



Creating an LLVM Runtime

Michael Kruse (*michael.kruse@amd.com*),
Joseph Huber (*joseph.huber@amd.com*)

Outline

Introduction

Let's Create a Runtime

Are We Done Yet?

Outline

Introduction

Why This Tutorial?

History

Rationale

Goal

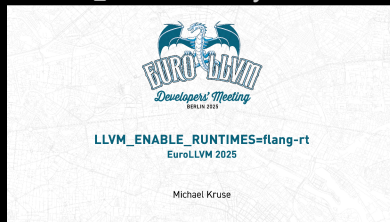
Let's Create a Runtime

Are We Done Yet?

About Me



- ▶ At AMD since 2024
- ▶ Working on
 - ▶ Fortran
 - ▶ OpenMP®
 - ▶ GPGPU Offloading
 - ▶ Loops
- ▶ Converted Flang's runtime to use the LLVM_ENABLE_RUNTIMES system



<https://youtu.be/AGrInJLxJOQ>

Why This Tutorial



1 hour ago

Member



After this change, it's no longer possible to run openmp tests in a non-bootstrapping build (i.e. without building clang at the same time).

Previously it was possible to run the tests in a standalone build, by building the `FileCheck` and `not` binaries and pointing at them with `OPENMP_LLVM_TOOLS_DIR` - like I did at [mstorsjo/llvm-mingw@20260210 /github/workflows/build.yml#L583-L612](https://mstorsjo.com/20260210/github/workflows/build.yml#L583-L612).

Now after this change (or before, when doing a runtimes build), running tests only works if there's a `FileCheck` target already in the build, i.e. if doing a bootstrapping build:

[llvm-project/openmp/cmake/OpenMPTesting.cmake](#)
Lines 52 to 74 in [c6425aa](#)

```
52     if (${OPENMP_STANDALONE_BUILD})
53         find_standalone_test_dependencies()
54
55         ## Set lit arguments.
56         set(DEFAULT_LIT_ARGS "-sv --show-unsupported --show-xfail")
57         if (MSVC OR XCODE)
58             set(DEFAULT_LIT_ARGS "${DEFAULT_LIT_ARGS} --no-progress-bar")
59         endif()
60         if ("${CMAKE_SYSTEM_NAME}" MATCHES "AIX")
61             set(DEFAULT_LIT_ARGS "${DEFAULT_LIT_ARGS} --time-tests --timeout=3000")
62         endif()
63         set(OPENMP_LIT_ARGS "${DEFAULT_LIT_ARGS}" CACHE STRING "Options for lit.")
```



Why This Tutorial

on Dec 8, 2025 • edited

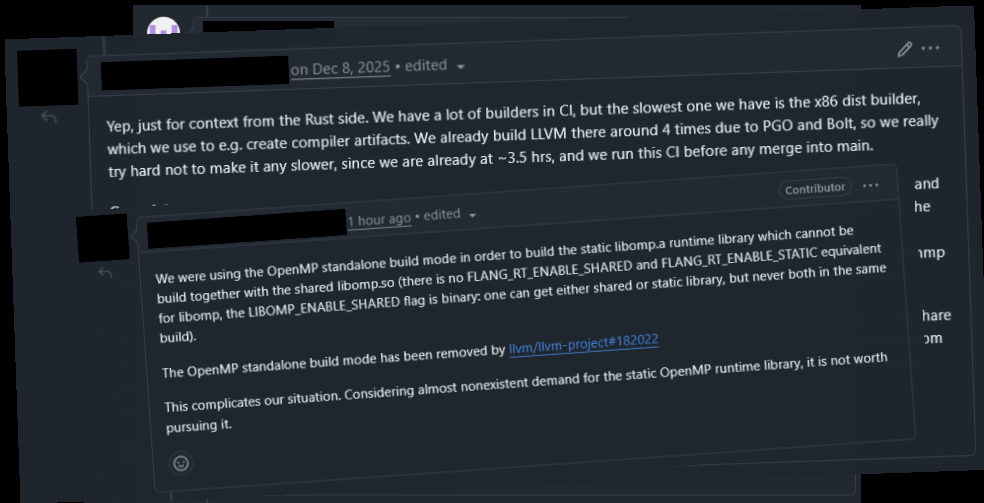
Yep, just for context from the Rust side. We have a lot of builders in CI, but the slowest one we have is the x86 dist builder, which we use to e.g. create compiler artifacts. We already build LLVM there around 4 times due to PGO and Bolt, so we really try hard not to make it any slower, since we are already at ~3.5 hrs, and we run this CI before any merge into main.

