Linaro Training Catalogue
2024

linaro™

Table of Contents

Learn Open Source development on Arm

Finding the information you need to succeed, while trying to navigate the peculiarities of the Open Source ecosystem, can be hard. In fact, at times, it can be quite daunting.

Having a friendly face or someone able to share the 'tricks of the trade' with you can be invaluable and can mean the difference between success and failure. If nothing else, having an experienced partner to fall back on can save you considerable time and money.

Linaro, with more than 14-years' experience of helping to drive co-operation within the Open Source community, has built a unique understanding of how the community operates. Working impartially with some of both the largest, as well as some of the smallest companies in the field, Linaro has developed an enviable reputation for both expertise and experience.

Linaro employs key developers and maintainers for many Open Source projects, including the Linux Kernel, OP-TEE and QEMU. Linaro usually features as a leading contributor to the Linux kernel, month on month. This in-depth knowledge and experience makes Linaro's Linux and Arm technologies training second-to-none.

We are proud of our training and hope this catalogue introduces the ways that Linaro training can work for you but we know it's not the same as interactive discussion! If you want to talk to us about training events for you and your teams please contact training@linaro.org and we'll be very happy to discuss further.

Training at Linaro

Linaro provides off-the-shelf or customized training on a variety of topics. We provide expert instructors who are real world engineers and are specialists in delivering hands-on training across Linux and Arm technologies.

Linaro's modular training format is highly flexible and can be delivered either Face-to-Face or online, at a time appropriate for your time zone.

Face-to-Face training

All Face-to-Face (F2F) training is conducted either at your own premises or at a suitable venue, such as a Hotel or Conference Centre, local to you and chosen by you.

F2F sessions include informal lecture presentations or discussion groups interspersed with full instructor-supported "Hands-on" Lab sessions. The Lab sessions provide attendees with valuable practical experience. They help consolidate the conceptual learning and prompt valuable discussion around those concepts, all adding to deepen understanding and confidence, post training.

Online training

Linaro can provide online training delivered using web conferencing platforms such as Zoom.

Online training sessions use exactly the same training materials and instructors as our Face-to-Face events but, because of the flexibility of the medium and the removal of the need to travel, online delivery gives the opportunity for a slower pace of delivery. Typically, a three-day F2F course is delivered over six 3-hour sessions, spread over a period of two, three or six weeks. The slower pace results in a less intense learning environment, giving trainees longer to get to know both the trainers and the subject matter. Experience has shown that this slower pace helps to develop important social relationships, often lost during online delivery methods. It can also make training easier to fit around the busy work schedules of your trainees.

To make best use of on-line time and to help students achieve success, the Lab sessions associated with online training courses are usually set as homework to be complete offline.   Students still receive the same level of instructor support as F2F students, just usually in the form of e-mail Q&A.

Our content is carefully designed to be suitable for both on-site and remote training.

On-site events consist of informal lecture-style delivery combined with lab exercises taken during the class. The trainer (or trainers in the case of large events) will spend their time during the lab sessions providing any assistance trainees may need to complete the lab sessions and, more generally, to discuss anything related to the training subject that trainees want to bring up.

Modular training - Learn what you need to know

Linaro has designed its training courses around a modular 'Topic' based structure, with each module representing approximately one half-day of F2F training, or a single on-line session. This catalogue introduces our training material as stand-alone courses but each course is built by combining several individual modules. Each module has a three letter code (e.g. TLC-01) and this modular structure allows us to offer you more than just a selection of off-the-shelf courses. It also allows us to offer customized courses by combining modules in different ways to better meet your requirements: you are free to 'pick & mix' any number of individual Topics, combining them to suit your specific need.

We also offer a library of booster modules which are not integrated as part of our full courses but which can be integrated into any of our courses as needed.

Practical 'Hands-on' learning: Your hardware, your choice!

Linaro's 'Hands-on' practical training sessions are designed to replicate the real world environment your developers will be familiar with. Typically, Linaro Lab sessions consist of carefully planned exercises designed to support the conceptual learning, helping students to consolidate their knowledge.

These practical exercises can be completed using QEMU-based virtual machines. A QEMU VM provides a consistent experience for all students that is easily supported by our instructors and is especially well suited for remote inline training.

However we know there are times when training on real hardware can provide a better experience for our students. In these cases we can offer additional services to adapt the lab books to allow some or all of the lab sessions to be undertaken on your own hardware (or on a commonly available community board, if that better suits the needs of your developers better). Undertaking lab sessions on the hardware devices that your trainees will use everyday can really close the gap between taking the training and applying what you have learned!

Quick Reference Guide

Comprehensive courses

Course	Level	F2F sessions	Online sessions	Prerequisites
Linux Kernel Development	Primer	2 days	4	C-programming experience required: Unix-like shells for file management, editing and invoking development tools.
Upstream Kernel Development	Intermediate	1 day	2	Trainees should understand how to write kernel drivers. No previous kernel community experience necessary.
Advanced Kernel Debugging	Advanced	2 days	4	Experience of Linux kernel programming together with traditional debug techniques such as log messages or stop/start debuggers.
Linux Kernel for Real-Time Systems	Intermediate	1 day	2	Experience of either real-time systems or integrating embedded Linux systems.
Hands-on introduction to Rust	Intermediate / Advanced	2-5 days	5	Significant programming experience and familiar with the fundamentals of computer science and software engineering.
OpenEmbedded and the Yocto Project	Primer / Intermediate	3 days	6	Prior embedded Linux experience, or eust copied the above form xperience using desktop Linux distributions with command line tools (shell scripting, etc).
Automatic validation with LAVA	Primer	1-5* days	3*	A general understanding of Linux as a user (Ubuntu, Debian). * Due to the nature of this topic it is strongly recommended that this course be combined with additional near-team goal-oriented consultancy to plan next steps.

