Embedded Software
- Linux Kernel & Device Driver -

Spring 2019

Syllabus

- **Instructors:**
  - Dongkun Shin
  - Office : Room 85470
  - E-mail : dongkun@skku.edu
  - Office Hours: Wed. 15:00-17:30 or by appointment

- **Lecture notes**
  - nyx.skku.ac.kr → Courses → Embedded Software (2019 Spring)
  - http://nyx.skku.ac.kr/?page_id=2079

- **Lecture notes and talks will be given in English.**
  - However, TA students are **not** fluent in English.

If you have any questions,

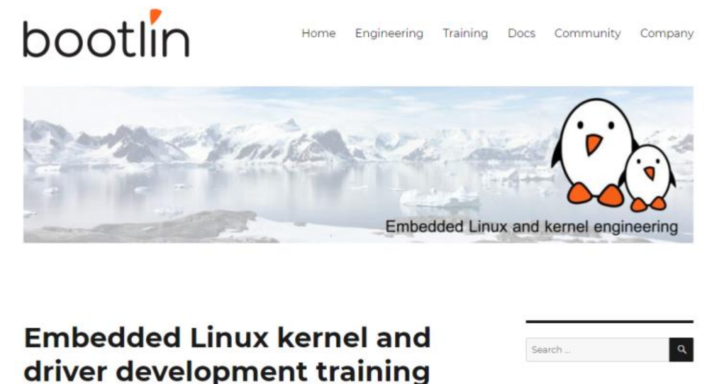
please feel free to interrupt me

in English

or Korean.

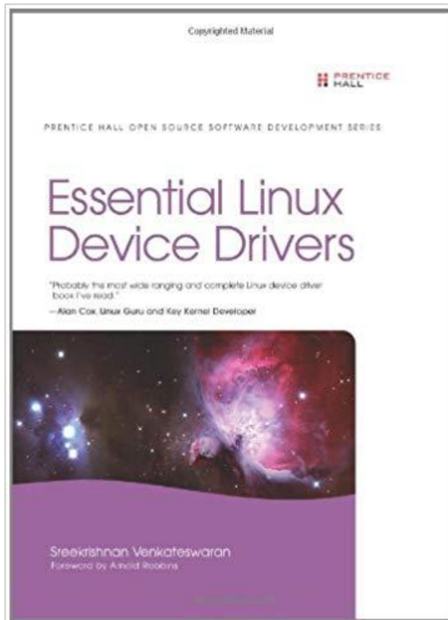
Syllabus (cont'd)

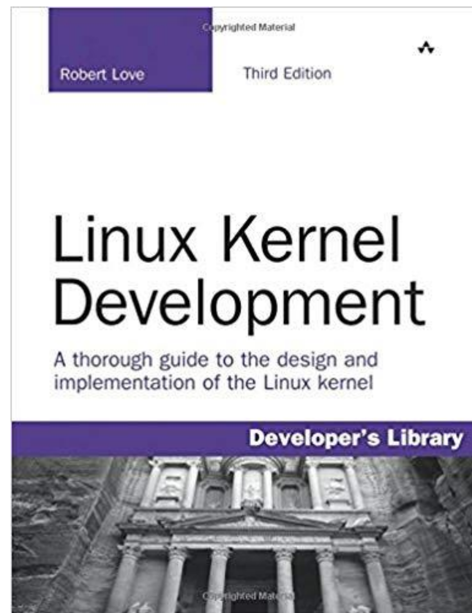
- **Main text**
  - Bootlin Embedded Linux kernel and driver development training materials (free!)
  - https://bootlin.com/training/kernel/

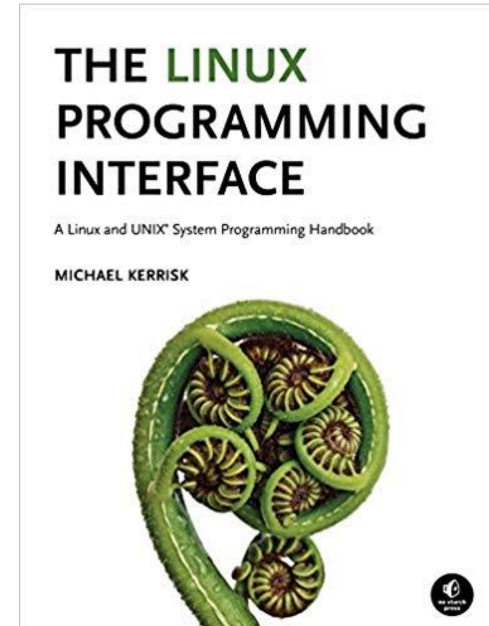


- **Grading policy (subject to change)**
  - Attendance: -5% (I'll check at random)
  - Midterm exam: 25%
  - Final exam: 25%
  - Assignment: 50% (Weekly Lab homework, Term Project)
  - If you cheat on tests and other assignments, you will fail the class.

