Cross compilation with Clang and LLVM tools

Peter Smith Linaro TCWG

Linaro connect
Bangkok 2019

Contents

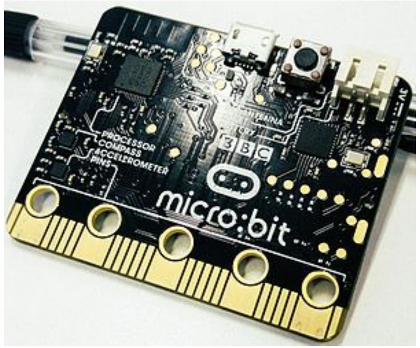
- Refresher on cross compilation.
- Clang/LLVM toolchains.
- Using Clang as a cross compiler.
- Additional cross compilation options.
- Using CMake.
- Differences between GCC and Clang.
- Assembling a Clang toolchain today.

Definitions

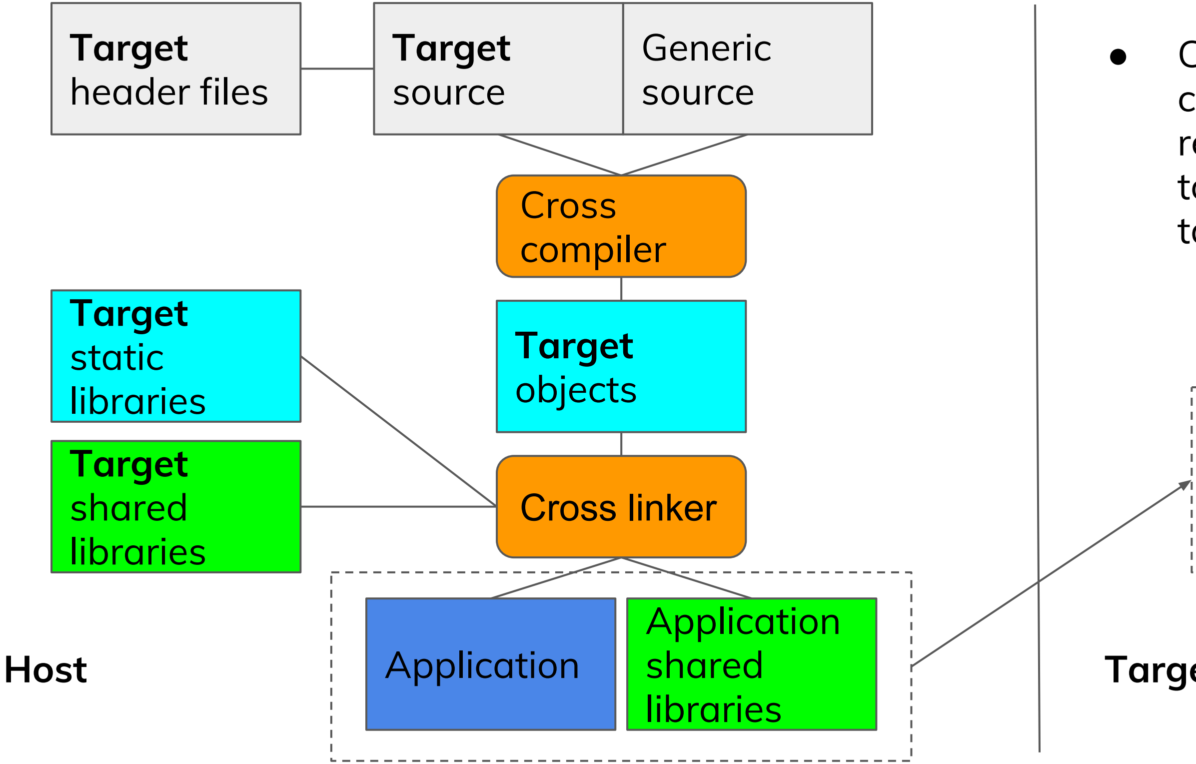
- **Host**
  - The system that we run the development tool on.
- **Target**
  - The system that we run the generated program on.
- **Native** compilation
  - Host is the same as Target.
- **Cross** compilation
  - Host is different to Target.