One of the compromises there is that we have two checkouts of LLVM - once as a standalone `llvm/llvm-project` build (A) and once as a submodule of `rustc` in `src/tools/llvm-project`. We use build A to compile C/C++ code, and keep rebuilding the second checkout (B) during bootstrapping.

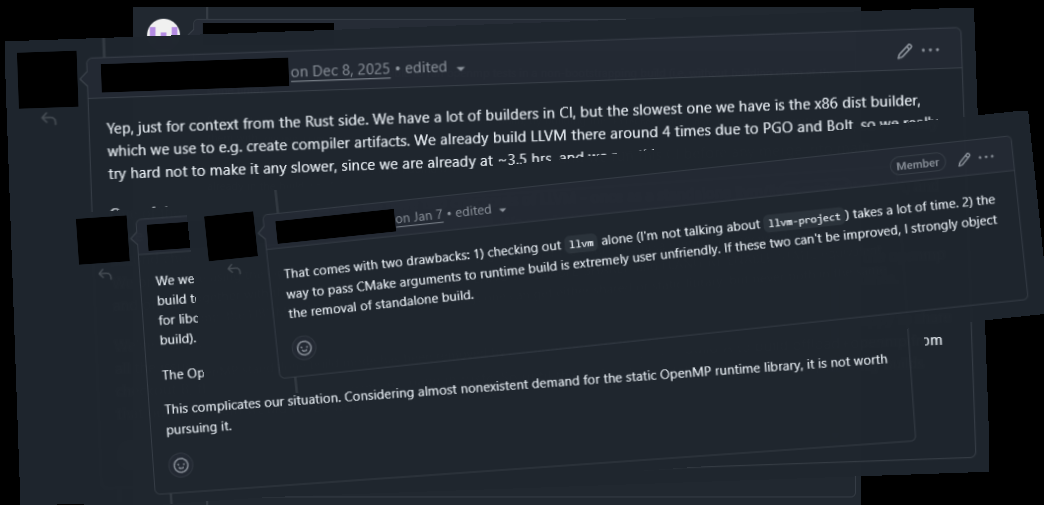
We only build clang as part of our LLVM Build A. However, `rustc` is based on LLVM B, and as such we'd also want the `openmp` and `offload` libraries to also be part of our LLVM Build B, which does not build the in-tree clang to save compile times.

We've tried for quite a bit to get the runtimes to build despite no in-tree clang or `lld` in LLVM B, and failed, I'm happy to share all the issues I ran into there. A standalone build instead allows me to use clang from build A to build `offload+openmp` from checkout B, that's compatible with what we have in `rustc`. So until that is fixed, standalone builds are likely the only builds that our infra team would be ok with.

