03-3 Toolchains

Embedded Linux system development

Cross-compiling toolchains

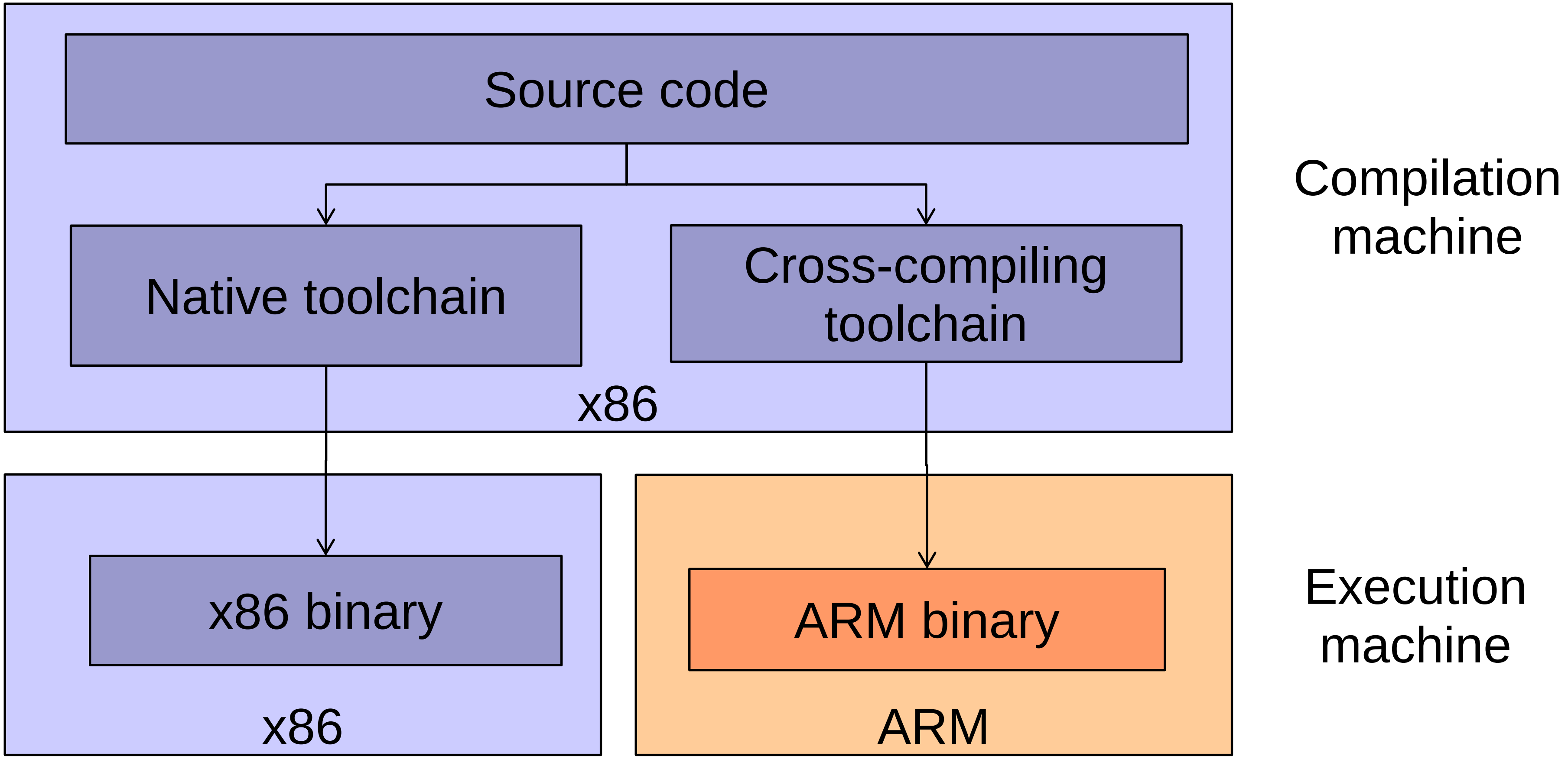
Thomas Petazzoni
Michael Opdenacker
Free Electrons

Latest update: 9/17/2018,
Document sources, updates and translations:
http://free-electrons.com/docs/toolchains
Corrections, suggestions, contributions and translations are welcome!