Motivation for cross compilation

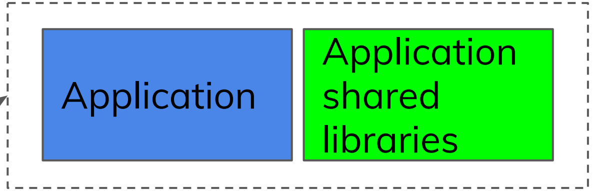
- Productivity boost when host is faster than target.
- Target can't run a C/C++ compiler.
- Need to build program for many architectures.
- Bootstrapping a compiler on a new architecture.



A cross compiled application



- Cross compilation requires a toolchain and target libraries.



Clang/LLVM Toolchains

- What is provided in an installation?
- What is missing?

Cross compilation and the LLVM toolchain

- Clang and other LLVM tools can work with multiple targets from the same host binary.
- Clang and LLD drivers can emulate the drivers of other toolchains.
- Controlled by the target triple.
- LLVM project does not have implementations of all the parts of toolchain.
- LLVM project includes some but not all of the library dependencies.

Toolchain components

Component	LLVM	GNU
C/C++ Compiler	`clang`	`gcc`
Assembler	`clang integrated assembler`	`as`
Linker	`ld.lld`	`ld.bfd, ld.gold`
Runtime	`compiler-rt`	`libgcc`
Unwinder	`libunwind`	`libgcc_s`
C++ library	`libc++abi, libc++`	`libsupc++, libstdc++`
Utils such as archiver	`llvm-ar, llvm-objdump etc.`	`ar, objdump etc.`
C library		`glibc, newlib`

Toolchain components used by Clang

- Defaults chosen at build time, usually favour GNU libraries for Linux targets, otherwise LLVM.
- Compiler runtime library
  - `--rtlib=compiler-rt`, `--rtlib=libgcc`.
  - compiler-rt needed for sanitizers but these are separate from builtins provided by libgcc.
- C++ library
  - `--stdlib=libc++`, `--stdlib=libstdc++`.
  - No run-time option to choose C++ ABI library, determined at C++ library build time.
- Linker
  - `-fuse-ld=lld`, `-fuse-ld=bfd`, `-fuse-ld=gold`.
  - Driver calls ld.lld, ld.bfd, ld.gold respectively.
- C-library choice can affect target triple
  - For example arm-linux-gnueabi, arm-linux-musleabi.

Using Clang as a Cross Compiler

- Deconstructing the Target Triple.
- Using a GCC toolchain to provide missing components.
- The Clang driver.
- How a Clang installation is laid out.

Target Triple

- General format of <**Arch**><**Sub-arch**>-<**Vendor**>-<**OS**>-<**Environment**>
  - **Arch** is the architecture that you want to compile code for
    - Examples include arm, aarch64, x86_64, mips.
  - **Sub-arch** is a refinement specific to an architecture
    - Examples include armv7a armv7m.
  - **Vendor** captures differences between toolchain vendors
    - Examples include Apple, PC, IBM.
  - **OS** is the target operating system
    - Examples include Darwin, Linux, OpenBSD, none.
  - **Environment** includes the ABI and object file format
    - Examples include android, elf, gnu, gnueabihf.
- Missing parts replaced with “unknown”.

Arm Target Triple

- <**Arch**>
  - arm or thumb. The -marm and -mthumb take precedence.
- <**Sub-arch**>
  - Accepts several forms, v7a, v7-a. -march, -mcpu take precedence.
- <**OS**>
  - linux, android, none (for bare metal)
- <**Environment**>
  - gnueabi, gnueabihf, eabi, eabihf, musleabi, musleabihf
    - hf is for hard-float, -mfloat-abi takes precedence.

