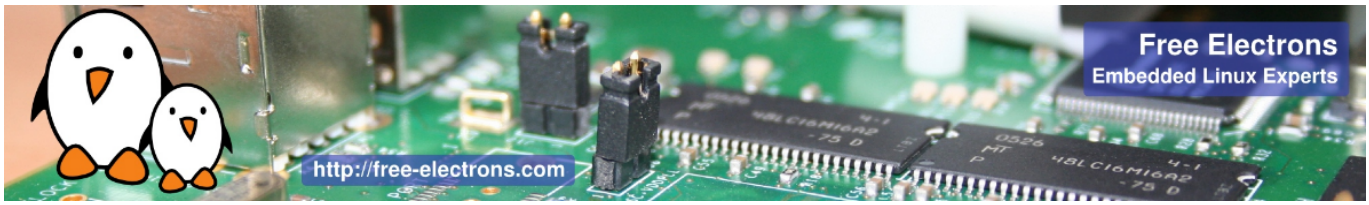
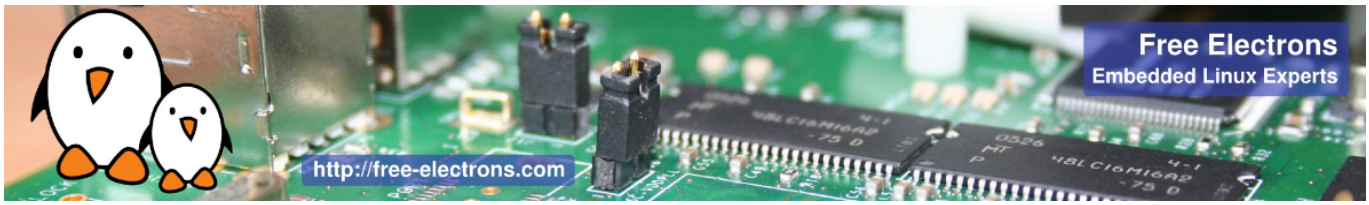



# Embedded Linux kernel and driver development training

5-day session

| | |
|---|---|
| **Title** | Embedded Linux kernel and driver development training |
| **Overview** | Understanding the Linux kernel<br>Developing Linux device drivers<br>Linux kernel debugging<br>Porting the Linux kernel to a new board<br>Working with the kernel development community<br>Practical labs with the ARM-based Beagle Bone Black board.<br><br>See our training materials on http://free-electrons.com/doc/training/linux-kernel. This way, you can check by yourself whether the course contents correspond to your needs. |
| **Duration** | **Five** days - 40 hours (8 hours per day).<br>50% of lectures, 50% of practical labs. |
| **Trainer** | One of the engineers listed on<br>http://free-electrons.com/training/trainers/ |
| **Language** | Oral lectures: English, French or German.<br>Materials: English. |
| **Audience** | People developing devices using the Linux kernel<br>People supporting embedded Linux system developers. |
| **Prerequisites** | **Solid experience in C programming**<br>In particular, participants must be familiar with creating and dealing with complex data types and structures, with pointers to such symbols, as well as with function pointers.<br><br>**Knowledge and practice of UNIX or GNU/Linux commands**<br>People lacking experience on this topic should get trained by themselves with our freely available on-line slides (http://free-electrons.com/docs/command-line/).<br><br>**Experience in embedded Linux development**.<br>Taking our Embedded Linux course (http://free-electrons.com/training/embedded-linux/) first is not an absolute prerequisite, but it will definitely help people lacking embedded Linux development experience. They will understand the development environment and board manipulations better, allowing them to concentrate on kernel code programming. |

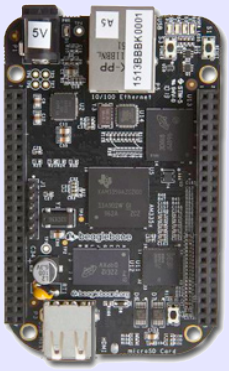



| | |
|---|---|
| **Required equipment** | **For on-site sessions only**<br>Everything is supplied by Free Electrons in public sessions.<br>• Video projector<br>• PC computers with at least 2 GB of RAM, and Ubuntu Linux installed in a **free partition of at least 20 GB. Using Linux in a virtual machine is not supported**, because of issues connecting to real hardware.<br>• We need Ubuntu Desktop 12.04 (32 or 64 bit, Xubuntu and Kubuntu variants are fine). We don't support other distributions, because we can't test all possible package versions.<br>• **Connection to the Internet** (direct or through the company proxy).<br>• **PC computers with valuable data must be backed up** before being used in our sessions. Some people have already made mistakes during our sessions and damaged work data. |
| **Materials** | Print and electronic copies of presentations and labs.<br>Electronic copy of lab files. |

