# Developing Embedded Linux Device Drivers (LFD435)

This instructor-led course is designed to show experienced programmers how to develop device drivers for embedded Linux systems, and give them a basic understanding and familiarity with the Linux kernel. Hands-on labs with a RISC-V based emulated development target allow students to practice what is learned in class.

**Duration:** 5 Days

## Prerequisites for this course

To make the most of this course, you must have:
Knowledge of basic kernel interfaces and methods such as how to write, compile, load and unload modules, use synchronization primitives, and the basics of memory allocation and management, such as is provided by LFD420 (Kernel Internals and Development). Pre-class preparation material will be provided before class.

## Outline for this course

Introduction
- Objectives
- Who You Are
- The Linux Foundation
- Copyright and No Confidential Information
- Linux Foundation Training
- Certification Programs and Digital Badging
- Linux Distributions
- Preparing Your System
- Things change in Linux
- Documentation and Links

Preliminaries
- Procedures
- Kernel Versions
- Kernel Sources and Use of git
- Hardware
- Staging Tree
- Labs

How to Work in OSS Projects **
- Overview on How to Contribute Properly
- Know Where the Code is Coming From: DCO and CLA
- Stay Close to Mainline for Security and Quality
- Study and Understand the Project DNA
- Figure Out What Itch You Want to Scratch
- Identify Maintainers and Their Work Flows and Methods

- Get Early Input and Work in the Open
- Contribute Incremental Bits, Not Large Code Dumps
- Leave Your Ego at the Door: Don't Be Thin-Skinned
- Be Patient, Develop Long Term Relationships, Be Helpful

Cross-Development Toolchain
- The Compiler Triplet
- Built-in Linux Distribution Cross Compiler
- Linaro
- CodeSourcery
- crosstool-ng
- Buildroot
- OpenEmbedded
- Yocto Project
- Labs

QEMU
- What is QEMU?
- Why use QEMU?
- Emulated Architectures
- Image Formats
- Labs

Booting a Target Development Board from uSD
- Why do we use uSD cards?
- Getting SW onto a uSD card
- Booting from flash
- Why is using uSD cards a bad idea?
- Labs

Booting a Target Development Board over Ethernet
- Using virtual Hardware
- An easier way to develop
- The Boot Sequence using TFTP and NFSroot
- Objectives of the Lab
- Labs

Kernel Configuration, Compilation, Booting
- Configuring the Kernel for the Development Board
- Labs

Device Drivers
- Types of Devices
- Mechanism vs. Policy
- Avoiding Binary Blobs
- Power Management
- How Applications Use Device Drivers
- Walking Through a System Call Accessing a Device
- Error Numbers

- printk()
- devres: Managed Device Resources
- Labs

Modules and Device Drivers
- The module driver() Macros
- Modules and Hot Plug
- Labs

Memory Management and Allocation
- Virtual and Physical Memory
- Memory Zones
- Page Tables
- kmalloc()
- get free pages()
- vmalloc()
- Slabs and Cache Allocations
- Labs

Character Devices
- Device Nodes
- Major and Minor Numbers
- Reserving Major/Minor Numbers
- Accessing the Device Node
- Registering the Device
- udev
- dev printk() and Associates
- file operations Structure
- Driver Entry Points
- The file and inode Structures
- Miscellaneous Character Drivers
- Labs

Kernel Features
- Components of the Kernel
- User-Space vs. Kernel-Space
- What are System Calls?
- Available System Calls
- Scheduling Algorithms and Task Structures
- Process Context
- Labs

Transferring Between User and Kernel Space
- Transferring Between Spaces
- put(get) user() and copy to(from) user()
- Direct Transfer: Kernel I/O and Memory Mapping
- Kernel I/O
- Mapping User Pages

- Memory Mapping
- User-Space Functions for mmap()
- Driver Entry Point for mmap()
- Accessing Files from the Kernel
- Labs

Platform Drivers
- What are Platform Drivers?
- Main Data Structures
- Registering Platform Devices
- An Example
- Hardcoded Platform Data
- The New Way: Device Trees
- Labs

Device Trees
- What are Device Trees?
- What Device Trees Do and What They Do Not Do
- Device Tree Syntax
- Device Tree Walk Through
- Device Tree Bindings
- Device Tree support in Boot Loaders
- Using Device Tree Data in Drivers
- Coexistence and Conversion of Old Drivers
- Labs

