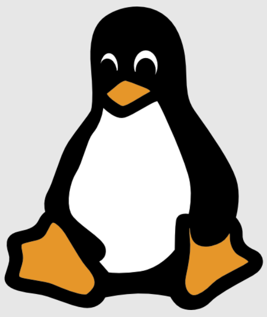


Embedded Linux kernel and driver development training

Online seminar

Course duration

7 half days – 28 hours

Language

Materials	English
Oral Lecture	English
	French

Trainer

One of the following engineers

- Grégory Clement
- Maxime Chevallier
- Miquèl Raynal
- Théo Lebrun

Contact

training@bootlin.com

+33 484 258 097

bootlin

bootlin.com

Audience

People developing devices using the Linux kernel
People supporting embedded Linux system developers.

Training objectives

- Be able to configure, build and install the Linux kernel on an embedded system.
- Be able to understand the overall architecture of the Linux kernel, and how user-space applications interact with the Linux kernel.
- Be able to develop simple but complete Linux kernel device drivers, thanks to the development from scratch of two drivers for two different hardware devices, that illustrate all the major concepts of the course.
- Be able to navigate through the device drivers mechanisms of the Linux kernel: Device Tree, device model, bus infrastructures.
- Be able to develop device drivers that communicate with hardware devices.
- Be able to develop drivers that expose functionality of hardware devices to Linux user-space applications: character devices, kernel subsystems.
- Be able to use the major kernel mechanisms needed for device driver development: memory management, locking, interrupt handling, sleeping, DMA.
- Be able to debug Linux kernel issues, using a variety of debugging techniques and mechanisms.

Prerequisites

- **Solid experience with the C programming language**: participants must be familiar with the usage of complex data types and structures, pointers, function pointers, and the C pre-processor.
- **Knowledge and practice of UNIX or GNU/Linux commands**: participants must be familiar with the Linux command line. Participants lacking experience on this topic should get trained by themselves, for example with our freely available on-line slides.
- **Minimal experience in embedded Linux development**: participants should have a minimal understanding of the architecture of embedded Linux systems: role of the Linux kernel vs. user-space, development of Linux user-space applications in C. Following Bootlin's Embedded Linux course allows to fulfill this pre-requisite.
- **Minimal English language level: B1**, according to the *Common European Framework of References for Languages*, for our sessions in English. See the CEFR grid for self-evaluation.

Pedagogics

- Lectures delivered by the trainer, over video-conference. Participants can ask questions at any time.
- Practical demonstrations done by the trainer, based on practical labs, over video-conference. Participants can ask questions at any time. Optionally, participants who have access to the hardware accessories can reproduce the practical labs by themselves.
- Instant messaging for questions between sessions (replies under 24h, outside of week-ends and bank holidays).
- Electronic copies of presentations, lab instructions and data files. They are freely available here.

Certificate

Only the participants who have attended all training sessions, and who have scored over 50% of correct answers at the final evaluation will receive a training certificate from Bootlin.

Disabilities

Participants with disabilities who have special needs are invited to contact us at *training@bootlin.com* to discuss adaptations to the training course.

Required equipement

Mandatory equipment:

- Computer with the operating system of your choice, with the Google Chrome or Chromium browser for videoconferencing.
- Webcam and microphone (preferably from an audio headset).
- High speed access to the Internet.

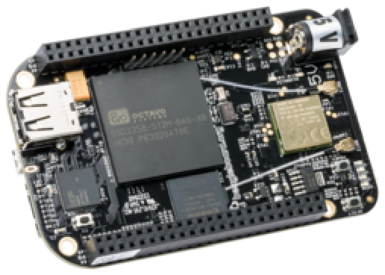
Optionnally, if the participants want to be able to reproduce the practical labs by themselves, they must separately purchase the hardware platform and accessories, and must have a PC computer with a native installation of Ubuntu Linux 24.04.