Arm Target Triple examples

- **`thumbv8-m.mainline-none-eabi`**
  - bare metal soft-floating point Armv8-m.mainline.
  - For M-class, thumb is always selected even if arm is in the triple.
- **`armv7a-linux-gnueabihf`**
  - Linux, hard-floating point Armv7-a)
- **`arm-linux-gnueabi -march=armv7-a -mfloat-abi=softfp`**
  - Linux, soft-floating calling convention, Armv7-a

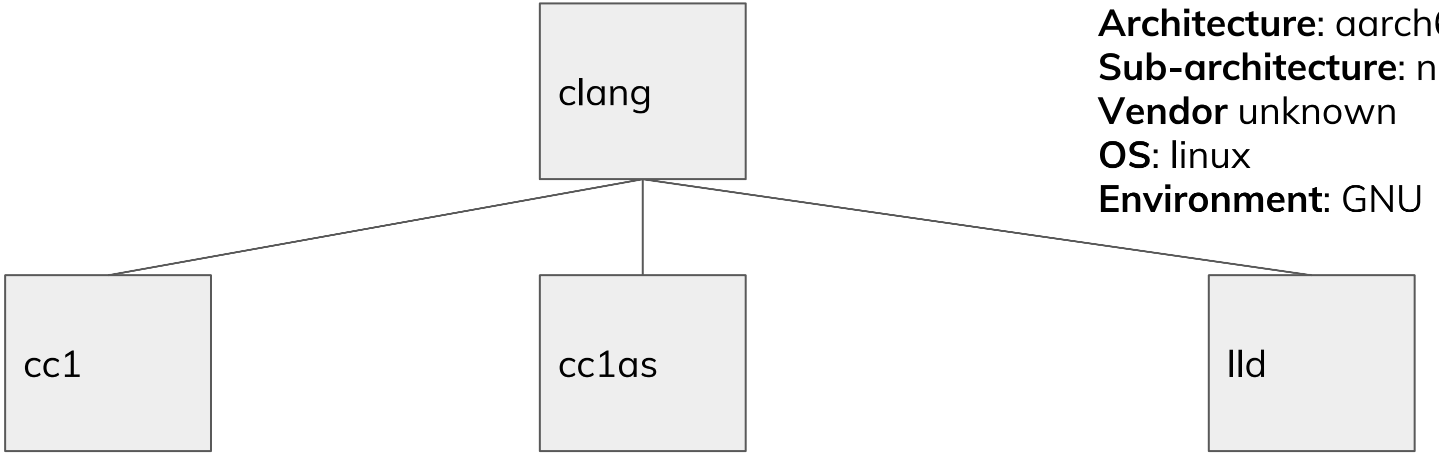
Clang driver and toolchains

- Driver mode
  - gcc, g++, cpp (preprocessor), cl (MSVC).
  - Set with option or inferred from filename clang, clang++, clang-cl.
- Target triple used to instantiate a ToolChain derived class
  - `arm-linux-gnueabihf` instantiates the Linux ToolChain (Linux.cpp, Gnu.cpp).
  - `arm-none-eabi` instantiates the bare metal ToolChain (BareMetal.cpp).
- Toolchain class has knowledge of how to emulate the native toolchain
  - Include file locations.
  - Library locations.
  - Constructing linker and non integrated assembler options.
  - Includes cross compilation emulation.
- Not all functionality is, or could realistically be, documented.

Clang Driver

```
clang --target=aarch64-linux-gnu func1.s hello.c -o hello
```

Architecture: aarch64
Sub-architecture: not applicable
Vendor unknown
OS: linux
Environment: GNU




```
clang-7.0 -cc1 -triple
aarch64-linux-gnu
-target-cpu=generic
-target-feature +neon
-target-abi aapcs
-Isystem /path/to/includes
...
```

```
clang-7.0 -cc1as -triple
aarch64-linux-gnu
-target-cpu=generic
-target-feature +neon
...
```

```
ld.lld -m aarch64linux
-dynamiclinker
/lib/ld-linux-aarch64.so
-L /path/to/system/libraries
-lc
...
```