Useful reading







Syllabus (cont'd)

Course Outline

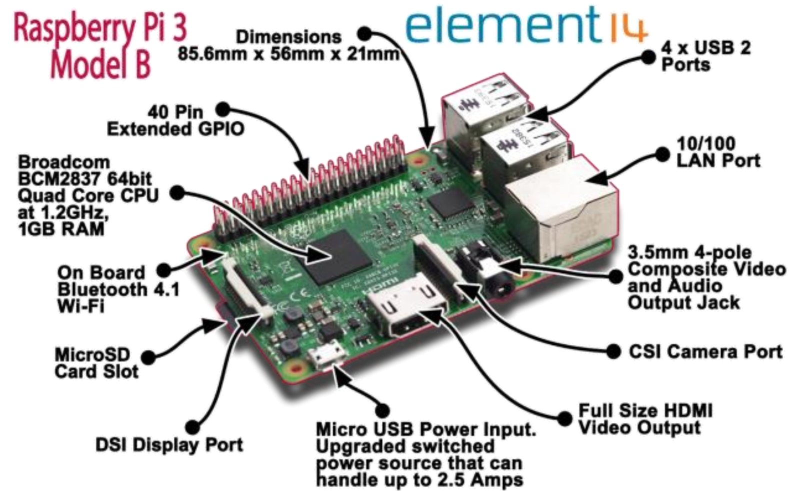
1. Intro & Development Setup
2. Linux Sources, Booting
3. Linux Kernel Module
4. Device Model
5. I2C Driver
6. Kernel Frameworks, Input Subsystem, Block Device
7. Memory Management, I/O Memory
8. Task Scheduling, Interrupt
9. Locking, Debugging
10. Power Management, DMA

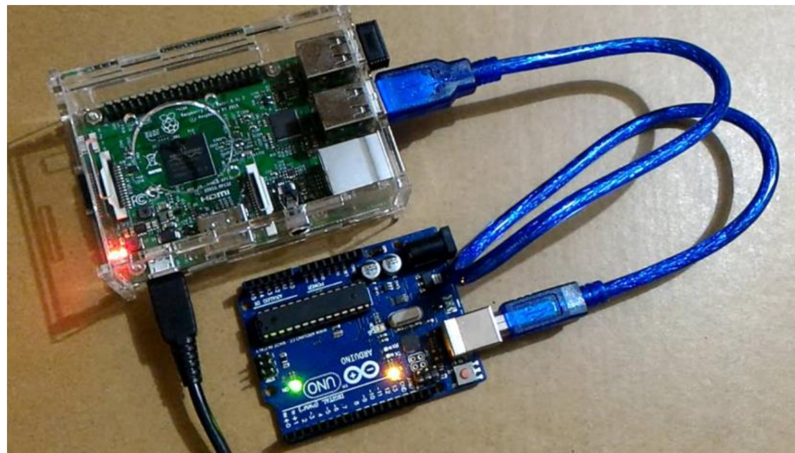
Syllabus (cont'd)

- **Assignments**
  - Lab Class Homework
  - Team project (Two students/Team)

- **Prerequisites (Mandatory)**
  - C programming
  - Software Practice 2 (Unix Shell)
  - Computer Architecture
  - Operating Systems

Lab





RasPi3 & Arduino Uno

Nintendo Nunchuk

https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

Schedule

	Lecture	Date
1	Intro, development setup	3/5, 3/7
2	Kernel sources, Booting	3/14, 3/21, 3/26
3	Linux kernel modules	3/28
4	Linux device model	4/4, 4/9
5	I2C API	4/11
	Mid-term Exam	4/23, 4/25
6	Kernel frameworks, Input subsystem, Block layer	4/30, 5,2
7	Memory management, I/O memory and ports	5/9, 5/14
8	Processes, scheduling, sleeping and interrupts	5/16, 5/23
8	Locking, Driver debugging techniques	5/30
10	Power management, DMA	6/11, 6/13
	Final Exam	6/18, 6/20