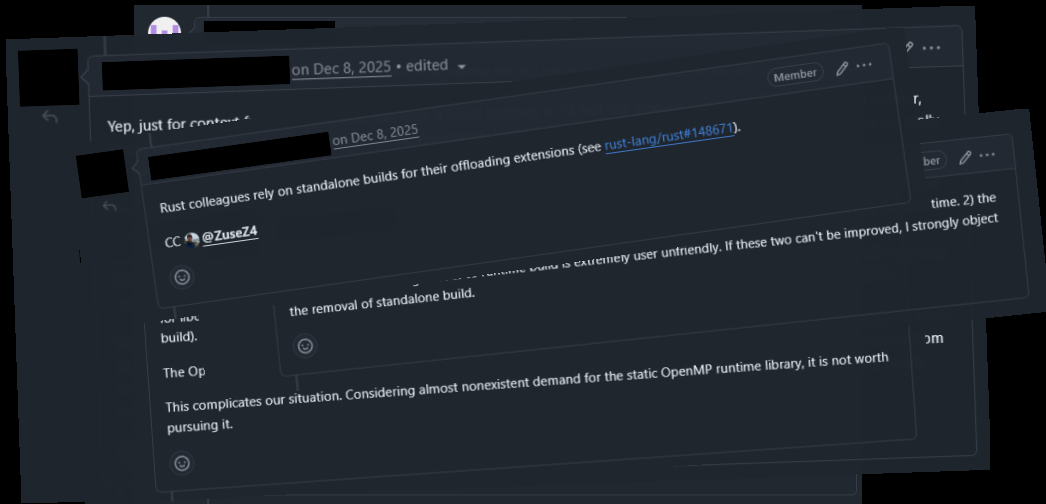
Why This Tutorial



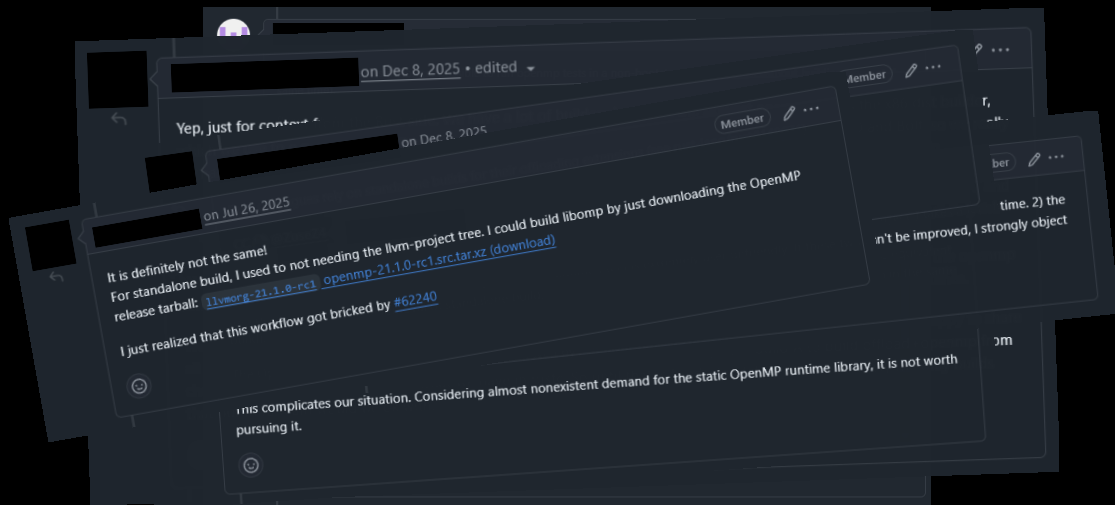
Why This Tutorial



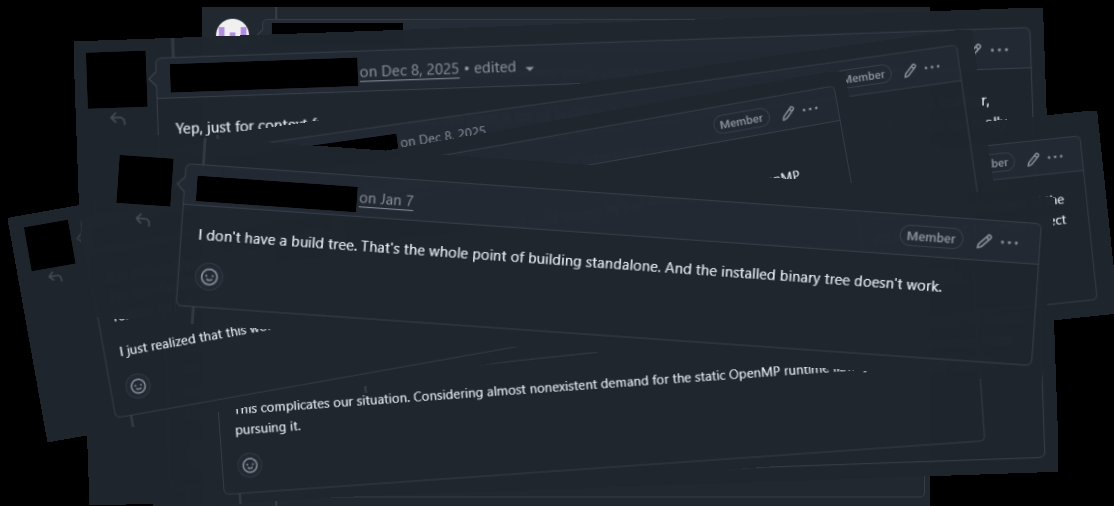
Why This Tutorial



Why This Tutorial



Why This Tutorial



History

1. Initially, each subproject had its own SVN repository
 - ▶ Subprojects categorized as “tools” (clang: 2009), “runtime” (libprofile), and “projects” (e.g., llvm-gcc)
 - ▶ An example project boilerplate existed

¹<https://reviews.llvm.org/D20992>

²<https://lists.llvm.org/pipermail/llvm-dev/2020-October/145997.html>

³<https://reviews.llvm.org/D26365>

⁴<https://reviews.llvm.org/D132478>

History

1. Initially, each subproject had its own SVN repository
 - ▶ Subprojects categorized as “tools” (clang: 2009), “runtime” (libprofile), and “projects” (e.g., llvm-gcc)
 - ▶ An example project boilerplate existed
2. “runtime” subdirectory disappears with dysfunct PGO removal (2013)