Anatomy of a Clang installation

- clang+llvm-7.0.1-x86_64.linux-gnu-ubuntu-16.04
  - `bin` (host user and developer tools such as clang, lld, llvm-mc)
  - `include`
    - `c++` (libc++)
    - `clang`, `clang-c`, `llvm`, `llvm-c` (for compiling against llvm as a library)
  - `lib`
    - `libc++`, `libc++abi`, `libgomp`, (for user programs, not namespaced by target)
    - `libclang`, `libllvm`, (for linking against llvm as a library)
    - `clang`
      - `7.0.0`
        - `include`
          - stdint.h, arm_acle.h, arm_neon.h and other compiler specific headers
          - sanitizers, xray subdirs
        - `lib`
          - `linux` (for OS=linux, can also be baremetal)
            - `libclang_rt.builtins-`**`x86_64`**`.a` (and other compiler-rt libs)
        - `share`
          - `asan_blacklist.txt` (and other sanitizer blacklists)
  - `share` scripts, man pages etc.

resource dir

compiler-rt

Using a Clang installation for cross compiling

- All the host tools will be able to cross compile for Arm
  - Assuming toolchain builder didn't exclude it from build.
- The libc++, libc++abi and all the LLVM/Clang libraries will be for the host.
  - Cannot use libc++ with the default library paths.
  - Linux defaults to libstdc++
- The compiler-rt library will also be for the host
  - It has the target in the name so it is possible to install Arm versions
  - Linux defaults to libgcc
- The remainder of the libraries will need to be found outside the installation
  - Linux distributions multiarch.
  - A separate GCC installation.

Finding the location of a GCC cross compiler

- A bunch of heuristics guided by two relevant command-line options
  - `--gcc-toolchain`
  - `--sysroot`
- Clang looks for `lib/gcc/<gcc-triple>/major.minor.patch` and `lib/gcc-cross/<gcc-triple>/major.minor.patch`
  - Prefixed first with `--gcc-toolchain`, then `--sysroot`.
  - The highest GCC version found is chosen.
  - GCC toolchain components assumed to be relative to this location.
- Includes and libraries searched for in `sysroot/usr`
- The Arm and Linaro toolchains need both `--gcc-toolchain` and `--sysroot`
  - `--gcc-toolchain=/path/to/install-dir`
  - `--sysroot=/path/to/install-dir/<gcc-triple>/libc`

Example multiarch on Ubuntu 16.04

```
$ clang hello.c --target=arm-linux-gnueabihf -o hello
$ qemu-arm -L /usr/arm-linux-gnueabihf hello
Hello World
$ clang hello.c -v --target=arm-linux-gnueabihf -o hello
Found candidate GCC installation: /usr/lib/gcc-cross/arm-linux-gnueabihf/5.4.0
Selected GCC installation: /usr/lib/gcc-cross/arm-linux-gnueabihf/5.4.0
ignoring nonexistent directory "/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /path/to/clang/lib/clang/9.0.0/include
 /usr/include/arm-linux-gnueabihf
 /usr/include
End of search list.
…
```

Potential for host pollution

Example aarch64-linux-gnu toolchain

```
$ clang hello.c --target=aarch64-linux-gnu -o hello
--gcc-toolchain=/work/gcc7a64 --sysroot=/work/gcc7a64/aarch64-linux-gnu/libc
$ qemu-aarch64 -L /work/gcc7a64/aarch64-linux-gnu/libc hello
Hello World
clang hello.c --target=aarch64-linux-gnu -o hello -v
--gcc-toolchain=/work/gcc7a64 --sysroot=/work/gcc7a64/aarch64-linux-gnu/libc
Found candidate GCC installation: /work/gcc7a64/lib/gcc/aarch64-linux-gnu/7.1.1
Selected GCC installation: /work/gcc7a64/lib/gcc/aarch64-linux-gnu/7.1.1
ignoring nonexistent directory "/work/gcc7a64/aarch64-linux-gnu/libc/usr/local/include"
ignoring nonexistent directory "/work/gcc7a64/aarch64-linux-gnu/libc/include"
#include "..." search starts here:
#include <...> search starts here:
 /path/to/clang/build/lib/clang/9.0.0/include
 /work/gcc7a64/aarch64-linux-gnu/libc/usr/include
End of search list.
```