Course	Level	F2F sessions	Online sessions	Prerequisites
Energy Aware Scheduler	Advanced	2 days	4	Experience of system-level Linux system debug and some background in power management.
KVM & Virtual I/O for 64-bit Arm Systems	Advanced	1 day	2	No prior experience necessary. A general understanding of virtualization or hardware emulation is useful.
Trusted firmware for A-profile Arm Systems	Intermediate	1 day	2	Experience of low-level OS or boot loader development using C
Introduction to OP-TEE	Intermediate	2 days	4	Comfortable using command line tools for software development.

Booster modules

Topic	Level	F2F sessions	Online sessions	Prerequisites
Reading (and writing) A64 assembler	Primer	0.5 day	1	Prior development experience using low-level languages such as C or Rust. No prior assembly experience is required.
WiFi: Linux implementation and debug	Intermediate	0.5 day	1	Existing kernel development experience is essential.

Linux Kernel Development

Linux Kernel Development is a short, fast paced introductory course to Linux kernel development. The course focuses strictly on driver development avoiding abstract discussion of kernel internals in order to keep the course as concise as possible..

Introduction to Kernel Development is a four module course equivalent to approximately two days face-to-face training and is suited to both face to face and remote delivery. It combines well with the Upstreaming course (2 additional modules) and/or the Advanced Kernel Debugging course (4 additional modules)

Trainees are expected to have C programming experience and to be comfortable working with Unix-like shells for file management, editing and invoking development tools. Trainees will learn how to write device drivers for Linux.

LKD-01: Introduction to Devicetree

Devicetree is a data structure for describing hardware and is used to describe both the System-on-Chip and board design of modern Arm embedded systems. Understanding Devicetree and the ecosystem around it is a vital concept to work on these platforms because it is fundamental to both board porting and debugging.

- From platform bus to devicetree
  - Historic overview
  - What device is (and is not)
- Devicetree as a tree
  - Nodes and properties
  - Devicetree source format
  - Flattened devicetree format
- Devicetree idioms
  - Separating SoC and board
  - SoC families
  - Minor revisions
- Devicetree bindings
  - Navigating existing devicetree bindings
  - Pinctrl bindings
  - Best practices for binding new hardware
- The Future (is already here): Devicetree, YAML and json-schema
- Lab/homework
  - Writing a devicetree for a custom board

LKD-02: Pragmatic Linux driver development - Part I

Before we can write drivers, we first need to be able to write code that executes in the kernel. The kernel is written in C but the style of kernel code is different to typical user-space C programs, using different library calls and idioms. In this module we will look at how to write kernel code and introduce important low-level services such as memory allocation and locking. We will then move on to look at the Linux Device Driver model in greater detail, setting us up for the hardware and time handling topics in Part II.

- First steps
  - Download, build, install and boot
  - Your first kernel module
- Exploring the kernel via character drivers
  - File operations (ops tables)
  - Memory allocation
  - Locking primitives
  - Why you shouldn't write a character driver!
- Linux device driver model
  - Devices, classes, buses
  - Driver binding
- Case study: omap watchdog
- Lab/homework
  - Write hello world kernel module
  - Your own membuf driver
  -

LKD-03: Pragmatic Linux driver development - Part II

This module focuses on handling hardware and managing the passage of time. I shows how the information about the hardware encoded in the devicetree can be accessed and used by drivers to communicate with hardware and receive interrupts. After this, we will look at the various time management techniques including measuring elapsed time, setting up precisely timed activity and how to efficiently handle imprecise timings such as timeouts. The module wraps up by covering device power management and is supported with exercise to guide trainees through the development of a simple but fully functional I2C driver.

- Hardware handling
  - Parsing device properties
  - Memory mapped I/O
  - Regmap
  - Interrupt handling
  - GPIO
- Time management
  - Measuring time
  - Delaying execution: busy waiting and sleeping
  - Timer-wheel timers and high-resolution timers
  - Deferred and delayed work to workqueues
- Device power management
  - System sleep
  - Runtime PM
- Lab/homework
  - I2C device/driver initialization
  - Complete I2C driver for a temperature sensor
  - Explore hwmon subsystem

LKD-04: Symbolic debugging for Linux kernel and userspace

Debugging kernel drivers often involves working across the user/kernel boundary studying the behavior of both the kernel and its userspace clients. Linux kernel and userspace already provide a good foundation for message/syslog based debugging. Attaching a traditional stop-the-world debugger to the kernel or to any process within it can augment the logs with additional information to help find bugs. In this module we will learn about how to use open-source stop-the-world debuggers on Linux systems, both to debug the kernel and user-space applications.

- Fundamentals
  - Kernel features for multi-process debugging
  - Compiler options for debugging
  - Debugging with gdb
  - Cross-debugging with gdbremote
- Case study: Debugger integration for VS Code (or Eclipse if preferred by trainees)
- Kernel debugging
  - kgdb and kdb
  - OpenOCD
- Case study: Debugging optimized code and code without debug information
- Lab/homework
  - Debugging userspace processes
  - Self-hosted kernel debugging
  - Kernel debugging using "fake JTAG"
  - Let's crash!

Upstream Kernel Development

Upstream Kernel Development shows trainees how to contribute to the Linux kernel. This includes understanding the role of kernel maintainers, how to best align their work with the Linux release schedule, how to handle feedback and what to do if things don't go to plan.

