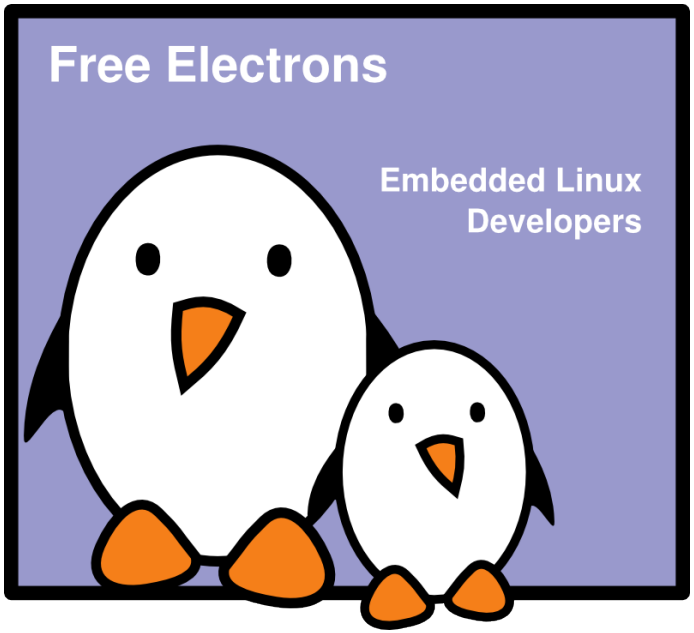
Embedded Linux Introduction



Thomas Petazzoni
Free Electrons
thomas.petazzoni@free-electrons.com

Thomas Petazzoni

- Thomas Petazzoni
  - Embedded Linux engineer and trainer at Free Electrons since January 2008
  - Linux user and developer since 2000
  - Given more than 120 days of embedded Linux training around the world
  - Major contributor to Buildroot, an opens-source, simple and fast embedded Linux build system

Free Electrons

- Free Electrons, specialized in Embedded Linux, since 2005
- Strong emphasis on community relation
- **Training**
  - *Embedded Linux system development*
  - *Linux kernel and device driver development*
  - All training materials freely available under a Creative Commons license.
- **Development and consulting**
  - Board Support Package development or improvement
  - Kernel and driver development
  - Embedded Linux system integration
  - Power-management, boot-time, performance audits and improvement
  - Embedded Linux application development

Free Electrons customers

ALSTOM Transport

ATMEL Everywhere You Are®

EBV Elektronik An Avnet Company

freescale semiconductor

Gamesa

IKUSI

logiplus

MICRONAS

MARVELL

FRANCE

Nokia Siemens Networks

SIEMENS

orange

THALES

Hydro Québec

Adeneo Embedded

Continental

Free Electrons trainings

Agenda

- **Introduction** : open-source and free software principles, advantages in the embedded space, hardware needed for embedded Linux
- **Open Source for embedded systems** : tools, bootloaders, kernel, system foundations, graphics and multimedia, networking, real-time, etc.
- **Development process of an embedded Linux system**
- **Commercial support, community support**
- **Android**
- **Conclusion**
- **Q&A**

About free software

Birth of free software

- **1983**, *Richard Stallman* : GNU project and concept of "free software". Start of development of *gcc*, *gdb*, *glibc*, etc. developed.
- **1991**, *Linus Torvalds* launches the LINUX project, an Unix-like operating system kernel. Together with GNU software and other free software components, it creates a complete and usable operating system: GNU/LINUX
- $\approx$ **1995**, Linux is more and more widely used on server systems.
- $\approx$ **2000**, Linux is more and more widely used in **embedded systems**
- $\approx$ **2005**, Linux is more and more widely used in desktop systems

Free software is no longer a *"new"* thing, it is well established since many years

Free software

A software is considered *free* when its license offers to all its users the following freedoms:

- Freedom to run the software, for any purpose
- Freedom to study how the software works, and change it
- Freedom to redistribute copies
- Freedom to distribute copies of modified versions

These freedoms are granted for both commercial and non-commercial use, without distinction.

Those freedoms imply that the source code is available, it can modified to match the needs of a given product, and the result can be distributed to customers $\Rightarrow$ good match for embedded systems!

Advantages of open-source in embedded systems

Re-using components

- The key advantage when using Linux and open-source components in embedded systems is the ability to **re-use existing components**.
- The open-source ecosystem already provides many components for standard features, from hardware support to network protocols, going through multimedia, graphic, cryptographic libraries, etc.
- As soon as a hardware, or a protocol, or a feature is wide-spread enough, high chance of having open-source components that support it.
- Allows to quickly design and develop complicated products, based on existing components.
- No-one should re-develop yet another operating system kernel, TCP/IP stack, USB stack or another graphical toolkit library.
- **Allows to focus on the added value of your product.**

Low cost

- Free software can be duplicated on as many devices as you want, free of charge.
- If your embedded system uses only free software, you can reduce the cost of software to zero. Even the development tools are free, unless you choose a commercial embedded Linux edition.
- **Allows to have an higher budget for the hardware or to increase the company's skills and knowledge**

Full control

- With open-source, you have the source code for all components in your system
- Allows unlimited modifications, changes, tuning, debugging, optimization, for an unlimited period of time
- Without locking or dependency from a third-party vendor
- To be true, non open-source components must be avoided when the system is designed and developed
- **Allows to have full control over the software part of your system**

Quality

- Many open-source components are widely used, on millions of systems
- Higher quality than what an in-house development can produce, or even proprietary vendors
- Of course, not *all* open-source components are of good quality, but most of the widely-used ones are.
- **Allows to design your system with high-quality components at the foundations**

Test of possible components

- Open-source being freely available, it is easy to get one 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 a high-quality support: you can directly contact the main developers of the component you are using
- Often better than traditionnal 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**

Taking part into the community

- Possibility of taking part into the development community of some of the components used in the embedded systems: bug reporting, test of new versions or features, patches that fix bugs or add new features, etc.
- Most of the time the open-source components are not the core value of the product: it's the **interest of everybody to contribute back**.
- For the **engineers**: a very motivating way of being recognized outside the company, communication with others in the same field, opening of new possibilities, etc.
- For the **managers**: motivation factor for engineers, allows the company to be recognized in the open-source community and therefore have more easily support and be more attractive to open-source developers

Drawbacks

- **Large choice**: community support or commercial support ? Which company for the support ? Which software solution ?
  - At the same time a strength and a drawback of free software and open source
- **New skills needed** compared to bare metal development or development with traditional embedded operating systems.
  - Need for training
  - Need for recruiting of new profiles
- **Licensing fear**
  - Generally over-exagerated

Hardware for embedded Linux

Processor 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 SoC (multimedia, industrial)
  - PowerPC (mainly real-time, industrial applications)
  - MIPS (mainly networking applications)
  - SuperH (mainly set top box and multimedia applications)
  - Blackfin (DSP architecture)
  - Microblaze (soft-core for Xilinx FPGA)
  - Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R
- 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-independant**

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

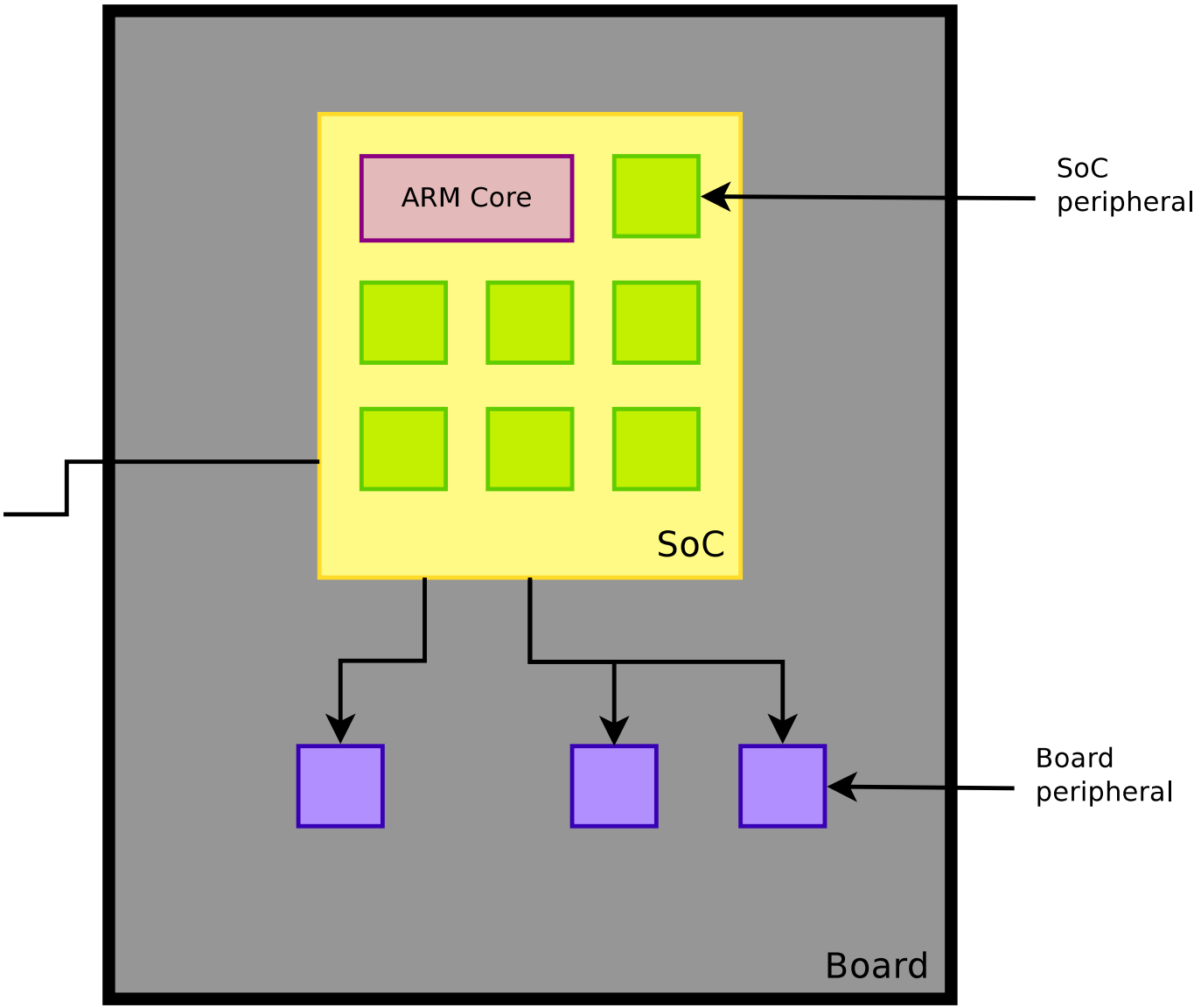
Communication

- The Linux kernel has support for many common communication busses
  - 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

