Over 35 Years Of Technology Training

Document Generated: 12/18/2025

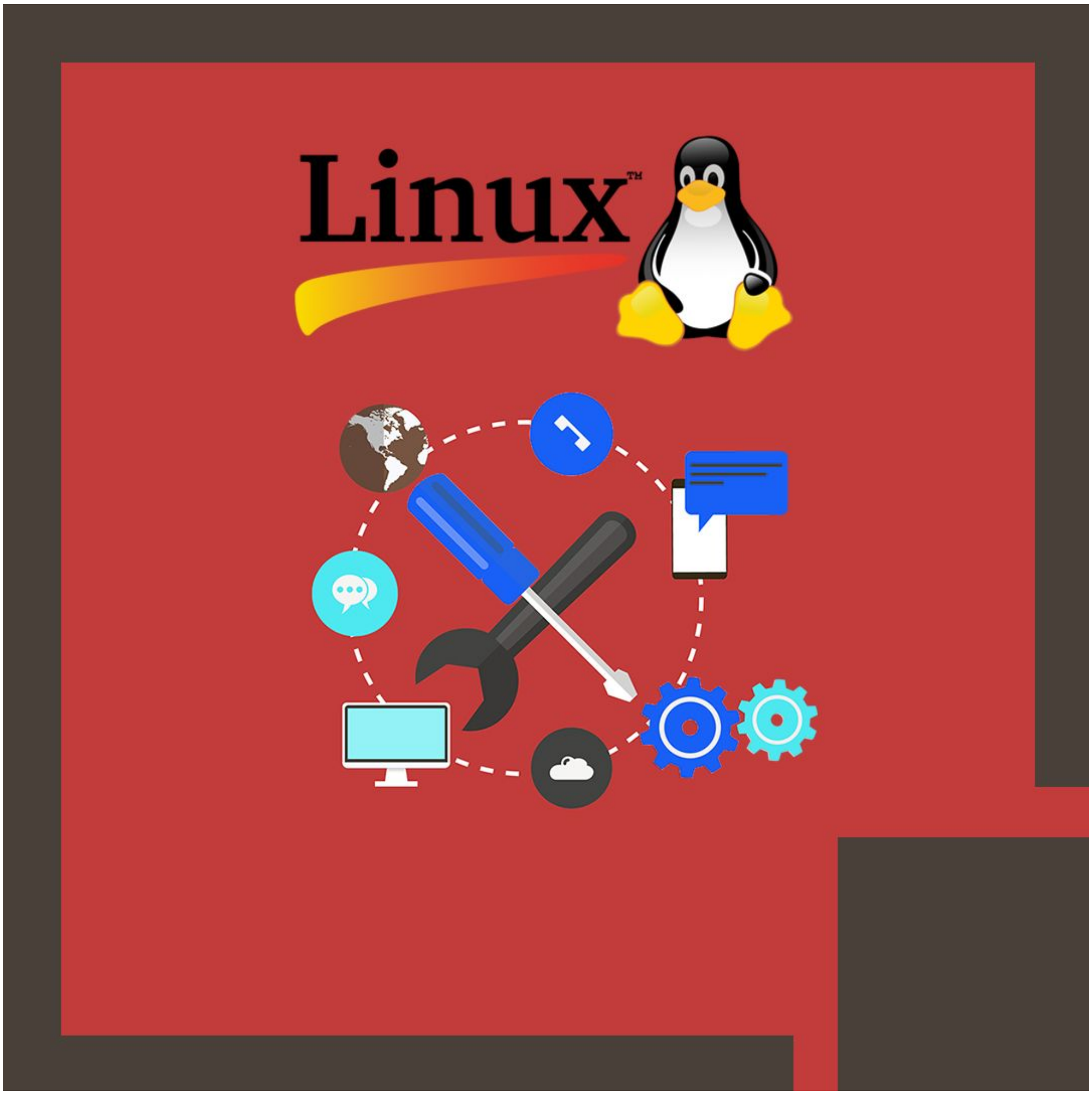
Learning Style: Virtual Classroom

Technology: Linux Foundation

Difficulty: Intermediate

Course Duration: 4 Days

Developing Linux Device Drivers (LFD430)



About this Course:

The Developing Linux Device Drivers (LFD430) is a comprehensive intermediate-level training program for professional App Developers and Language Programmers who want to attain a better understanding of developing Linux System Device Drivers. The primary objective of this course is to give an overview of major Linux device drivers and the appropriate Linux methods and APIs for Kernel device interfaces.