Upstream Kernel Development is a two module course, equivalent to approximately a day face-to-face training. It is suited to both face to face and remote delivery.

This course covers the "social" aspects of upstream kernel development. It is best suited to trainees who already understand how to write kernel drivers but have not previously worked alongside the wider kernel community. Trainees will learn how to confidently participate in upstream kernel development and, in particular, how to contribute new kernel drivers.

UPS-01: Mechanics

There is a mature process for proposing changes to the upstream Linux kernel. In this module, we go through details of every stage of that process, combining it with examples and interpretation to help trainees take part more confidently in upstream kernel development.

- What is Upstreaming?
- How the Linux project is organised
- How to Upstream?
    - Patch Preparation
    - Patch Creation
    - Patch Posting
    - Feedback
    - Maintenance
    - Worked example
- Lab/homework
    - Further reading
    - Minarai - Learning by watching

UPS-02: Tips, tools and techniques

Upstreaming means meeting community expectations and navigating social situations that emerge on mailing list and in other communication channels. In this module we will help trainees better understand these social processes and introduce them to tools that can help them meet community expectations with less effort. These combine to give trainees the skills they need for more successful upstreaming.

- Talking with upstream
- Tags
- Thinking like a maintainer
- Kernel versioning and release flow
- Sharp tools and smart techniques
    - Train your style
    - Filtering mailing lists
    - Source navigation
    - Static checkers
    - Handling regressions and bisectability testing
- What do do when you are ignored
- Case study: Mailbox upstreaming

Advanced kernel debugging

Advanced kernel debugging introduces a variety of kernel-specific debug tools and techniques that can be used to debug complex system level problems on Linux systems. The course commences with a high level overview of different debugging use-cases and uses them to introduce tools that can be used to solve different problems. After that we study three advanced tools in-depth: ftrace, eBPF and perf

Advanced kernel debugging is a four module course equivalent to approximately two days face-to-face training and is suited to both face to face and remote delivery.

Trainees are expected to have experience of Linux kernel programming together with traditional debug techniques such as log messages or stop/start debuggers. Trainees will learn a wide variety of powerful debugging techniques using tools that are already integrated into the Linux kernel.

AKD-01: Kernel debug stories

Kernel debugging involves learning a wide variety of tools whose scope is very different to a traditional stop-the-world debugger. By focusing on use cases rather than on the detail of exactly how each tool works we are able to cover a wide variety of advanced debug tools in a short space of time. This provides both a foundation and a real-world context for detailed examination of different debug tools.

- Overview of tracing, profiling and stop-the-world
- Using automatic tools to fail early
  - Review of several common kernel config options to fail early
  - Failing early with PROVE_LOCKING
  - Kernel hardware assisted address sanitizer (KHWASAN) with and without the memory tagging extension
  - Failing early in production with KFENCE
- Case studies
  - "I can't reproduce but my customer can": performance of printk, Arm memory map, ftrace for function tracing, kdump/kcrash
  - "My XYZ missed its deadline": alternative ftrace tracers, perf, lock statistics, Coresight & OpenCSD, eBPF
  - "My board just stopped dead": initial triage, JTAG, SoC level debug, ftrace, ramoops, how to read an oops
  - "I'm sure this used to work": bisection, ktest.pl
- Use case: "My board just randomly failed"
  - Investment ahead of time
  - Debugging when problems appear
- Lab/homework
  - initcall_debug experiments
  - DYNAMIC_DEBUG experiments
  - Explore DebugFS

AKD-02: Tracing with ftrace

Ftrace is a kernel-based tracing framework for Linux. Ftrace is not merely a function tracer but can also be used for a variety of different tracing scenarios utilizing both the tracing framework and the many tracepoints spread throughout the kernel. This module introduces trainees to this powerful tool, showing both how to configure the tracer using basic shell commands as well as providing an introduction to additional tools that can visualize and process trace results.

- Fundamentals
  - Using profiling hooks to instrument function call/return
  - Static tracepoints
  - Kprobe events
- Function tracing
  - Controlling ftrace via debugfs and from the kernel command line
  - Function call tracing
  - Function call and return tracing
  - Filtering trace events
  - Handling static tracepoints
  - Dynamically creating tracepoints
- Other tracers
  - Latency tracer
  - Branch tracer
- Tools
  - trace-cmd
  - KernelShark
  - LISA
- Case studies
  - How to examine complex internal interactions
  - Studying scheduler decisions
- Lab/homework
  - Catch-all function tracing
  - Targeted function tracing
  - Function graph tracing
  - Dynamic tracing experiments

AKD-03: Debugging with eBPF

The extensions in eBPF, the extended Berkeley Packet Filter, transformed BPF into a powerful in-kernel virtual machine and this ultimately allowed eBPF programs to be attached to many parts of the kernel. By attaching eBPF programs to static and dynamic tracepoints we can perform fast, safe, dynamic analysis of running systems.

- Using eBPF for debugging
- Dive into eBPF implementation
  - Instruction set (ISA)
  - Verifier

  - Maps
  - Worked example
- eBPF tools
  - Kernel examples
  - Ply - lightweight, easy to learn, easy to deploy eBPF frontend
  - BCC - combining eBPF and scripting languages
  - bpftrace - digging through kernel data structures
- Case studies
  - Gathering statistics about run states
  - Who is hammering a library function?
  - Hunting leaks
  - Reusing tools from other developers
- Lab/homework
  - Explore eBPF tools: ply, bpftrace and bcc
  - Adopting eBPF tools to a new kernel
  - Userspace probes

AKD-04: Using perf on Arm platforms

Perf is a powerful tool for profiling and debugging the Linux kernel. In addition to providing a way to exploit the device's performance counters, perf also provides support for multiple profiling techniques including both software and hardware tracing.