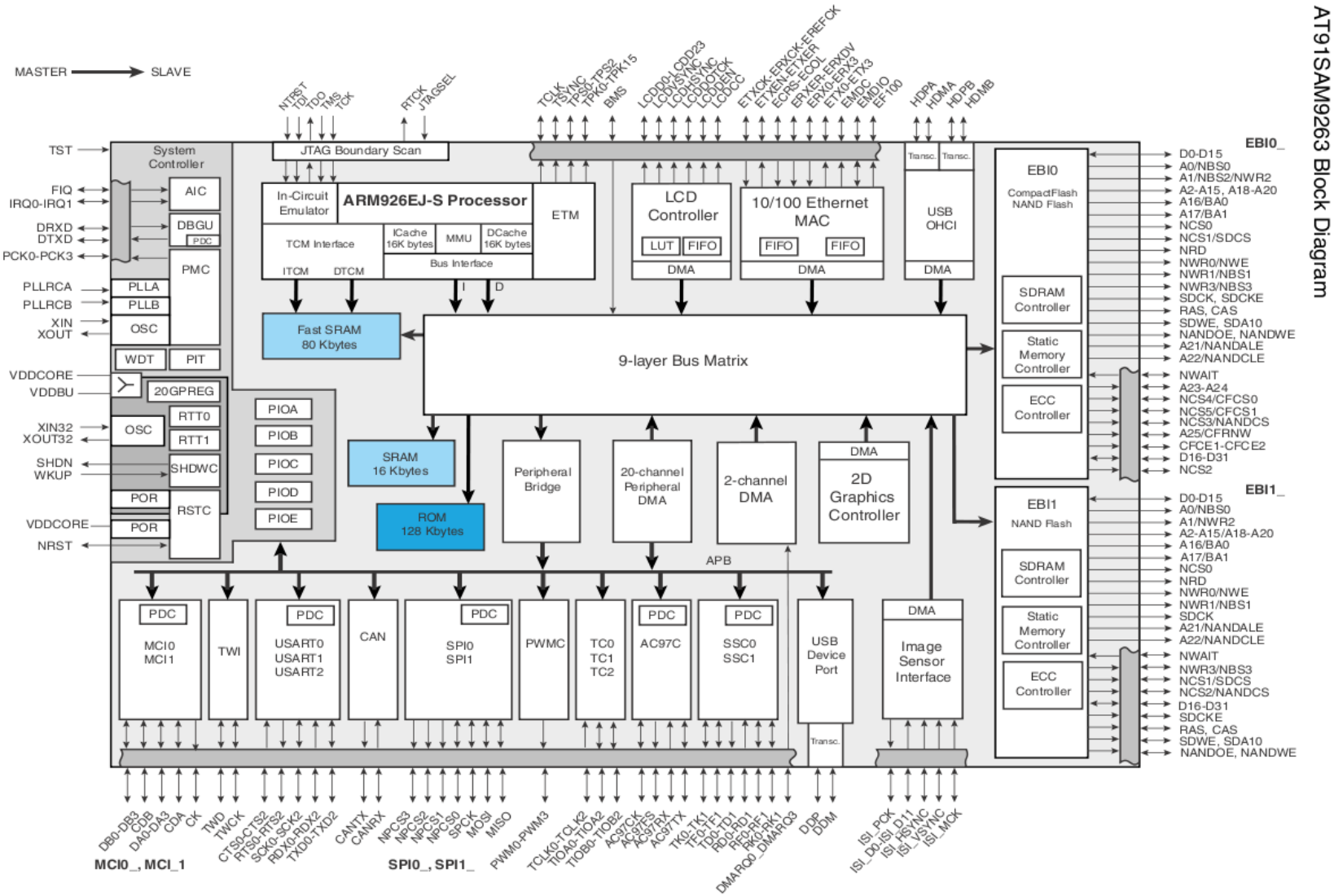
ARM and System-on-Chip

- ARM is one of the very popular architecture used in embedded Linux systems
- ARM designs **CPU cores** (instruction sets, caches, MMU, etc.) and sells the design to licensees
- The licensees are **founders** (Texas Instruments, Freescale, ST Ericsson, Atmel, etc.), they integrate an ARM core with many peripherals, into a chip called a **SoC**, for System-on-chip
- Each founder provides different models of SoC, with **different combination of peripherals**, power, power consumption, etc.
- The concept of SoC allows to reduce the number of peripherals needed on the board, and therefore the cost of designing and building the board.
- **Linux supports SoC from most vendors, but not all**, and not with the same level of functionality.

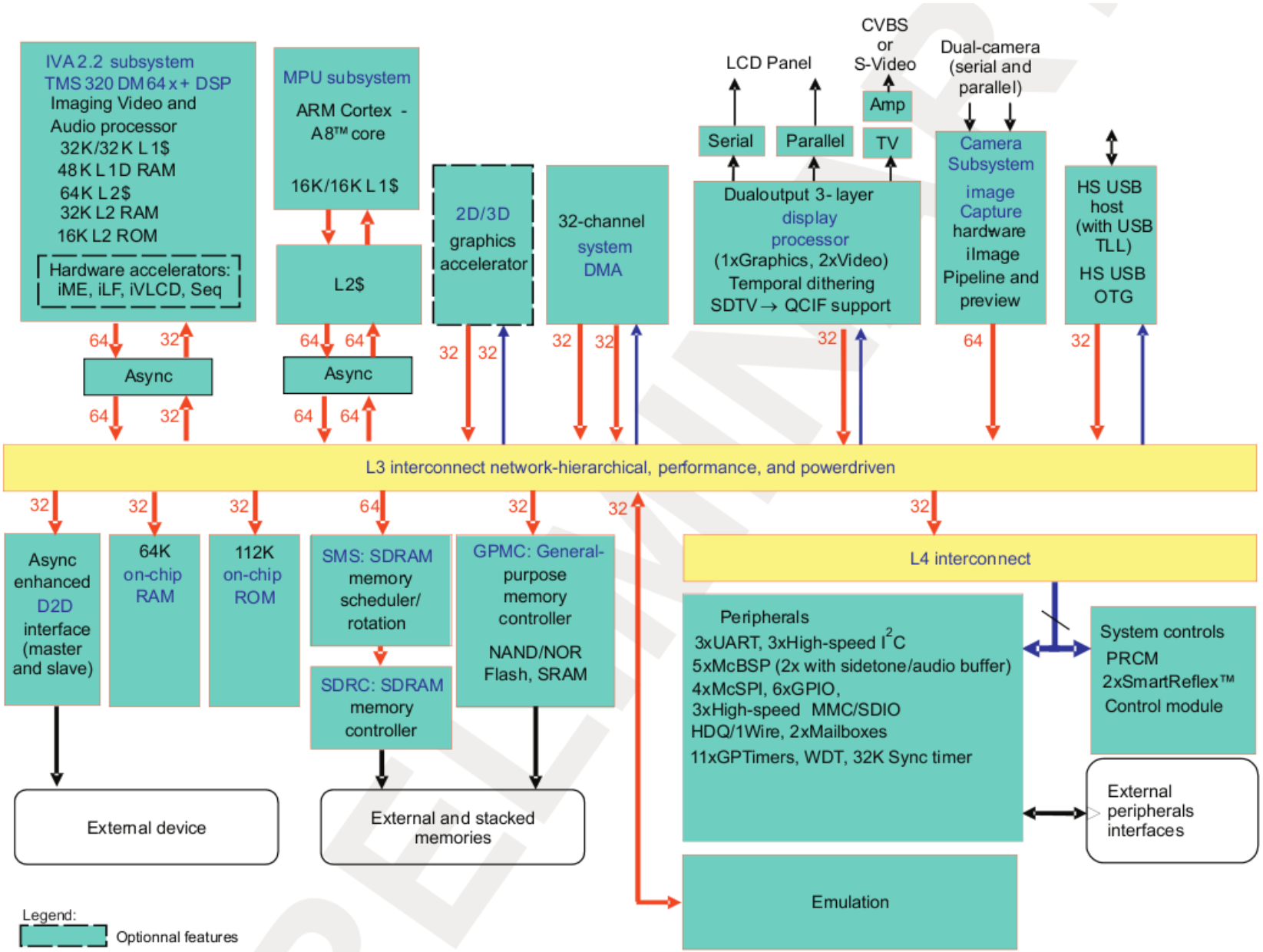
ARM and System-on-Chip



Atmel AT91SAM9263



Texas Instruments OMAP3430



Type of hardware platforms

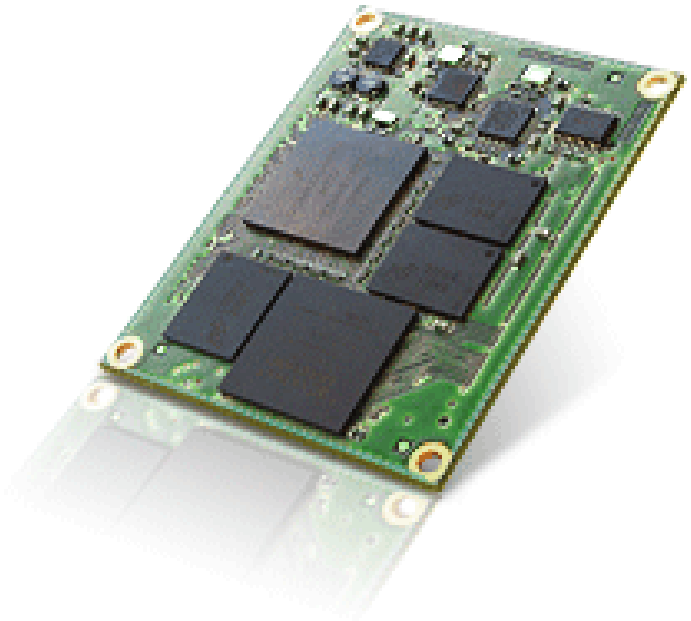
- **Evaluation platforms** from the SoC vendor. Usually expensive, but many peripherals are built-in. Generally unsuitable for real products.
- **Component on Module**, a small board with only CPU/RAM/Flash and a few other core components, with connectors to access all other peripherals. Can be used to build end products for small to medium quantities.
- **Community development platforms**, a new trend to make a particular SoC popular and easily available. Those 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.

Gumstix

- **TI OMAP3530 (600 Mhz, ARM Cortex-A8, PowerVR SGX, DSP)**
- Component on Module
- 256 MB RAM
- 256 MB NAND
- Bluetooth, Wifi
- uSD
- $229
- Development boards available at $ 49-229, with many peripherals: LCD, Ethernet, UART, SPI, I2C, etc.
- `http://www.gumstix.com`

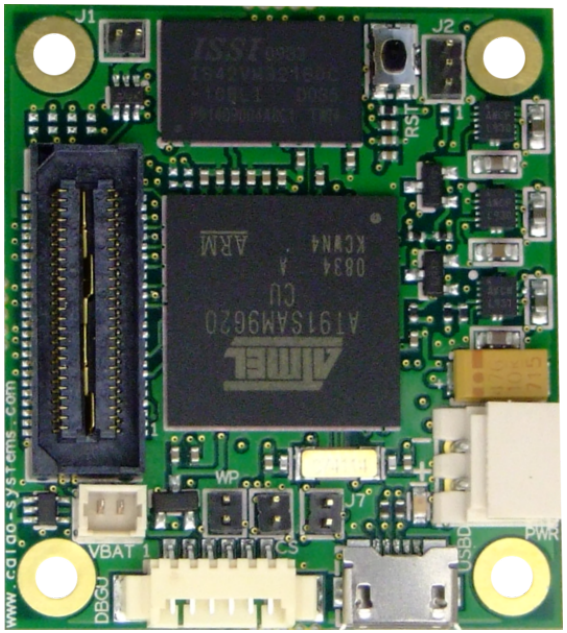
Armadeus

- **Freescale i.MX 27, 400 Mhz**
- Component on Module
- 128 MB RAM
- 256 MB NAND
- FPGA Xilinx Spartan 3A
- 116 €
- Development boards available at 130-230 €, with peripherals (Ethernet, serial, etc.) and access to UART, SPI, I2C, PWM, GPIO, USB, SD, LCD, Keypad, etc.
- `http://www.armadeus.com`