## Hardware

The hardware platform used for the practical labs of this training session is the **BeagleBone Black** board, which features:

- An ARM AM335x processor from Texas Instruments (Cortex-A8 based), 3D acceleration, etc.
- 512 MB of RAM
- 2 GB of on-board eMMC storage (4 GB in Rev C)
- USB host and device
- HDMI output
- 2 x 46 pins headers, to access UARTs, SPI buses, I2C buses and more.

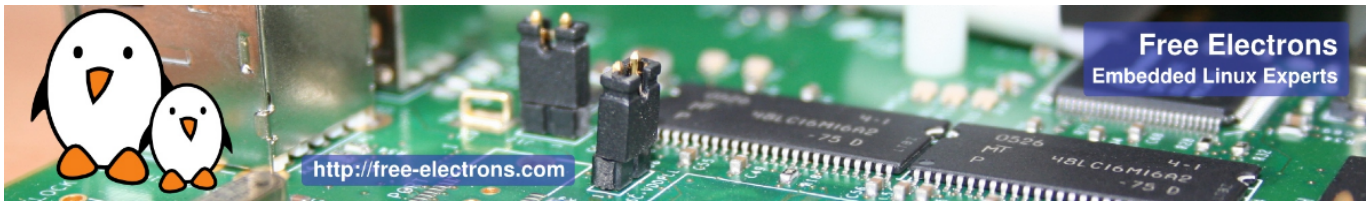


## Labs

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 those 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.




# Day 1 - Morning

## Lecture - Introduction to the Linux kernel

- Kernel features
- Understanding the development process.
- Legal constraints with device drivers.
- Kernel user interface (/proc and /sys)
- User space device drivers

## Lecture - Kernel sources

- Specifics of Linux kernel development
- Coding standards
- Retrieving Linux kernel sources
- Tour of the Linux kernel sources
- Kernel source code browsers: cscope, Kscope, Linux Cross Reference (LXR)

## Lab - 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.

# Day 1 - Afternoon

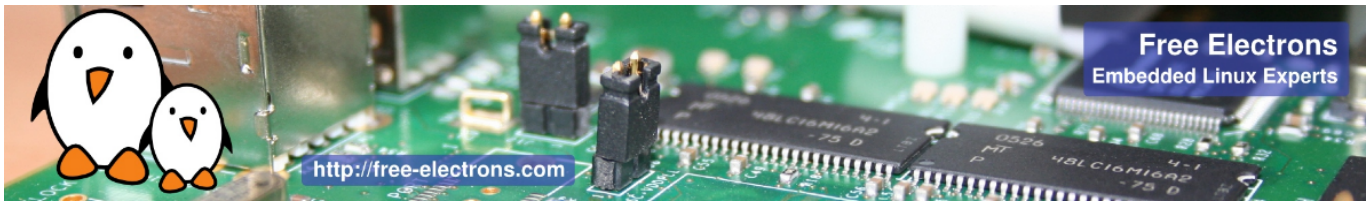
## Lecture - Configuring, compiling and booting the Linux kernel

- Kernel configuration.
- Native and cross compilation. Generated files.
- Booting the kernel. Kernel booting parameters.
- Mounting a root filesystem on NFS.

## Lab - Kernel configuration, cross-compiling and booting on NFS

*Using the BeagleBone Black board*

- Configuring, cross-compiling and booting a Linux kernel with NFS boot support.