- Fundamentals
  - Basics of statistical profiling
  - Time based profiling
  - Event based profiling
- Using perf with tracing tools
  - Profile with ftrace
  - Profile with probes
  - Profile with CoreSight
- Case studies
  - Identifying cache related bottlenecks
  - Feedback directed optimization using hardware profilers
- Lab/homework
  - System-wide flat profiling
  - Single process trace
  - Call graph profiling

Using the Linux kernel for real-time systems

Using the Linux kernel for real-time systems is a primer on the real-time behavior of the Linux kernel. It covers the system and library calls that are most critical to building real-time applications and then takes a closer look at the different ways preemptive scheduling can be configured and implemented within the Linux kernel.

This is a two module course, equivalent to approximately a day face-to-face training. It is suited to both face to face and remote delivery.

Trainees must have experience of either real-time systems or integrating embedded Linux systems. Trainees will learn about the trade-offs involved in the different kernel preemption modes together with an overview of the tools you can use to study system behavior of real-time Linux systems.

RTL-01: Managing real time activity

Linux kernel and it's associated low-level libraries, provides APIs for its applications that are richer and more complex than those of a traditional RTOS. Additionally traditional RTOS features, such as interrupt locking, are unavailable. In this module, we will introduce the API needed to build real-time applicaitons for Linux. We'll also look at how virtual memory can impact real-time behaviour and also at some other common system integration pitfalls that can harm real-time performance.

- Pthreads and scheduler classes
    - Normal scheduling: SCHED_OTHER, SCHED_BATCH and SCHED_IDLE
    - Priority based scheduling: SCHED_FIFO and SCHED_RR
    - Earliest deadline first scheduling: SCHED_DEADLINE
- Pthread mutexes
    - PTHREAD_PRIO_NONE
    - PTHREAD_PRIO_INHERIT
    - PTHREAD_PRIO_PROTECT
- Condition variables
- Signal handling
- Clocks and timers
    - Available clocks
    - Alarms and interval timers
    - POSIX timers
    - Precise sleeping
- Virtual memory locking
- System integration
    - Poll/select
    - Capabilities
    - Real-time controllers for cgroups
- Profiling tools
- Lab/homework
    - Real-time threads
    - Setup wakeup latency test for RT thread
    - Locking memory
    - Allow non-root users to launch RT apps

RTL-02: Real time implementation and analysis

In this module we will look at real-time in a more formal way by discussing the Linux scheduler and different contributors towards scheduling latency. This is followed by a tour of the kernel features that can be used to tune and improve critical latencies within a Linux system. The module then wraps up by looking more deeply at time and time keeping in Linux systems.

- Workload classification
- Linux kernel scheduler implementation
- Scheduler side topics (that impact real time performance)
- Latency analysis
- Preemption evolution
  - Preemption only for user space
  - Voluntary preemption
  - Full preemption
  - Preempt RT
- Timers and time keeping
  - Clock sources and clock events
  - Scheduling-clock ticks and the timer wheel
  - High resolution timers
- Unexpected behaviors: tasklets!
- Lab/homework
  - Running a kernel with and without PREEMPT_RT
  - PREEMPT_RT on your hardware

A Hands-on Introduction to Rust

Rust is an expressive high-level multi-paradigm language but many other languages offer this. Rust is unusual for being all these things whilst simultaneously being a memory-safe language suited to systems and bare-metal programming!

A hands-on introduction to Rust is a five module course equivalent to approximately two and a half days face-to-face training and is suited to both face to face and remote delivery.

This course divides its attention roughly equally between language, standard library and the ecosystem of tools and crates. This triple focus on language, library and ecosystem allows A hands-on introduction to Rust to complement other Rust learning resources. It provides a broad foundation of practical skills that allow trainees to get started quickly whilst also offering a springboard for further study.

Trainees are expected to have significant previous programming experience and to be familiar with the fundamentals of computer science and/or software engineering. Trainees will be introduced to a wide variety of Rust language and library features. Trainees will learn how to write, test and benchmark Rust code and will be provided with a solid foundation on which to build further Rust skills on-the-job.

RST-01: Getting started with Rust

This module introduces both the why and the how of Rust. Students will learn about the unique combination of strengths provided by the language before touring several short programs to help them understand the general structure of Rust programs. After this comes a comprehensive overview of Rust's syntax and data-types. This is followed by a short overview of I/O in Rust and a demonstration of how to set up the build tools. The primer leaves trainees fully equipped to write their first Rust programs.

- Why Rust?
- Examples: hello, fib, sleep
- A Rust primer
  - Fundamentals
  - Data structures
  - Arrays, slices, strings and vectors
  - Basic I/O
- Demo: Build tools

RST-02: Ecosystem and Libraries

Understanding the Rust ecosystem is important for new Rust programmers. Even kernel and systems programmers who expect to strictly limit their crates used in their programs will benefit from access to the wide variety of third-party open-source code that can help them refine their understanding of the language. During this session trainees will also learn how to make their code more idiomatic and adopt the simple and effective error handling patterns using language concepts such as the try operator.