Limitations of Clang’s Driver

- No support for specs files
  - Clang has configuration files that can be used to partially emulate.
- No support for Linux or bare metal multilib
  - Android multilib support is available.
- Heuristics to find GCC installation are opaque and incomplete
  - Will work with well known distros.
- With multiarch there is a danger of host include pollution
  - `/usr/local/include` and `/usr/include` on constructed include path.
- Not easy to use libc++ when using `--gcc-toolchain`, `--sysroot`
  - These are in Clang’s include and lib dir, not GCC’s.
  - Need to either specify include and library dirs, or copy libc++ to GCC installation.

Additional cross compilation options

- Using configuration files.
- Using libc++ and compiler-rt.
- bare metal for embedded systems.

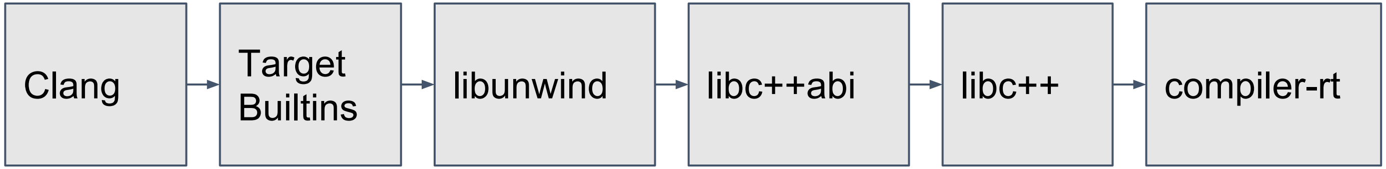
Using Clang configuration files

- Can be used to emulate some of the features of specs files
- `clang --config <config file>`
    - Contain command line options.
    - Can include other configuration files.
- Search path for config file can be set at build time
    - User dir: `-DCLANG_CONFIG_USER_DIR`
    - System dir: `-DCLANG_CONFIG_SYSTEM_DIR`
    - Directory where clang executable resides.
- Can select a config file automatically via renaming/symlinking clang
    - `armv7l-clang` is equivalent to `clang --config armv7l.cfg`
- No conditional features like in specs file
    - Would need a configuration file for each combination of choices.
    - For example `armv6-m-rdimon-nano-clang`
    - Could make individual config files that could be included.

Using libc++, compiler-rt

- So far we've been using a Clang installation with cross-tools but host LLVM libraries
    - Our libraries have come from a GCC cross toolchain.
    - Will need compiler-rt to use of sanitizers.
- Need to obtain target builds of LLVM libraries
    - Rebuild them from source.
    - Copy from a native Target Clang installation.
- Building from source is the only way to eliminate any dependencies on GNU libraries such as libgcc and libunwind.
- Using libc++ when cross compiling requires us to manually set the C++ include path to avoid conflicts with libstdc++.

Building Compiler-rt and Libc++ from source



1. Clang is needed for builtins and sanitizers, can skip if you already have a recent Clang.
2. The builtins are the equivalent of libgcc, a static library. These are a small part of compiler-rt that we need to build first.
3. The LLVM unwinder may use builtins from libunwind, needs C++ header from libc++
4. The libc++abi C++ runtime uses the unwinder from LLVM unwind
5. The libc++ C++ standard library uses libc++abi
6. The remainder of compiler-rt includes the sanitizers, which need libc++