Schedule

Lab	Date
Kernel Sources & Raspberry Pi Board Setup	3/12
Kernel Configuration & Kernel Building	3/19
Writing modules	4/2
Linux device model for an I2C driver	4/16
Communicate with the Nunchuk over I2C	4/18
Expose the Nunchuk functionality to user space	5/7
UART platform driver 1	5/21
UART platform driver 2	5/28
Locking, Kernel Debugging	6/4
Term Project Due Date	6/24

Rights to copy

- © Copyright 2004-2019, Bootlin
- **License: Creative Commons Attribution - Share Alike 3.0**
- https://creativecommons.org/licenses/by-sa/3.0/legalcode
- You are free:
  - to copy, distribute, display, and perform the work
  - to make derivative works
  - to make commercial use of the work
- Under the following conditions:
  - Attribution. You must give the original author credit.
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only
- under a license identical to this one.
  - For any reuse or distribution, you must make clear to others the license terms of this work.
  - Any of these conditions can be waived if you get permission from the copyright holder.
- Your fair use and other rights are in no way affected by the above.

- **Document sources:** https://git.bootlin.com/training-materials/

Birth of free software

- 1983, Richard Stallman, GNU project and the free software concept.
  - Beginning of the development of gcc, gdb, glibc and other important tools
- 1991, Linus Torvalds, Linux kernel project, a Unix-like operating system kernel.
  - Together with GNU software and many other open-source components: a completely free operating system, GNU/Linux
- 1995, Linux is more and more popular on server systems
- 2000, Linux is more and more popular on embedded systems
- 2008, Linux is more and more popular on mobile devices
- 2010, Linux is more and more popular on phones

Free software?

- A program is considered free when its license offers to all its users the following four freedoms
  - Freedom to run the software for any purpose
  - Freedom to study the software and to change it
  - Freedom to redistribute copies
  - Freedom to distribute copies of modified versions
- Those freedoms are granted for both commercial and non-commercial use
- They imply the availability of source code, software can be modified and distributed to customers
- Good match for embedded systems!

Examples of embedded systems running Linux

Wireless routers

Chromecast

Bike computers

Robots

Smart Watch

CCTV

Processor and architecture

- The Linux kernel and most other architecture-dependent component support a wide range of 32 and 64 bits architectures
  - x86 and x86-64, as found on PC platforms, but also embedded systems (multimedia, industrial)
  - ARM, with hundreds of different SoCs (all sorts of products)
  - RiscV, the rising architecture with a free instruction set (from high-end cloud computing to the smallest embedded systems)
  - PowerPC (mainly real-time, industrial applications)
  - MIPS (mainly networking applications)
  - SuperH (mainly set top box and multimedia applications)
  - c6x (TI DSP architecture)
  - Microblaze (soft-core for Xilinx FPGA)
  - Others: ARC, m68k, Xtensa...