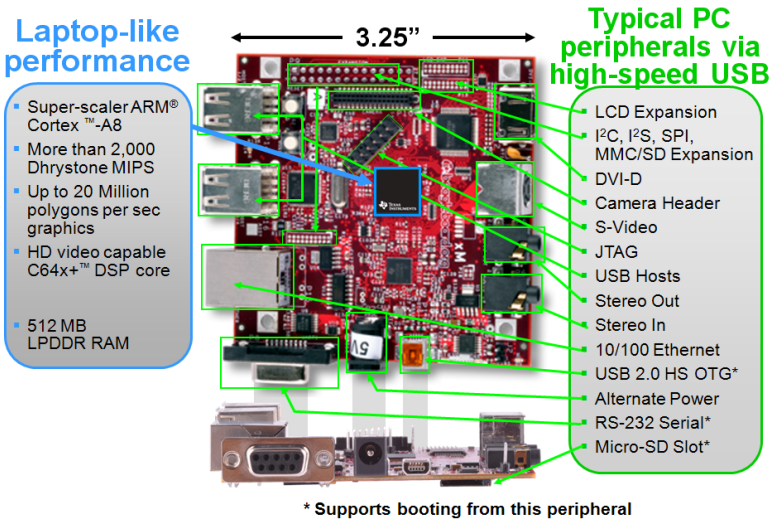
Calao TNY A9G20

- **Atmel AT91 9G20, 400 Mhz**
- Component on Module
- 64 MB RAM
- 256 MB NAND
- 152 €
- Many expansion boards available
- `http://www.calao-systems.com`



BeagleBoard

- **TI OMAP DM3730 (1 Ghz, cortex A8, PowerVR GPU, DSP)**
- Community development platform
- 512 MB RAM
- eMMC, microSD, Ethernet
- HDMI, S-Video, Camera, audio
- Expansion: USB, I2C, SPI, LCD, UART, GPIO, etc.
- $ 149
- `http://beagleboard.org`



Snowball

- **ST Ericson AP9500 (dual cortex A9, MALI GPU)**
- Community development platform
- 1 GB RAM
- 4-8 GB eMMC, microSD
- HDMI, S-Video, audio
- Wifi, Bluetooth, Ethernet
- Accelerometer, Magnetometer, Gyrometer, GPS
- Expansion: USB, I2C, SPI, LCD, UART, GPIO, etc.
- 169 € to 244 €
- `http://igloocommunity.org`

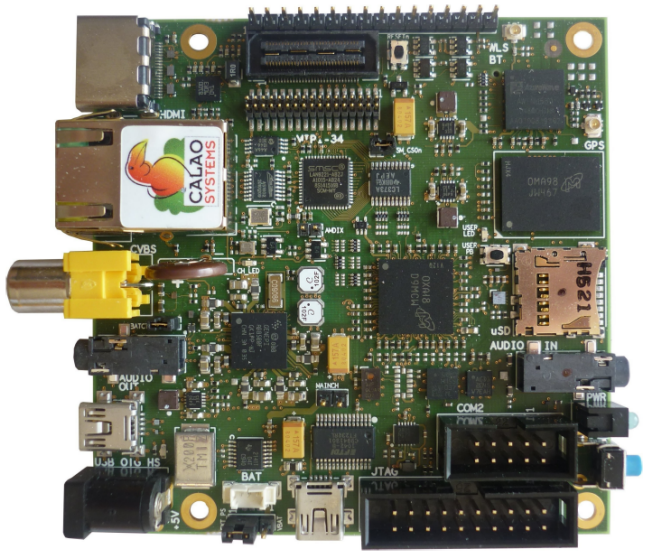
ARM

Cortex
Low-Power Leadership from ARM

Dual Cortex A9

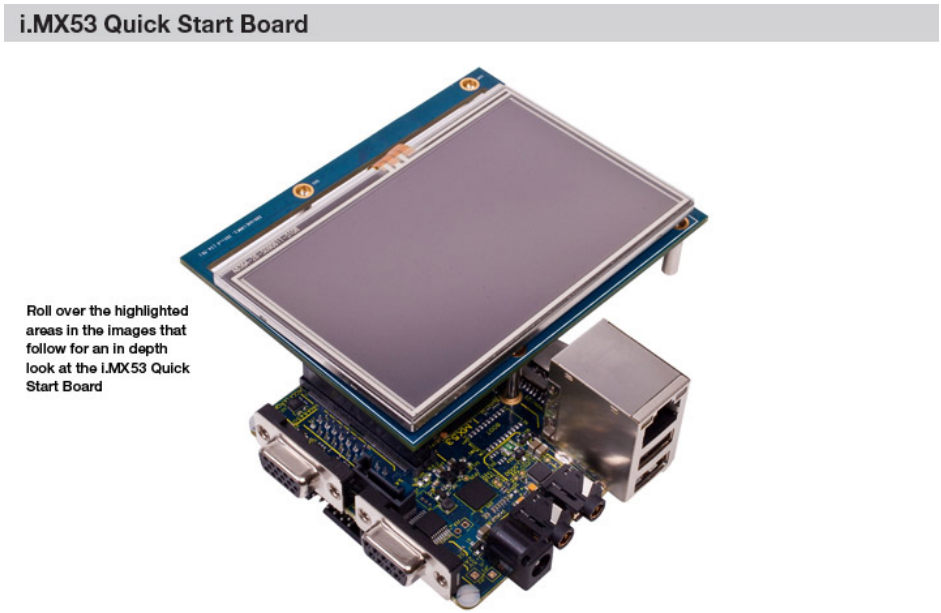
NEON™

mali



Freescale Quick Start

- **Freescale I.MX53 (1 Ghz Cortex A8)**
- Community development platform
- 1 GB RAM
- 4-8 GB eMMC, microSD
- LVDS, LCD, VGA, HDMI, audio
- Accelerometer, SD/MMC, uSD, SATA, Ethernet, USB
- Expansion: I2C, SPI, SSI, LCD, Camera
- 149 €

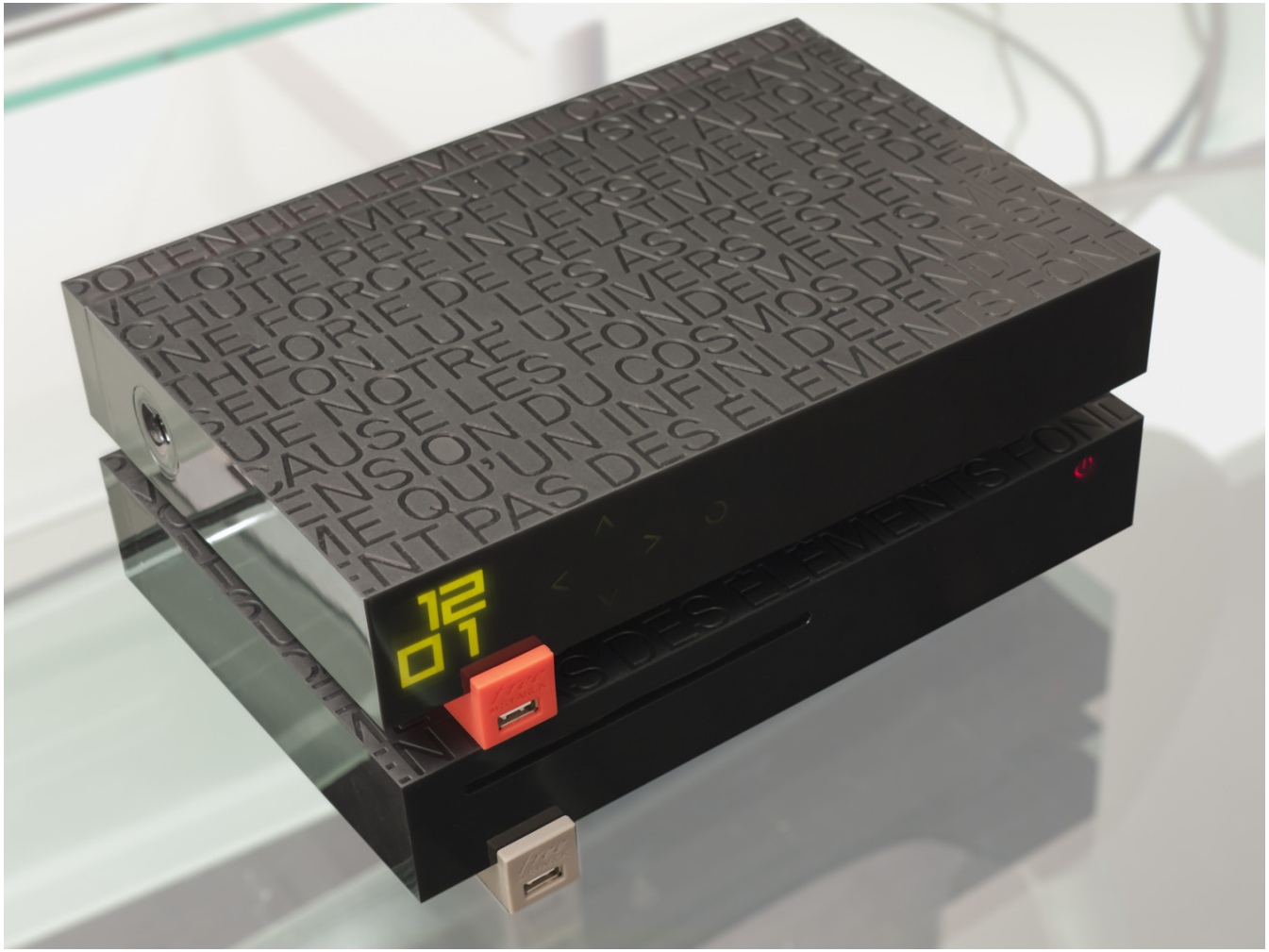


Criterias for choosing the hardware

- Make sure the hardware you plan to use is already supported by the Linux kernel, and 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 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 a properly supported hardware in the official Linux kernel and a poorly-supported hardware, there will be huge differences in development time and cost.