Hardware platform for practical labs

BeagleBone Black

BeagleBone Black or **BeagleBone Black Wireless** board

- An ARM AM335x (single Cortex-A8) processor from Texas Instruments
- USB powered
- 512 MB of RAM
- 2 or 4 GB of on-board eMMC storage
- USB host and device
- HDMI output
- 2 x 46 pins headers, to access UARTs, SPI buses, I2C buses and more.
- Ethernet or WiFi



BeaglePlay

BeaglePlay board

- Texas Instruments AM625x (4xARM Cortex-A53 CPU)
- SoC with 3D acceleration, integrated MCU and many other peripherals.
- 2 GB of RAM
- 16 GB of on-board eMMC storage
- USB host and USB device, microSD, HDMI
- 2.4 and 5 GHz WiFi, Bluetooth and also Ethernet
- 1 MicroBus Header (SPI, I2C, UART, ...), OLDI and CSI connector.

Training Schedule

Half day 1

Lecture	Introduction to the Linux kernel	▪ Roles of the Linux kernel ▪ Kernel user interface (/proc and /sys) ▪ Overall architecture ▪ Versions of the Linux kernel ▪ Kernel source tree organization
Demo	Downloading the Linux kernel source code	▪ Download the Linux kernel code from Git
Lecture	Linux kernel source code	▪ Specifics of Linux kernel development ▪ Coding standards ▪ Stability of interfaces ▪ Legal aspects, licensing ▪ Organization of the kernel community ▪ The release schedule and process: release candidates, stable releases, long-term support, etc.
Demo	Kernel sources	▪ Making searches in the Linux kernel sources: looking for C definitions, for definitions of kernel configuration parameters, and for other kinds of information. ▪ Using the UNIX command line and then kernel source code browsers.
Lecture	Configuring, compiling and booting the Linux kernel	▪ Kernel configuration. ▪ Native and cross compilation. Generated files. ▪ Booting the kernel. Kernel booting parameters. ▪ Mounting a root filesystem on NFS.
Demo	Kernel configuration, cross-compiling and booting on NFS	▪ Configuring, cross-compiling and booting a Linux kernel with NFS boot support.

Half day 2

Lecture	Linux kernel modules	▪ Linux device drivers ▪ A simple module ▪ Programming constraints ▪ Loading, unloading modules ▪ Module dependencies ▪ Adding sources to the kernel tree
Demo	Writing modules	▪ Write a kernel module with several capabilities. ▪ Access kernel internals from your module. ▪ Set up the environment to compile it
Lecture	Describing hardware devices	▪ Discoverable hardware: USB, PCI ▪ Non-discoverable hardware ▪ Extensive details on Device Tree: overall syntax, properties, design principles, examples ▪ YAML bindings and meta hardware description to verify Device Tree content
Demo	Describing hardware devices	▪ Create your own Device Tree file ▪ Configure LEDs connected to GPIOs ▪ Describe an I2C-connected device in the Device Tree