- The Rust Ecosystem
  - Rust Project organization
  - Crates, cargo and crates.io

    - Building with Cargo
- Idiomatic Rust
    - Introducing traits
    - Example: wc
    - Handling errors and panics in Rust

RST-03: The Rust Type System

Having covered the fundamentals of the language and ecosystem, at this stage trainees are ready to fully explore the Rust type system. Here we will deep dive into the language features that support a high level approach to problem solving when compared to imperative languages such as C. This involves functional programming, generic programming and the tools Rust provided for polymorphism.

- Programming in a functional style
    - Closures and iterators
- Generics
- Traits
- Polymorphism
    - Fundamentals
    - Boxes and polymorphism
- Design patterns and case studies
- Multi-paradigm but no objects?

RST-04: Taming the Borrow Checker

The borrow checker is the defining feature of Rust and it has had a profound influence over the design of the language. A Hands-On Introduction to Rust is structured to allow trainees to learn the language, library and tools with minimal interference from the borrow checker. Nevertheless no introduction to Rust can be complete without studying the borrow checker, how lifetimes can be managed with a single thread and how the library supports multi-threaded data sharing and resource tracking. Trainees will also study adjacent topics including unsafe Rust and foreign function interfacing.

- Borrow checking
    - Basics
    - Avoiding problems
- Lifetimes
    - Lifetime elision
    - Explicit lifetimes
- Library tools
    - Reference counting
    - Threading
- Unsafe Rust
    - Foreign function interfacing
- Wrap up

RST-05: Embedded Rust and Rust for Linux

In this module we will look at the differences between regular Rust and embedded Rust. Differences are simultaneously both modest and profound: cross-compilation, unavailability of the standard library and, very often, extensive Foreign Function Interfacing (FFI). We will learn embedded Rust firstly by exploring how to interface Rust to an existing embedded RTOS (our case study is based on Zephyr but the concepts and principles apply much more broadly). Following that we explore one of the most comprehensive “embedded” Rust environments, Rust4Linux.

- Embedded Rust
  - What is Embedded Rust?
  - Case study: Zephyr FFI example
- Rust4Linux
  - Minimal example
  - Kernel flavoured Rust
  - Case study: PHY driver
  - Case study: NVMe driver
  - Wrap up

OpenEmbedded and the Yocto Project

OpenEmbedded and the Yocto Project is a complete introduction to developing and maintaining Linux distributions using OpenEmbedded tools and Yocto Project releases.

We start off by compiling a complete Linux distribution from scratch before moving on to look at different ways to customize the resulting image to make it into a base for application development or porting. We'll also look at how to adapt the Linux kernel that is built as part of the distribution.

Initially the training focuses on experimental customization because this approach makes some of the OpenEmbedded concepts more tractable. However after this initial focus on experimentation we start to look more deeply at how recipes work and the best ways to manage your distributions in a reliable and maintainable manner.

OpenEmbedded and the Yocto Project is a three day course when delivered face-to-face and also delivered in remote format as a six module course.

Trainees are expected to either have prior embedded Linux experience or experience using desktop Linux distributions with command line tools (shell scripting, etc).

OYP-01: OpenEmbedded/Yocto Project – Getting Started

There are many different ways to build operating systems using OpenEmbedded tools. During Getting Started we ensure all trainees start from the same point by looking at how to build and boot a specific example system.

Normally we will guide trainees through the process of running the Yocto Project reference distro (poky) using an emulated Qemu-based platform. However, alternatively, we can customise the lab book so that your trainees learn everything they need using your own choice of platform and OpenEmbedded distribution. This is a great way to close the gap between training and real-world practice of what you learn!

The rest of this session is spent building up basic vocabulary for Linux distributions. In particular we will review the major components and contrast their roles within both desktop and embedded systems.

- Getting started
    - Your first build
    - Describing what happened during your first build
    - Booting using QEMU
- Anatomy of a typical Linux distribution
    - Bootloader
    - Kernel
    - Init system and device manager
    - Libraries, applications and services
    - Package management
    - Summary

OYP-02: OpenEmbedded experimenter’s guide

The experimenter’s guide is entirely focused on how a developer can customise their own builds. We will show how to add extra components to a build, how to compile and run software that was not supplied as part of the base distribution components, and how to tweak customizable packages such as the Linux kernel.

- Adding additional packages to the build
- Building an SDK
- Cross-compiling userspace code
- Configuring the kernel build
- Source code changes

OYP-03 & -04: Important OpenEmbedded concepts

This is a double-length module and, at its center, it focuses on how recipes become packages. After a brief introduction studying how the Yocto Project is organized we begin to introduce how the OpenEmbedded tools interpret the metadata that drives the build and packaging process.

- OpenEmbedded and the Yocto Project
  - Introduction to (embedded) Linux distros
  - History of OpenEmbedded and the Yocto Project
  - Linaro RPB and OpenEmbedded
- Important OpenEmbedded concepts
  - Metadata
  - Build environment
  - Recipes
  - Dependencies and packages
  - Configuration files
  - Bitbake
- Build workflow
  - Build workflow
  - Anatomy of the build folder
  - Run scripts and log files

OYP-05: Layers and troubleshooting

Layers are vital to help developers make their distributions maintainable. They do this by making it easy to separate metadata so that different developers can work independently in different parts of the distribution.

After covering layers, trainees will learn how to debug common problems that they may encounter. This includes learn how to fix metadata problems using tools to see how the build has changed over time and how layers interact with each other. We also look at tools to help us fix problems that the distro inherits from its upstream sources.