Professionals Linux Developers and C# & C/C++ Developers will learn the art of developing Linux device drivers and can utilize these learnings to help businesses maximize productivity. Businesses are always on the lookout for a proficient App Developers who can optimize and manage Linux system applications seamlessly. On average, an Embedded Linux Developer earns $107,500 annually and there is a great demand of professionals having proficiency in Linux device driver development.

Course Objective:

The core objective of this course is to help professionals gain a better understanding and sound knowledge of the following key concepts:

- Numerous Kinds of Linux Device Drivers
- Fundamental of Linux APIs and Kernel Hardware and Software Device Interface
- Best Practices and Module Techniques for Linux
- Debugging and Developing Linux Device Drivers
- Getting to Know Popular Linux Distributions

Audience:

This course is tailored for the following group of professionals:

- Programmers and App Developers
- C# & C/C++ Developers
- Professional Linux Developers

Prerequisites:

Professionals planning to enroll in the Developing Linux Device Drivers (LFD430) course must comply with the following prerequisites:

- Fundamental Knowledge of Kernel Interfaces including loading and unloading modules, creating & compiling interfaces, and synchronizing primitives.
- Basic Know-how of Management and Memory Allocation Essentials such as LFD420 Linux Kernel Development and Internals.

Course Outline:

Introduction

- Objectives
- Who You Are
- The Linux Foundation
- Linux Foundation Training
- Linux Distributions
- Platforms
- Preparing Your System
- Using and Downloading a Virtual Machine
- Things change in Linux
- Documentation and Links
- Course Registration

Preliminaries

- Procedures
- Kernel Versions
- Kernel Sources and Use of git
- Rolling Your Own Kernel
- Hardware
- Staging Tree

How to Work in OSS Projects **

- Overview on How to Contribute Properly
- 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

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

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
- Labs

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

- Buses and Ports
- Memory Barriers
- Registering I/O Ports
- Reading and Writing Data from I/O Registers
- Allocating and Mapping I/O Memory
- Accessing I/O Memory
- Access by User – ioperm(), iopl(), /dev/port
- Labs

PCI

- What is PCI?
- PCI Device Drivers
- Locating PCI Devices
- Accessing Configuration Space
- Accessing I/O and Memory Spaces

- PCI Express
- Labs

Platform Drivers**

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

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

Network Drivers I: Basics

- Network Layers and Data Encapsulation
- Datalink Layer
- Network Device Drivers
- Loading/Unloading
- Opening and Closing
- Labs

Network Drivers II: Data Structures

- net_device Structure
- net_device_ops Structure
- sk_buff Structure
- Socket Buffer Functions
- netdev_printk() and Associates
- Labs

Network Drivers III: Transmission and Reception

- Transmitting Data and Timeouts
- Receiving Data
- Statistics
- Labs

Network Drivers IV: Selected Topics

- Multicasting **
- Changes in Link State
- ioctls
- NAPI and Interrupt Mitigation
- NAPI Details
- TSO and TOE
- MII and ethtool **

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

Power Management

- Power Management
- ACPI and APM
- System Power States
- Callback Functions
- Labs

Block Drivers

- What are Block Drivers?
- Buffering
- Registering a Block Driver
- gendisk Structure
- Request Handling
- Labs

Closing and Evaluation Survey

- Evaluation Survey

**** These sections may be considered in part or in whole as optional. They contain either background reference material, specialized topics, or advanced subjects. The instructor may choose to cover or not cover them depending on classroom experience and time constraints.**