Embedded systems using Linux

Consumer device: Internet box

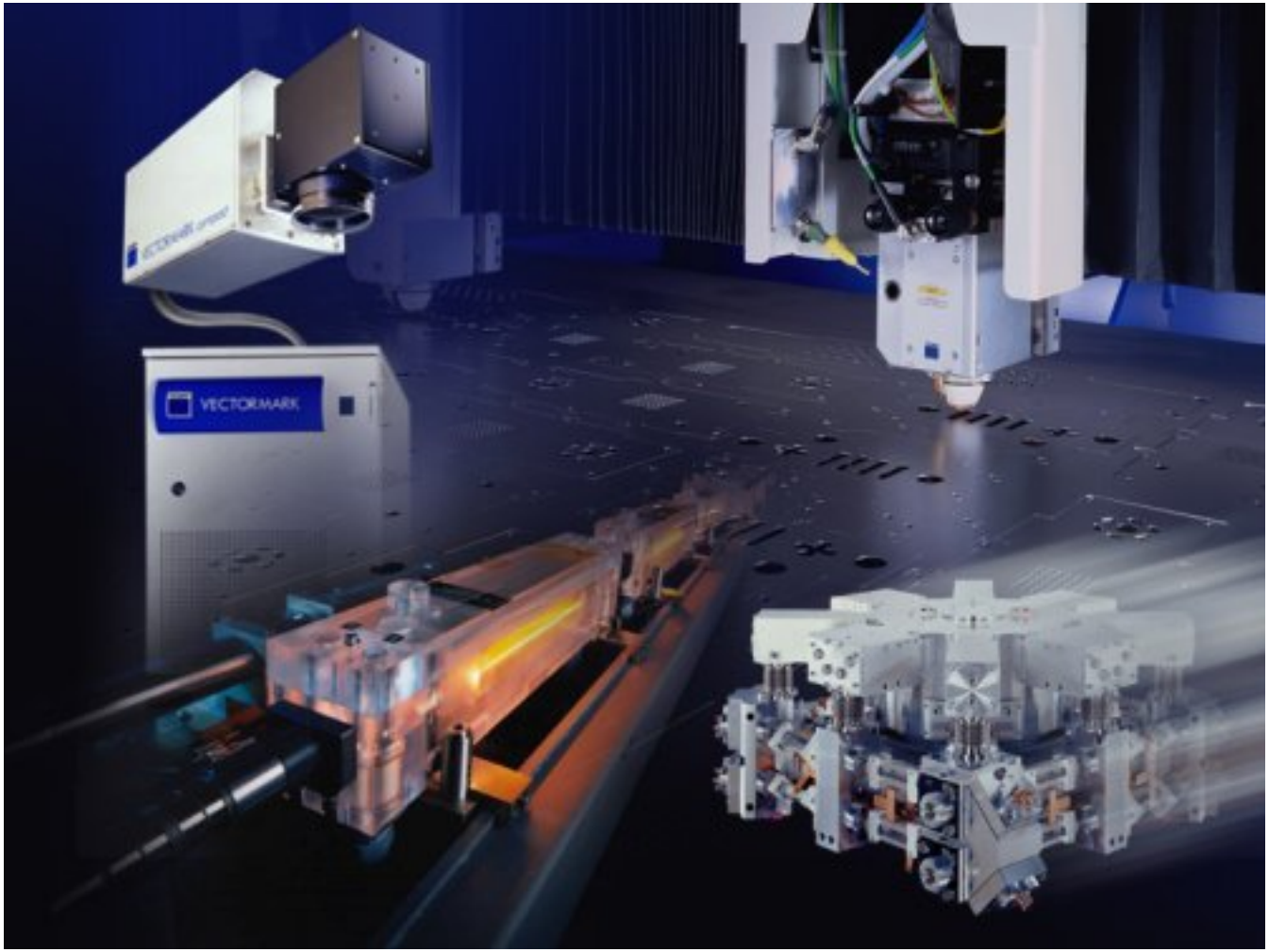


Consumer device: Network Attached Storage

Consumer device: Television

Professionnal device : point of sale terminal

Industrial system: laser cutter



Industrial system: wind turbine

Industrial system: cow milking

Industrial system: snow removal equipment

Industrial system: sea pollution detection system

Industrial system: viticultural machine

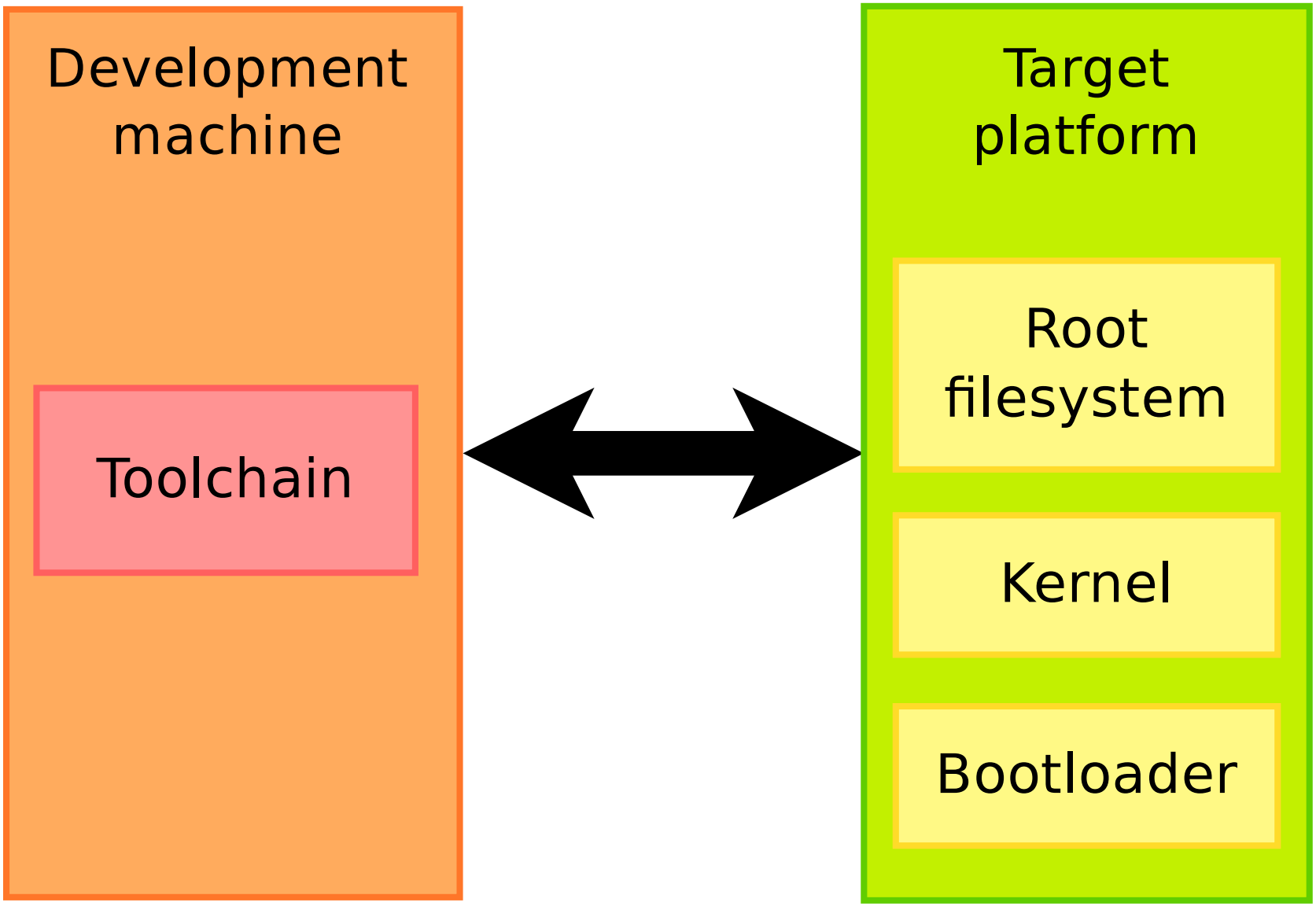
Open-source components for embedded systems

Four major components

Every embedded Linux system needs four major components to work:

- **Toolchain**, which doesn't run on the target platform, but allows to generate code for the target from a development machine.
- **Bootloader**, which is responsible for the initial boot process of the system, and for loading the kernel into memory
- **Linux Kernel**, with all the device drivers
- **Root filesystem**, which contains all the applications and libraries

Four major components



Toolchain

The toolchain is usually a *cross-compilation toolchain* : it runs on a development machine and generates code for the embedded platform. It has the following components:

- **binutils**, the binary manipulation utility including an *assembler* and a *linker*
- **gcc**, the C/C++ (and more) compiler, which is the standard in the open-source world
- **a C library**, which offers the POSIX interface to userspace applications. Several C library are available: *glibc*, *eglibc* and *uClibc*, with different size/features.
- **gdb**, the debugger, which allows remote debugging

Getting a cross-compilation toolchain

- **Pre-compiled toolchains** are the easiest solution. The toolchains from *CodeSourcery* are very popular.
  `http://www.codesourcery.com/`
- **Crosstool-NG** is a tool that automates the process of generating the toolchain. Allows more flexibility than pre-compiled toolchains.
  `http://ymorin.is-a-geek.org/projects/crosstool`
- **Embedded Linux build systems** are also usually capable of generating their own cross-compilation toolchain.

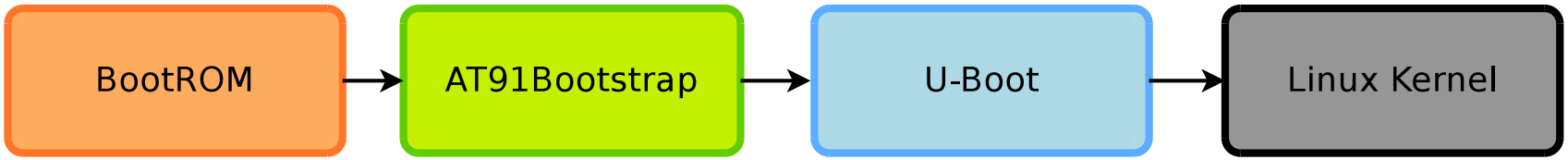
Make sure to get a toolchain that matches your hardware and your needs.
The toolchains provided by the hardware vendors are often old and rusty.

Bootloader: principle

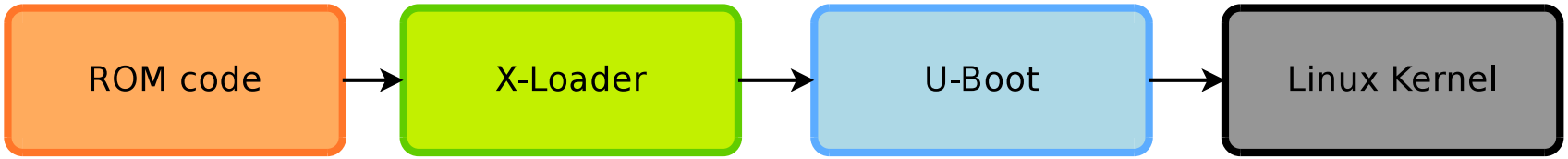
- The role of the bootloader is to initialize some basic hardware peripherals, load the Linux kernel image and run it.
- The boot process of most recent embedded processors is the following:
  1. The processor **executes code in ROM**, to load a first-stage bootloader from NAND, SPI flash, serial port or SD card
  2. The **first stage bootloader** initializes the memory controller and a few other peripherals, and loads a second stage bootloader. No interaction is possible with this first stage bootloader, and it is typically provided by the CPU vendor.
  3. The **second stage bootloader** offers more features: usually a shell, with commands. It allows to manipulate the storage devices, the network, configure the boot process, etc. This bootloader is typically generic and open-source.

Bootloader: principle

Atmel AT91



Texas Instruments OMAP3



Open-source bootloaders

- **U-Boot** is the de-facto standard in open-source bootloaders. Available on ARM, PowerPC, MIPS, m68k, Microblaze, x86, NIOS, SuperH, Sparc. Huge hardware support available, large number of features (networking, USB, SD, etc.)
  `http://www.denx.de/wiki/U-Boot`
- **Barebox**, a newer open-source bootloader, with a cleaner design than U-Boot, but less hardware support for the moment.
  `http://www.barebox.org`
- **GRUB**, the standard for x86 PC.
  `http://www.gnu.org/software/grub/`

Make sure your hardware comes with one of these well-known open-source bootloaders.