- Understanding layers
    - Layers and bitbake
    - Commonly imported layers
    - Tools: bitbkake-layers and devtool
    - Contributor's guide
- Troubleshooting
    - Buildhistory
    - Bitbake variable debugging
    - How to modify source code locally
    - Runtime debug

OYP-06: Advanced OpenEmbedded

Metadata drives almost everything that happens within an OpenEmbedded build. That covers much more than just packaging software with recipes. In this session we cover a variety of these advanced topics. Threaded through this session is details on how to build customized images for custom boards based on a distribution that is constructed around your requirements. We also look at tools to promote code-reuse within recipes as well as techniques like packageconfig and virtual packages that allow recipes to be shared by all distributions created using OpenEmbedded.

- Classes
- Image recipes
- Packageconfig
- Virtual packages
- Machine and distro configuration
- Alternative toolchains

Automated validation with LAVA

This course is a beginner tutorial covering both LAVA usage and LAVA administration. Trainees will gain practical experience of LAVA by spinning up a micro-instance on either their own workstations or on a shared test server. The micro-instance is based on docker containers that work together to provide a LAVA instance for experimentation. The LAVA instance is complemented by other components to provide a complete example CI loop based around LAVA.

Automated validation with LAVA is a hybrid training/consultancy programme comprising a three module course, equivalent to approximately a day of face-to-face training, together with customer-led consultancy time. The additional consultancy time is flexible and intended to ensure customers successfully meet their near-term goals for LAVA whether those goals are adding support for particular boards, integrating specific test suites, migrating the LAVA micro-instance to work with existing CI infrastructure or something else entirely!

This course does not require any specific prior experience except for a general understanding of Linux as a user (Ubuntu, Debian). Trainees will learn how to use LAVA to perform automated testing on real hardware as well as how to set up and maintain a LAVA lab instance.

LVA-01: LAVA for users - Part I

In this module, we look at the basics of what LAVA and the role it plays within a complex heterogeneous CI system by managing the devices-under-test. As part of this we look at how LAVA jobs are expressed and how test cases and test suites for Unix-like systems can be integrated using the LAVA test shell.

- What is LAVA?
- Writing and Submitting LAVA jobs
- LAVA test shell
- Lab/homework
    - Your first LAVA job
    - Writing your own tests

LVA-02: LAVA for users - Part II

This starts by looking at how LAVA interacts with other parts of the CI system. As part of this we will look at how LAVA colates results from each test job and the XML-RPC and RESTful APIs to automate interaction with LAVA. After we'll tour some advanced LAVA concepts, including multi-node test jobs. We also look at how fully existing test suites into LAVA, both for POSIX and for microcontroller test shells.

- Exploring results
- LAVA APIs
- Multinode test job
- Hacking session
- Integrating a test framework
- Monitor and Interactive tests
- Misc Tricks and Tips
- Lab/homework
    - Qemu boot time chart
    - Explore hacking session

LVA-03: LAVA for administrators

This module is a primer introducing how to run and grow a LAVA lab. It introduces basic administrative features for LAVA, including how to install LAVA and how to increase its scope by adding new users, workers and test devices.

- LAVA lab layouts
- Components and services
- Installing and updating LAVA
- Enabling SSL
- Adding users
- Adding workers
- Adding devices
- LAVA state machines
- Closing the CI loop

Energy Aware Scheduler

Energy Aware Scheduler provides a detailed introduction to how Energy Aware Scheduling works and how to develop energy models of your system to ensure that both the scheduler and the thermal manager make the best decisions possible. We will also look at tools for debugging and tuning scheduler decisions

Energy Aware Scheduler is a four module course. It has equivalent to approximately two days face-to-face training although the exact time to deliver varies depending on available equipment for lab exercises. It is suited to both face to face and remote delivery.

Trainees are expected to have experience of system level debug of Linux systems and some background in power management. Trainees will learn about how to deploy and tune EAS into embedded Linux systems (including Android).

EAS-01: Introduction for energy aware scheduling

This module is a primer introducing the goals and implementation of the energy aware scheduler.

- Energy Scheduling primer
- Task Size Estimation
    - CPU Capacity
    - Per-entity Load Tracking
    - Choosing the right OPP
- Power Estimation
    - Ways to Model Energy
    - Linux Energy Model subsystem
- Putting it together
- Lab/homework
    - Enable CPUFreq driver
    - Enable CPU capacity and energy model
    - Run workload with scheduler statistics

EAS-02: Capacity and power modeling

In this module we will look at how to characterize the behavior of your system in order to generate an energy model for it. The energy aware scheduler, as well as other kernel features for thermal tuning, rely on models that can predict the energy costs of operating the device in different modes. Models based on measurable real-world units have largely replaced the complex heuristics that dominated early attempts to schedule big.LITTLE systems. This module helps trainees calculate a suitable model using the same real-world units.

- Modeling Plan
    - Options
    - Objectives
    - Measuring
- Model building
    - Controlling Data

    - Testing Modeling Assumptions
    - Fitting the Model
- Lab/homework
    - Build your own model

EAS-03: Scheduler tuning

The energy aware scheduler has been partnered with multiple different tuning approaches since it first appeared in the kernel. In this module we tour this accumulated history showing how different branches of EAS have been tuned.

- Accumulated history of EAS tuning
    - SchedFreq replaced by schedutil
    - SchedTune
    - UTIL_EST
    - UTIL_EST_FASTUP
    - uclamp
    - Cleanup of UTIL_EST_FASTUP
- Scheduler tuning interfaces
- Android PowerHAL integration
- Lab/homework
    - Per-task uclamps
    - Applying uclamps globally
    - Applying uclamps using cgroups

EAS-04: Analysis of Scheduling Behavior

This module is an introduction to tooling that is commonly used to study, tune and debug scheduler behavior. Initially we cover tools that can generate "interesting" use-cases, either synthetically or by automatically deploying real-world programs to the device-under-test. This is followed up by introducing a number of different tools that can be used to monitor or visualize scheduler behavior.

