Linux for embedded systems

LECTURER: TRẦN NGUYÊN NGỌC

EMAIL: NGOCTN@SOICT.HUST.EDU.VN; NGOC.TRANNGUYEN@HUST.EDU.VN

OFFICE: 405-B1 HUST

Content

1. Why should we use Linux for embedded systems?
2. Example of embedded systems using Linux
3. Embedded hardware for Linux systems
4. Embedded Linux system architecture
5. Embedded Linux development environment
6. Cross-compiling toolchains
7. Toolchain options
8. Obtaining a toolchain

Why should we use Linux for embedded systems?

- Re-using components
  - Open-source
  - Many components for standard features
  - Wide-spread enough, high chance of having open-source components
  - Quickly design and develop complicated products.
  - Don’t re-develop yet another operating system kernel, TCP/IP stack, USB stack or another graphical toolkit library.
  - Focus on the added value of your product.
- Low cost
  - Free of charge.
  - Can reduce the cost of software licenses to zero.
    - The development tools are free, unless you choose a commercial embedded Linux edition.
  - You still need substantial learning and engineering efforts
  - Allows to have a higher budget for the hardware or to increase the company’s skills and knowledge

Why should we use Linux for embedded systems? (continue)

- Full control
  - Open-source
  - Unlimited modifications, changes, tuning, debugging, optimization, for an unlimited period of time
  - Without lock-in or dependency from a third-party vendor
  - Non open-source components must be avoided when the system is designed and developed
  - Have full control over the software part of your system
- Quality
  - Many open-source components are widely used, on millions of systems
  - Usually higher quality than what an in-house development can produce, or even proprietary vendors
  - Most widely-used open-source components have good quality.
  - Allows to design your system with high-quality components at the foundations

Why should we use Linux for embedded systems? (continue)

- Eases testing of new features
  - Open-source being freely available, it is easy to get a piece of software and evaluate it
  - Allows to easily study several options while making a choice
  - Much easier than purchasing and demonstration procedures needed with most proprietary products
  - Allows to easily explore new possibilities and solutions
- Community support
  - Open-source software components are developed by communities of developers and users
  - This community can provide high-quality support: you can directly contact the main developers of the component you are using. The likelyhood of getting an answer doesn't depend what company you work for.
  - Often better than traditional support, but one needs to understand how the community works to properly use the community support possibilities
  - Allows to speed up the resolution of problems when developing your system

Embedded hardware for Linux systems

Processor and architecture

- 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...

RAM and storage

- 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
- Not necessarily interesting to be too restrictive on the amount of RAM/storage: having flexibility at this level allows to re-use as many existing components as possible.

Communication

- The Linux kernel has support for many common communication buses
  - I2C
  - SPI
  - CAN
  - 1-wire
  - SDIO
  - USB
- And also extensive networking support
  - Ethernet, Wifi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
  - Firewalling, advanced routing, multicast