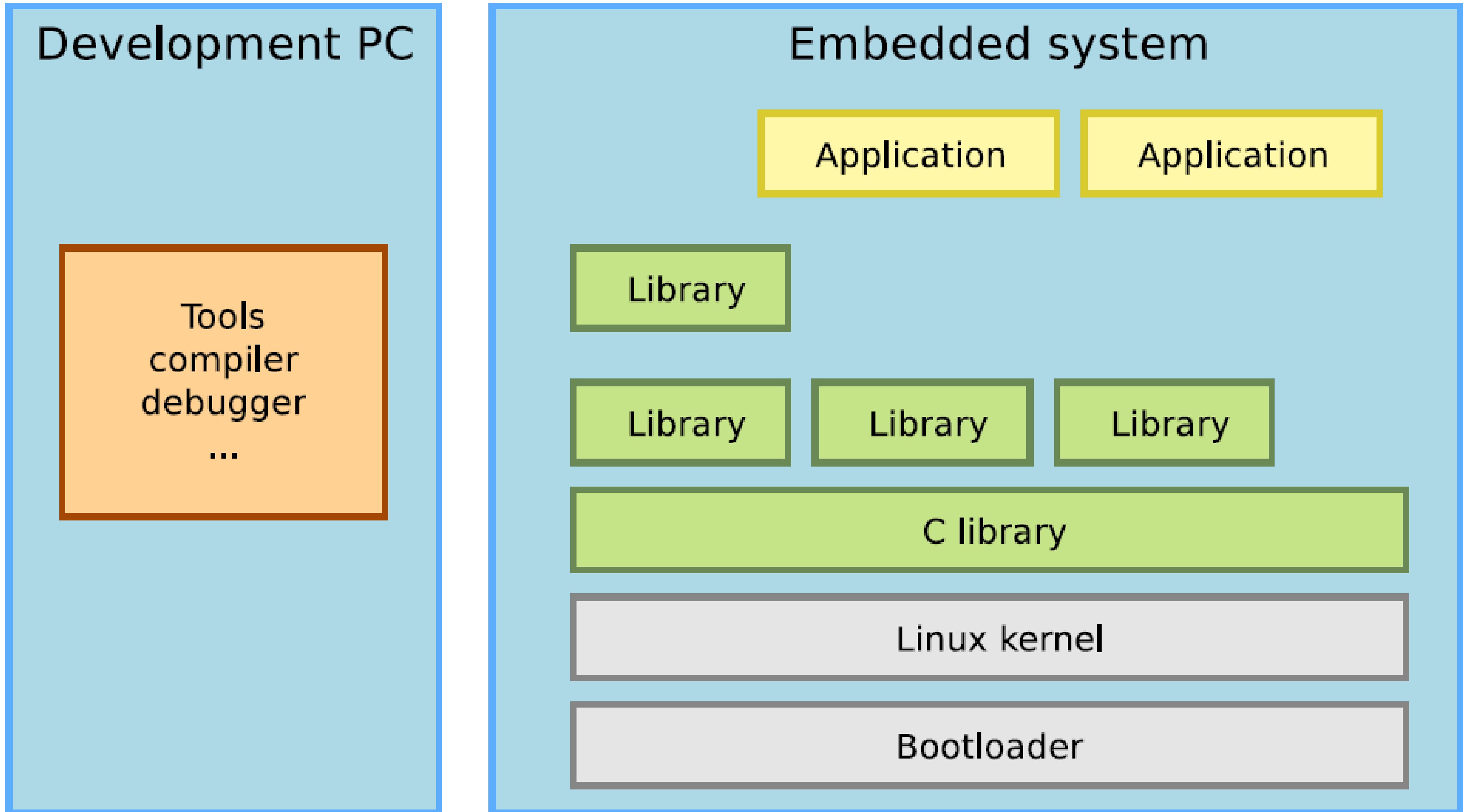
- Both MMU and no-MMU architectures are supported, even though no-MMU architectures have a few limitations.
- Linux is not designed for small microcontrollers.
- Besides the toolchain, the bootloader and the kernel, all other components are generally architecture-independent

RAM, Storage, Communication

- RAM: a very basic Linux system can work within 8 MB of RAM, but a more realistic system will usually require at least 32 MB of RAM. Depends on the type and size of applications.
- Storage: a very basic Linux system can work within 4 MB of storage, but usually more is needed.
  - Flash storage is supported, both NAND and NOR flash, with specific filesystems
  - Block storage including SD/MMC cards and eMMC is supported
- Support for many common communication busses
  - I2C, SPI, CAN, 1-wire, SDIO, USB
- Extensive networking support
  - Ethernet, Wifi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
  - Firewalling, advanced routing, multicast

1. Development Setup

Embedded Linux System Architecture



Embedded Linux is the usage of the Linux kernel and various open-source components in embedded systems

Software components

- **Cross-compilation toolchain**
  - Compiler that runs on the development machine, but generates code for the target
- **Bootloader**
  - Started by the hardware, responsible for basic initialization, loading and executing the kernel
- **Linux Kernel**
  - Contains the process and memory management, network stack, device drivers and provides services to userspace applications
- **C library**
  - The interface between the kernel and the userspace applications
- **Libraries and applications**
  - Third-party or in-house