## Day 2 - Morning

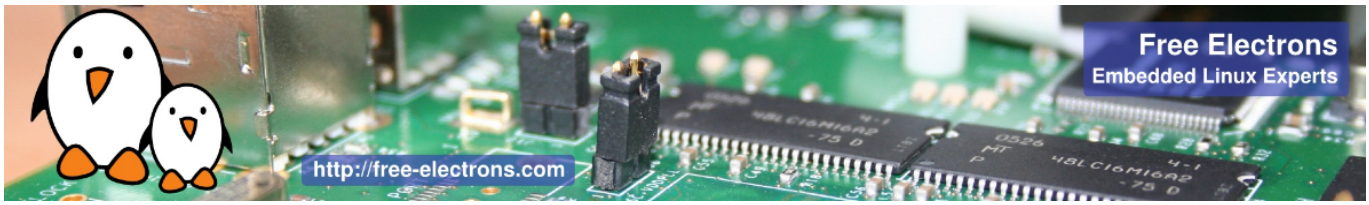
| Lecture - Linux kernel modules | Lab - Writing modules |
|---|---|
| • Linux device drivers<br>• A simple module<br>• Programming constraints<br>• Loading, unloading modules<br>• Module dependencies<br>• Adding sources to the kernel tree | *Using the BeagleBone Black board*<br>• Write a kernel module with several capabilities.<br>• Access kernel internals from your module.<br>• Setup the environment to compile it |

## Day 2 - Afternoon

| Lecture - Linux device model | Lab - Linux device model for an I2C driver |
|---|---|
| • Understand how the kernel is designed to support device drivers<br>• The device model<br>• Binding devices and drivers<br>• Platform devices, Device Tree<br>• Interface in user space: `/sys` | *Using the BeagleBone Black board*<br>• Implement a driver that registers as an I2C driver<br>• Modify the Device Tree to list an I2C device<br>• Get the driver called when the I2C device is enumerated at boot time |

## Day 3 - Morning

| Lecture - Introduction to the I2C API | Lecture - Pin muxing |
|---|---|
| • The I2C subsystem of the kernel<br>• Details about the API provided to kernel drivers to interact with I2C devices | • Understand the *pinctrl* framework of the kernel<br>• Understand how to configure the muxing of pins |




**Lab - Communicate with the Nunchuk over I2C**

*Using the BeagleBone Black board*

- Configure the pin muxing for the I2C bus used to communicate with the Nunchuk
- Extend the I2C driver started in the previous lab to communicate with the Nunchuk via I2C

## Day 3 - Afternoon

**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

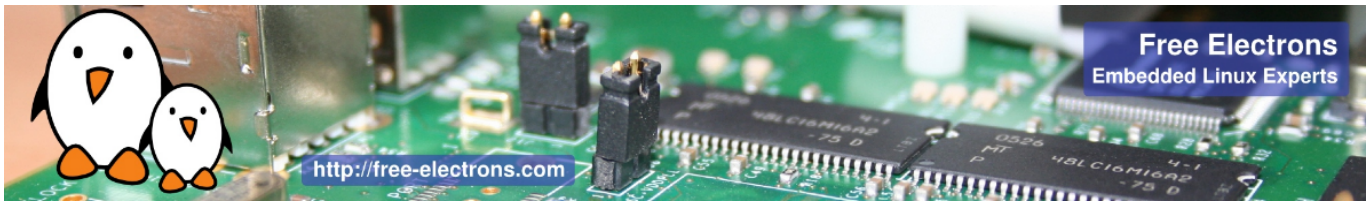
**Lecture - The input subsystem**

- 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

**Lab - Expose the Nunchuk functionality to user space**

*Using the BeagleBone Black board*

- Extend the Nunchuk driver to expose the Nunchuk features to user space applications, as a *input* device.
- Test the operation of the Nunchuk using sample user space applications




# Day 4 - Morning

**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()`.

**Lecture - I/O memory and ports**

- I/O register and memory range registration.
- I/O register and memory access.
- Read / write memory barriers.

**Lab - Minimal platform driver and access to I/O memory**

*Using the BeagleBone Black board*

- 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.