Types of hardware platforms

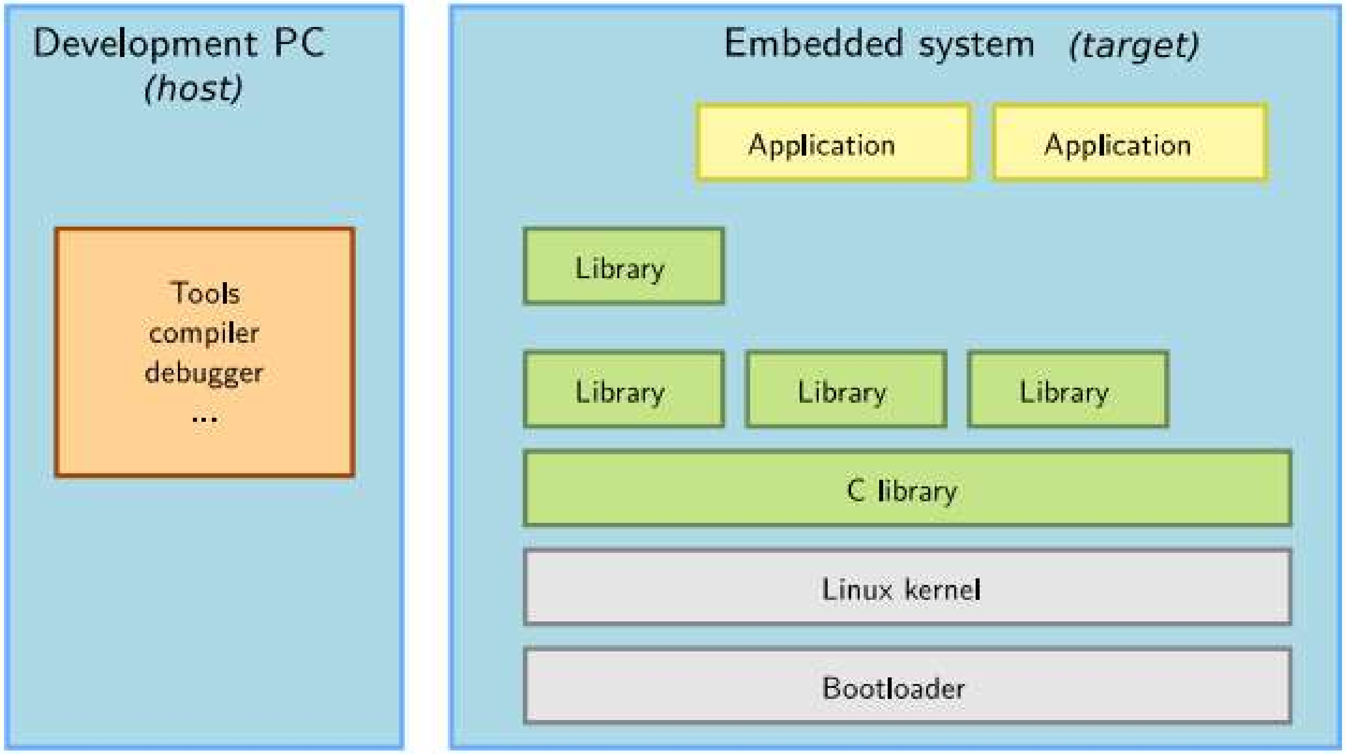
- Evaluation platforms from the SoC vendor. Usually expensive, but many peripherals are built-in. Generally unsuitable for real products.
- Component on Module, a small board with only CPU/RAM/flash and a few other core components, with connectors to access all other peripherals. Can be used to build end products for small to medium quantities.
- Community development platforms, to make a particular SoC popular and easily available. These are ready-to-use and low cost, but usually have less peripherals than evaluation platforms. To some extent, can also be used for real products.
- Custom platform. Schematics for evaluation boards or development platforms are more and more commonly freely available, making it easier to develop custom platforms.

Criteria for choosing the hardware

- Make sure the hardware you plan to use is already supported by the Linux kernel, and has an open-source ***bootloader***, especially the SoC you're targeting.
- Having support in the official versions of the projects (kernel, bootloader) is a lot better: quality is better, and new versions are available.
- Some SoC vendors and/or board vendors do not contribute their changes back to the mainline Linux kernel. Ask them to do so, or use another product if you can. A good measurement is to see the delta between their kernel and the official one.
- Between properly supported hardware in the official Linux kernel and poorly-supported hardware, there will be huge differences in development time and cost.

Embedded Linux system architecture

Host and target



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 user space applications
- C library
  - The interface between the kernel and the user space applications
- Libraries and applications
  - Third-party or in-house