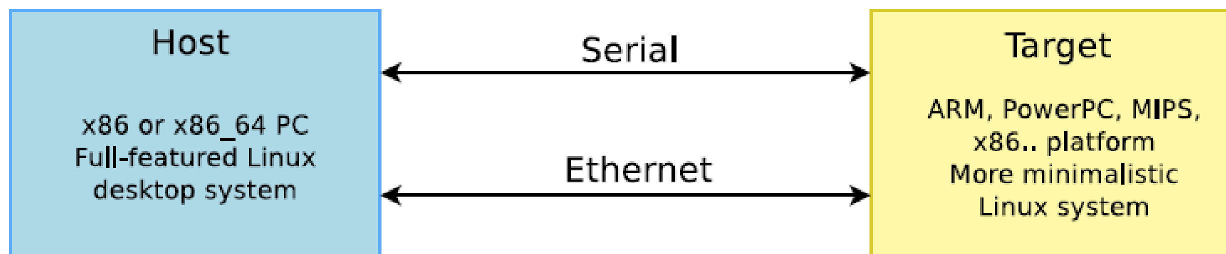
Embedded Linux work

- Several distinct tasks are needed when deploying embedded Linux in a product:

- Board Support Package (BSP) development
  - A BSP contains a bootloader and kernel with the suitable device drivers for the targeted hardware
- System integration
  - Integrate all the components, bootloader, kernel, third-party libraries and applications and in-house applications into a working system
- Development of applications
  - Normal Linux applications, but using specifically chosen libraries

Host vs. target

- When doing embedded development, there is always a split between
  - The host, the development workstation, which is typically a powerful PC
  - The target, which is the embedded system under development
- **Serial Port (RS232 cable)**
  - input/output from target (for monitoring system)
  - File transfer is possible but slow
  - Windows: HyperTerminal, Linux: Minicom, Picocom, Gtkterm, Putty, etc.
- **Network Port (LAN UTP cable)**
  - for high-speed file transfer
  - Monitor of control target board using Telnet
  - TFTP, NFS, etc.
- **JTAG Port (JTAG cable, Parallel cable)**
  - download boot-loader from PC to target
  - Jflash



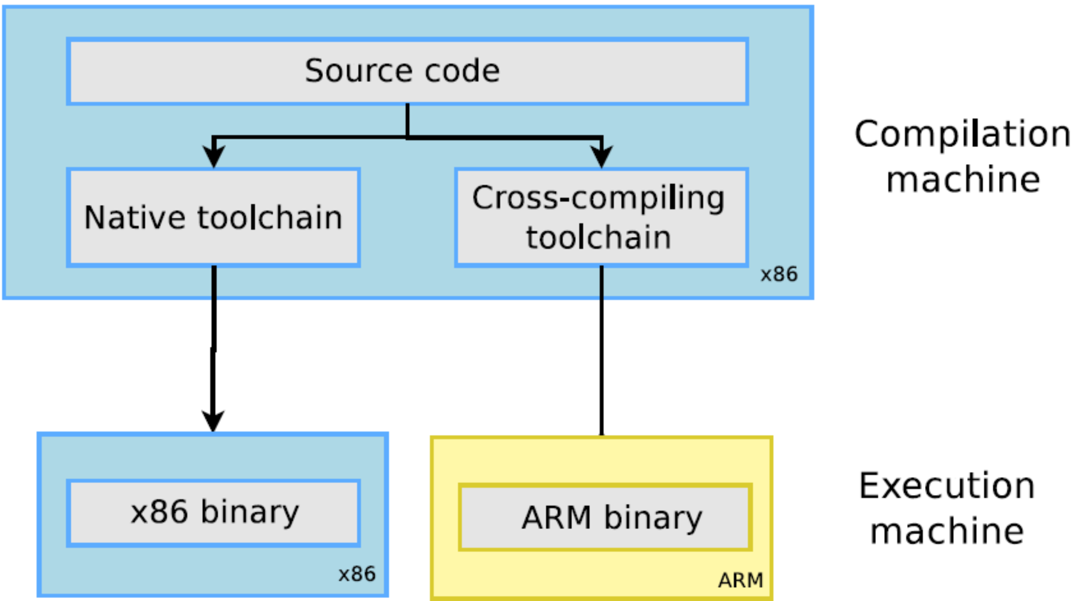
Cross-Compiling Toolchains

- The usual development tools available on a GNU/Linux workstation is a native toolchain
  - runs on your workstation and generates code for your workstation, usually x86

