Anatomy of cross-compilation toolchains

Thomas Petazzoni
Bootlin
thomas.petazzoni@bootlin.com
Thomas Petazzoni

- **CTO and Embedded Linux engineer at Bootlin**
  - Embedded Linux specialists.
  - Development, consulting and training.
  - [http://bootlin.com](http://bootlin.com)

- **Contributions**
  - **Kernel support for the Marvell Armada** ARM SoCs from Marvell
  - Major contributor to **Buildroot**, an open-source, simple and fast embedded Linux build system

- Living in **Toulouse**, south west of France
I am **not a toolchain developer**. Not pretending to know everything about toolchains.

- Experience gained from building simple toolchains in the context of Buildroot
- Purpose of the talk is to give an introduction, not in-depth information.
- Focused on simple gcc-based toolchains, and for a number of examples, on ARM specific details.
  - Will not cover advanced use cases, such as LTO, GRAPHITE optimizations, etc.
  - Will not cover LLVM
What is a cross-compiling toolchain?

- A set of tools that allows to build source code into binary code for a *target platform* different than the one where the build takes place
  - Different CPU architecture
  - Different ABI
  - Different operating system
  - Different C library
- Three *machines* involved in the build process
  - **build** machine, where the build takes place
  - **host** machine, where the execution takes place
  - **target** machine, for which the programs generate code
- Native toolchain: `build == host == target`
- Cross-compilation toolchain: `build == host != target`
- Corresponds to the `--build`, `--host` and `--target` *autoconf* configure script arguments
  - By default, automatically guessed by *autoconf* to be for the current machine
Toolchain tuple

- autoconf defines the concept of *system definitions*, represented as **tuples**
- A system definition describes a system: CPU architecture, operating system, vendor, ABI, C library
- Different forms:
  - `<arch>-<vendor>-<os>-<libc/abi>`, full form
  - `<arch>-<os>-<libc/abi>`
- Components:
  - `<arch>`, the CPU architecture: arm, mips, powerpc, i386, i686, etc.
  - `<vendor>`, (mostly) free-form string, ignored by *autoconf*
  - `<os>`, the operating system. Either *none* or *linux* for the purpose of this talk.
  - `<libc/abi>`, combination of details on the C library and the ABI in use
Toolchain tuple examples

- **arm-foo-none-eabi**, bare-metal toolchain targeting the ARM architecture, from vendor *foo*
- **arm-unknown-linux-gnueabihf**, Linux toolchain targeting the ARM architecture, using the EABIhf ABI and the glibc C library, from an *unknown* vendor
- **armeb-linux-uclibcgneabi**, Linux toolchain targeting the ARM big-endian architecture, using the EABI ABI and the uClibc C library
- **mips-img-linux-gnu**, Linux toolchain targeting the MIPS architecture, using the glibc C library, provided by Imagination Technologies.
Two main values for `<os>`

- **none** for **bare-metal** toolchains
  - Used for development without an operating system
  - C library used is generally **newlib**
  - Provides C library services that do not require an operating system
  - Allows to provide basic *system calls* for specific hardware targets
  - Can be used to build bootloaders or the Linux kernel, cannot build Linux userspace code

- **linux** for **Linux** toolchains
  - Used for development with a Linux operating system
  - Choice of Linux-specific C libraries: **glibc**, **uclibc**, **musl**
  - Supports Linux system calls
  - Can be used to build Linux userspace code, but also bare-metal code such as bootloaders or the kernel itself
There are four core components in a Linux cross-compilation toolchain:

1. binutils
2. gcc
3. Linux kernel headers
4. C library

In addition to these, a few dependencies are needed to build gcc itself.
“collection of binary tools”

Main tools

- **ld**, the linker. Links multiple object files into a shared library, an executable, or another object file.
- **as**, the assembler. Takes architecture-specific assembler code in text form, and produces a corresponding object file with binary code.

Debugging/analysis tools and other tools

- addr2line, ar, c++filt, gold, gprof, nm, objcopy, objdump, ranlib, readelf, size, strings, strip

Needs to be configured for each CPU architecture: your native x86 binutils cannot produce ARM code.