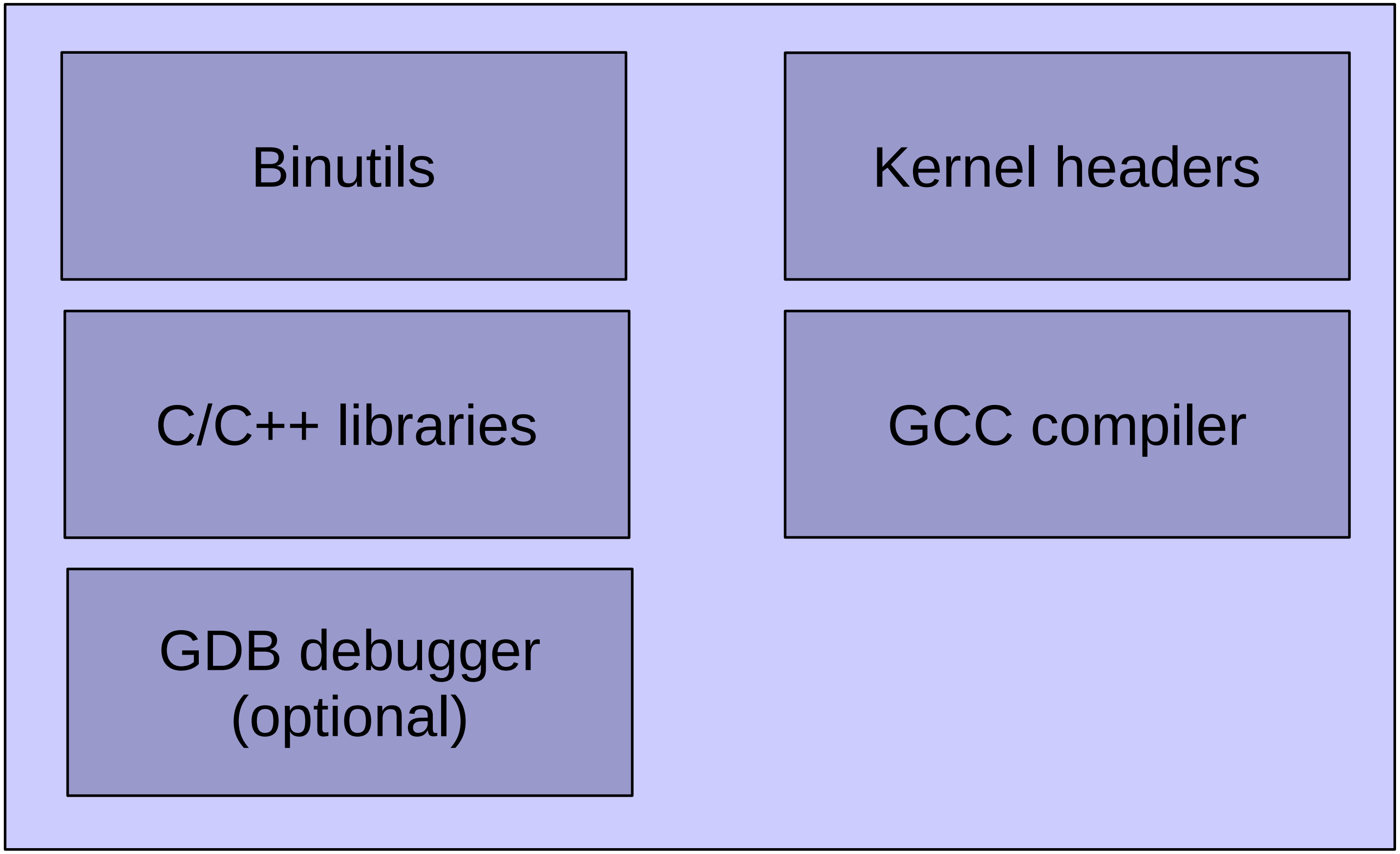
Definition (1)

- The usual development tools available on a GNU/Linux workstation is a **native toolchain**
- This toolchain runs on your workstation and generates code for your workstation, usually x86
- For embedded system development, it is usually impossible or not interesting to use a native toolchain
  - The target is too restricted in terms of storage and/or memory
  - The target is very slow compared to your workstation
  - You may not want to install all development tools on your target.
- Therefore, **cross-compiling toolchains** are generally used. They run on your workstation but generate code for your target.