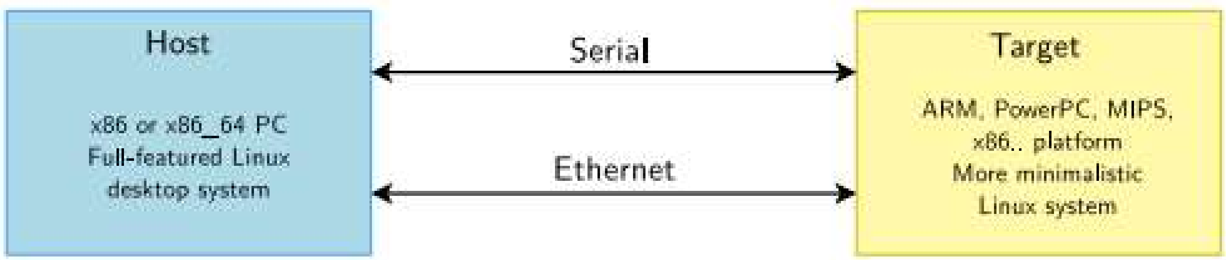
Embedded Linux development environment

Embedded Linux solutions

- Two ways to switch to embedded Linux
  - Use solutions provided and supported by vendors like MontaVista, Wind River or TimeSys. These solutions come with their own development tools and environment. They use a mix of open-source components and proprietary tools.
- Use community solutions. They are completely open, supported by the community.
- OS for Linux development
  - Recommend to use Linux as the desktop operating system to embedded Linux developers, for multiple reasons
- Any good and sufficiently recent Linux desktop distribution can be used for the development workstation
  - Ubuntu, Debian, Fedora, openSUSE, Red Hat, etc

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
- They are connected by various means: almost always a serial line for debugging purposes, frequently an Ethernet connection, sometimes a JTAG interface for low-level debugging



Need to install necessary packages/software/programs

Principle of software management

Software components	Manage sofware	How to manage
• Executable files • Libraries • Configuration files • Temporary files	• Install • Remove • Reconfigure • Get information	• Independent • Script for each software • Software database • Tools

Install software from source codes

Install source code

Compile source code
- Install dependent packages if needed

Install the software
- Installation scripting

Configure software
- Configuration scripting

Remove software
- Removal scripting

Scripts for all above operations

Makefile, Automake, make, other developing tools

Install software from management program

Program to install/remove/configure

Check the conflict with other software

Use tools/programs to manage software

- Software are packed as packages
- Software database
- Detect software conflict (redundance, missing, different versions)

Tools to manage software

	Redhat	Debian
Manage packages	rpm	dpkg
Package Management System	yum, urpm*, dnf	apt-*
Interactive interface	dselect, taskshell	aptitude
GUI	krpm, yumex	synaptic
Package repositories		/etc/apt/sources.list

apt-*

Same as yum but for Debian distributions

Some operations

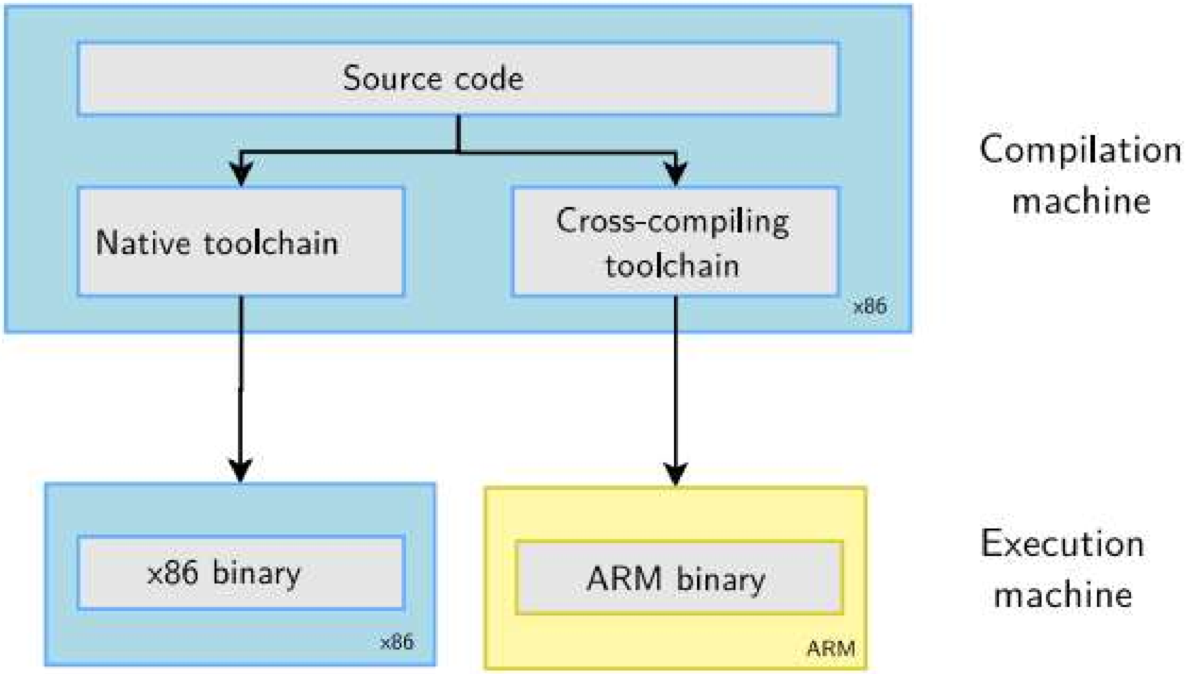
- apt-get install/ download/ remove/ source application
- apt-get clean
- apt-get update/ upgrade
- Note: New Debian OS: apt is a subset of apt-get or apt-cache