Pretty straightforward to cross-compile, no special dependencies are needed.

```
./configure --target=arm-buildroot-linux-gnueabihf --with-sysroot=PATH
```
GNU Compiler Collection

Front-ends for many source languages: C, C++, Fortran, Go, etc.

Back-ends for many CPU architectures.

Provides:

- The compiler itself, cc1 for C, cc1plus for C++. Only generates assembly code in text format.
- The compiler driver, gcc, g++, which drives the compiler itself, but also the binutils assembler and linker.
- Target libraries: libgcc (gcc runtime), libstdc++ (the C++ library), libgfortran (the Fortran runtime)
- Header files for the standard C++ library.

Building gcc is a bit more involved than building binutils: two steps are needed, see later.
In order to build a C library, the Linux kernel headers are needed: definitions of system call numbers, various structure types and definitions.

In the kernel, headers are split between:

- User-space visible headers, stored in `uapi` directories: `include/uapi/`, `arch/<ARCH>/include/uapi/asm`
- Internal kernel headers.

Installation takes place using
```
make ARCH=.. INSTALL_HDR_PATH=... headers_install
```

- The installation includes a sanitation pass, to remove kernel-specific constructs from the headers.
- As of Linux 4.8, installs 756 header files.
Which version of the kernel headers should be used in a toolchain?

The kernel to userspace ABI is backward compatible.

Therefore, the version of the kernel used for the kernel headers must be the same version or older than the kernel version running on the target system.

Otherwise the C library might use system calls that are not provided by the kernel.

Examples:

- Toolchain using 3.10 kernel headers, running 4.4 kernel on the target → OK
- Toolchain using 4.8 kernel headers, running 4.4 kernel on the target → NOK

Linux 3.13.0 headers

$ cat arm-none-linux-gnueabi/libc/usr/include/linux/version.h

#define LINUX_VERSION_CODE 199936
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
C library

- Provides the implementation of the POSIX standard functions, plus several other standards and extensions
- Based on the Linux system calls
- Several implementations available:
  - glibc
  - uClibc-ng (formerly uClibc)
  - musl
  - bionic, for Android systems
  - A few other more special-purpose: newlib (for bare-metal), dietlibc, klibc
- After compilation and installation, provides:
  - The dynamic linker, ld.so
  - The C library itself libc.so, and its companion libraries: libm, librt, libpthread, libutil, libnsl, libresolv, libcrypt
  - The C library headers: stdio.h, string.h, etc.
C library: glibc

- GNU C Library
- **De-facto standard** of Linux C libraries
- Used in virtually all common desktop/server distributions
- **Full-featured**
  - Supports for numerous architectures or operating systems
  - No support for noMMU platforms
  - No support for static linking
- **ABI backward** compatibility
- Almost no configurability
- Used to be “too big” for embedded, but no longer necessarily the case.
- LGPLv2.1 or later
- https://www.gnu.org/software/libc/
C library: uClibc/uClibc-ng

- Started in 2000
- High-level of configurability
- Supports many architectures, include some not supported by glibc
- Supports only Linux as operating system
- No ABI backward compatibility
- Supports numerous no-MMU architectures: ARM noMMU, Blackfin, etc.
- No longer related to uClinux
- Support for static linking
- Original uClibc project dead (last release in May 2012), but the uClibc-ng fork is very active and is the de-facto replacement.
- LGPLv2.1
- http://uclibc-ng.org/
C library: musl

- Started in 2011
- **MIT licensed**
- Very active development
- Support for ARM, ARM64, i386, Microblaze, MIPS(64), OpenRisc, PowerPC(64), SuperH, x86-4
- Recently, **noMMU support** was added for SuperH2, for the J-core Open Processor
- No configurability
- **Small**, even smaller than uClibc, especially for static linking scenarios
- Strict conformance to standards (stricter than glibc, uClibc), causes a few build issues with a number of packages
- Nice comparison of the three main C libraries: http://www.etalabs.net/compare_libcs.html
- http://www.musl-libc.org/
C library: size comparison