Definition (2)



Components

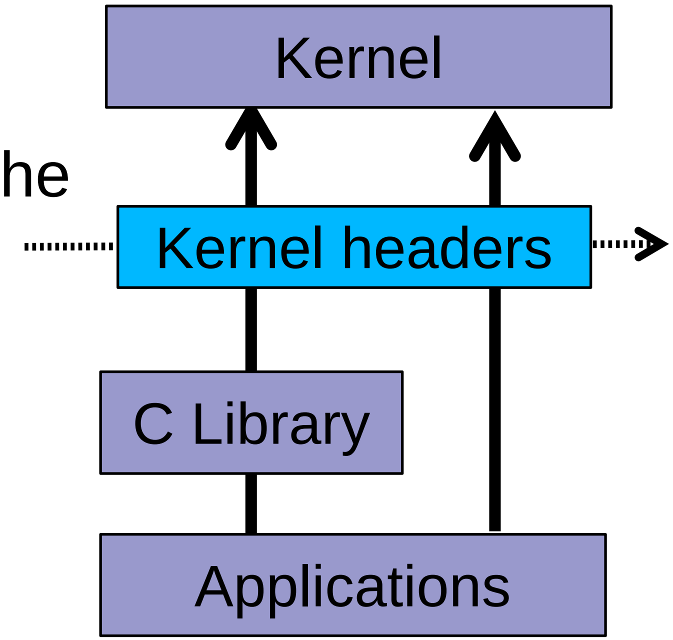

Binutils
Kernel headers
C/C++ libraries
GCC compiler
GDB debugger
(optional)

binutils

- **Binutils** is a set of tools to generate and manipulate binaries for a given CPU architecture
- **as**, the assembler, that generates binary code from assembler source code
- **ld**, the linker
- **ar**, **ranlib**, to generate .a archives, used for libraries
- **objdump**, **readelf**, **size**, **nm**, **strings**, to inspect binaries. Very useful analysis tools !
- **strip**, to strip useless parts of binaries in order to reduce their size
- http://www.gnu.org/software/binutils/
- GPL license

Kernel headers (1)

- The C library and compiled programs needs to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.



- Available in <linux/...> and <asm-generic/...> and a few other directories corresponding to the ones visible in /usr/include/ in the kernel sources

/usr/include

```
bone$ cd /usr/include
bone$ ls arm-linux-gnueabihf/asm
auxvec.h        hwcap.h    kvm_para.h    poll.h          setup.h        socket.h    termbits.h
bitsperlong.h   ioctl.h    mman.h        posix_types.h   shmbuf.h       sockios.h   termios.h
byteorder.h     ioctls.h   msgbuf.h      ptrace.h        sigcontext.h   statfs.h    types.h
errno.h         ipcbuf.h   param.h       resource.h      siginfo.h      stat.h      unistd.h
fcntl.h         kvm.h      perf_regs.h   sembuf.h        signal.h       swab.h
bone$ ls asm-generic
auxvec.h        int-l64.h    kvm_para.h      poll.h          shmbuf.h          socket.h    termbits.h
bitsperlong.h   int-ll64.h   mman-common.h   posix_types.h   shmparam.h        sockios.h   termios.h
errno-base.h    ioctl.h      mman.h          resource.h      siginfo.h         statfs.h    types.h
errno.h         ioctls.h     msgbuf.h        sembuf.h        signal-defs.h     stat.h      ucontext.h
fcntl.h         ipcbuf.h     param.h         setup.h         signal.h          swab.h      unistd.h
```

Kernel headers (2)

- System call numbers, in </include/asm/unistd.h>

```
#define __NR_exit               1
#define __NR_fork               2
#define __NR_read               3
```

- Constant definitions, here in </include/asm-generic/fcntl.h>, included from </include/asm/fcntl.h>, included from </include/linux/fcntl.h>

```
#define O_RDWR        00000002
```

- Data structures, here in </include/asm/stat.h>

```
struct stat {
        unsigned long  st_dev;
        unsigned long  st_ino;
                        [...]
};
```

Kernel headers (3)