¹<https://reviews.llvm.org/D20992>

²<https://lists.llvm.org/pipermail/llvm-dev/2020-October/145997.html>

³<https://reviews.llvm.org/D26365>

⁴<https://reviews.llvm.org/D132478>

History

1. Initially, each subproject had its own SVN repository
 - ▶ Subprojects categorized as “tools” (clang: 2009), “runtime” (libprofile), and “projects” (e.g., llvm-gcc)
 - ▶ An example project boilerplate existed
2. “runtime” subdirectory disappears with dysfunct PGO removal (2013)
3. Desire to separate runtimes^{1,2} (2016–2020)
 - ▶ Somewhat different from compile-time utils/tools/compiler

¹<https://reviews.llvm.org/D20992>

²<https://lists.llvm.org/pipermail/llvm-dev/2020-October/145997.html>

³<https://reviews.llvm.org/D26365>

⁴<https://reviews.llvm.org/D132478>

History

1. Initially, each subproject had its own SVN repository
 - ▶ Subprojects categorized as “tools” (clang: 2009), “runtime” (libprofile), and “projects” (e.g., llvm-gcc)
 - ▶ An example project boilerplate existed
2. “runtime” subdirectory disappears with dysfunct PGO removal (2013)
3. Desire to separate runtimes^{1,2} (2016–2020)
 - ▶ Somewhat different from compile-time utils/tools/compiler
4. Introduction of LLVM_ENABLE_PROJECTS³ (2016)
5. Introduction of LLVM_ENABLE_RUNTIMES⁴ (2022)
 - ▶ First for libc++

¹<https://reviews.llvm.org/D20992>

²<https://lists.llvm.org/pipermail/llvm-dev/2020-October/145997.html>

³<https://reviews.llvm.org/D26365>

⁴<https://reviews.llvm.org/D132478>

Rationale

Why LLVM_ENABLE_RUNTIMES?

Build Host	Cross-Compile Target	Compiler Target
The system we build on ▶ cmake, ninja, cc, ... ▶ llvm-tblgen, mlir-tblgen, ...	Where Clang will run ▶ clang, opt, lld, ... ▶ LLVMSupport{.a/.so}, LLVMPolly.so, ...	What Clang will compile for ▶ a.out ▶ compiler-rt, libc, libc++, ...

- ▶ Automate multiple build-steps
 - ▶ Instead of: Build clang → build compiler-rt for x86_64 → build compiler-rt for arm64 → ...
- ▶ Reduce per-runtime boilerplate/differences between build modes
- ▶ Runtimes compiled (Stage 2) using just-built Clang

Rationale

Why LLVM_ENABLE_RUNTIMES?

Build Host	Cross-Compile Target	Compiler Target
The system we build on ▶ cmake, ninja, cc, ... ▶ llvm-tblgen, mlir-tblgen, ...	Where Clang will run ▶ clang, opt, lld, ... ▶ LLVMSupport{.a/.so}, LLVMPolly.so, ...	What Clang will compile for ▶ a.out ▶ compiler-rt, libc, libc++, ...

Stage 1

Stage 2

- ▶ Automate multiple build-steps
 - ▶ Instead of: Build clang → build compiler-rt for x86_64 → build compiler-rt for arm64 → ...
- ▶ Reduce per-runtime boilerplate/differences between build modes
- ▶ **Runtimes compiled (Stage 2) using just-built Clang**

Tutorial Runtime

What is our tutorial runtime going to do?

Our Runtime

```
void _dublin_hello_world() {  
    puts("Dia daoibh, a dhomhain!\n");  
}
```

- ▶ Use LLVM™ best-practices
- ▶ Linux® only
- ▶ Prioritize conciseness and principles over
 - ▶ ... features
 - ▶ ... comments
 - ▶ ... unrelated boilerplate
- ▶ Point out shortcomings of the current system

Outline

Introduction

Let's Create a Runtime

Setup

Step 1: Register with the LLVM™ Build System

Step 2: Build a Library Artifact

Step 3: An Example That Uses the Library

Step 4: Build Modes

Step 5: CMake Cache Files

Step 6: Artifact Output Location

Step 7: Installation

Step 8: Shared and Static Libraries

Step 9: Regression Testing

Step 10: Unittests

Step 11: Sphinx Docs

Step 12: Doxygen Docs

Step 13: Cross-Compilation

Step 14: Accelerator Offloading

Step 15: Depending on Other LLVM™ Libraries / C++

Are We Done Yet?