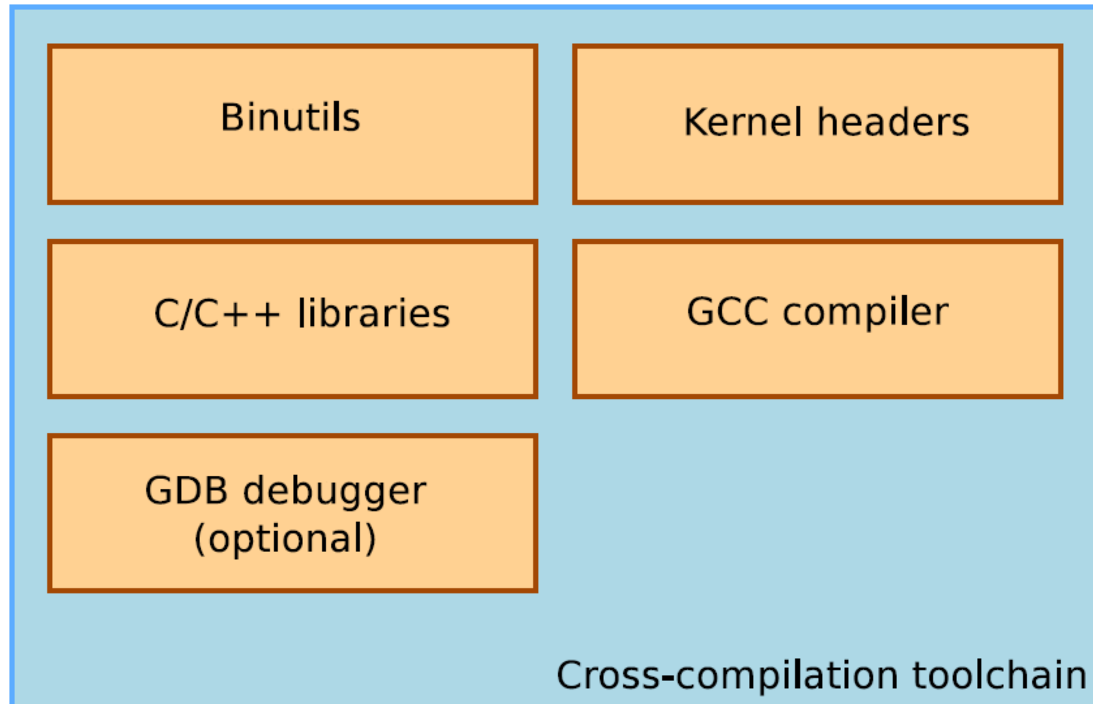
- For embedded system development, it is usually impossible or not interesting to use a native toolchain
  - The target is too restricted in terms of storage and/or memory
  - The target is very slow compared to your workstation
  - You may not want to install all development tools on your target.

- Cross-compiling toolchains run on your workstation but generate code for your target.