- Workload Generation
    - rt-app
    - Arm Workload Automation
- Analysis Tools
    - Perfetto
    - sched-analyzer
    - Idlestat
    - Tracepoints
    - LISA
- Lab/homework
    - Tracing C-states
    - LISA

KVM and Virtual I/O for 64-bit Arm Systems

This course is an introduction to the implementation of KVM on Arm systems and how virtio and VFIO can be used to provide accelerated access to host devices from within guest systems.

KVM and Virtual I/O for 64-bit Arm systems is a two module course, equivalent to approximately a day face-to-face training. It is suited to both face to face and remote delivery.

This course does not require any specific prior experience but a general understanding of how to use virtualization or hardware emulation is useful. Trainees will learn the internal implementation of KVM and device access.

KVM-01: KVM for Armv8

In this module will discuss the foundation hardware features that KVM is built on before giving an more detailed overview of the implementation, without and with Virtual Host Extensions (VHE).

- ARMv8 virtualization extension
- KVM software stack
- ARMv8 KVM implementation (No-VHE)
    - General working flow
    - Initialization
    - Context management
    - Memory management
    - Exception handling
    - Interrupt controller
    - Timer
- ARMv8 KVM implementation (VHE)
    - What's VHE and why?
    - Initialization and working flow
    - Exception handling

KVM-02: Device access using virtio and VFIO

A hypervisor can present simulated hardware to its guests but simulating real hardware is slow and, potentially, error prone. Practical virtual systems often require more efficient (and secure) access to the underlying hardware resources. This module introduces two different but complementary approaches to providing virtual machines with more direct access to hardware resources.

- Virtualization drivers programming models
- Virtio
    - Virtio implementation
    - Story for enabling virtio network device
- VFIO
    - VFIO brief introduction
    - VFIO PCI device driver programming
    - Stories for deployment VFIO on Arm platforms

Trusted Firmware for A-profile Arm systems

This is an introductory course helping developers learn about Trusted firmware-A and how it can be used to implement early stage bootloaders, PSCI and secure world switching on Armv8 and Armv9 systems.

Trusted Firmware for A-profile Arm systems is a four module course equivalent to approximately two days of face-to-face training and is suited to both face to face and remote delivery.

Trainees are expected to have experience of using C for low-level programming such as OS or bootloader development. Trainees will learn about the role of the Trusted Firmware within the system, why this is useful for Armv8 and Armv9 systems, They will also learn how to integrate the reference bootloaders into existing and future systems.

TFA-01: Introduction to Trusted Firmware-A

This module is a primer on the role of Trusted Firmware-A (TF-A) with a modern Arm system. This includes sections on the underlying hardware features that motivate it and the features TF-A provides as part of the overall system architecture. This is complimented by a summary of how TF-A communicates with other system components.

- Arm A-Profile Architecture evolution
  - From Armv7-A to Armv8-A
  - From Armv8-A to Armv9-A
- About Trusted Firmware-A project
  - History and origins
- Trusted Firmware-A as EL3 firmware
  - Firmware components
- Handling Secure Monitor Call
- Context management
- Power State Coordination Interface (PSCI)
- Lab/homework
  - Build and boot TF-A
  - Examining PSCI flow using a debugger

TFA-02: Trusted Firmware-A reference bootloaders

Trusted Firmware-A includes not only a reference implementation for the EL3 firmware on 64-bit Arm systems, it also includes a selection of reference bootloaders that can be used to ensure TF-A and other security critical components are signature checked and loaded before control is passed to the rich OS. This module is a comprehensive introduction to those bootloaders and how to use them.

- Boot flows
  - Bootloaders image terminology
- Image organization
  - Firmware Image Package (FIP)
- Console API framework

  - Log levels
  - Crash reporting
- IO storage abstraction layer
- BL2 image parameter passing
- Locking primitives
- Device tree
  - Firmware Configuration Framework
- Lab/homework
  - Examining TF-A boot flow using a debugger
  - Build and analyze a FIP

TFA-03: Firmware Security

This module studies the security specific features of Trusted Firmware-A in greater depth. This includes how the threat model has been developed and how the bootloaders provide an implementation of Arm security standards. The module also covers how TF-A can exploit a TPM to provide measured boot together with a summary of the TF-A features to update and encrypt firmware content.

- Generic threat model
- Trusted Board Boot
  - Chain of Trust
  - Authentication Framework
- Measured boot
- Firmware update
- Firmware encryption
- Lab/homework
  - Enable Trusted Board Boot
  - Enable firmware encryption

TFA-04: Secure/Realm world interfaces

This module looks in more detail and the services Trusted Firmware-A provides to other system components. This covers the interfaces it presents to the rich OS and hypervisor, sometimes called Normal World, as well as the interfaces offered to the Secure World in typical Armv8 systems or by the Realm World in the more complex Armv9 designs.

- EL3 runtime interface (firmware)
  - Arm architecture services
  - Standard services
  - SiP/OEM specific services
- EL3 runtime interface (Secure world)
  - Secure-EL1 payload dispatcher
  - Interrupt handling
  - FF-A standard protocol
  - Secure Partition Manager (SPM)

- EL3 runtime interface (Realm world)
  - Realm Management Monitor (RMM)
  - Granule Protection Tables Library
- Lab/homework
  - Write your own EL3 runtime service
  - Enable Test Secure Payload (TSP)
  - OP-TEE dispatcher flow

Introduction to OP-TEE

Introduction to OP-TEE is a comprehensive introduction to OP-TEE and to trusted application development. The course includes guides to important OP-TEE tasks such as building, porting and debugging.