<table>
<thead>
<tr>
<th>Library</th>
<th>glibc</th>
<th>uclIBC</th>
<th>musl</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld, dynamic linker</td>
<td>121 KB</td>
<td>25 KB</td>
<td>N/A</td>
</tr>
<tr>
<td>libc</td>
<td>878 KB</td>
<td>286 KB</td>
<td>437 KB</td>
</tr>
<tr>
<td>libcrypt</td>
<td>30 KB</td>
<td>17 KB</td>
<td>N/A</td>
</tr>
<tr>
<td>libdl</td>
<td>9.5 KB</td>
<td>9 KB</td>
<td>N/A</td>
</tr>
<tr>
<td>libm</td>
<td>414 KB</td>
<td>37 KB</td>
<td>N/A</td>
</tr>
<tr>
<td>libnsl</td>
<td>54 KB</td>
<td>4.7 KB</td>
<td>N/A</td>
</tr>
<tr>
<td>libnss_dns</td>
<td>14 KB</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>libnss_files</td>
<td>30 KB</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>libpthread</td>
<td>105 KB</td>
<td>76 KB</td>
<td>N/A</td>
</tr>
<tr>
<td>libresolv</td>
<td>54 KB</td>
<td>4.7 KB</td>
<td>N/A</td>
</tr>
<tr>
<td>librt</td>
<td>22 KB</td>
<td>13 KB</td>
<td>N/A</td>
</tr>
<tr>
<td>libutil</td>
<td>9.5K</td>
<td>4.7 KB</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>1741 KB</strong></td>
<td><strong>477 KB</strong></td>
<td><strong>437 KB</strong></td>
</tr>
</tbody>
</table>

ARM Cortex-A9 toolchain built with the Thumb-2 instruction set, using Buildroot. gcc 4.9, binutils 2.26, musl 1.1.15, glibc 2.23, uclibc-ng 1.0.17
Several math libraries are needed to build gcc

They are compiled for the host machine, i.e., they are not needed on the target

- **mpfr**, multiple-precision floating-point computations. Used since gcc 4.3 to evaluate and replace at compile-time calls to built-in math functions having constant arguments with their mathematically equivalent results
- **gmp**, dependency of mpfr
- **mpc**, for computation of complex numbers. Used since gcc 4.5 to evaluate calls to complex built-in math functions having constant arguments and replace them at compile time with their mathematically equivalent result
The build process for a regular Linux cross-compilation toolchain is in fact fairly easy:

1. Build *binutils*
2. Build the dependencies of *gcc*: *mpfr*, *gmp*, *mpc*
3. Install the Linux kernel headers
4. Build a first stage *gcc*: no support for a C library, support only for static linking
5. Build the *C library* using the first stage *gcc*
6. Build the final *gcc*, with C library and support for dynamic linking
Overall build process: example in Buildroot
Concept of sysroot

- The *sysroot* is the **logical root directory for headers and libraries**
- Where *gcc* looks for headers, and *ld* looks for libraries
- Both *gcc* and *binutils* are built with `--with-sysroot=<SYSROOT>`
- The kernel headers and the C library are installed in `<SYSROOT>`
- If the toolchain has been moved to a different location, *gcc* will still find its sysroot if it’s in a subdir of `--prefix`
  - `--prefix=/home/thomas/buildroot/arm-uclibc/host/usr`
  - `--with-sysroot=/home/thomas/buildroot/arm-uclibc/host/usr/arm-buildroot-linux-uclibcgnueabihf/sysroot`
- Can be overridden at runtime using *gcc*’s `--sysroot` option.
- The current *sysroot* can be printed using the `-print-sysroot` option.
Most toolchains provide a **single sysroot** with the C library and gcc runtime libraries.

These libraries, built for the target, are optimized for a **specific architecture variant and ABI**.

Need to have one toolchain for each architecture variant or ABI.

**Multilib** toolchains contain multiple **sysroot**, each having a version of the target libraries for different architecture/ABI variants.