Setup

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-0-main>

📄 README.md

- ▶ Update Readme for tutorial

📄 .gitignore

- ▶ Make git ignore install* subdirs
(like build* already is)

```
$HOME/bin/cmake
```

```
/usr/bin/cmake -G Ninja \
-D CMAKE_C_COMPILER_LAUNCHER=ccache \
-D CMAKE_CXX_COMPILER_LAUNCHER=ccache \
-D CMAKE_DISABLE_PRECOMPILE_HEADERS=ON \
"$@"
```

Register with the LLVM™ Build System

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-1-basic>

📄 `llvm/CMakeLists.txt`

📄 `runtimes/CMakeLists.txt`

▶ `Allow LLVM_ENABLE_RUNTIME=dublin`

📄 `dublin/CMakeLists.txt`

📄 `dublin/CODE_OWNERS.TXT`

📄 `dublin/LICENSE.TXT`

▶ Subproject boilerplate

Step 2: Build a Library Artifact

- 📄 `lib/Dublin/dublin.c`
- 📄 `include/dublin/Dublin/dublin.h`
 - ▶ Library sources
- 📄 `lib/Dublin/CMakeLists.txt`
 - ▶ Build instructions for `libdublin.a`

Standard Subproject Layout

```
subproject
├── docs
│   └── *.rst, *.md
├── examples
│   └── Example1
│       └── *.c, *.cpp
├── test
│   ├── Unit
│   │   └── lit.cfg.py, lit.site.cfg.py.in
│   ├── Component1
│   │   └── *
│   ├── Component2
│   │   └── *
│   └── lit.cfg.py, lit.site.cfg.py.in
└── unittests
    ├── Component1
    │   └── Component1Test.cpp
    └── Component2
        └── Component1Test.cpp
```

```
├── cmake
│   ├── Modules/modules
│   │   └── *.cmake
│   └── caches
│       └── *.cmake
├── include
│   └── subproject
│       ├── Component1
│       │   └── component1.h
│       └── Component2
│           └── component2.h
├── lib
│   ├── Component1
│   │   ├── component1.c
│   │   └── component1-internal.h
│   └── Component2
│       └── component2.c
└── utils
    └── utility
        └── utility.cpp
```

An Example That Uses the Library

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-3-example>

📄 `examples/dublin-hello/dublin-hello.c`

▶ Example source

📄 `examples/dublin-hello/CMakeLists.txt`

▶ Example build instructions

Common What To Build Options

- ▶ `DUBLIN_INCLUDE_<component>`
 - ▶ Executes `add_subdirectory(<component>)`
 - ▶ `ninja <component>` can be used
 - ▶ Defaults to `LLVM_INCLUDE_<component>`
- ▶ `DUBLIN_BUILD_<component>`
 - ▶ `ninja all` also builds `<component>`
 - ▶ Generally, steers which target use `ALL/EXCLUDE_FROM_ALL`
 - ▶ Defaults to `LLVM_BUILD_<component>`

Build Modes

 CMakeLists.txt

- ▶ Detect unsupported legacy build modes

Build Modes

Legacy Build Modes

As project: `cmake -S <llvm-project>/llvm -D LLVM_ENABLE_PROJECTS=dublin`

Standalone: `cmake -S <llvm-project>/dublin`

- ▶ Intended to be superseded by runtimes build modes¹
- ▶ If your build infrastructure still uses this, please consider updating
- ▶ Even “standalone” mode needs files from sibling directories
A consequence of the monorepository

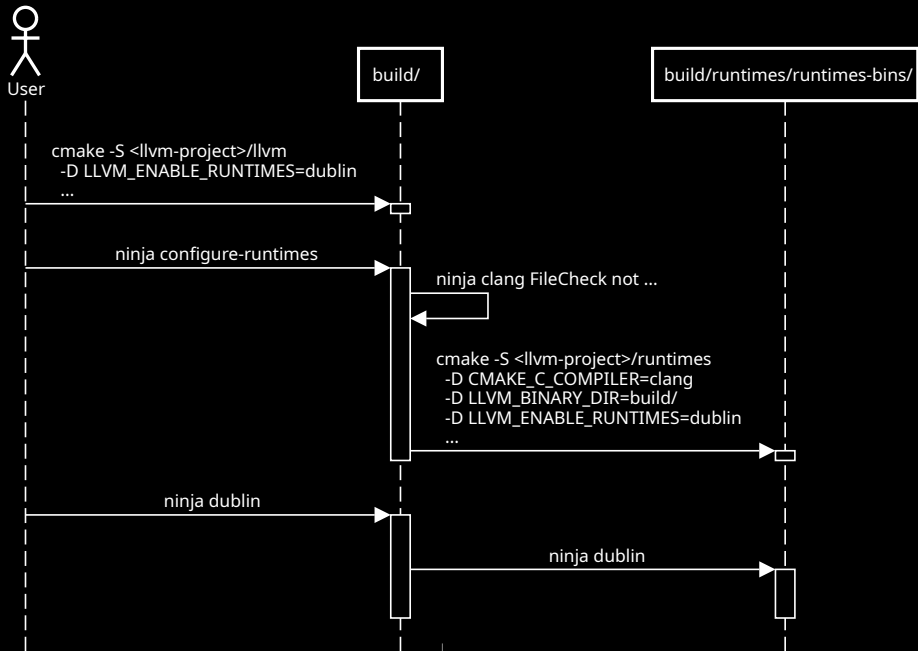
Runtimes Build Modes

Bootstrapping: `cmake -S <llvm-project>/llvm -D LLVM_ENABLE_RUNTIMES=dublin`

Default/Standalone: `cmake -S <llvm-project>/runtimes -D LLVM_ENABLE_RUNTIMES=dublin`

