Open Source C/C++ embedded toolchains using LLVM

EMBO++ 2023

Peter Smith
February 2023

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
  - 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.

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 different 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.
  - Local-bounds sanitizer
  - 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;
}
```

-fsanitze=undefined –fsanitize-trap=undefined

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

-fsanitze=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}
```

// -fsanitize-no-recover=undefined to get call to

// __ubsan_handle_mul_overflow_abort

Kernel Control Flow Integrity KCFI

- 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.

```c
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
```

Address and Memory Sanitizer

- Uses shadow memory to record state of “normal” memory.
  - 1 byte of shadow memory for every 8-bytes of normal memory
    - Requires 8-byte alignment of memory allocations.
    - Red zones between allocations.
    - Shadow Address = (Address >> 3) + Offset
  - Shadow memory byte value of k
    - (0 <= k <= 8) describes how many of these bytes are accessible and 8 – k that are not.
    - K < 0 describes types of red zone (heap, stack, globals).
- Sanitizer runtime
  - Intercepts malloc, free and several other C-library functions.
  - Relies on symbol pre-emption of sanitizer runtime before libc.
- Not currently supported for bare-metal systems
  - Possible for known platforms, the Linux kernel address sanitizer ksan is a bare-metal system.
  - Very difficult to make a “just works” solution in an embedded toolchain.

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 the 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.

Multilib (example from GNU Arm Embedded Toolchain)

```
        thumb   / v6-m.base    /  nofp   /    libc.a libc_nano.a crt0.o nosys.specs
                  v8-m.base    /  nofp   /    libc.a libc_nano.a crt0.o nosys.specs
                  v8-m.main    /  nofp   /    libc.a libc_nano.a crt0.o nosys.specs
lib /
                  v8-m.main+dp /  softfp /    libc.a libc_nano.a crt0.o nosys.specs
                               /  hard   /    libc.a libc_nano.a crt0.o nosys.specs

        arm     / v5te         /  softfp /    libc.a libc_nano.a crt0.o nosys.specs
                                  hard   /    libc.a libc_nano.a crt0.o nosys.specs
```

`-march` (or `-mcpu`) `-mfloat-abi` mapped to library and include directories

GCC specs files for a toolchain

- GCC driver program is controlled by an internal specs file (-dumpspecs).
- Additional specs files that modify the internal specs file can be inserted.
- GNU Embedded toolchain uses these specs files for a few key parameters.
  - Rename libraries, add additional libraries.
  - Rename startup code and other additional startup code.
  - Deal with other specs files.