# Day 4 - Afternoon

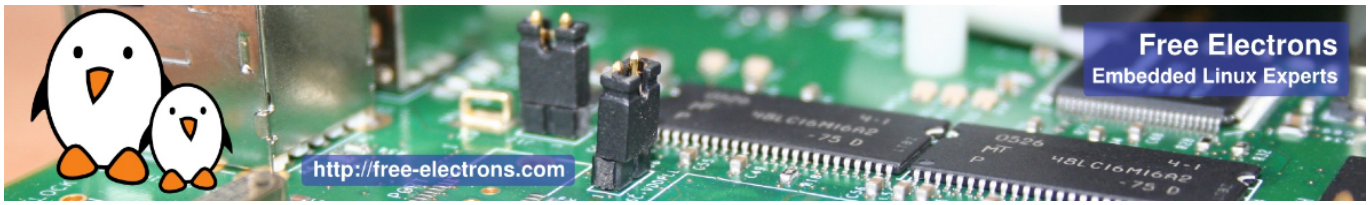
**Lecture - The misc kernel subsystem**

- What the *misc* kernel subsystem is useful for
- API of the *misc* kernel subsystem, both the kernel side and user space side

**Lab - Output-only serial port driver**

*Using the BeagleBone Black board*

- 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

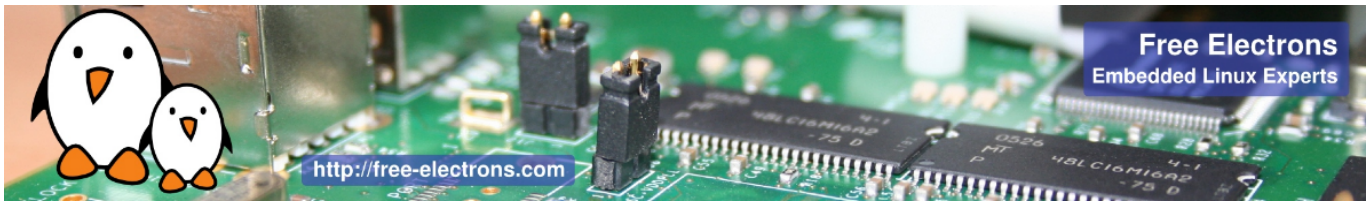



| Lecture - Processes, scheduling, sleeping and interrupts | Lab - Sleeping and handling interrupts in a device driver |
|---|---|
| • Process management in the Linux kernel.<br>• The Linux kernel scheduler and how processes sleep.<br>• Interrupt handling in device drivers: interrupt handler registration and programming, scheduling deferred work. | *Using the BeagleBone Black board*<br>• Adding read capability to the character driver developed earlier.<br>• Register an interrupt handler.<br>• Waiting for data to be available in the `read()` file operation.<br>• Waking up the code when data is available from the device. |

## Day 5 - Morning

| Lecture - Locking | Lab - Locking |
|---|---|
| • Issues with concurrent access to shared resources<br>• Locking primitives: mutexes, semaphores, spinlocks.<br>• Atomic operations.<br>• Typical locking issues.<br>• Using the lock validator to identify the sources of locking problems. | *Using the BeagleBone Black board*<br>• Observe problems due to concurrent accesses to the device.<br>• Add locking to the driver to fix these issues. |

| Lecture - Driver debugging techniques | Lab - Investigating kernel faults |
|---|---|
| • Debugging with printk<br>• Using Debugfs<br>• Analyzing a kernel oops<br>• Using kgdb, a kernel debugger<br>• Using the Magic SysRq commands<br>• Debugging through a JTAG probe | *Using the BeagleBone Black board*<br>• Studying a broken driver.<br>• Analyzing a kernel fault message and locating the problem in the source code. |




# Day 5 - Afternoon

**Lecture - ARM board support and SoC support**

- Understand the organization of the ARM support code
- Understand how the kernel can be ported to a new hardware board

**Lecture - Marvell Armada mainline support**

- Process of mainlining the Marvell Armada support
- Timeline of the mainlining process
- Location of Marvell Armada specific code in the kernel

**Lab - Building and booting mainline on Armada 385 RD**

- Building and configuring a little endian mainline kernel
- Booting on Marvell Armada 385 RD
- Exploring useful *debugfs* entries
- Building and booting a big endian mainline kernel

**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 - The Linux kernel development process**

- Organization of the kernel community
- The release schedule and process: release candidates, stable releases, long-term support, etc.
- Legal aspects, licensing.
- How to submit patches to contribute code to the community.