A C++ example using libc++ and sanitizers

- Test is a slightly modified version of the ubsan example
  - Modified to throw an exception.
- We want to use as much of the LLVM libraries as possible
  - `compiler-rt`
  - `libc++`, `libc++abi`, `libunwind`

```
#include <iostream>
#include <string>

int func(void) {
    throw std::string("Hello World\n");
    return 0;
}

int main(int argc, char** argv) {
    try {
        func();
    } catch (std::string& str) {
        std::cout << str;
    }
    int k = 0x7fffffff;
    k += argc; //signed integer overflow
    return 0;
}
```

A C++ example using libc++ and sanitizers

```
$root=/path/to/clang/install_dir
$gcctoolchain=/path/to/gcc
$sysroot=${gcctoolchain}/aarch64-linux-gnu/libc
$ ${root}/bin/clang++ --target=aarch64-linux-gnu -fsanitize=undefined \
                      --rtlib=compiler-rt --stdlib=libc++ \
                      -nostdinc++ -I${root}/include/c++/v1 \
                      -Wl,-L${root}/lib \
                      --sysroot ${sysroot} \
                      --gcc-toolchain=${gcctoolchain} \
                      -rpath ${root}/lib \
                      example.cpp -o example

$ qemu-aarch64 -L ${sysroot} example
Hello World
example.cpp:16:7: runtime error: signed integer overflow: 2147483647 + 1 cannot be
represented in type 'int'
```

Cross compiling for embedded systems

- Clang has a bare metal driver for Arm that is selected with `--target=arm-none-eabi`
- The functionality is somewhat bare
  - Setup include paths for libc++.
  - A kind of multiarch for the compiler-rt builtins.
  - Implicitly adding `-lc -lm`
  - Defaults to LLD, compiler-rt and the integrated assembler.
- Limitations
  - No support for finding an arm-none-eabi-gcc toolchain.
  - No support for specs files or multilib.
  - LLD support for linker scripts and embedded features good but not perfect.
- May need to use GNU supporting tools arm-none-eabi-gcc-objcopy
  - There are LLVM equivalents but they are not drop in replacements.
- A skeleton that you have to provide all the details yourself
  - Functional but not user-friendly.

Obtaining compiler-rt builtins

- Unless `--nostdlib` is used, Clang will add -lclang_rt.builtins.<arch>.a
- The only way to get armv6m, armv7m, armv8m versions is to build from source
- Guide available at https://llvm.org/docs/HowToCrossCompileBuiltinsOnArm.html
- Place built libraries into `lib/clang/7.0.0/lib/baremetal` directory

Using CMake

- Providing the target, sysroot and GCC toolchain
- Overriding the linker

Cross compilation with CMake

- Useful CMake options
  - https://cmake.org/cmake/help/v3.14/manual/cmake-toolchains.7.html
  - To skip link stage of trycompile stage
    - -DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY
  - Select Clang as compiler and assembler
    - -DCMAKE_C_COMPILER=clang
    - -DCMAKE_CXX_COMPILER=clang++
    - -DCMAKE_ASM_COMPILER=clang
  - Set --sysroot with -DCMAKE_SYSROOT
  - Set --gcc-toolchain
    - -DCMAKE_C_COMPILER_EXTERNAL_TOOLCHAIN
    - -DCMAKE_CXX_COMPILER_EXTERNAL_TOOLCHAIN
  - Set --target
    - -DCMAKE_C_COMPILER_TARGET
    - -DCMAKE_CXX_COMPILER_TARGET
    - -DCMAKE_ASM_COMPILER_TARGET

Overriding the linker with CMake

- CMake will by default use the compiler driver
- For bare metal it can be useful to use arm-none-eabi-gcc instead of clang
  - Support for specs files
  - More mature linker script support
- Can use CMAKE_C_LINK_EXECUTABLE and CMAKE_CXX_LINK_EXECUTABLE to override.

```
set(CMAKE_C_LINK_EXECUTABLE "/path/to/arm-none-eabi-gcc <FLAGS>
<CMAKE_C_LINK_FLAGS> <LINK_FLAGS> <OBJECTS> -o <TARGET>
<LINK_LIBRARIES> -lc")

set(CMAKE_CXX_LINK_EXECUTABLE "/path/to/arm-none-eabi-g++ <FLAGS>
<CMAKE_C_LINK_FLAGS> <LINK_FLAGS> <OBJECTS> -o <TARGET>
<LINK_LIBRARIES> -lc")
```