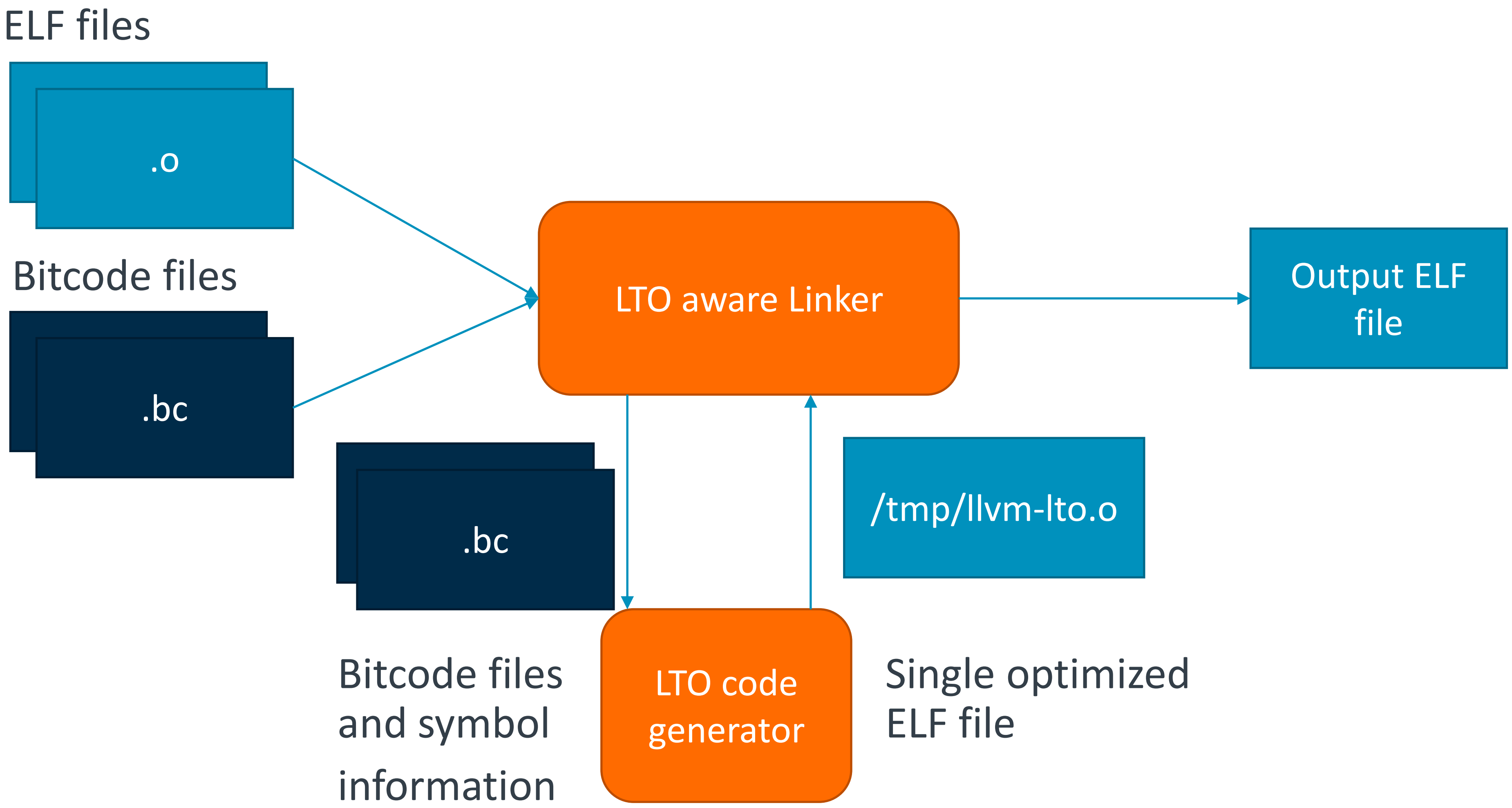
```
%rename link_gcc_c_sequence                nosys_link_gcc_c_sequence

*nosys_libgloss:
-lnosys

*nosys_libc:
%{!specs=nano.specs:-lc} %{specs=nano.specs:-lc_nano}

*link_gcc_c_sequence:
%(nosys_link_gcc_c_sequence) --start-group %G %(nosys_libc) %(nosys_libgloss) --end-group
```


Clang config files

- Collections of command line options.
- Searched for in same path as the clang executable.
- `--config=<config-file.cfg>`
- Can use `<CFGDIR>` for relative path names
- File below is `armv7em_hard_fpv4_sp_d16_semihost.cfg`

```
--target=armv7em-none-eabi -mfloat-abi=hard -march=armv7em -mfpu=fpv4-sp-d16
-fuse-ld=lld
-fno-exceptions
-fno-rtti
--sysroot <CFGDIR>/../lib/clang-runtimes/armv7em_hard_fpv4_sp_d16
<CFGDIR>/../lib/clang-runtimes/armv7em_hard_fpv4_sp_d16/lib/crt0-semihost.o
-lsemihost
```

