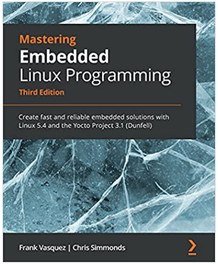
Fundamentals of Embedded Linux

Chris Simmonds

NDC Techtown 2022

License

About Chris Simmonds



- Consultant and trainer
- Author of *Mastering Embedded Linux Programming*
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops

"Looking after the Inner Penguin" blog at `https://2net.co.uk/`

Mastodon: @csimmonds@fosstodon.org
`https://fosstodon.org/@csimmonds`

`https://uk.linkedin.com/in/chrisdsimmonds/`

- What is embedded computing?
- Embedded Linux
- Embedded build systems
- Real-time
- Open source licenses
- Conclusion

What is embedded computing?

- No formal definition
- Basically, "code running on a computer inside a device that you do not think of as being a computer"
- Characteristics include
  - single purpose
  - not end-user programmable
  - designed for price - so minimum hardware necessary
  - has power constraints - e.g. battery power
  - has power dissipation constraints - e.g. no cooling fan

What kind of computer?

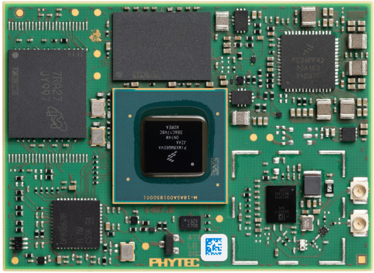
- Microcontroller (MCU)
  - small, low power, low performance, $0.20 to $10
  - CPU, RAM, flash storage and peripherals all on one chip
  - microwave oven, washing machine, remote sensor, ...
- Microprocessor (MPU)
  - CPU, RAM, storage and peripherals on separate chips
  - high power, high performance, high cost
  - mostly x86 architecture
- System on Chip (SoC)
  - MPU with on-chip peripherals
  - mostly ARM architecture

SBC, SoM and custom hardware

- Categories of Embedded Linux hardware
  - SBC - Single Board Computer: ready-to-go, e.g. Raspberry Pi
  - SoM - System-on-Module: SoC, plus supporting cicuitary integrated on to a module which plugs into a custom designed base board.
  - Custom hardware: board designed for a specific purpose

A typical SoM

- NXP i.MX 8M SoC (4 core ARM Cortex-A53 1.5 GHz)
- 1 to 8 GiB LPDDR4 RAM
- 8 to 64 GiB eMMC flash storage
- Vivante GPU
- Video Processor, Display controller, M4 MCU (built in)
- Ethernet, WiFi and Bluetooth
- HDMI and DSI display; CSI camera
- USB, PCIe



Phytec phyCORE-i.MX 8M
55 mm x 40 mm

- What is embedded computing?
- Embedded Linux
- Embedded build systems
- Real-time
- Open source licenses
- Conclusion

What kind of operating system?

- MCU
  - really small MCUs run code bare metal
  - mid and high end (32-bit) use a Real Time Operating System (RTOS) such as Zephyr, NuttX, FreeRTOS, ...
- Embedded MPU and SoC
  - Predominantly Embedded Linux

Why not Linux for MCU? Very few MCUs have virtual memory. Even if they do, they generally don't have enough RAM and storage to run Linux

Devices running Embedded Linux

You use Embedded Linux every day

Why Embedded Linux?