Example of the Sourcery CodeBench ARM toolchain:

```
$ arm-none-linux-gnueabi-gcc -print-multi-lib
.
armv4t;@march=armv4t
thumb2;@mthumb@march=armv7-a
```

Three sysroots: ARMv5, ARMv4 and ARMv7 Thumb-2.
The compiler automatically selects the right `sysroot` depending on the gcc flags:

```
$ arm-none-linux-gnueabi-gcc -march=armv5te -print-sysroot
.../bin/../../arm-none-linux-gnueabi/libc
$ arm-none-linux-gnueabi-gcc -march=armv4t -print-sysroot
.../bin/../../../arm-none-linux-gnueabi/libc/armv4t
$ arm-none-linux-gnueabi-gcc -march=armv7-a -mthumb -print-sysroot
.../bin/../../../../arm-none-linux-gnueabi/libc/thumb2
```

Each `sysroot` has a different library variant:

```
$ readelf -A arm-none-linux-gnueabi/libc/lib/ld-2.18.so
  Tag_CPU_name: "5TE"
  Tag_CPU_arch: v5TE
$ readelf -A arm-none-linux-gnueabi/libc/armv4t/lib/ld-2.18.so
  Tag_CPU_name: "4T"
  Tag_CPU_arch: v4T
$ readelf -A arm-none-linux-gnueabi/libc/thumb2/lib/ld-2.18.so
  Tag_CPU_name: "7-A"
  Tag_CPU_arch: v7
  Tag_THUMB_ISA_use: Thumb-2
```
Toolchain contents

Cross-compilation toolchain generated by Buildroot

- arm-buildroot-linux-uclibcgnueabihf/
- bin/
- include/
- lib/
- libexec/
- share/
Toolchain contents

- arm-buildroot-linux-uclibcgnueabihf/
  - bin/
    - Limited set of *binutils* programs, without their cross-compilation prefix. Hard links to their counterparts with the prefix. This is where *gcc* finds them.
  - include/c++/4.9.4/
    - Headers for the C++ standard library, installed by *gcc*
    - Interestingly, they are not part of the *sysroot* per-se.
  - lib/
    - The *gcc* runtime libraries, built for the target
    - *libatomic*, provides a software implementation of *atomic* built-ins, when needed
    - *libgcc*, the main *gcc* runtime (optimized functions, 64-bit division, floating point emulation)
    - *libitm*, transactional memory library
    - *libstdc++*, standard C++ library
    - *libsupc++*, subset of *libstdc++* with only the *language support* functions
  - sysroot/
    - lib/, usr/lib/: C library and *gcc* runtime libraries (shared and static)
    - usr/include/, Linux kernel and C library headers

- bin/
Toolchain contents

- arm-buildroot-linux-uclibcgnueabihf/
- bin/
  - arm-buildroot-linux-uclibcgnueabihf- prefixed tools
  - From binutils: addr2line, ar, as, elfedit, gcov, gprof, ld, nm, objcopy, objdump, ranlib, readelf, size, strings, strip
  - From gcc: c++ (same as g++), cc (same as gcc), cpp, g++, gcc, gcc-ar, gcc-nm, gcc-ranlib
  - The gcc-{ar,nm,ranlib} are wrappers for the corresponding binutils program, to support Link Time Optimization (LTO)
- include/
- lib/
- libexec/
- share/
Toolchain contents

- arm-buildroot-linux-uclibcgnueabihf/
- bin/
- include/
  - Headers of the host libraries (gmp, mpfr, mpc)
- lib/
- libexec/
- share/
Toolchain contents

- arm-buildroot-linux-uclibcgnueabihf/
- bin/
- include/
- lib/
  - gcc/arm-buildroot-linux-uclibcgnueabihf/4.9.4/
    - crtbegin*.o, crtend*.o, object files handling constructors/destructors, linked into executables
    - include/, headers provided by the compiler (stdarg.h, stdint.h, stdatomic.h, etc.)
    - include-fixed/, system headers that gcc fixed up using fixincludes
    - install-tools/, also related to the fixincludes process
    - libgcc.a, libgcc_eh.a, libgcov.a, static variants of the gcc runtime libraries
    - ldscripts/, linker scripts provided by gcc to link programs and libraries
  - Host version of gmp, mpfr, mpc, needed for gcc
- libexec/
- share/
Toolchain contents