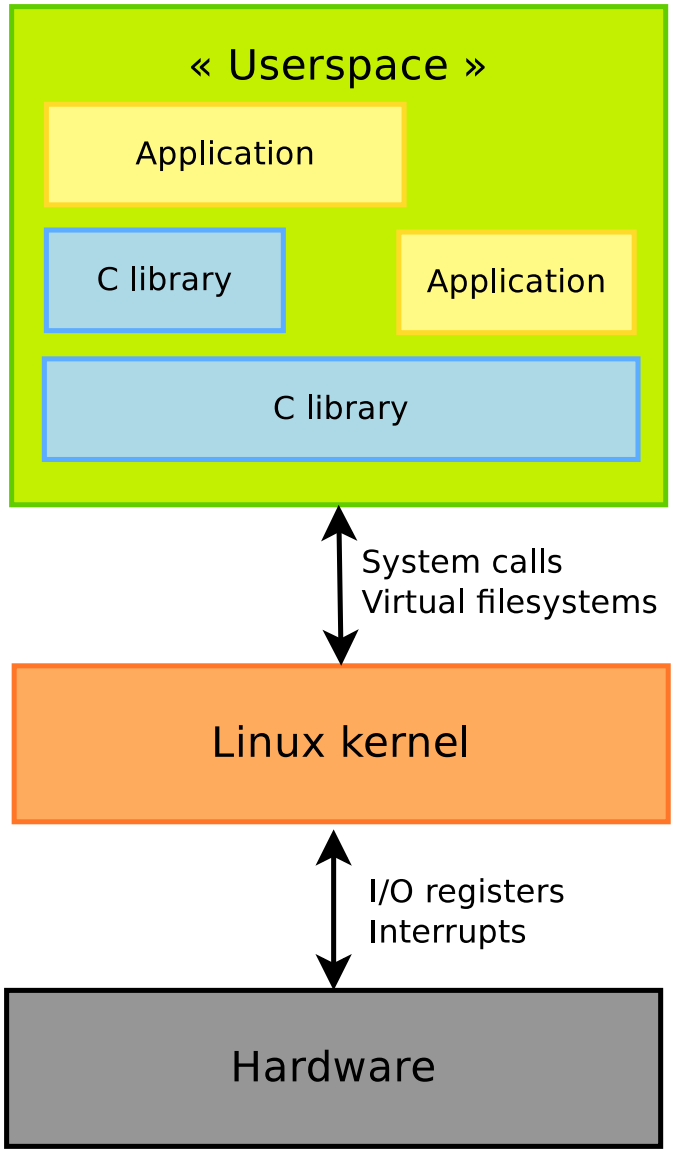
Linux kernel

- The Linux kernel is a core piece of the system. It provides the major following features:
  - Process management
  - Memory management
  - Inter-process communication, timers
  - Device drivers for the hardware: input, sound, network, storage, graphics, data acquisition, timers, GPIO, etc.
  - Filesystems
  - Networking
  - Power management
- The Linux kernel has thousands of options to selectively enable or disable features depending on the system needs.
- Under the GPLv2 license.
- `http://www.kernel.org`

Kernel vs. userspace

- The **Linux kernel** runs in *priviledged* mode. It can access and control the hardware. Its role is to multiplex the available ressources, and provide a coherent and consistent interfaces to userspace.
- **Userspace** is the set of applications and libraries that run on the system. They work in *unpriviledged mode*. They must go through the kernel to access the system ressources, including the hardware. The kernel provides isolation between applications.

Kernel vs. userspace



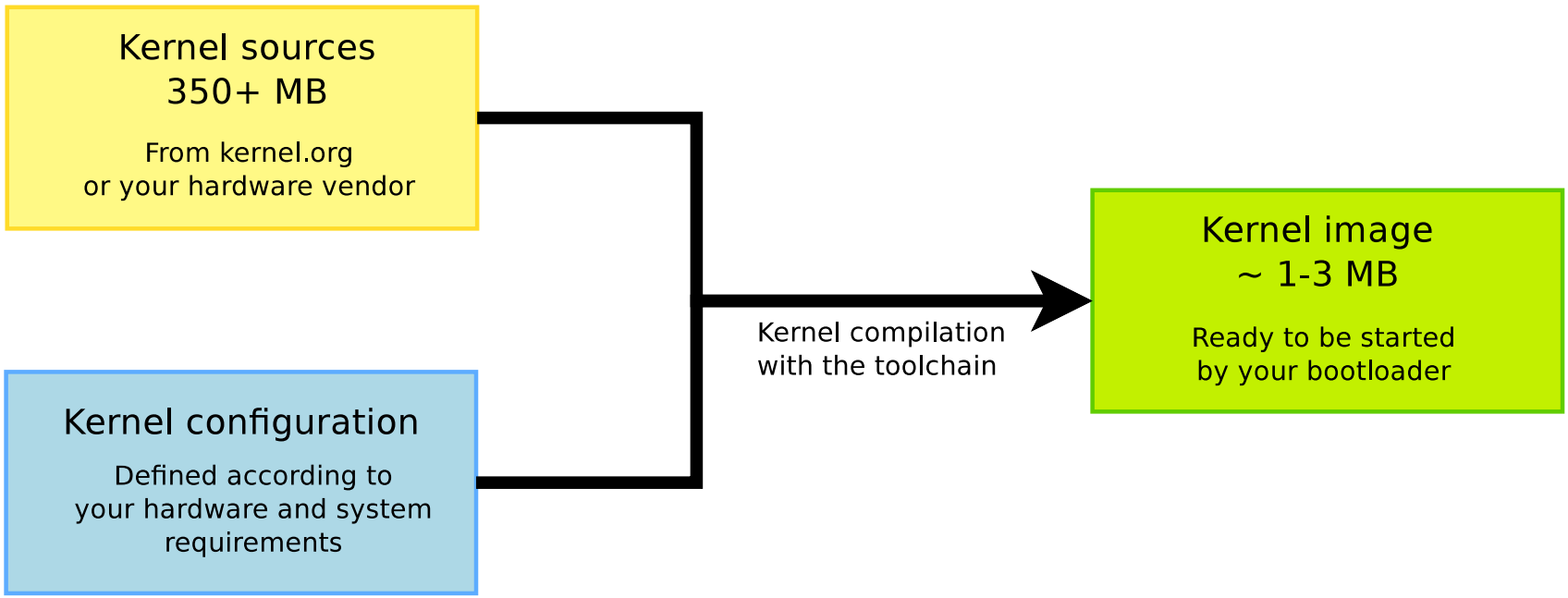
The kernel for your platform

The Linux kernel is highly-portable accross architectures.
Therefore, the code is split into several levels:

- Code **generic** to all architectures: ARM, x86, PowerPC, etc.
- Code **specific to an architecture**
- Code **specific to a System-on-Chip**: Atmel AT91, TI OMAP3, Freescale i.MX, etc.
- Code **specific to a board**, which consists of a particular SoC with additional peripherals on the board

If your SoC is well supported, the only part of the code that needs to be modified is the part specific to the board. Except if you need additional device drivers.

Kernel compilation



Kernel programming

- The programming environment inside the Linux kernel is **very different** from the one in userspace: different API, different constraints, different mechanisms
  - No standard C library, a specific API is available
  - No memory protection
- Programming in the kernel is typically needed to
  - **adapt the kernel to a particular board**
  - **write device drivers**
- There are many ressources on kernel programming:
  - *Linux Device Drivers*, book from O'Reilly
  - *Essential Linux Device Drivers*, book from Prentice Hall
  - Kernel programming training materials from Free Electrons

Root filesystem

- In a Linux system, applications, libraries, configuration and data are stored into files in a *filesystem*
- A global single hierarchy of directories and files is used to represent all the files in the system, regardless of their storage medium or location.
- A particular filesystem, called the **root filesystem** is mounted at the root of this hierarchy.
- This *root filesystem* typically contains all files needed for the system to work properly.

Filesystems

The Linux kernel supports a wide-range of filesystem types:

- **ext2, ext3, ext4** are the default filesystem types for Linux. They are usable on block devices.
- **jffs2, ubifs** are the filesystems usable on Flash devices (NAND, NOR, SPI flashes). Note that SD/MMC cards or USB keys are not Flash devices, but block devices.
- **squashfs** is a read-only highly-compressed filesystem, appropriate for all system files that never change.
- **vfat, nfts**, the Windows-world filesystems, are also supported for compatibility
- **nfs, cifs** are the two most important network filesystems supported

Minimum contents of a root filesystem

- An **init** application, which is the first application started by the kernel when the system boots. *init* is in charge of starting shells, system services and applications.
- A **shell** and associated tools, in order to interact with the system. The shell will typically operate over a serial port.
- The **C library**, which implements the POSIX interface, used by all applications.
- A set of **device files**. Those are special files that allow applications to perform operations on the devices managed by the kernel.
- A **typical Unix hierarchy**, with the `bin`, `dev`, `lib`, `sbin`, `usr/bin`, `usr/lib`, `usr/sbin`, `proc`, `sys`
- The *proc* and *sysfs* **virtual filesystems** mounted

Busybox

- In a normal Linux system, all core components are spread into different projects and not implemented with embedded constraints in mind
- Busybox provides a highly-configurable compilation of all the basic commands needed in a Linux system: `cp`, `mv`, `sh`, `wget`, `grep`, `init`, `modprobe`, `udhcpc`, `httpd`
- All those commands are compiled into a single binary, and the commands are symbolic links to this binary. Very space-efficient on systems where static compilation is used.
- Under the GPLv2 license
- `http://www.busybox.net`

Application development in a basic system

- With just Busybox and the C library, we have a fully working embedded Linux system
- The C library implements the well-known POSIX interface, which provides an API for process control, signals, file and directory operations, timers, pipes, the C standard library (string functions, etc.), memory management, semaphores, shared memory, thread management, etc.
- This is an already very comfortable environment to develop applications in C or C++ that can interact with the hardware devices, do some computations, react on hardware devices.
- Thanks to Busybox, you can even easily provide a Web server to monitor the system.

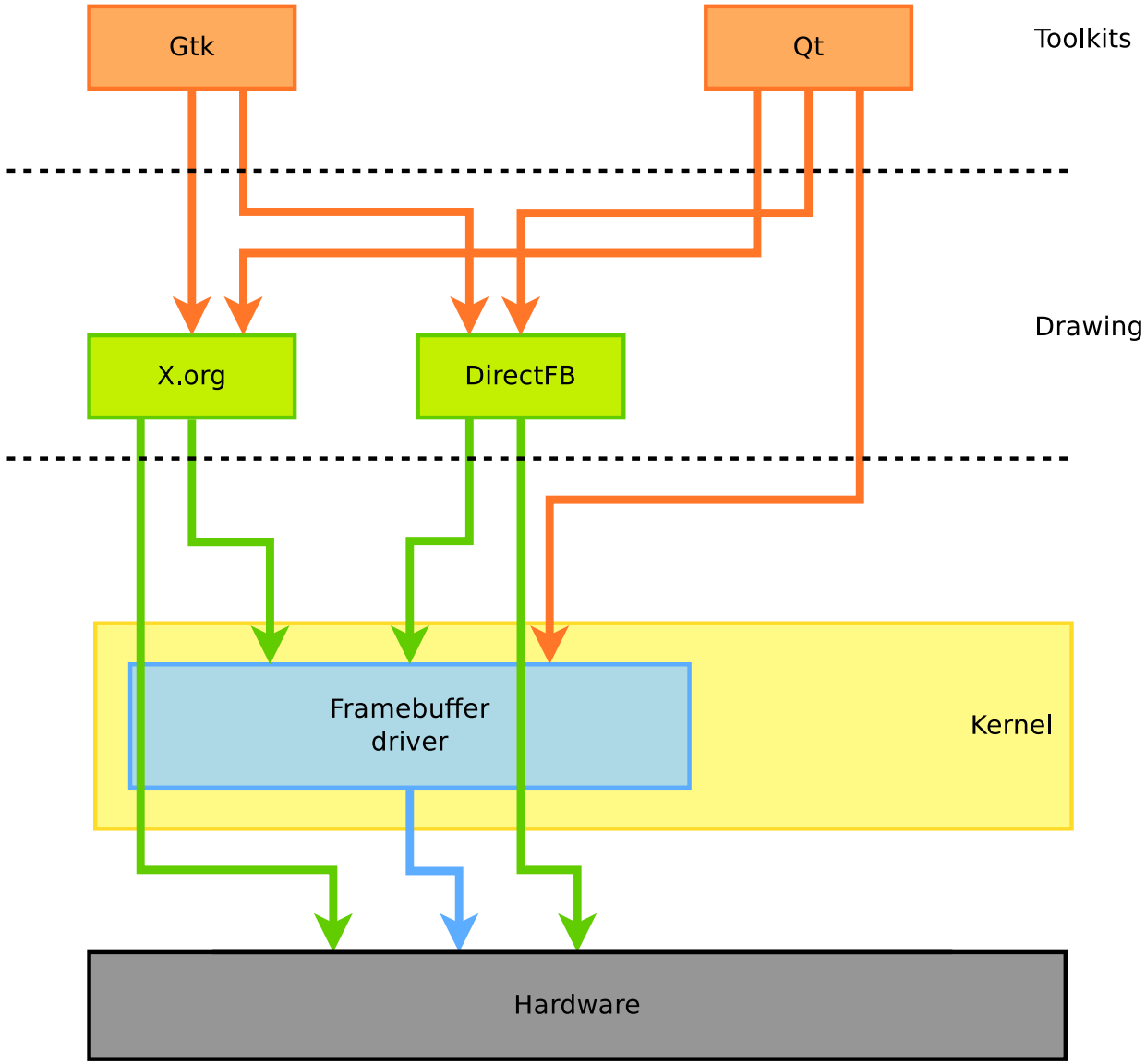
Interaction with the hardware

There are different ways of interacting with the hardware:

- The kernel has a device driver for the device, in which case it can be accessed either through a *device file* in `/dev` with the standard file API or through the networking API.
- Userspace applications running as `root` can use the `/dev/mem` special device to access directly the physical memory, or better use the *UIO* framework of the kernel.
- For the SPI and I2C busses, there are special `/dev/spidevX` and `/dev/i2cdevX` to send/receive messages on the bus, without having a kernel device driver. For USB, the `libusb` library allows userspace USB device drivers.