This course is a four module course equivalent to approximately two days face-to-face training and is suited to both face to face and remote delivery.

Trainees must be comfortable using command line tools for software development. Trainees will learn how to port OP-TEE to new platforms, how to deploy trusted applications using OP-TEE and be confident using the debug tooling to troubleshoot.

TEE-01: Introduction to OP-TEE

OP-TEE is a Trusted Execution Environment (TEE) for Arm systems built to architecture neutral specifications from GlobalPlatform. This module is a primer that introduces TEEs and how Arm TrustZone can be used to create secure implementations. It also a primer on practical elements related to OP-TEE including how to build, install and update it.

- Introduction to TEE
  - Generic TEE architecture
  - Overview of ARM TrustZone
  - ARM TrustZone based TEE
- About OP-TEE
  - Open Portable TEE (OP-TEE)
  - History and origins
- Build OP-TEE
  - OP-TEE gits
  - OP-TEE build environment
- Lab/homework
  - OP-TEE hands-on
  - Build OP-TEE from scratch

TEE-02: OP-TEE concepts and TA development

This module is an application developer’s guide to OP-TEE. It introduces the APIs and build system support that allows you to implement your own Trusted Applications running in OP-TEE as well as how to communicate with it from your application running in the Rich OS.

- OP-TEE main concepts
  - Architecture
  - Trusted Applications
  - Shared memory
  - Crypto layer
  - Secure storage

- Build TA from scratch and run
  - Write a TA from scratch
  - Build and sign a TA
- Lab/homework
  - Build and test hello world TA
  - Write key generation TA
  - Write key storage TA
  - Write data signing TA

TEE-03: OP-TEE porting and interfaces

In contrast to TEE-02, this module is a system integration engineer's guide to OP-TEE. It includes details on how to port OP-TEE to new platforms and to configure it appropriately. We also look at how OP-TEE and some of its built-in Trusted Applications are used to provide cryptographic services to the rich OS.

- OP-TEE porting
  - Add a new platform
  - Porting guidelines
- OP-TEE interfaces
  - GP TEE user-space clients
  - PKCS#11 user-space clients
  - Linux Kernel interface
  - SMC interface
  - Kernel clients (TEE bus)
  - Boot-loader clients
- Lab/homework
  - Review existing OP-TEE platform ports
  - Explore Linux kernel interface

TEE-04: OP-TEE advanced concepts and debug

This module introduces several advanced features of OP-TEE, including how to write Trusted Applications with deeper access to OP-TEE allowing them to implement features that are not possible when using but the standardized GlobalPlatform APIs. As part of this trainees will also be introduced to debugging techniques to allow find and fix problems when they occur.

- OP-TEE advanced concepts
  - Pseudo TAs
  - TA loading
  - Interrupt handling
  - Thread handling
  - MMU
  - Pager
  - Devicetree
  - Virtualization
  - Secure boot

- Debugging under OP-TEE
  - GlobalPlatform return code origins
  - OP-TEE log levels
  - Abort dumps / call stack
  - ftrace for Linux TEE driver
  - ftrace in OP-TEE
  - Profiling using gprof
  - Benchmark framework
  - GDB using QEMU
- Lab/homework
  - Debug TA crash

Single module boosters

These single module boosters can be used to enrich our training courses with additional content that is of particular benefit to you and your teams.

A64-01: Reading (and writing) A64 assembler

Reading (and writing) A64 assembler is a two hour primer in the basics of the A64 instruction set with a focus on learning to read basic ALU, load/store and branch instructions allowing trainees to see the structure of assembler programs, especially those produced by the compiler. The module does not cover every mnemonic because reference manuals are a better way to do that. Instead this course introduces the programmer's models and the fundamental "look and feel" of the instruction set allowing trainees both to confidently debug low-level code and examine the compiler output of critical functions looking for optimization opportunities.

- Overview
    - Register model
    - Procedure call standard
- Integer processing instructions
- Loads and stores
    - Data width
    - Addressing modes
- Branches and condition flags
    - A64 condition flags and condition codes
    - Branches
    - Conditional select
- Floating point and SIMD
    - Loads and stores
    - NEON fundamentals
    - Floating point compare
- Worked examples
    - DF-II biquad filters (floating point)
    - 16-tap FIR filters (NEON)
- Wrap up
- Lab/homework
    - Revisiting memcpy()
    - Revisiting biquad_step()
    - C library functions

LWF-01: WiFi - Linux implementation and debug

WiFi - Linux implementation and debug is an introduction to the Linux WiFi stack designed to orient developers and give them a head start in debugging. Both FullMAC and SoftMAC devices will be covered but there is a particular focus on understanding the difference between legacy and upstream SoftMAC implementations.

This is a single module course consisting of approximately 90 minutes of lecture format material.

Trainees must have existing kernel development experience. Trainees will learn about the structure of the Linux WiFi stack and what techniques are most commonly used to find and fix problems.

- WiFi/IEEE 802.11 basics
- IEEE 802.11 software stack
  - Stack overview
  - cfg80211 and mac80211
  - FullMAC and SoftMAC drivers
  - Userspace WiFi management
- Debugging tools and techniques
  - Standard debugging techniques
  - iw tool
  - Packet capture
  - Hybrid techniques
- Case studies
  - Unexpected disconnection during voice call
  - Unable to scan hidden access points

Linaro Training Catalogue 2024

training@linaro.org