- **Moore's law**: complex hardware requires complex software
- **Free**: you have the **freedom** to get and modify the code, making it easy to adapt and extend
- **Functional**: supports a (very) wide range of hardware
- **Up to date**: the kernel has a 10 week release cycle
- **Free**: there is no charge for using the source code

Minimum hardware spec

- 32 or 64-bit processor architecture with memory management unit (MMU)
  - examples: ARM, x86, RISC-V
- At least 16 MiB RAM (*)
- At least 4 MiB storage (*), usually flash memory

(*) It is possible to build Linux systems with less RAM and flash, but it requires non-trivial effort

Open source ecosystem

The main players are:

- Open source community
  - A loose alliance of developers, working on 1000's of individual projects, some funded by companies with a commercial interest
- SoC vendors
  - Customise upstream code (e.g. Linux kernel, toolchain) to ensure it works well on their platforms
- SBC and SoM vendors: further customisation
- Commercial embedded Linux vendors: offer support and services

Pain points

- Lack of support for your particular hardware (always check with the manufacturer before you design a component in)
- The rapid update cycle does not fit well with the slower cycle for embedded projects
- SoC/SoM/SBC vendors do not always push fixes and features as quickly as we would like

Distro or build from source?

- Distro, e.g. Debian or Ubuntu
  - binary packages
  - package manager, e.g. apt or dnf
  - native compile
- Build from source
  - bespoke operating system
  - optimised for the hardware and task
  - cross compile

Distro: pluses

- vast number of packages, ready to install
- no compilation time
- little or no setup time - switch on and go

Distro: minuses

- Requires hardware support
  - only works out-of-the-box on commodity hardware such as PC or Raspberry Pi
- Native development means compiling on a compatible machine
  - OK for PC, but not scalable for Raspberry Pi
- Too big
  - e.g. Ubuntu Core is 500 MB, but we only have 256MB flash memory
- Software update via package manager is not robust
  - we need atomic update, e.g. the entire root filesystem image

Distro: real world

- Common on embedded PC
- Common on embedded Raspberry Pi, e.g. Raspberry Pi Compute Modules
- Mostly low volume systems where human intervention can correct problems

Boards need Board Support Packages

- The Board Support Package (BSP) is everything you need to run Linux on a particular board
- A BSP consists of
  - Bootloader
  - Linux kernel
  - Kernel drivers specific to the board
  - Device tree (ARM)
  - Libraries to support vendor-specific components such as accelerated graphics
  - Boot scripts and run-time configuration files
  - Firmware binaries for on-chip peripherals(*)

(*) some of the on-chip peripherals are actually MCUs and require firmware to be loaded at boot-time, e.g. WiFi and Bluetooth interfaces. Usually not open source

BSP

ARM, RISC-V

- U-Boot source and configuration
- Linux source and configuration
- Device tree
- Recipes for Yocto or Buildroot

x86

- BIOS (part of motherboard)
- In some cases, kernel drivers for non-generic or proprietary hardware(*)

(*) Mainline Linux is enough to boot and use most x86 hardware

Elements of embedded Linux

Every embedded Linux project has these four elements:

- Toolchain: to compile all the other elements
- Bootloader: to initialise the board and load the kernel
- Kernel: to manage system resources
- Root filesystem: to run applications

Toolchain

- toolchain = C/C++ compiler + linker + C library + debugger
  - Compiler, linker and debugger: either GCC or Clang
  - C library: either glibc or musl libc

Types of toolchain

- Native toolchain
  - Install and develop on the target
- Cross toolchain
  - Build on development system, deploy on target
  - Keeps target and development environments separate

Cross toolchains are the most common for embedded development

Bootloader

- Open source bootloders include:
  - Das U-Boot
  - Barebox
  - Little Kernel
  - GRUB 2 (for X86 and X86_64)
- The role of the bootloader is to:
  - Initialise the board
  - Load a Linux kernel, kernel command line, device tree and initial ramfs
  - System maintenance, e.g. flash system images, run diagnostics

Kernel

- Non x86 boards seldom use mainline Linux
- Most cases, Kernel comes from the SoC vendor (not an ideal situation)
- Vendor kernel has
  - initialisation code for the chip
  - adaptation for chip features (e.g. Qualcomm energy-aware scheduling (*))
  - drivers for on chip peripherals ("IP blocks") - some will be proprietary, shipped as binary kernel modules
- Vendors give code updates less often than mainline - maybe only one per yrear
- Vendors are not very good at pushing feature upstream

(*)A few years ago, the Qualcomm vendor kernel had 25,000 patches that were not in mainline