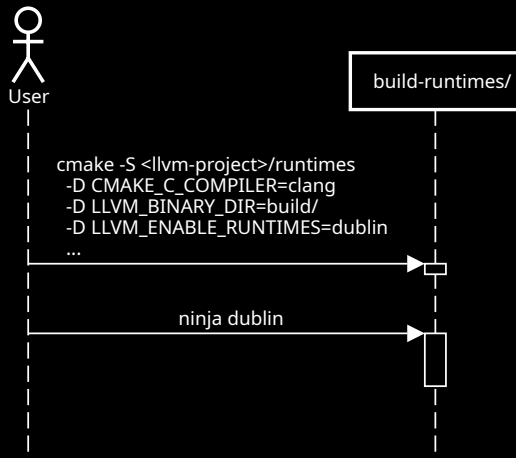
- ▶ Shared boilerplate code in `<llvm-project>/runtimes/CMakeLists.txt`

Bootstrapping Runtimes Build Mode



Default/Standalone Runtimes Build Mode

Skip the Middleman



Default/Standalone Runtimes Build Mode

```
runtimes/CMakeLists.txt
```

```
project(Runtimes C CXX ASM)
```

```
find_package(LLVM HINT ${LLVM_BINARY_DIR})
```

```
# more boilerplate code
```

```
foreach (runtime ${LLVM_ENABLE_RUNTIMES})
```

```
    add_subdirectory(${runtime})
```

```
endforeach ()
```

CMake Cache Files

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-5-cachefiles>

 `cmake/caches/runtimes-bins.cmake`

- ▶ Pre-populated CMake parameters as a bootstrapping build would set them

Artifact Output Location

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-6-location>

- 📄 CMakeLists.txt, cmake/Modules/GetToolchainDirs.cmake
 - ▶ Determine standard locations
- 📄 lib/Dublin/CMakeLists.txt
 - ▶ Set artifact output location

Library Locations

LLVM_ENABLE_PER_TARGET_RUNTIME_DIR=ON

Installation Prefix Dir

```
$PREFIX
├── bin
│   └── clang
└── lib
    └── <target triple>
        ├── *.so, *.a
```

- Accessible to any compiler when installed (including e.g., gcc)

Resource Dir

```
$PREFIX
├── bin
│   └── clang
└── lib
    └── clang
        ├── <llvm version>
        │   └── lib
        │       └── <target triple>
        │           ├── *.a
```

- Accessible to only this version of Clang (and Flang)

Installation

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-7-install>

📄 CMakeLists.txt

- ▶ Set install output location

📄 lib/Dublin/CMakeLists.txt

- ▶ Library installation

📄 include/CMakeLists.txt

- ▶ Header installation

Shared and Static Libraries

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-8-shared>

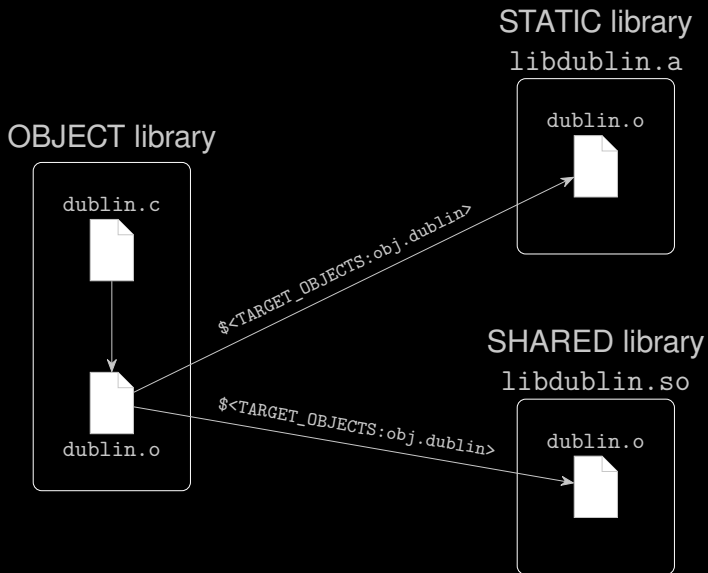
📄 `cmake/Modules/AddDublin.cmake`

- ▶ If building Shared and Static library, use an OBJECT library
- ▶ Wrapper around CMake's native `add_library`
- ▶ AddLLVM's `llvm_add_library` main service is linking to LLVM™ libraries
Target triple confusion. Do not use.