The choice of one solution over another depends on the type of device, and the need of interaction with existing kernel subsystems.

Graphics


Gtk
Qt
Toolkits
Drawing
X.org
DirectFB
Framebuffer
driver
Kernel
Hardware

Drawing

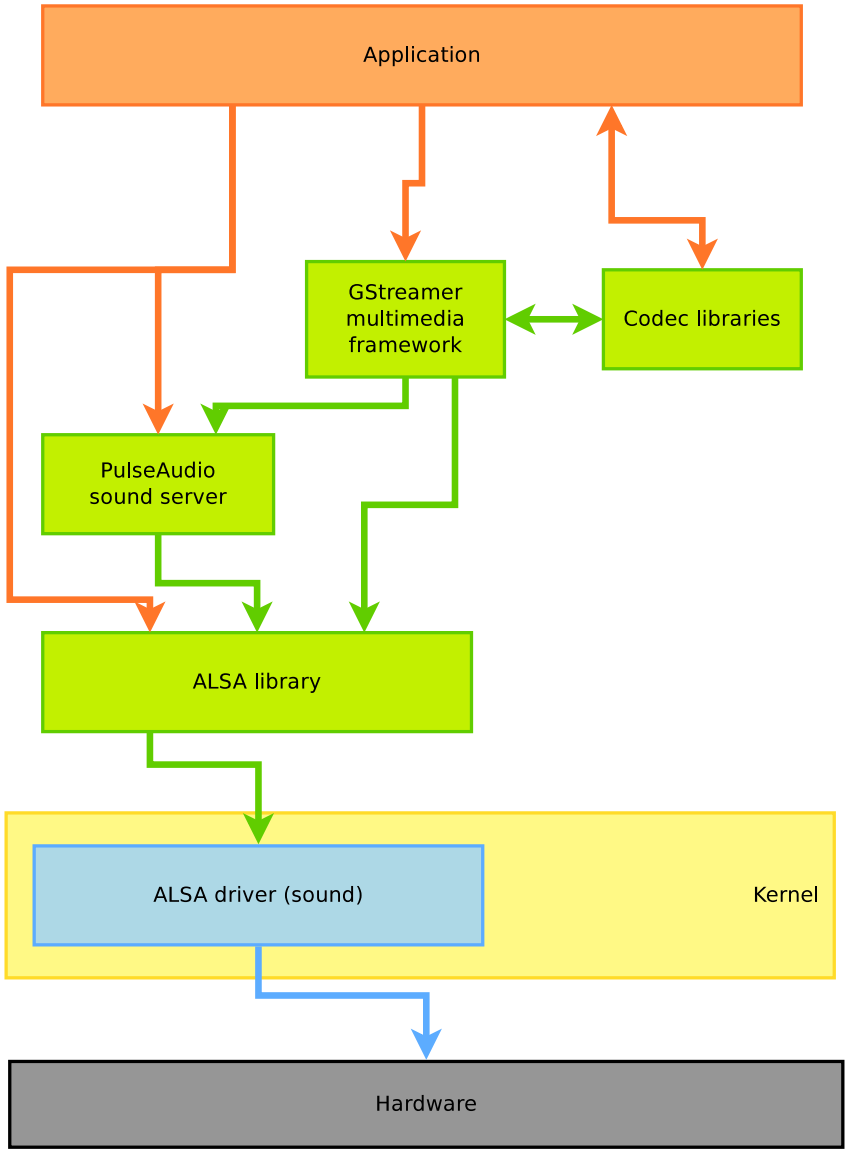
- **X.org**, `http://www.x.org`
  - A client/server approach. The server manages the hardware (graphics and input), and provides a protocol to clients. The clients are all applications that want to draw things and receive input events.
  - The solution used on all desktop Linux systems. The X Server implements the well-known *X11* protocol.
  - Client libraries are available, but applications can hardly be written against them: too low-level. *Toolkits* are used on top of it.
- **DirectFB**, `http://www.directfb.org`
  - A library over the kernel framebuffer driver, which allows handles input events.
  - More lightweight than the X Server, but without the *X11* protocol compatibility, and with a bit less features.
  - The API can directly be used to develop applications, but *toolkits* can also be used on top of it.

Toolkits

Toolkits provide high-level API to build graphical interfaces with windows, buttons, text inputs, drop-down lists, check boxes, canvas, etc.

- **Qt**, `http://qt.nokia.com`
  - A complete development framework in C++, with event management, networking, timers, threads, XML... and graphics
  - Used as the foundation for the KDE desktop environment on Linux systems, but also very popular on embedded systems.
  - Works on X.org, on the framebuffer on or top of DirectFB
- **Gtk**, `http://www.gtk.org`
  - Also a complete development framework, in C.
  - Used as the foundation for the GNOME desktop environment on Linux systems. Probably less popular on embedded systems.
  - Works on X.org and on DirectFB

Sound


Application
GStreamer multimedia framework
Codec libraries
PulseAudio sound server
ALSA library
ALSA driver (sound)
Kernel
Hardware

Sound stack

- **ALSA** is the kernel subsystem for sound devices
  `http://www.alsa-project.org`
- **alsa-lib** is the userspace library that allows applications to use ALSA drivers. It is a fairly low-level library.
- **PulseAudio** is a sound server. It manages multiple audio streams, can adjust their volume independently, redirect them dynamically, etc.
  `http://www.pulseaudio.org`
- **GStreamer** is a multimedia framework. With a collection of input, output, decoding and encoding plugins, one can build custom pipelines, to encode, decode and display video or audio streams
  `http://www.gstreamer.org`

Video stack

- *X.org* or the *framebuffer* are typically used as video-output devices
- The **Video4Linux** kernel subsystem supports video-input devices and some video-output devices that do overlay.
- The **GStreamer** multimedia framework video decoding and encoding

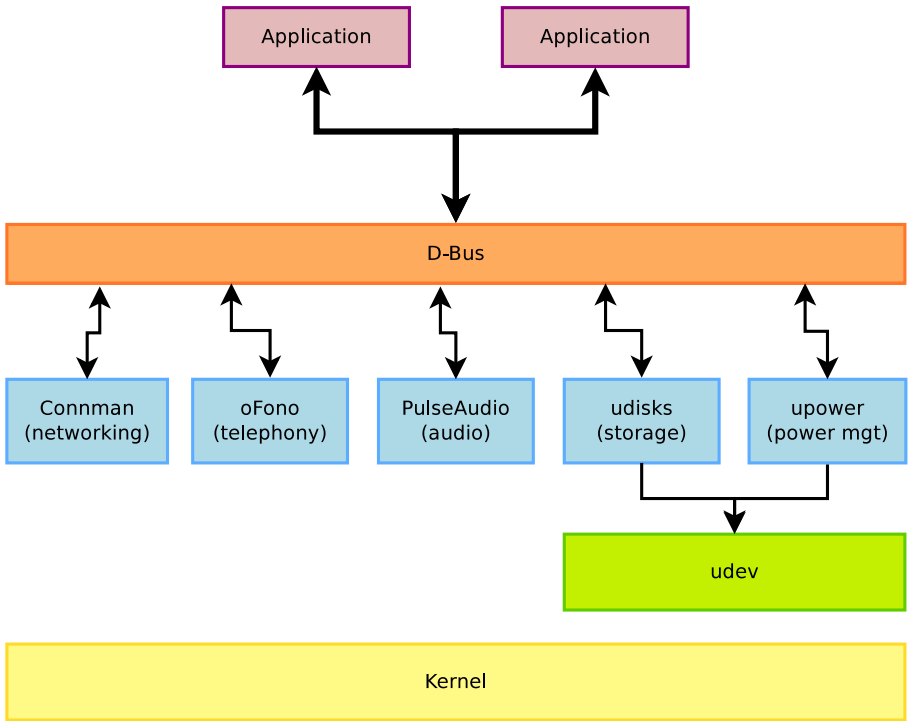
Networking

Linux is well-known system its networking capabilities:

- Support for Ethernet, CAN, Wifi, Bluetooth devices in the kernel, and associated userspace configuration applications
- Web servers: `httpd` in Busybox, lighttpd, boa, etc.
- SSH servers: Dropbear, OpenSSH
- Cryptography/VPN: OpenSSL, OpenVPN
- GPRS/Modem: `pppd`
- Firewall: *Netfilter* in the kernel, *iptables* command in userspace
- And also: mail server, SNMP, NTP, etc.

The service infrastructure

- Standardization of *services* around a inter-process communication bus: **D-Bus**. Allows programs to offer services to other programs, and to signal them events in the system.
- Used to split the low-level mechanisms from the graphical interfaces.