Device tree

- The kernel needs to know details about hardware
  - to decide which drivers to initialise
  - to configure device parameters such as register addresses and IRQ
- Sources of information:
  - firmware ACPI tables (x86 and ARM server)
  - bus enumeration, e.g. PCI
  - hard coded structures
  - **device tree** (ARM, RICK-V, PPC, MIPS, and others)

Root filesystem

- The user space part of the operating system
- The rootfs contains code to boot and start essential services
  - init daemon
  - other daemons started by init (network services, authentication services, monitoring and logging services, etc)
- System libraries
- Configuration files
- ... and anything else essential to the system ...

- What is embedded computing?
- Embedded Linux
- Embedded build systems
- Real-time
- Open source licenses
- Conclusion

Embedded build systems

- Building the four elements by hand is time consuming
- Embedded build systems make it easy

Tool	Notes
buildroot	Small, menu-driven
OpenWrt	A variant of Buildroot for network devices
OpenEmbedded	General purpose
Yocto Project	General purpose, wide industry support, complex

Buildroot

- One of the first embedded build systems (2001)
  - (OpenEmbedded started two years later)
- Web: `https://buildroot.org`
- As well as the **root filesystem**, can also build **toolchain**, **bootloader**, and **kernel**
- Architectures: ARM, RISC-V, x86, PowerPC, and many more...
- Packages: over 2500
- Board configs: over 250

OpenEmbedded

- `www.openembedded.org`
- Based on recipes grouped together into **meta layers**
- The recipes are processed by a task scheduler named **BitBake**
- Recipes generate packages as RPM (default)
- In other words, OpenEmbedded is a tool to create a custom Linux distribution

OpenEmbedded Core

- The core of OpenEmbedded, **oe core**, is the basis of several build systems
  - OpenEmbedded itself
  - Poky (part of the Yocto Project)
  - ELDK (from Denx)
  - Mentor Graphics Linux
  - ... and others

The Yocto Project

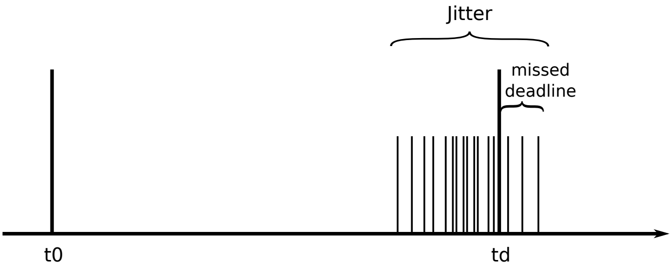
- The Yocto Project is a Linux Foundation project to maintain a build system for embedded Linux
- Consists of
  - oe-core, shared with OpenEmbedded
  - BitBake: shared with OpenEmbedded
  - Poky, the distribution metadata
  - Reference BSPs including BeagleBone
  - Documentation, which is extensive
  - Toaster: a graphical user interface for Yocto

- What is embedded computing?
- Embedded Linux
- Embedded build systems
- Real-time
- Open source licenses
- Conclusion

Real-time

- Embedded computing is often associated with real-time processing
- Real-time = computation that must be completed before a deadline
- Examples:
  - controlling the motion of a robot
  - displaying a video stream
- Otherwise, the task is non-real-time
- Example:
  - compiling a program: the result is just as good if it takes one second or one minute

Real-time metrics



- To make a program real-time you need to reduce **jitter** by increasing **determinism**
- There are two things to consider:
  - how long before the deadline? Shorter deadlines are harder to hit
  - how much do you care about missing the deadline? The more you care, the harder it is

Soft or hard?