📄 `lib/Dublin/CMakeLists.txt`

- ▶ Use `AddDublin.cmake` instead of `add_library`

OBJECT Libraries



Regression Testing

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-9-test>

📄 `dublin/test/Dublin/dublin-hello.c`

- ▶ The regression test:
 - ▶ Build the source
 - ▶ Link to `libdublin(.a/.so)`
 - ▶ Run
 - ▶ Expect output: “Dia daoibh, a dhomhain!”

📄 `dublin/test/CMakeLists.txt`

- ▶ Testing boilerplate

📄 `dublin/test/lit.cfg.py`

📄 `dublin/test/lit.site.cfg.py.in`

- ▶ `llvm-lit` configuration
 - ▶ How to find regression tests (`lit.formats.ShTest`)
 - ▶ Setup execution environment
 - ▶ Forward build configuration

Unittests

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-10-unittest>

📄 `unittests/Dublin/DublinTest.cpp`

▶ Google Test-based test code

📄 `test/Unit/lit.cfg.py`

📄 `test/Unit/lit.site.cfg.py.in`

▶ `llvm-lit` configuration

How to find unittests (`lit.formats.GoogleTest`)

Sphinx Docs

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-11-docs>

- docs/index.md
 - ▶ The documentation
- docs/conf.py
- docs/CMakeLists.txt
 - ▶ Sphinx configuration
- docs/_theme
 - ▶ HTML boilerplate

Doxygen Docs

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-12-doxygen>

📄 `lib/Dublin/dublin.c`

- ▶ Add Doxygen comment

📄 `docs/doxygen-mainpage.dox`

- ▶ Doxygen frontpage content

📄 `docs/doxygen.cfg.in`, `docs/CMakeLists.txt`

- ▶ Doxygen configuration

Cross-Compilation

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-13-cross>

No modifications necessary !

Cross-Compilation

- ▶ `-D LLVM_RUNTIME_TARGETS="default;aarch64-linux-gnu"`

Compile runtimes also for arm64

- ▶ `-D RUNTIMES_CMAKE_ARGS=-DLLVM_USE_LINKER=lld`

Ubuntu's default `ld.bfd` does not support arm64

Cross-Compilation

```
$builddir/  
├─ runtimes/  
│   ├─ runtimes-bins/  
│   │   └─ CMakeCache.txt  
│   └─ runtimes-aarch64-linux-gnu-bins/  
│       └─ CMakeCache.txt
```

- ▶ LLVM_RUNTIME_TARGETS=default
 - ▶ $\approx$ x86_64-unknown-linux-gnu
 - ▶ Executable on host
 - ▶ NOT necessarily same ABI as LLVM™ (e.g., incompatible LLVMSupport.a)
- ▶ LLVM_RUNTIME_TARGETS=aarch64-linux-gnu
 - ▶ Cross-compilation target

Accelerator Offloading

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-14-offload>

📄 CMakeLists.txt

📄 cmake/Modules/AddDublin.cmake

▶ Accommodate limitations of GPGPU targets

📁 examples/dublin-openmp

📁 examples/dublin-hip

📁 examples/dublin-cuda

▶ Use of GPGPU-side libdublin.a from various offload languages

Depending on Other LLVM™ Libraries / C++

<https://github.com/Meinersbur/llvm-dublin/tree/dublin-15-deps>

- 📄 lib/Dublin/dublin.cpp
- 📄 include/dublin/Dublin/dublin.h
- 📄 examples/dublin-hello/dublin-hello.cpp
 - ▶ Convert to C++
- 📄 cmake/Modules/AddDublin.cmake
- 📄 ../third-party/unittest/CMakeLists.txt
 - ▶ Link to libc++, regardless of compiler default

Compiler-RT Builtins



Outline

Introduction

Let's Create a Runtime

Are We Done Yet?

More Topics

Build System Improvements

Not Covered Topics

- ▶ Platform Introspection

`check_source_compiles`

- ▶ Platform Support

Windows®/MacOS/AIX/Fuchsia/...

- ▶ Multilib

`lib32/lib64`

- ▶ Versioning

`libdublin.a.1.1`

- ▶ Export libraries

`find_package(Dublin)`

- ▶ Continuous Integration

- ▶ Fortran

- ▶ ...