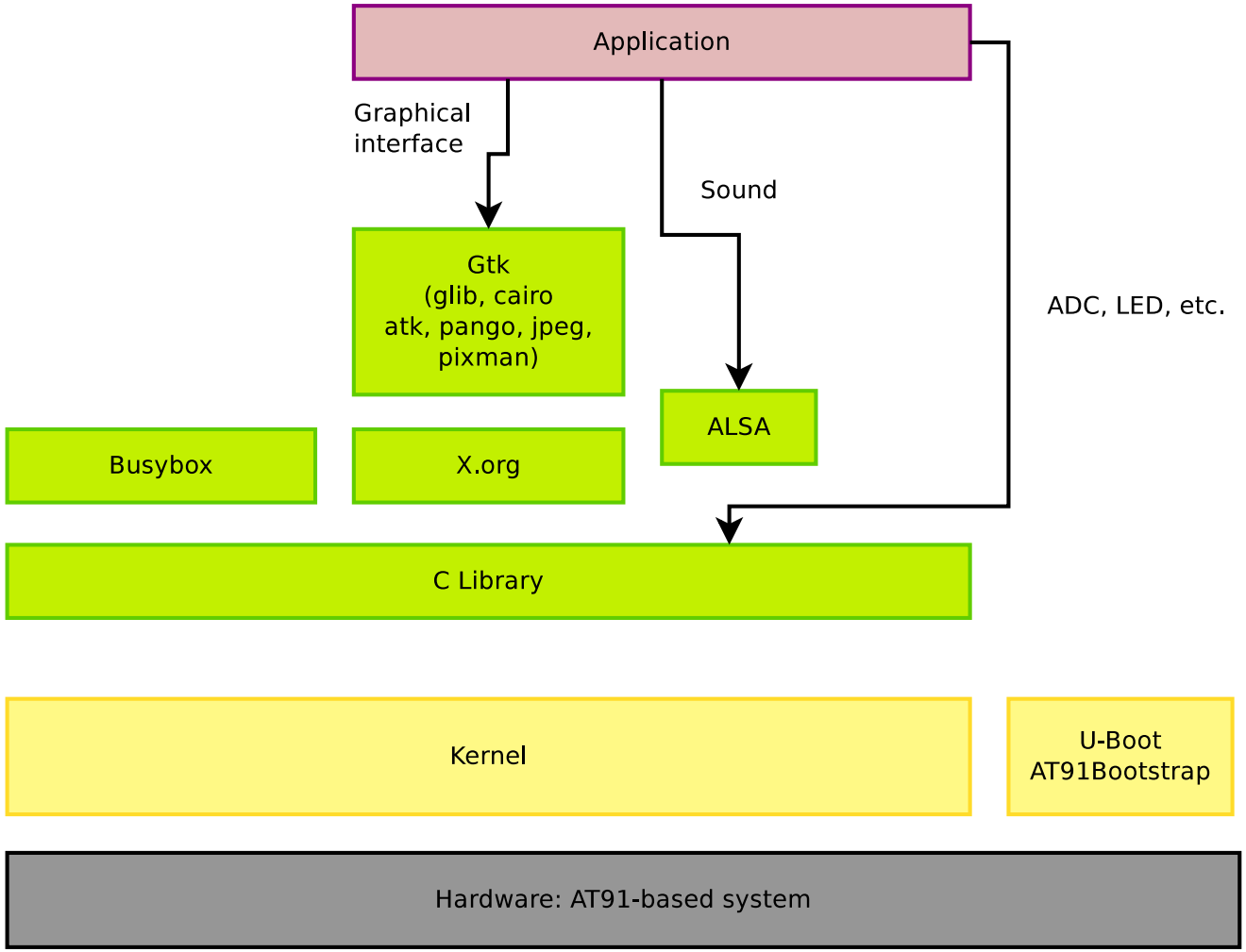
Real-time

Three solutions for real-time in Linux:

- Use the **standard kernel**. It has been improved over the years for real-time applications (kernel preemption, high-resolution timers, priority inheritance, support for the POSIX real-time API, etc.)
- Use the **PREEMPT_RT** patches. Those are patches against the Linux kernel that further improves its behaviour for real-time (more complete preemption, interrupt handlers in threads, etc.)
  `http://rt.wiki.kernel.org`
- Use one of the **co-kernel solutions** : **Xenomai** or **RTAI**. Real-time tasks are scheduled by a dedicated real-time core, and Linux runs as a low-priority task.
  `http://www.xenomai.org`
  `http://www.rtai.org`

Example: exercise bike

System that shows the performance and progression of the bike user, with a graphical interface and connection to the hardware.



Embedded Linux development process

Three major steps

- Board Support Package development
- System integration
- Application development

Board Support Package

- The BSP is the base of the system, that heavily depends on the hardware: *toolchain*, *bootloader* and *Linux kernel*
- Important questions
  - Are the bootloader and Linux kernel versions **sufficiently recent** ? With too old versions you miss features, and more importantly, you loose all community support.
  - Is the support available in the **mainline official versions** of the bootloader and the kernel ? This is the best solution, as it guarantees that you will benefit from updated versions.
  - If they are provided by the hardware vendor, how big is the delta with the official version ? When the delta is too large, it is hard to upgrade to newer versions $\rightarrow$ you will be blocked.
  - Are there binary drivers ? They will prevent you to upgrade the kernel.
- These components are critical. Don't rely on old versions with huge modifications from the hardware vendor.
- Make sure you keep separate: the official version from which the development was started, the hardware vendor

System integration

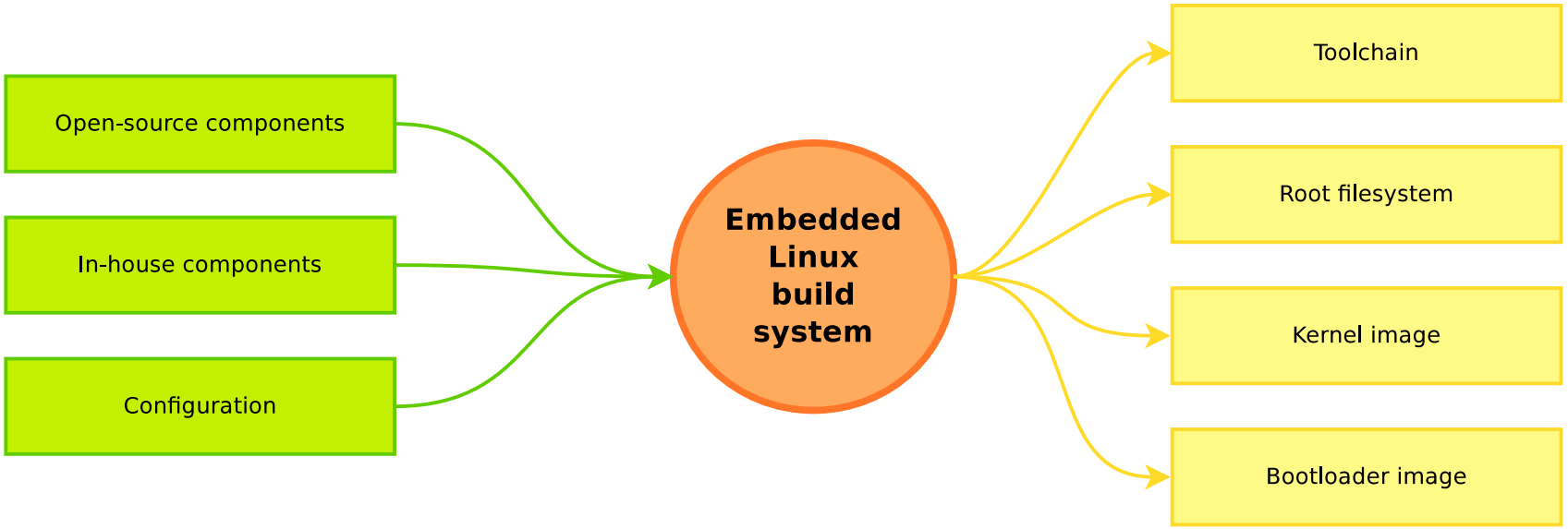
- Integrate all the open-source components needed for your system and your custom libraries and applications.
- This involves configuring and cross-compiling a lot of components, with sometimes complex dependencies.
- Don't do this by hand, and don't re-invent the wheel by writing your own build system.

System integration: solutions

- Some **binary distributions**, such as Debian, are available for embedded architectures (ARM, PowerPC, MIPS, etc.).
  - Advantages: everything is already compiled, easy to add/remove components, nice package management system.
  - Drawbacks: not a lot of control on the component configuration, code not necessarly optimized for your hardware platform, fairly big system that often needs to be stripped down, no mechanism to reproduce the build.
- **Embedded Linux build systems**, that build from source all the elements of a Linux system and generates the root filesystem image (and more)
  - Advantages: huge control over the system and components configuration, automated mechanism to reproduce the build, lightweight system.
  - Drawbacks: need to learn a new tool, long compilation times.

Do not use the demonstration root filesystem of the hardware vendor as a starting point: you have no way to reproduce the build and you don't control its components.

Embedded Linux build system : principle



Some common open-source build systems:

- Buildroot, `http://www.buildroot.org`
- OpenEmbedded, `http://www.openembedded.org`
- Yocto, `http://www.yoctoproject.org`
- and others: PTXdist, OpenBricks, OpenWRT, etc.
- Note: the hardware vendor specific build systems are usually of bad quality. Replace it with an independent, community-driven build system.

Embedded Linux build system: example of Buildroot

- A **configuration interface** similar to the kernel one allows to define all aspects of the system: CPU architecture, software components needed, filesystem type for the root filesystem image, kernel version and configuration, bootloader version and configuration, etc.
- Once the configuration is done, *Buildroot* **takes care of all the steps**: downloading, extracting, patching, configuring, compiling and installing all components, in the right order.
- More than **600 software components** already available
- **Simple to use**, regular stable releases, active community
- Very easy to add new software components, either open-source or in-house
- Drawbacks: full rebuilds often needed, no package management system on the target.

Choosing open-source components

There are several criterias to look at when choosing an open-source component:

- **Component quality**. Is there sufficient documentation ? Is the component widely used (presence in embedded Linux build systems is a good indicator)
- **Community vitality**. Is the component still being developed actively ? Is the community responsive to bug reports and questions ? When was the last stable release ? Is there regular activity in the revision control system ?
- **License**. Does the license of the component matches the requirement of your product ?
- **Technical requirements**. Does it offer the features you need ? Is it appropriate in terms of storage, CPU and memory consumption ?

→ Components already available in embedded Linux build systems are often a good starting point.

Application development

- Application development in embedded Linux systems is just the same as developping application in a normal Linux system: same development tools, same libraries.
- The standard language in Linux is C. All the system, and many libraries are written in this language.
- C++ is also widely used.
- Many interpreted languages are available: Lua, Python, Perl, PHP. Do not neglect their usefulness for non-performance sensitive parts, since they typicall allow faster development.

Application and system debugging

- **Cross-debugging** with `gdb` and `gdbserver`
  - `gdbserver` runs on the target, and controls the execution of the application to debug
  - `gdb` and potentially one of its graphical interfaces, runs on the development machine, and talks to `gdbserver` through Ethernet or serial port.
- System call tracing with `strace` and library call tracing with `ltrace`
- **System-wide tracing** with *LTTng*, `http://lttng.org`
- **System-wide profiling** with *OProfile*, `http://oprofile.sourceforge.net`

Licenses

- The free software licenses grants to everyone the set of four **fundamental freedoms**, but they also have some **requirements**.
- They fall into two main categories:
  - **Copyleft** licenses, that require modified versions to be distributed under the same license.
  - **Non-copyleft** licenses, that do not require modified versions to be distributed under the same license: they can be kept proprietary.
- The Free Software Foundation and the Open Source Initiative have a list of licenses together with their opinion on them:
  `http://www.gnu.org/licenses/license-list.html`
  `http://opensource.org/licenses/index.html`

GPL

- General Public License
- The **major copyleft license**, covers $\approx 50\%$ of the free software projects
- For example: Busybox, Linux Kernel, U-Boot, etc.
- Requires **derivate works to be released under the same license**, including applications relying on a library licensed under the GPL.
- The license requires you to ship the complete source code of the GPL components together with your product, including your modifications to these components, and attribution must be kept.
- No need to distribute the source code before the product is distributed.
- License already enforced multiple times in court.
- Even though the kernel is GPL, all userspace does not need to be under the GPL

GPL and kernel modules

- As the Linux kernel is under GPL, modifications to it must be released under the GPL
- Some kernel code can be written as *modules*, which can be dynamically loaded/unloaded at runtime.
- There is a **debate whether kernel modules are derivative works** of the kernel or not.
- No final answer on the question, opinions vary.
- Generally, the community opinion, including many kernel developers, is that proprietary kernel modules are bad.
- *"We, the undersigned Linux kernel developers, consider any closed-source Linux kernel module or driver to be harmful and undesirable."*,
- **Easier to make your kernel code GPL, and leave the added-value parts in userspace**

LGPL

- Lesser General Public License
- A **weaker copyleft license, used for many libraries**
- For example: Gtk, Qt, alsa-lib and most libraries
- Requires *derivative works* to be released under the same license, but non-free applications can be linked against a LGPL library, as long as the library can be replaced (dynamic linking is used in general)
- The license also requires you to ship the complete source code of the LGPL components, including your modifications.
- Beware that a few libraries are under the GPL: *readline* library, *MySQL* library, etc.