Differences between GCC and Clang

- Assembler
- Predefined Macros
- LLD

GCC/Clang assembler differences

- Clang use its integrated assembler by default
  - Uses the same target selection options as Clang.
  - Is fussier on syntax than gas.
  - Does not permit coprocessor syntax for VFP/Neon in Armv7-a and above.
  - Unified Assembly Language (UAL) only.
  - Does not support `.altmacro`
  - Single pass so doesn't support some symbolic calculations.
- Can use -fno-integrated-as to use the GNU assembler
  - Uses heuristics to match target options to gas command line.

GCC/Clang general differences

- Clang will define many, but not all macros that GCC defines.
- In particular it claims to be GCC 4.2.1
  - `__GNUC__ 4`
  - `__GNUC_MINOR__ 2`
  - `__GNUC_PATCHLEVEL__ 1`
- There are Clang equivalents that ideally should be used instead
  - `__clang__ 1`
  - `__clang_major__ 9`
  - `__clang_patchlevel__ 0`
- Can get list via `clang -dM -E - < /dev/null`

LLD Limitations and differences

- LLD defaults to `-ztext` (no dynamic relocations in `.text` section)
- LLD has limited support for `-N` (`--omagic`), and no support for `-n` (`--nmagic`)
    - Can simulate with `-zmax-page-size=1`
- `PT_LOAD` program header generation not identical to ld.bfd
    - Can use PHDRS to select these manually.
- LLD tends to support subset of features that upstream Clang supports.
- LLD Linker Script support is not perfect
    - Some syntax differences "()" for address not allowed.
    - Chain of references
        - `aliasto__text = aliasto__text1;`
        - `aliasto__text1 = aliasto__text2;`
        - `aliasto__text2 = __text;`

Clang/LLVM toolchains today

- How to support Clang in your open source project.
- Assembling a Clang based toolchain today.

Supporting Clang in your open source project

- Rely on the cross-compilation options of the build system
  - Clang/LLVM can be cross-compiled using CMake.
  - Burden on the user to get the options right for their system.
  - Beware hidden host dependencies
    `-DLLVM_TABLEGEN=/path/to/host/llvm-tablegen`
- Supply the toolchain
  - Android NDK, Google Chrome are large enough projects to provide their own toolchains.
    - `chromium/src/third_party/llvm-build/Release+Asserts/bin`
  - Build systems pre-configured with all the paths and include directories.
- Customize Clang with via a new OS, Vendor, Enviroment
  - Conditionally alter existing ToolChain, for example Android modifies Linux.
  - Write a new ToolChain class that is selected via OS, Vendor, Environment
    - Linux, Fuchsia, FreeBSD ...

How could we assemble a Clang toolchain today?

- A linux toolchain akin to the arm-linux-gnueabihf toolchain
  - Arrange the directory structure such that Clang can find it from the sysroot.
  - Set the default sysroot at build time, `DEFAULT_SYSROOT`.
  - Choose either libc++ or libstdc++ so that the includes don't clash.
  - Provide target compiler-rt libraries including sanitizers.
  - LLD as default linker.
- A bare metal embedded toolchain akin to arm-none-eabi toolchain
  - Partition libraries and includes by directory.
  - Provide compiler-rt builtins for the various m-class architectures.
  - Provide config files to replace specs-files and multilib
    - armv6m.rdimon.nano.cfg
  - Essentially provide the necessary command line options for the most common cases.
  - Config files could be separated into separate pieces so that users could construct their own.
  - Would probably need a GNU binutils, with ld.bfd as the default linker.
- In both cases some GNU libraries would need to be compiled with GCC.

Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org

Linaro connect
Bangkok 2019