- Soft real-time
  - Missing deadline is OK some of the time
  - example: video processing: nobody will notice one or two dropped frames
- Hard real-time
  - Missing deadline is never acceptable: in extreme cases may cause injury or death
  - example: robot welding system

Real-time Linux

Linus Torvalds, Kernel Summit 2006:

"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT_RT"

Real-time in user space

Linux scheduling policies

- SCHED_NORMAL
  - "Completely Fair Scheduler": tries to give each thread a fair share of CPU time
- SCHED_FIFO
  - Threads have static priorities between 1 and 99
- Scheduler runs SCHED_FIFO threads in priority order (99 is highest) first
- ... then runs SCHED_NORMAL threads

Note: SCHED_FIFO requires CAP_SYS_NICE, which requires root privileges by default

Real-time in kernel space

- Linux is a source of non determinism, caused by scheduling latencies, interrupt handling, kernel locks
- Enabling kernel preemption is a big help
- `CONFIG_PREEMPT`
  - Reduces jitter to milliseconds
  - enabled on most embedded kernels
- `PREEMPT_RT`
  - Reduces jitter to 100s microseconds or less
  - Only just been integrated into mainline Linux (after more than 10 years of effort)

Note: increasing determinism by enabling preemption reduces throughput

Lesson: real time systems are not "fast" systems

Software update

- Updates need to be atomic
- But, package managers (apt, dnf) are not atomic
    - loose power at the wrong time leads to inconsistent set of packages and so a bricked device
- Instead, embedded devices use **Image update**, which can be made atomic

Software update

Commonly used update agents

- swupdate
- RAUC - Robust Auto-Update Controller
- Mendor.io (open source with commercial support)

- What is embedded computing?
- Embedded Linux
- Embedded build systems
- Real-time
- Open source licenses
- Conclusion

Working with open source licenses

- Open source licenses grant the *freedom* to modify and redistribute the source code
- Open source licenses can be divided into two groups
  - "permissive", such as BSD, MIT and Apache
  - "copyleft" - GPL (General Public License)
- The license should be part of each package of code
- typically in a file named `LICENSE` or `COPYING`
- also as a comment at the beginning of each source file

Permissive licenses

- In general, these licenses state that you can create derivative works so long as you
  - Don’t change copyright notices
  - Don’t change the limited warranty notice
- You don’t need to distribute source code

I am not a lawyer. Please consult your legal department for clarification

GPL v2

- Version 2 of the General Public License says
  - you can create derivative works
  - you must distribute source code to end users
    - by public server
    - or by "written offer": a promise to supply code on request
  - you are creating a derivative work if you link with code or a library licensed under GPL

Note: I am not a lawyer. Please consult your legal department for clarification

LGPL v2

- The lesser GPL (LGPL) is mostly applied to library code
- Allows linking to a library without creating a derivative work
  - i.e. you can write proprietary programs that link dynamically with LGPL libraries
  - static linking is a more complex legal issue: don't do it

Note: I am not a lawyer. Please consult your legal department for clarification

GPL v3 and LGPL v3

- Adds "The right to tinker"
  - it must be possible to replace the GPL v3 components of any device
  - also known as the "anti Tivoization clause"
- and protection against patent threats
  - You must provide every recipient with any patent licenses necessary to exercise the rights that the GPLv3 gives them
- and many other details...

Note: I am not a lawyer. Please consult your legal department for clarification

Secure boot and (L)GPL v3

- GPL3 with secure boot is possible so long as you have a "dev board" mode in which an unsigned kernel can boot and run all GPL v3 runtime components, but not (necessarily) any proprietary components

Here is a talk by Bradly Khun and Behan Webster on this topic:

`https://events19.linuxfoundation.org/wp-content/uploads/2017/11/Safely-Copylefted-Cars-Reexamining-GPLv3-Installation-Information-Requirements-ALS-Bradley-Kuhn-Beh` `pdf` Note: I am not a lawyer. Please consult your legal department for clarification

- What is embedded computing?
- Embedded Linux
- Embedded build systems
- Real-time
- Open source licenses
- Conclusion

Questions?

Slides at
https://2net.co.uk/slides/fundamentals-of-embedded-linux-csimmonds-ndctechtown-2022.pdf

@2net_software

https://uk.linkedin.com/in/chrisdsimmonds/