Non-copyleft licenses

Many non-copyleft licenses are widely used. They **do not require derivate works to be distributed under the same license**, but the original project must still be credited.

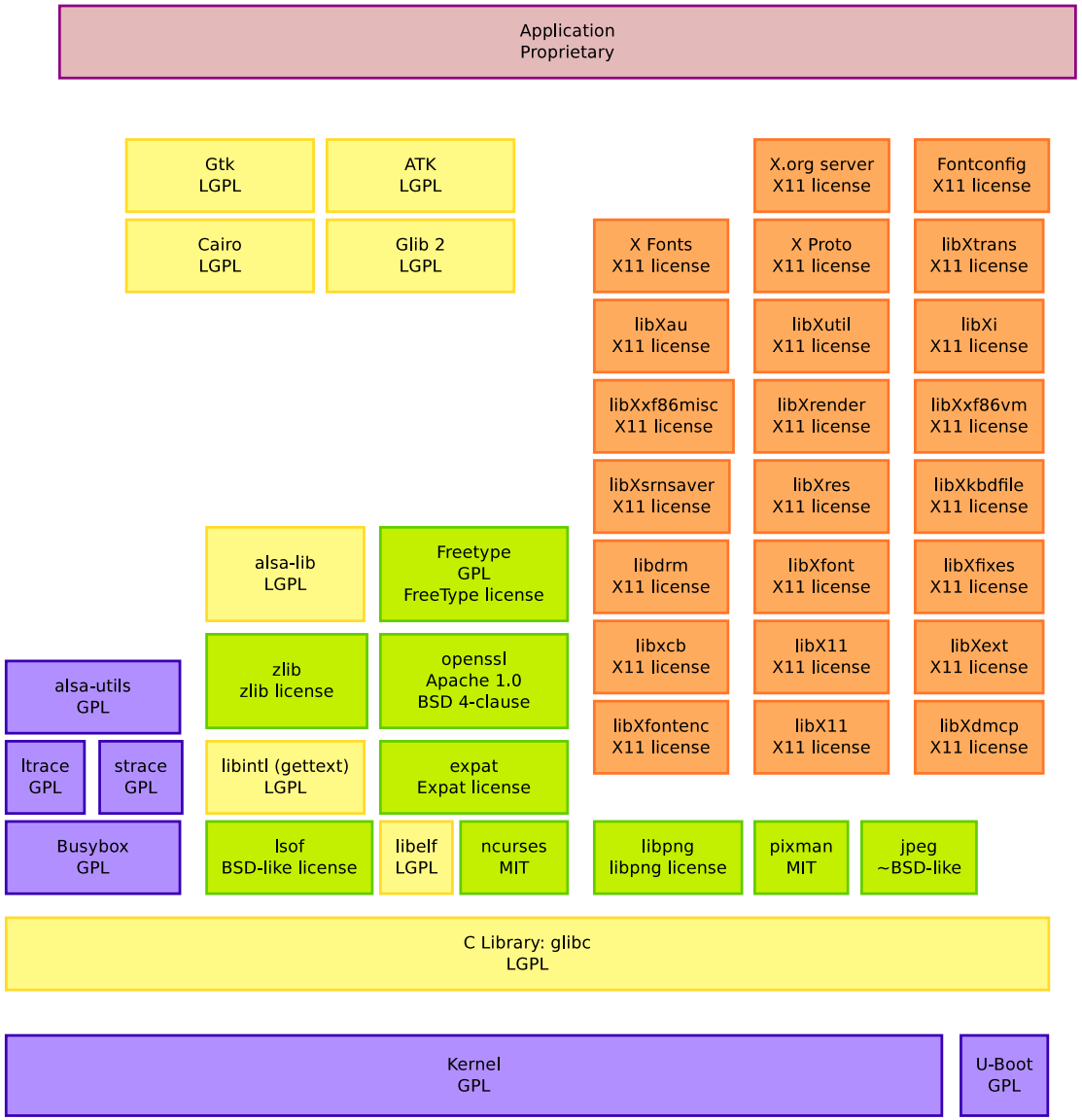
- BSD license
- Apache license
- MIT license
- Artistic license
- X11 license

You are not required to distribute the source code for these components and you can integrate code from these components into your proprietary applications.

Licensing good practices

- Keep **an accurate and complete list of all components** you use in your product, together with their respective license
- Make sure that the license of a component matches your requirements before basing all your product and development on top of it
- Keep your **changes separate from the original version** of the components. This allows for easier upgrades, but also to respect the licenses that want the changes from the original version to be clearly identified.
  Possible methods: stack of patches with *Quilt*, or branches in a revision control system.
- Do not copy/paste GPL/LGPL code into the parts of your system that must remain proprietary.
- **Comply with the licenses** as soon as your product starts being shipped.

Licensing analysis



Working on embedded Linux

- **Use Linux on the development station**
  - All community tools are developed by Linux developers, using Linux as their desktop operating system. Trying to use Windows or Mac OS to do embedded Linux development will lead to difficulties
  - Linux on the embedded system is the same as the Linux on the desktop: many good embedded Linux engineers are just long-time Linux users
- **Have a good e-mail client**
  - Needed to interact with the community
  - Your e-mail client must support *threading*, text-only e-mails (no HTML) and proper wrapping
  - Don't use Outlook or Lotus Notes, but instead Thunderbird, Evolution, KMail, Claws-Mail, etc.

Working on embedded Linux

- **Have a good and unfiltered network connectivity**
  - Need to have access to many ressources: Git and SVN repositories, IRC channels for discussion with the community, mailing-lists
  - Having a standard SMTP server is also useful to send patches
- **Control your system components, build procedure and use revision control systems**
  - Don't rely on prebuilt kernels or root filesystems, make sure you have all the source code and the documentation or scripts to rebuild all your system from scratch.
  - Use revision control systems to keep track of the changes you make to the different components you use in your system.
  - A significant part of working with embedded Linux is integration: it has to be done cleanly.

Support : full commercial solutions

- Vendors such as *Montavista*, *Windriver* or *Timesys*
- Provides integrated Board Support Package, system building tools, application development tools for embedded Linux, together with support.
- Advantages: single known representative to deal with, supposedly well-tested solutions and comprehensive support
- Drawbacks: dependency on vendor specific tools, vendor specific kernel and component versions, lock-in, high cost, support not necessarly that good

Support : community

- The developers of the different components and the open-source community as a whole generally provides good and timely support.
- Through mailing-lists, IRC.
- Need to understand how the community works, or better be part of it, to benefit from good support.
- Advantages: small cost, generally very quick and efficient feedback, allows your engineers to gain knowledge
- Drawbacks: support only for recent versions of the components, no clear representative, need to have some knowledge of how the community works, doesn't work for closed code

Support : commercial support to community solutions

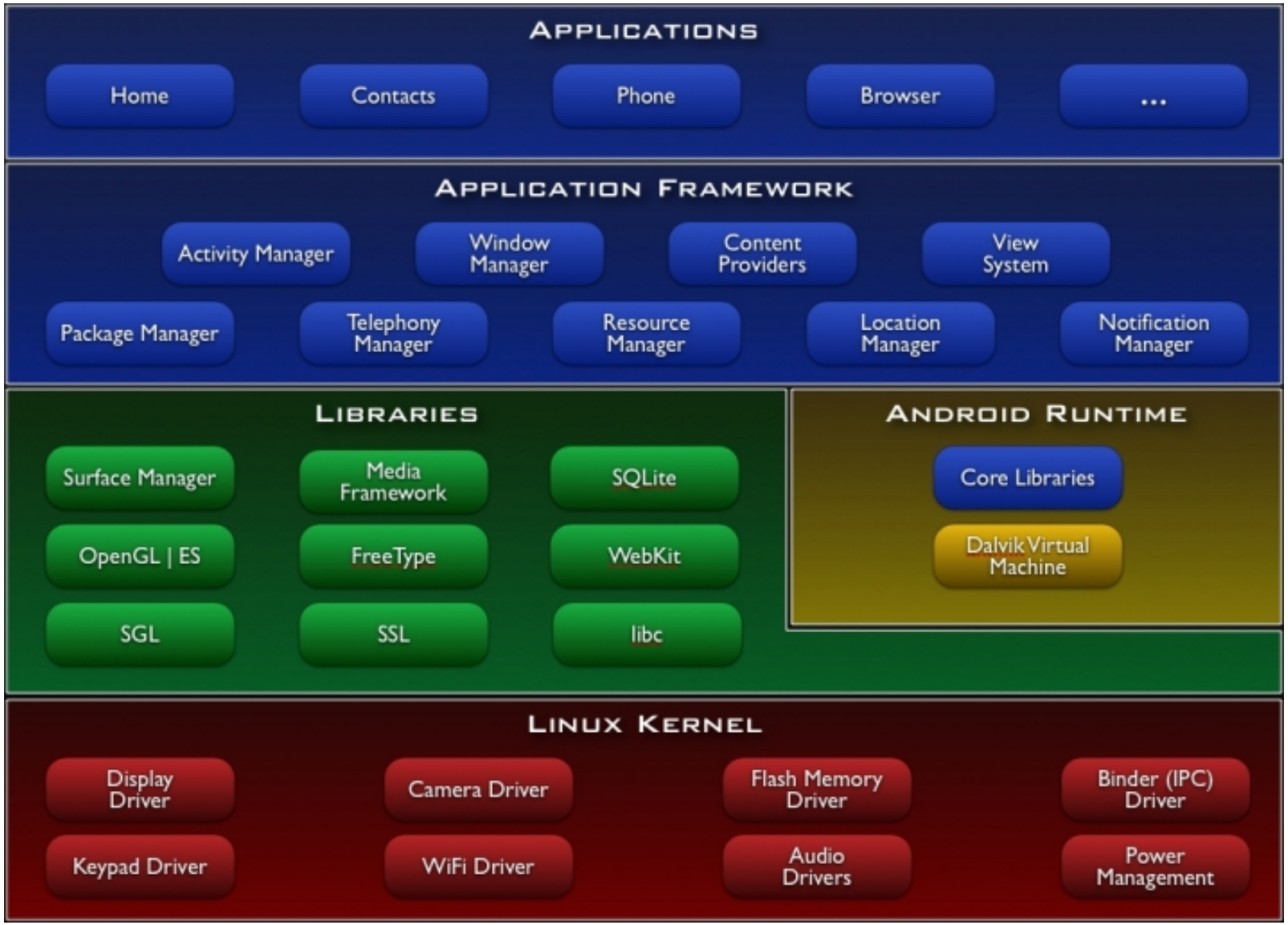
- Companies that do not have any specific product, and provide support for existing open-source components
- Companies such as *Free Electrons*, *DENX*, and hundreds of other small to medium sized companies.
- Advantages: single known representative, usage of well-known open source components so that you remain independent from the support provider, support that cares about your specific problem even if old components are used
- Drawbacks: ?

Android

Android

- *Android* is the famous Linux-based embedded operating system developed by Google, mainly targeted at phones, but usable for other applications as well
- Not a traditional Linux system: Google has only re-used the Linux kernel and a few userspace components, but the large majority of the system is Android-specific
  $\rightarrow$ need of specific skills and knowledge to work on Android
- The kernel needs fairly major modifications to be Android-compatible (power management, inter-process communication, etc.)
- Most components developed by Google are licensed under the Apache license, the kernel is the only component under the GPL.

Android architecture



Conclusion

- Linux and the open source world offers a wide range of components and tools for embedded system development
- Those components have many advantages: focus on added value, low cost, complete control, etc.
- Support is available, both from the community or commercial companies
- **So what about Linux in your next embedded product ?**

Questions ?

Thomas Petazzoni
thomas.petazzoni@free-electrons.com