Potential Runtimes Build Improvements

- ▶ Inform LLVM™ developers and OS distributions
(Build modes) developer documentation missing, runtime template

Potential Runtimes Build Improvements

- ▶ Inform LLVM™ developers and OS distributions
(Build modes) developer documentation missing, runtime template
- ▶ Make RUNTIMES_* options discoverable
`set(RUNTIMES_CMAKE_ARGS "" CACHE STRING "Help")`

Potential Runtimes Build Improvements

- ▶ Inform LLVM™ developers and OS distributions
(Build modes) developer documentation missing, runtime template
- ▶ Make RUNTIMES_* options discoverable
`set(RUNTIMES_CMAKE_ARGS "" CACHE STRING "Help")`
- ▶ Unify the CMake code of all runtimes
Boilerplate diverged significantly since the SVN days

Potential Runtimes Build Improvements

- ▶ Inform LLVM™ developers and OS distributions
(Build modes) developer documentation missing, runtime template
- ▶ Make RUNTIMES_* options discoverable
`set(RUNTIMES_CMAKE_ARGS "" CACHE STRING "Help")`
- ▶ Unify the CMake code of all runtimes
Boilerplate diverged significantly since the SVN days
- ▶ Fat libraries support
MacOS Universal Binaries, arm64ec, Offload, ...

Potential Runtimes Build Improvements

- ▶ Inform LLVM™ developers and OS distributions
(Build modes) developer documentation missing, runtime template
- ▶ Make RUNTIMES_* options discoverable
`set(RUNTIMES_CMAKE_ARGS "" CACHE STRING "Help")`
- ▶ Unify the CMake code of all runtimes
Boilerplate diverged significantly since the SVN days
- ▶ Fat libraries support
MacOS Universal Binaries, arm64ec, Offload, ...
- ▶ Avoid special cases
 - ▶ GPGPU targets
 - ▶ LLVM_RUNTIME_TARGETS=default
 - ▶ Compiler-RT “builtins”

Potential Runtimes Build Improvements

- ▶ Inform LLVM™ developers and OS distributions
(Build modes) developer documentation missing, runtime template
- ▶ Make RUNTIMES_* options discoverable
`set(RUNTIMES_CMAKE_ARGS "" CACHE STRING "Help")`
- ▶ Unify the CMake code of all runtimes
Boilerplate diverged significantly since the SVN days
- ▶ Fat libraries support
MacOS Universal Binaries, arm64ec, Offload, ...
- ▶ Avoid special cases
 - ▶ GPGPU targets
 - ▶ LLVM_RUNTIME_TARGETS=default
 - ▶ Compiler-RT “builtins”
- ▶ Writing runtimes in C++
Option whether to use default stdlib or libc++

Potential Runtimes Build Improvements

- ▶ Inform LLVM™ developers and OS distributions
(Build modes) developer documentation missing, runtime template
- ▶ Make RUNTIMES_* options discoverable
`set(RUNTIMES_CMAKE_ARGS "" CACHE STRING "Help")`
- ▶ Unify the CMake code of all runtimes
Boilerplate diverged significantly since the SVN days
- ▶ Fat libraries support
MacOS Universal Binaries, arm64ec, Offload, ...
- ▶ Avoid special cases
 - ▶ GPGPU targets
 - ▶ LLVM_RUNTIME_TARGETS=default
 - ▶ Compiler-RT “builtins”
- ▶ Writing runtimes in C++
Option whether to use default stdlib or libc++
- ▶ Get rid of LLVM_ENABLE_PER_TARGET_RUNTIME_DIR

Potential Runtimes Build Improvements

- ▶ Inform LLVM™ developers and OS distributions
(Build modes) developer documentation missing, runtime template
- ▶ Make RUNTIMES_* options discoverable
`set(RUNTIMES_CMAKE_ARGS "" CACHE STRING "Help")`
- ▶ Unify the CMake code of all runtimes
Boilerplate diverged significantly since the SVN days
- ▶ Fat libraries support
MacOS Universal Binaries, arm64ec, Offload, ...
- ▶ Avoid special cases
 - ▶ GPGPU targets
 - ▶ LLVM_RUNTIME_TARGETS=default
 - ▶ Compiler-RT “builtins”
- ▶ Writing runtimes in C++
Option whether to use default stdlib or libc++
- ▶ Get rid of LLVM_ENABLE_PER_TARGET_RUNTIME_DIR
- ▶ QEMU (CMAKE_CROSSCOMPILING_EMULATOR)
- ▶ ...

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED “AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

LLVM is a trademark of LLVM Foundation. The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board. Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries. Windows is a registered trademark of Microsoft Corporation in the US and/or other countries. Other names used herein are for identification purposes only and may be trademarks of their respective companies

©2026 Advanced Micro Devices, Inc. All rights reserved.

AMD 