Half day 3

Lecture	Pin muxing	▪ Understand the *pinctrl* framework of the kernel ▪ Understand how to configure the muxing of pins
Demo	Pin muxing	▪ Configure the pinmuxing for the I2C bus used to communicate with the Nunchuk ▪ Validate that the I2C communication works using user space tools
Lecture	Linux device model	▪ Understand how the kernel is designed to support device drivers ▪ The device model ▪ Binding devices and drivers ▪ Platform devices, Device Tree ▪ Interface in user space: `/sys`
Lecture	Introduction to the I2C API	▪ The I2C subsystem of the kernel ▪ Details about the API provided to kernel drivers to interact with I2C devices
Demo	Communicate with the Nunchuk over I2C	▪ Explore the content of `/dev` and `/sys` and the devices available on the embedded hardware platform. ▪ Implement a driver that registers as an I2C driver. ▪ Communicate with the Nunchuk and extract data from it.
Half day 4		
Lecture	Kernel frameworks	▪ Block vs. character devices ▪ Interaction of user space applications with the kernel ▪ Details on character devices, `file_operations`, `ioctl()`, etc. ▪ Exchanging data to/from user space ▪ The principle of kernel frameworks
Lecture	The input subsystem	▪ Principle of the kernel *input* subsystem ▪ API offered to kernel drivers to expose input devices capabilities to user space applications ▪ User space API offered by the *input* subsystem
Demo	Expose the Nunchuk functionality to user space	▪ Extend the Nunchuk driver to expose the Nunchuk features to user space applications, as a *input* device. ▪ Test the operation of the Nunchuk using `evtest`
Lecture	Memory management	▪ Linux: memory management - Physical and virtual (kernel and user) address spaces. ▪ Linux memory management implementation. ▪ Allocating with `kmalloc()`. ▪ Allocating by pages. ▪ Allocating with `vmalloc()`.
Half day 5		
Lecture	I/O memory	▪ I/O memory range registration. ▪ I/O memory access. ▪ Memory ordering and barriers
Demo	Minimal platform driver and access to I/O memory	▪ Implement a minimal platform driver ▪ Modify the Device Tree to instantiate the new serial port device. ▪ Reserve the I/O memory addresses used by the serial port. ▪ Read device registers and write data to them, to send characters on the serial port.

Lecture	The misc kernel subsystem	▪ What the *misc* kernel subsystem is useful for ▪ API of the *misc* kernel subsystem, both the kernel side and user space side
Demo	Output-only serial port driver	▪ Extend the driver started in the previous lab by registering it into the *misc* subsystem ▪ Implement serial port output functionality through the *misc* subsystem ▪ Test serial output from user space

Half day 6

Lecture	Processes, scheduling, sleeping and interrupts	▪ Process management in the Linux kernel. ▪ The Linux kernel scheduler and how processes sleep. ▪ Interrupt handling in device drivers: interrupt handler registration and programming, scheduling deferred work.
Demo	Sleeping and handling interrupts in a device driver	▪ Adding read capability to the character driver developed earlier. ▪ Register an interrupt handler. ▪ Waiting for data to be available in the `read()` file operation. ▪ Waking up the code when data is available from the device.
Lecture	Locking	▪ Issues with concurrent access to shared resources ▪ Locking primitives: mutexes, semaphores, spinlocks. ▪ Atomic operations. ▪ Typical locking issues. ▪ Using the lock validator to identify the sources of locking problems.
Demo	Locking	▪ Add locking to the current driver

Half day 7

Lecture	DMA: Direct Memory Access	▪ Peripheral DMA vs. DMA controllers ▪ DMA constraints: caching, addressing ▪ Kernel APIs for DMA: `dma-mapping`, `dmaengine`, `dma-buf`
Demo	DMA: Direct Memory Access	▪ Setup streaming mappings with the `dma` API ▪ Configure a DMA controller with the `dmaengine` API ▪ Configure the hardware to trigger DMA transfers ▪ Wait for DMA completion
Lecture	Driver debugging techniques	▪ Debugging with printing functions ▪ Using Debugfs ▪ Analyzing a kernel oops ▪ Using kgdb, a kernel debugger ▪ Using the Magic SysRq commands
Demo	Investigating kernel faults	▪ Studying a broken driver. ▪ Analyzing a kernel fault message and locating the problem in the source code.
Lecture	Power management	▪ Overview of the power management features of the kernel ▪ Topics covered: clocks, suspend and resume, dynamic frequency scaling, saving power during idle, runtime power management, regulators, etc.
Lecture	If time left	▪ mmap

Labs

The practical labs of this training session use the following hardware peripherals to illustrate the development of Linux device drivers:

- A Wii Nunchuk, which is connected over the I2C bus to the BeagleBone Black board. Its driver will use the Linux *input* subsystem.
- An additional UART, which is memory-mapped, and will use the Linux *misc* subsystem.

While our explanations will be focused on specifically the Linux subsystems needed to implement these drivers, they will always be generic enough to convey the general design philosophy of the Linux kernel. The information learnt will therefore apply beyond just I2C, input or memory-mapped devices.