Interrupts and Exceptions
- What are Interrupts and Exceptions?
- Exceptions
- Asynchronous Interrupts
- MSI
- Enabling/Disabling Interrupts
- What You Cannot Do at Interrupt Time
- IRQ Data Structures
- Installing an Interrupt Handler
- Labs

Timing Measurements
- Kinds of Timing Measurements
- Jiffies
- Getting the Current Time
- Clock Sources
- Real Time Clock
- Programmable Interval Timer
- Time Stamp Counter
- HPET
- Going Tickless

Kernel Timers

- Inserting Delays
- What are Kernel Timers?
- Low Resolution Timer Functions
- Low Resolution Timer Implementation
- High Resolution Timers
- Using High Resolution Timers
- Labs

Ioctls
- What are ioctls?
- Driver Entry point for ioctls
- Defining ioctls
- Labs

Unified Device Model and sysfs
- Unified Device Model
- Basic Structures
- Real Devices
- sysfs
- kset and kobject examples
- Labs

Firmware
- What is Firmware?
- Loading Firmware
- Labs

Sleeping and Wait Queues
- What are Wait Queues?
- Going to Sleep and Waking Up
- Going to Sleep Details
- Exclusive Sleeping
- Waking Up Details
- Polling
- Labs

Interrupt Handling: Deferrable Functions and User Drivers
- Top and Bottom Halves
- Softirqs
- Tasklets
- Work Queues
- New Work Queue API
- Creating Kernel Threads
- Threaded Interrupt Handlers
- Interrupt Handling in User-Space
- Labs

Hardware I/O- Memory Barriers
- Allocating and Mapping I/O Memory
- Accessing I/O Memory

Direct Memory Access (DMA)**
- What is DMA?
- DMA Directly to User
- DMA and Interrupts
- DMA Memory Constraints
- DMA Masks
- DMA API
- DMA Pools
- Scatter/Gather Mappings
- Labs

Memory Technology Devices (Flash Memory Filesystems)
- What are MTD Devices?
- NAND vs. NOR vs. eMMC
- Driver and User Modules
- Flash Filesystems

USB Drivers
- What is USB?
- USB Topology
- Terminology
- Endpoints
- Descriptors
- USB Device Classes
- USB Support in Linux
- Registering USB Device Drivers
- Moving Data
- Example of a USB Driver
- Labs

Kernel Architecture I
- UNIX and Linux **
- Monolithic and Micro Kernels
- Object-Oriented Methods
- Main Kernel Components
- User-Space and Kernel-Space

Kernel Programming Preview
- Task Structure
- Memory Allocation
- Transferring Data between User and Kernel Spaces
- Object-Oriented Inheritance - Sort Of
- Linked Lists
- Jiffies
- Labs

Modules
- What are Modules?
- A Trivial Example
- Compiling Modules
- Modules vs Built-in
- Module Utilities
- Automatic Module Loading
- Module Usage Count
- Module Licensing
- Exporting Symbols
- Resolving Symbols **
- Labs

Kernel Architecture II- Processes, Threads, and Tasks
- Kernel Preemption
- Real Time Preemption Patch
- Labs

Kernel Configuration and Compilation
- Installation and Layout of the Kernel Source
- Kernel Browsers
- Kernel Configuration Files
- Kernel Building and Makefiles
- initrd and initramfs
- Labs

Kernel Style and General Considerations
- Coding Style
- Using Generic Kernel Routines and Methods
- Making a Kernel Patch
- sparse
- Using likely() and unlikely()
- Writing Portable Code, CPU, 32/64-bit, Endianness
- Writing for SMP
- Writing for High Memory Systems
- Power Management
- Keeping Security in Mind
- Labs

Race Conditions and Synchronization Methods
- Concurrency and Synchronization Methods
- Atomic Operations
- Bit Operations
- Spinlocks
- Seqlocks
- Disabling Preemption
- Mutexes
- Semaphores

- Completion Functions
- Read-Copy-Update (RCU)
- Reference Counts
- Labs

Memory Addressing
- Virtual Memory Management
- Systems With and Without MMU and the TLB
- Memory Addresses
- High and Low Memory
- Memory Zones
- Special Device Nodes
- NUMA
- Paging
- Page Tables
- page structure
- Labs

Memory Allocation
- Requesting and Releasing Pages
- Buddy System
- Slabs and Cache Allocations
- Memory Pools
- kmalloc()
- vmalloc()
- Early Allocations and bootmem()
- Memory Defragmentation
- Labs