Cross-Compiling Toolchains



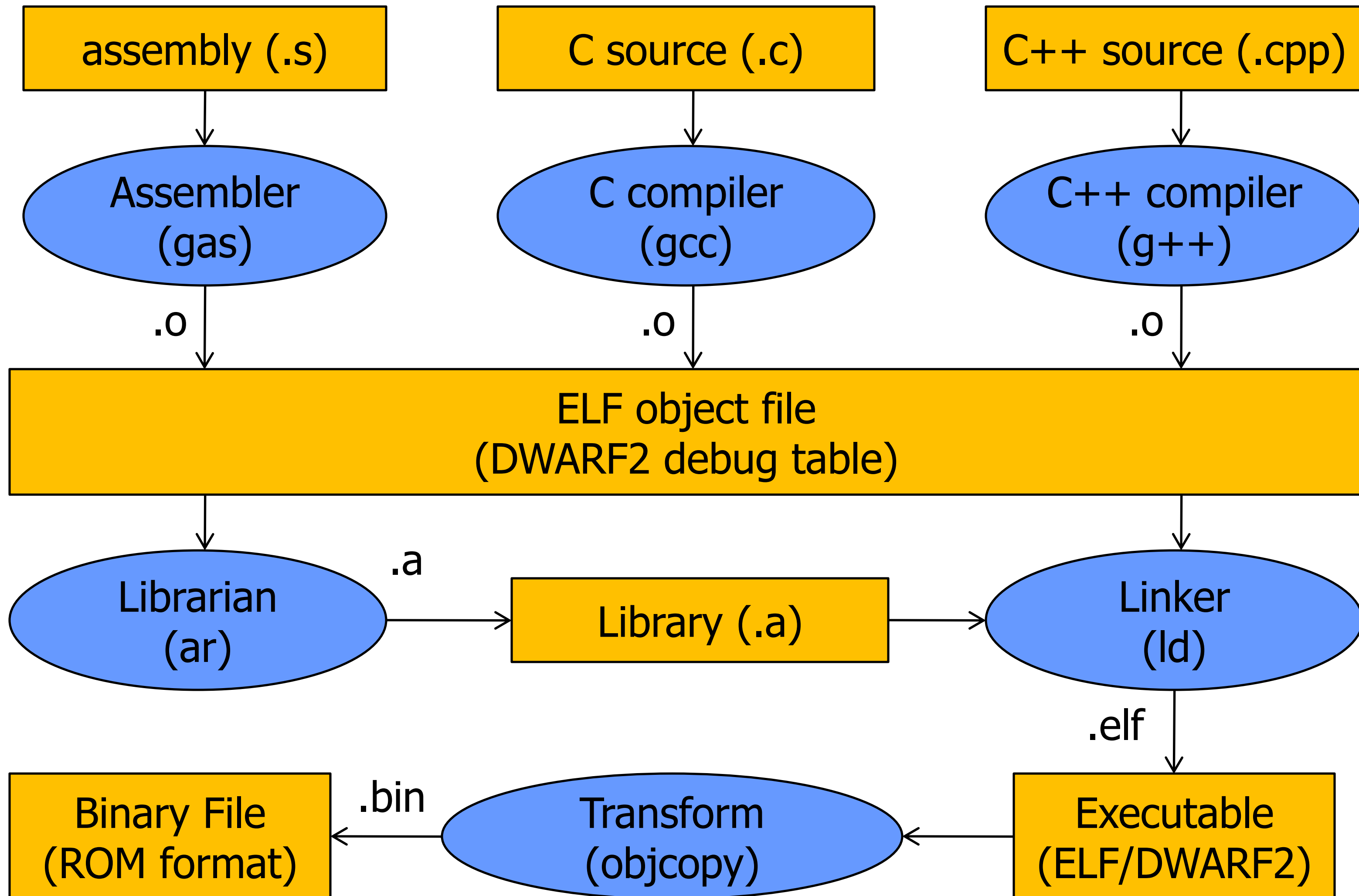
Toolchain Components



Binutils

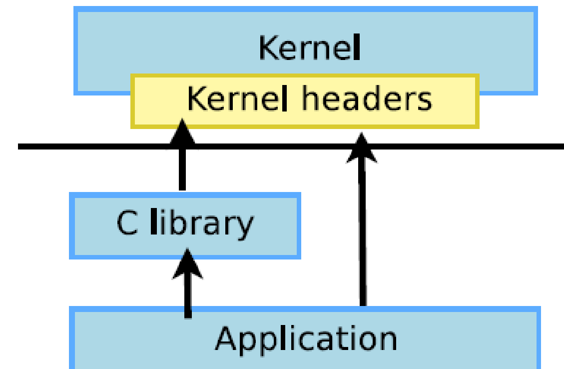
- a set of tools to generate and manipulate binaries for a given CPU architecture
  - as, the assembler, that generates binary code from assembler source code
  - ld, the linker
  - ar, ranlib, to generate .a archives, used for libraries
  - objdump, readelf, size, nm, strings, to inspect binaries. Very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

GNU Binutils



Kernel headers

- The C library and compiled programs needs to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Compiling the C library requires kernel headers, and many applications also require them.
- Available in <linux/...> and <asm/...> and a few other directories corresponding to the ones visible in include/ in the kernel sources



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

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

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

```
#define O_RDWR 00000002
```

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

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

C/C++ compiler

- GCC: GNU Compiler Collection, the famous free software compiler
- http://gcc.gnu.org/
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, Go, etc. Can generate code for a large number of CPU architectures, including ARM, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, x86, x86_64, IA64, Xtensa, etc.
- Available under the GPL license, libraries under the GPL with linking exception.
- Alternative: Clang / LLVM compiler
  - (http://clang.llvm.org/) getting increasingly popular and able to compile most programs (license: MIT/BSD type)

