



arm

Open Source C/C++ embedded toolchains using LLVM

FOSDEM 2023

Peter Smith
February 2023

arm

What makes an embedded toolchain?

Hosted vs Embedded toolchains

Hosted

- + We know what OS programs will run on
 - Can depend OS syscalls to implement library functions.
- + Toolchain is often run on the OS by the user.
- + Platform will often provide the C/C++ libraries rather than the toolchain.
- + Platform will sometimes provide the linker and assembler.
- + Typically all of the standard library will be available.
- + Target a platform interface rather than the specific hardware running on a device.

Embedded toolchains often freestanding

- + No Assumption of an OS
 - C-library may implement part of an OS.
- + Subset of standard library available
 - For example no thread support and no high-resolution timers.
- + Static linking only.
- + Target specific hardware running on the end device.
 - Can ship with many binary library variants.
- + Toolchain is almost never run on the embedded device
 - Cross-compilation.

arm

Why do we want to use
LLVM for an embedded
toolchain?

LLVM project advantages

- + Take advantage of the amount of runtime configuration offered by clang.
 - Clang can be both a hosted native compiler and a freestanding cross-compiler for many targets.
 - Can provide an Arm and AArch64 toolchain in a single package.
- + Code-generation can be more mature than GCC for certain targets
 - Armv8.1-M M-profile Vector Extension (MVE) as included with Cortex-M55.
 - Armv8.1-M Pointer Authentication and Branch Target Identification.
- + Take advantage of clang/llvm features such as CFI and sanitizers.
- + Familiarity with clang and llvm ecosystem.
- + Diversity of implementation
 - Compilers have different warnings and find bugs.

Aside: Using LLVM sanitizers in embedded systems

- + Code-generation is not usually a problem, however there are dependencies on the sanitizer runtime.
- + Some of the sanitizers have an option for a minimal or no runtime
 - Undefined behavior Sanitizer UBSAN
 - Control Flow Integrity Sanitizer (CFI), requires LTO.
 - Kernel Control Flow Integrity Sanitizer (KCFI)
- + Sanitizer uses an undefined instruction to force a trap, or call out to a user-defined function.
 - `-fsanitize=undefined -fsanitize-trap=undefined`

Undefined Behavior Sanitizer UBSAN

Can be run with traps-only or minimal runtime (needs a trivial implementation)

C Code

```
int mul(int x, int y) {  
    return x * y;  
}
```

-fsanitize=undefined – fsanitize-trap=undefined

```
mul:  
    smull    r0, r1, r0, r1  
    cmp.w    r1, r0, asr #31  
    it       eq  
    bxeq     lr  
    .inst.n 0xdefe // undef
```

-fsanitize=undefined – fsanitize-minimal-runtime

```
mul:  
    push     {r4, lr}  
    smull    r4, r0, r0, r1  
    cmp.w    r0, r4, asr #31  
    bne      .LBB0_2  
    mov      r0, r4  
    pop      {r4, pc}  
  
.LBB0_2:  
    bl       __ubsan_handle_mul_overflow_minimal  
    mov      r0, r4  
    pop      {r4, pc}
```

Kernel Control Flow Integrity KFCI

- + A simplified form of the CFI Sanitizer that does not require LTO.
 - -fsanitize=kcfi
- + Functions are given a signature that can be checked when calling via a function pointer
 - Trap if signatures do not match.

```
typedef int Fptr(void);
```

```
int function(void) {  
    return 0;  
}
```

```
int call(Fptr* fp) {  
    return fp();  
}
```

```
.long    0x36b1c5a6 // Signature
```

```
function:
```

```
.fnstart
```

```
movs     r0, #0
```

```
bx       lr
```

```
call:
```

```
ldr      r1, [r0, #-4] // load sig from ptr
```

```
movw     r2, #0xc5a6    // expected sig
```

```
movt     r2, #0x36b1
```

```
cmp      r1, r2
```

```
bne      .LBB1_2
```

```
bx       r0
```

```
.LBB1_2:
```

```
.inst.n 0xdefe // Undef
```


arm

Components of an embedded toolchain

llvm-project nearly there, but missing a C-library

LLVM Component	GNU embedded equivalent	Description
clang, clang++	gcc, g++	Compiler
clang integrated assembler	as	Assembler
ld.lld	ld.bfd, ld.gold	Linker
llvm-objdump, llvm-readelf, ...	objdump, readelf, ...	Binutils
compiler-rt	libgcc	Compiler runtime library
libunwind	libgcc	Unwinder
libc++abi	libsupc++.a	C++ ABI library
libc++	libstdc++	C++ standard library
libc [*]	newlib [**]	C library

- [*] LLVM libc isn't yet suitable for use in embedded systems.
- [**] Newlib isn't part of the GNU project, but there are hooks in the GCC configure scripts for building a toolchain.

Building an LLVM toolchain

- + Not an easy as it should be
 - Building the tools is not difficult.
 - Cross-compiling the runtime libraries for all supported architecture variants is a bigger challenge.
- + Arm has an open-source recipe for building an LLVM embedded toolchain using the picolibc C-library at <https://github.com/ARM-software/LLVM-embedded-toolchain-for-Arm>
 - Primarily build-scripts for Linux and Windows (using mingw).
 - Binaries available for corresponding to LLVM 13, 14 and 15 releases.
 - Supports M-profile architectures along with experimental AArch64 support.
 - Builds with CMake and meson (for Picolibc).
 - Approach likely to be adaptable for other Targets that are supported by LLVM and Picolibc.