Cross-compiling toolchains

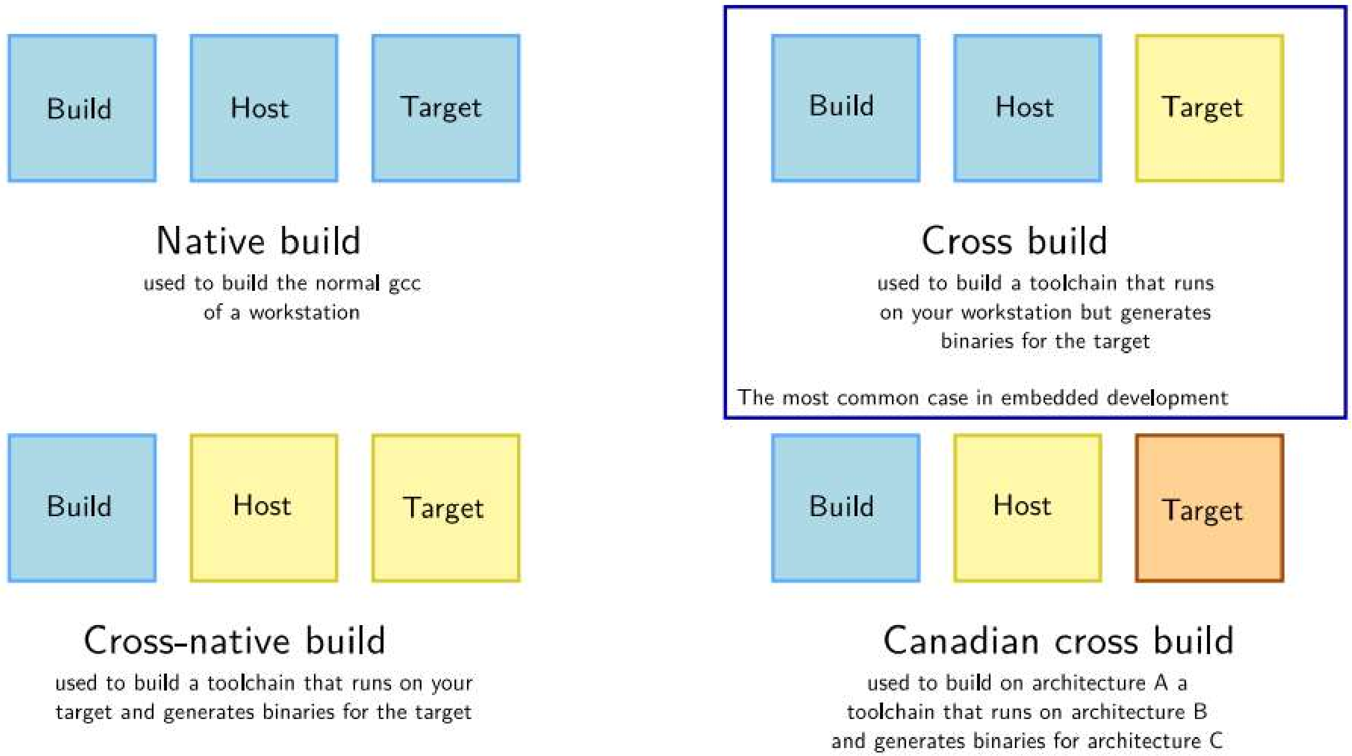
Toolchain

- 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.