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*, *musl*, *klibc*, *newlib*...
- The choice of the C library must be made at cross-compiling toolchain generation time, as the GCC compiler is compiled against a specific C library.
- Comparing libcs by feature: http://www.etalabs.net/compare_libcs.html

glibc

- 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
- By default, quite big for small embedded systems. On armv7hf, version 2.23: libc: 1.5 MB, libm: 492 KB,
- But some features not needed in embedded systems can be configured out (merged from the old *eglibc* project).
- http://www.gnu.org/software/libc/

uClibc

- http://uclibc-ng.org/
- A continuation of the old uClibc project, license: LGPL
- Lightweight C library for small embedded systems
  - High configurability: many features can be enabled or disabled through a menuconfig interface.
  - Supports most embedded architectures, including MMU-less ones (ARM Cortex-M, Blackfin, etc.). The only library supporting ARM noMMU.
  - No guaranteed binary compatibility. May need to recompile applications when the library configuration changes.
  - Some glibc features may not be implemented yet (real-time, floating-point operations...)
  - Focus on size rather than performance
  - Size on armv7hf, version 1.0.24: libc: 652 KB,
- Actively supported, but Yocto Project stopped supporting it.

musl C library

- http://www.musl-libc.org/
- A lightweight, fast and simple library for embedded systems
- Created while uClibc’s development was stalled
- In particular, great at making small static executables
- Permissive license (MIT)
- Supported by build systems such as Buildroot and Yocto Project.
- Used by the Alpine Linux distribution
  - (https://www.alpinelinux.org/), fitting in about 130 MB of storage.

glibc vs uclibc-ng vs musl - small static executables

- Let’s compile and strip a hello.c program **statically** and compare the size
- With gcc 6.3, armel, musl 1.1.16: **7,300** bytes
- With gcc 6.3, armel, uclibc-ng 1.0.22 : **67,204** bytes.
- With gcc 6.2, armel, glibc: **492,792** bytes

bionic

- Android-specific custom library for C compiler (libc)
- Not glibc: License, Size, Speed issues
- BSD license
- ARM/x86 support only
- Partial pthreads, No SysV IPC, No STL(Standard Template Library)

ABI (Application Binary Interface)

- When building a toolchain, the ABI used to generate binaries needs to be defined
- ABI defines
  - calling conventions (how function arguments are passed, how the return value is passed, how system calls are made)
  - organization of structures (alignment, etc.)
- All binaries in a system must be compiled with the same ABI, and the kernel must understand this ABI.
- On ARM, two main ABIs: OABI and EABI (Embedded ABI)
  - Nowadays everybody uses EABI
- On MIPS, several ABIs: o32, o64, n32, n64

Floating point support

- Some processors have a floating point unit, some others do not.
  - For example, many ARMv4 and ARMv5 CPUs do not have a floating point unit. Since ARMv7, a VFP unit is mandatory.
- For processors having a floating point unit, the toolchain should generate hard float code, in order to use the floating point instructions directly
- For processors without a floating point unit, two solutions
  - Generate hard float code and rely on the kernel to emulate the floating point instructions. This is very slow.
  - Generate soft float code, so that instead of generating floating point instructions, calls to a user-space library are generated
- Decision taken at toolchain configuration time
- Also possible to configure which floating point unit should be used

CPU optimization flags

- A set of cross-compiling tools is specific to a CPU architecture (ARM, x86, MIPS, PowerPC)
- However, with the -march=, -mcpu=, -mtune= options, one can select more precisely the target CPU type
  - For example, -march=armv7 -mcpu=cortex-a8
- At the toolchain compilation time, values can be chosen.
- They are used:
  - As the default values for the cross-compiling tools, when no other -march, -mcpu, -mtune options are passed
  - To compile the C library
- Even if the C library has been compiled for armv5t, it doesn't prevent from compiling other programs for armv7

Obtaining a Toolchain

- Building a cross-compiling toolchain by yourself is a difficult and painful task!
  - Toolchain building utilities
  - Buildroot, PTXdist, OpenEmbedded/Yocto Project
- Get a pre-compiled toolchain
  - Advantage: it is the simplest and most convenient solution
  - Drawback: you can't fine tune the toolchain to your needs
- Determine what toolchain you need: CPU, endianism, C library, component versions, ABI, soft float or hard float, etc.
- Check whether the available toolchains match your requirements.
- Possible choices
  - Toolchains packaged by your distribution
    - Ubuntu example: sudo apt install gcc-arm-linux-gnueabihf
  - Sourcery CodeBench toolchains, now only supporting MIPS, NIOS-II, AMD64, Hexagon. Old versions with ARM support still available through build systems (Buildroot...)
  - Toolchain provided by your hardware vendor.