Usability compared to a GNU Toolchain

+ Multilib support

- Derive a library path based on input command line options such as march and mcpu.

+ Specs file

- GNU Toolchain uses these to select semihosting and newlib-nano.
- `--specs=nano.specs --specs=rdimon.specs`

+ LLVM Embedded Toolchain for Arm uses clang config files

- Need more of these than equivalent specs files.
- `--config armv6m_soft_nofp_semihost`

+ Long tail of small incompatibilities from existing open source projects:

- Options only supported by GNU tools.
- GNU as vs LLVM integrated assembler.
- Command line defaults for some options different.
- LLD linker script differences.

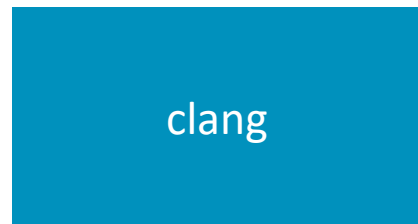
arm

Clang Bare Metal Driver

Clang Drivers

- + Clang driver is responsible for parsing command line arguments and launching sub-processes to perform the desired actions.
 - clang -cc1 for C/C++ compiler.
 - clang-cc1as for integrated assembler.
 - lld for linking.
- + Clang driver instantiates ToolChain classes to handle specific use-cases.
- + Target-triple **<Arch><Sub-arch><Vendor><OS><Environment>** selects ToolChain.

clang -target=aarch64-Linux-gnu hello.c



clang -cc1

```
clang-15.0 -cc1 -triple aarch64-Linux-gnu  
-target-cpu="generic" -target-feature +neon  
-Isystem /path/to/includes  
...
```

lld

```
ld.lld -m aarch64Linux  
-dynamiclinker /lib/ld-linux-aarch64.so  
-L /path/to/system/libraries  
...
```

Clang driver selection per target

Driver.cpp:getToolChain

lib/Driver/Toolchains

`--target=arm-none-eabi`



BareMetal

Requires:

`(toolchains::BareMetal::handlesTarget(Target))`

`--target=arm-linux-gnu`



Linux

`--target=x86_64-unknown`



Generic_GCC

Clang bare metal driver

- + Defaults to LLD for linking.
- + Defaults to the LLVM runtimes
 - Compiler-rt, libc++, libc++abi, libunwind.
- + Uses the clang integrated assembler.
- + Relies on the user to have the right include and library path for the C-library.
- + Has some hard-coded RISC-V multilib support.

arm

Ongoing work and community involvement

Data Driven Multilib support

- + Multilib support in Clang is currently hard-coded
 - Not possible for every embedded toolchain to have an upstream description.
- + GCC has a configure time selection that maps command-line options to directories.
- + RFC <https://discourse.llvm.org/t/rfc-multilib/67494>
- + Different mechanism than GCC as clang driver has more information on target capabilities
 - Still in active development. Feedback on the RFC welcome and in any patches linked from it welcome.

Future work

+ Upstream buildbots for runtimes

- compiler-rt, libc++, libc++abi, libunwind.

+ Making LTO more useable with Linker Scripts

- Link-Time Attributes for LTO: <https://www.youtube.com/watch?v=OkGsMrVd2y8>
 - + Prevent some cross-module optimizations between different memory regions.
 - + Allow for easier placement of sections.

+ Improving code-coverage

- MC/DC: Enabling easy-to-use safety-critical code coverage analysis with LLVM
https://www.youtube.com/watch?v=RmX_8GxxTbs
 - + Not strictly embedded, but a lot of safety-critical systems are embedded systems.
- Runtime suitable for embedded toolchain.

+ LLVM libc

- Focussed on hosted use case for now, but hopes to have scalable implementations suitable for embedded systems.

Arm specific work

- + Big endian for AArch32
 - <https://reviews.llvm.org/D140201>
 - <https://reviews.llvm.org/D140202>
 - <https://reviews.llvm.org/D140203>
- + Cortex-M Security Extensions (CMSE) Support
 - <https://reviews.llvm.org/D139092>

How can I contribute?

- + LLVM Embedded Toolchains Working Group sync up
 - Every 4 weeks on a Thursday at 17:00 GMT.
 - Calendar link at <https://llvm.org/docs/GettingInvolved.html#online-sync-ups>
 - Agenda and meeting notes available at <https://discourse.llvm.org/t/llvm-embedded-toolchains-working-group-sync-up/63270/19>
- + Discourse at <https://discourse.llvm.org/>
- + Bug reports
 - LLVM bugs <https://github.com/llvm/llvm-project/issues>
 - LLVM embedded toolchain for Arm bugs <https://github.com/ARM-software/LLVM-embedded-toolchain-for-Arm/issues>
- + Round tables and panels at LLVM developer meetings.

arm

Thank You

Danke

Gracias

Grazie

谢谢

ありがとう

Asante

Merci

감사합니다

धन्यवाद

Kiitos

شكرًا

ধন্যবাদ

תודה