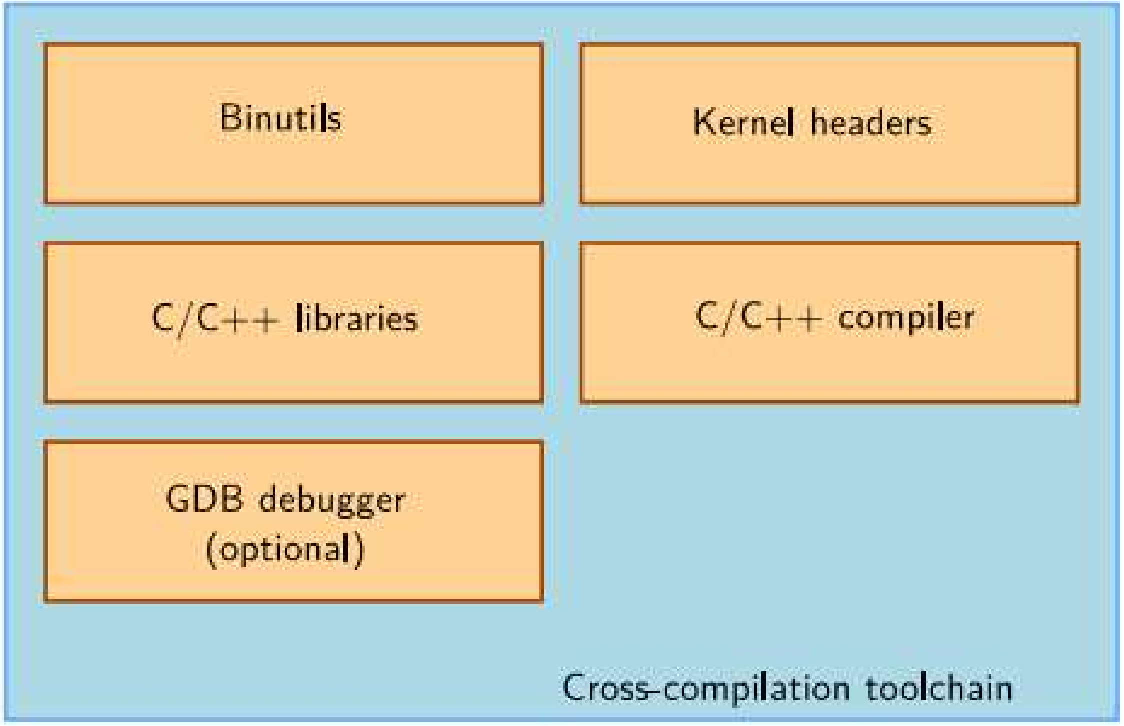
Toolchain


Source code
Native toolchain
Cross-compiling toolchain
x86
x86 binary
x86
ARM binary
ARM
Compilation machine
Execution machine

Different toolchain build procedures



Components


Binutils
Kernel headers
C/C++ libraries
C/C++ compiler
GDB debugger
(optional)
Cross-compilation toolchain

Building or obtaining a toolchain

Obtaining a toolchain

- Build a toolchain manually
  - Flexible
  - A difficult and painful task
- Get a pre-compiled toolchain
  - Advantage: simplest and most convenient solution
  - Disadvantage: you cannot fine tune the toolchains to your needs
  - Make sure the toolchain you find meets your requirements: CPU, endianesss, C library, component version, etc.
- Possible choice:
  - Toolchains packaged by your distribution
  - Sourcery CodeBench toolchains, now only supporting MIPS, NIOS-II, AMD64, Hexagon. Old versions with ARM support still available through build systems

Toolchain building utilities

- Another solution is to use utilities that automate the process of building the toolchain
  - Same advantage as the pre-compiled toolchains: you don’t need to mess up with all the details of the build process
  - But also offers more flexibility in terms of toolchain configuration, component version selection, etc.
  - They also usually contain several patches that fix known issues with the different components on some architectures
  - Multiple tools with identical principle: shell scripts or Makefile that automatically fetch, extract, configure, compile and install the different components
- Some solutions
  - Crosstool-ng
  - Buildroot
  - PTXdist
  - OpenEmbedded/ Yocto Project