- arm-buildroot-linux-uclibcgnueabihf/
- bin/
- include/
- lib/
- libexec/
  - gcc/arm-buildroot-linux-uclibcgnueabihf/4.9.4/
    - cc1, the actual C compiler
    - cc1plus, the actual C++ compiler
    - collect2, program from gcc collecting initialization functions, wrapping the linker
    - install-tools/, misc gcc related tools, not needed for the compilation process
    - liblto_plugin.so.0.0.0, lto-wrapper, lto1, related to LTO support (outside of the scope of this talk)
- share/
Toolchain contents

- arm-buildroot-linux-uclibcgnueabihf/
- bin/
- include/
- lib/
- libexec/
- share/
  - documentation (man pages and info pages)
  - translation files for gcc and binutils
Architecture tuning

- GCC provides several configure-time options to tune for a specific architecture/CPU variant: `--with-arch`, `--with-cpu`, `--with-abi`, `--with-fpu`
- These define the default architecture/CPU variant for which GCC will generate code.
- They can be overridden at runtime using the `-march`, `-mcpu`, `-mabi`, `-mfpu` options.
- However, be careful: parts of the toolchain are built for the target!
  - The GCC runtime libraries
  - The C library, dynamic linker, and startup code
- They are built together with the rest of the toolchain, so it’s important to know with what optimization level they were built!
- Passing `-march=armv5te` is not sufficient to make your binary work on ARMv5 if your toolchain originally targets ARMv7.
- Read the GCC documentation, section *Machine-dependent options* to get the complete list of possible values.
ABI: definition

- ABI = Application Binary Interface

- From the point of a toolchain, the ABI defines:
  - How function calls are made (so-called calling convention)
    - How arguments are passed: in registers (which ones?), on the stack, how 64-bits arguments are handled on 32 bits architectures
    - How the return value is passed
  - Size of basic data types
  - Alignment of members in structures
  - When there is an operating system, how system calls are made

- Object files from different ABIs cannot be linked together (especially important if you have pre-built libraries or executables!)

- For a given CPU architecture, there can potentially be an infinite number of ABIs: ABIs are just specifications on how to use the CPU architecture

- Need to understand the ABIs for each architecture.
**ABI:** example of ARM 32

- **OABI:** obsolete ABI. Forced the use of hard-float instructions, which required emulation of floating-point operations in the kernel. No longer supported anywhere.

- **EABI**, standardized by ARM. Allows mixing hard-float code with soft-float code. Floating point arguments passed in integer registers.
  - Hard-float code: uses floating point instructions directly.
  - Soft-float code: emulates floating point instructions using a userspace library provided by gcc.

- **EABIf**: also standardized by ARM. Requires a floating point unit: only hard-float code. Floating point arguments passed in floating point registers.

- **gcc options**
  - EABI soft-float: `-mabi=aapcs-linux -mfloat-abi=soft`
  - EABI hard-float: `-mabi=aapcs-linux -mfloat-abi=softfp`
  - EABIf: `-mabi=aapcs-linux -mfloat-abi=hard`
Difference between toolchain and SDK

- Toolchain: just the compiler, binutils and C library
- SDK: a toolchain, plus a number (potentially large) of libraries built for the target architecture, and additional native tools helpful when building software.
- Build systems such as OpenEmbedded or Yocto can typically:
  - Use an existing toolchain as input, or build their own toolchain
  - In addition to producing a root filesystem, they can also produce a SDK to allow application developers to build applications/libraries for the target.
How to get a cross-compilation toolchain

- Pre-built
  - From your distribution. Ubuntu and Debian have numerous cross-compilers readily available.
  - From various organization: Linaro provides ARM and AArch64 toolchains, Mentor provides a few free Sourcery CodeBench toolchains, Imagination provides MIPS toolchains, etc.

- Built it yourself
  - Crosstool-NG, tool specialized in building cross-compilation toolchain. By far the most configurable/versatile.
  - Embedded Linux build systems generally all know how to build a cross-compilation toolchain: Yocto/OpenEmbedded, Buildroot, OpenWRT, etc.
References

- Crosstool-NG documentation, https://github.com/crosstool-ng/crosstool-ng/blob/master/docs/
- Binutils documentation, https://sourceware.org/binutils/docs/
Thanks for your attention!

Questions?

Thomas Petazzoni
thomas.petazzoni@bootlin.com

Slides under CC-BY-SA 3.0