changing the header machine field from ARM to i386, from ARM64 to x86_64
    - Invoke GNU objcopy
    - Restore header fields

Releases

- All toolchains work as cross compilers to all 4 current Windows architectures (i686, x86_64, ARM, ARM64)
  - Cross compilers from Unix (Binary releases for x86_64 and aarch64 Linux)
  - Native toolchains running on all 4 architectures (Binary releases of all 4)
- Buildable from source on macOS too, but no binary releases there

Building

- Bootstrapped as a cross compiler, from scratch every time
  1. Build LLVM (using host compiler - normally Linux)
  2. Install mingw-w64 headers, build base mingw-w64 libraries
  3. Build compiler-rt -> Usable as C compiler
  4. Build libunwind/libcxxabi/libcxx -> Usable as C++ compiler
- For building toolchains for Windows:
  5. Cross compile LLVM using toolchain built above
  6. Copy the sysroots (import libraries, runtimes) from above into the crossbuilt toolchain

Uptake

- The Clang/LLVM based build configuration is getting picked up by others too
  - MSYS2 have added environments that use Clang and LLD as default tools
    - Currently 1825 packages out of 2090 successfully built in this configuration
- LLVM-MinGW remains my reference environment
  - Useful for people wanting a small, self-contained (cross) toolchain

Testing on commodity HW

- On Linux, run a nightly build of latest llvm-project, latest mingw-w64
  - Build LLVM with assertions enabled. If building a previously untested git version of LLVM, this is a must.
  - Run a small set of smoke tests to validate the toolchain before “installing” it as latest nightly build
    - Running tests locally in Wine on x86, in Wine on ARM on a connected devboard via ssh
  - Keep a one-week window of older nightly toolchains for easy testing when there’s regressions
- Build (latest git of) a bunch of open source projects of interest with the toolchain
  - Most miscompiles are caught by failed asserts
  - Build-only testing of some projects, running test suites for others
  - Only running tests that can be run in Wine
    - Won’t catch incorrect use of APIs in the tested projects
    - Will catch most cases of the compiler not doing the right thing

Testing on commodity HW

- For small/important things, test all combinations every night
  - All ffmpeg, dav1d tests run for all architectures every time
    - Often catches misoptimizations if they don’t trigger asserts
- For heavier things, do round-robin testing with one configuration per night
  - mingw-w64 can be configured with two C runtimes as default, UCRT or msvcrt.dll (and UCRT can be configured in two ways)
  - Build VLC for one architecture per night (not running any tests, but storing built version)
  - Won’t find all regressions immediately, but gives sensible test coverage at a tolerable runtime
    - Most regressions are found within a couple days
    - The sooner a regression is found, the easier it is to handle it

Testing on commodity HW

- Building LLVM takes a lot of time
- Bisecting can be very slow
- Run a `nice`d background job that continuously pulls the latest main branch, builds and populates ccache
  - Much smaller hit when updating the actual llvm-project workdir used for development
- A 5 GB ccache allows caching built object files around 1-2 weeks back
- Bisecting most regressions within the cached region should be fairly quick
  - … or let’s say, less painful than without a cache

Testing on commodity HW

- By having things tested with projects of interest continuously, it should be possible to release anytime
  - Earlier did releases with random git snapshots of LLVM
  - Nowadays settled down on doing releases based on LLVM stable releases

Thanks!

https://github.com/mstorsjo/llvm-mingw

martin@martin.st