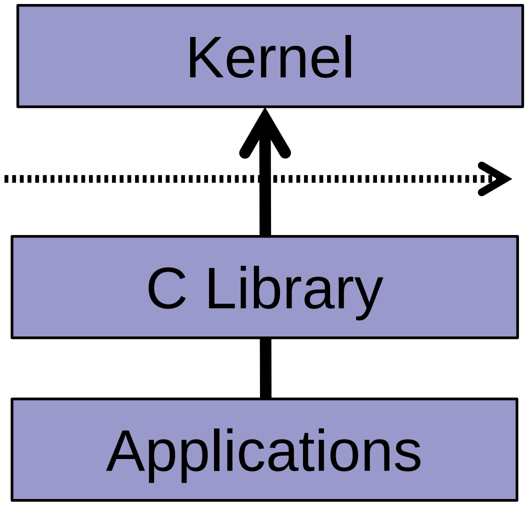
- The kernel-to-userspace ABI is backward compatible
- Binaries generated with a toolchain using kernel headers older than the running kernel will work without problem, but won't be able to use the new system calls, data structures, etc.
- Binaries generated with a toolchain using kernel headers newer than the running kernel might work on if they don't use the recent features, otherwise they will break
- Using the latest kernel headers is not necessary, unless access to the new kernel features is needed
- The kernel headers are extracted from the kernel sources using the headers_install kernel Makefile target.

GCC compiler

- GNU C Compiler, the famous free software compiler
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, and
- Generate code for a large number of CPU architectures, including **ARM**, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, i386, x86_64, IA64, Xtensa, etc.
- http://gcc.gnu.org/
- Available under the GPL license, libraries under the LGPL.

C library

- The C library is an essential component of a Linux system
- Interface between the applications and the kernel
- Provides the well-known standard C API to ease application development
- Several C libraries are available: glibc, uClibc, eglibc, dietlibc, newlib, etc.
- The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific C library.



glibc

http://www.gnu.org/software/libc/

- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Of course, actively maintained
- Quite big for small embedded systems: approx 2.5 MB on arm (version 2.9 - libc: 1.5 MB, libm: 750 KB)
- 2016-08-05: glibc 2.24 released.
- 2018-08-01: glibc 2.28 released

uClibc

http://www.uclibc.org/ from CodePoet Consulting

- License: LGPL
- Lightweight C library for small embedded systems
- High configurability: many features can be enabled or disabled through a menuconfig interface
- Works only with Linux/uClinux, works on most embedded architectures
- No stable ABI, different ABI depending on the library configuration
- Focus on size rather than performance
- Small compile time

uClibc (2)

- Most of the applications compile with uClibc. This applies to all applications used in embedded systems.
- Size (arm): 4 times smaller than glibc!
  uClibc 0.9.30.1: approx. 600 KB (libuClibc: 460 KB, libm: 96KB)
  glibc 2.9: approx 2.5 MB
- Used on a large number of production embedded products, including consumer electronic devices
- Actively maintained, large developer and user base
- Now supported by MontaVista, TimeSys and Wind River.
- **15 May 2012, uClibc 0.9.33.2 Released**

Honey, I shrunk the programs!

C program	Compiled with shared libraries		Compiled statically	
	glibc	uClibc	glibc	uClibc
Plain "hello world" (stripped)	5.6 K (glibc 2.9)	5.4 K (uClibc 0.9.30.1)	472 K (glibc 2.9)	18 K (uClibc 0.9.30.1)
Busybox (stripped)	245 K (older glibc)	231 K (older uClibc)	843 K (older glibc)	311 K (older uClibc)

Executable size comparison on ARM

Installing and using a precompiled toolchain

- Follow the installation procedure proposed by the vendor
- Usually, it is simply a matter of extracting a tarball at the proper place
- Then, add the path to toolchain binaries in your PATH:
  export PATH=/path/to/toolchain/bin/:$PATH

Or

```
host$ export ARCH=arm
host$ export CROSS_COMPILE= arm-linux-gnueabihf-
host$ PATH=$PATH:~/BeagleBoard/bb-kernel/dl/gcc-linaro-5.3-2016.02-x86_64_arm-linux-gnueabihf/bin
host$ ${CROSS_COMPILE}gcc helloWorld.c
```