Link Time Optimization (LTO) tool flow



Linker script problems with LTO

With thanks to Bringing link-time optimization to the embedded world. 2017 DevMeeting

```
SECTIONS {
  .rom : { ROM.o(.text) } >rom
  .text : { *(.text .text.*) } > flash
  .data : { *(.data .data.*) } > ram AT>flash
  .bss : { *(.bss .bss.*) } > ram
}
```

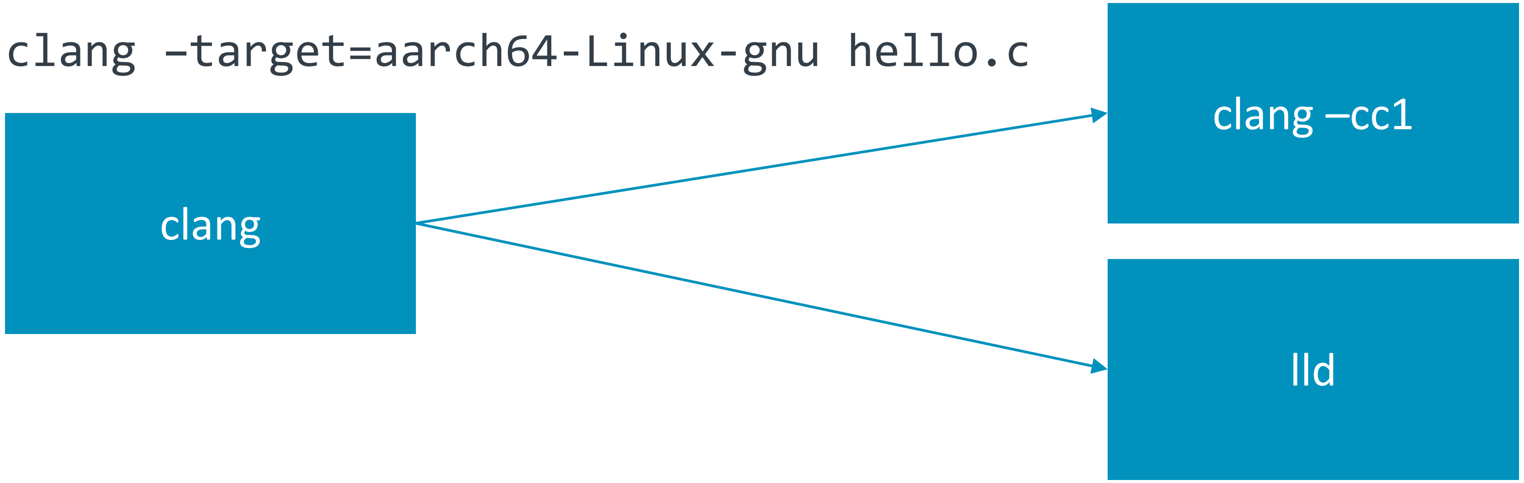
- LTO loses the file name part of the input section description.
- LTO is not aware of Output Section boundaries
  - Can inline from "fast" into "slow" memory.
  - Can constant merge into an Output Section that may not be loaded at time of access.
- LTO can aggressively remove code and data that looks unused, like the vector table!
- Linker needs to add information from the linker script prior to link time code-generation.

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-15.0 -cc1 -triple aarch64-Linux-gnu
-target-cpu=“generic” -target-feature +neon
-Isystem /path/to/includes
…
```

```
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.

Clang linux driver and clang built linux

- For several architectures and configs clang can build the linux kernel
  - https://github.com/ClangBuiltLinux
  - Some kernel configurations need the GNU assembler.
- Instructions available in the kernel build system documentation
  - https://docs.kernel.org/kbuild/llvm.html
- Linux kernel has minimal, to no, requirements on the runtime
  - No C-library required.
  - For many targets and compile time options, no use of runtime (compiler-rt).
- Yocto project has some support for Clang
  - https://github.com/kraj/meta-clang

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
    - https://github.com/ARM-software/LLVM-embedded-toolchain-for-Arm/issues/197
- 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
- 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.

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה