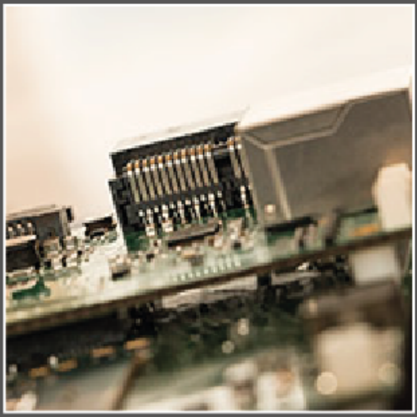
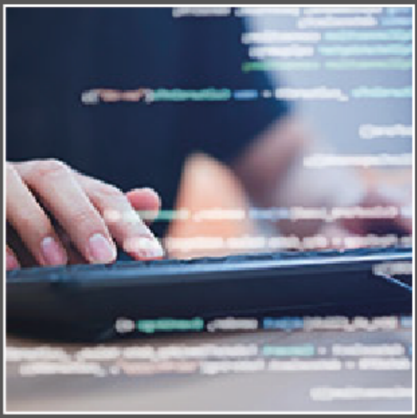
Linaro Developer Services
Training Catalogue
February 2023





Linaro
Developer Services

Training at Linaro

Since 2010, Linaro has helped to drive open source software development on Arm. Linaro Developer Services has expert level knowledge of the open source technologies that you rely on combined with a deep understanding of the communities that build it. Linaro hands-on-training shares that expertise to help you leverage Linux and Arm technology.

Linaro Developer Services provides off-the-shelf or customised 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.

Our courses are flexible, can be delivered on-site or remotely and incorporate many emerging technologies, together with the latest best practices.

On-site or remote training

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

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

Note: *Like many other companies, Linaro has imposed travel restrictions due to the coronavirus pandemic. Our on-site training offer is significantly impacted by this and is subject to constraints based on the locations our trainers call home.*

Remote events use the same training materials as our on-site events, however our remote events provide the option to adopt a much slower delivery cadence. A course that takes four days of intense face-to-face contact can instead be delivered as eight two and a half hour sessions over a period of four or even eight weeks. This often fits better into trainee's busy schedules. Additionally, by giving trainees longer to get to know us, we can develop some of the social relationships that are more difficult to form when using web conferencing tools.

For remote events, the lab exercises are set as homework. To ensure trainees are successful with the lab session we provide remote support via e-mail to provide any help and support that is required. We also encourage trainees to use the same support e-mail service if they have any specific questions in order to provide the same level of access to trainers that we provide with face-to-face training..

Modular training

Almost all Linaro training courses are based on a modular structure. Each module represents approximately half-day of face-to-face training or one remote session.

This catalogue introduces our training material as complete stand-alone courses. Most of these courses are composed of several modules. This modular structure allows us to offer you a selection of coherent off-the-shelf courses but also allows us to offer customized courses by combining modules in different ways to meet your specific requirements.

To tailor the material we can borrow modules from other courses, we can include modules from our library of booster modules or, if needed, we can work with you to define fully custom training modules to help you meet your specific needs.

Your hardware, your choice!

There are other ways we can customize training to make your trainees more productive. For example our lab sessions typically consist of exercises to help trainees consolidate their learning.

Our off-the-shelf options include exercises that can be completed using QEMU-based virtual machines. This is a great default and QEMU is especially well suited for remote delivery because it provides a consistent experience for all trainees and is easy to provide support for.

However, it can be useful to deliver training that uses your hardware for as many exercises as possible tailoring. allows us to deliver some or all of these lab sessions using the real hardware devices that your trainees will use everyday.

Contact us

We hope this catalogue is useful to show how Linaro training can work for you. If you want to discuss training events for you and your teams please contact training@linaro.org and we'll be happy to discuss further.

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.

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: Devicetree, YAML and json-schema
- Lab/homework
  - Qemu devicetree experiments

Pragmatic Linux driver development - Part I

- First steps
  - Changing the kernel configuration

    - Your first kernel module
    - Exploring the kernel via character drivers
    - Locking primitives
- Why you shouldn't write a character driver!
- Updating the device tree
- Linux device driver model
- Copying and adapting an existing driver
- Lab/homework
    - Write hello world kernel module
    - The membuf misc driver experiments

Pragmatic Linux driver development - Part II

- Parsing device properties
- Handling hardware
    - Memory mapped I/O
    - Regmap
    - Interrupt handling
- Memory allocation
- Time, timers and housekeeping
- Case study: Industrial I/O
- Portable drivers and compile testing
- Lab/homework
    - I2C device/driver experiments
    - Communicating with temperature sensor
    - Explore hwmon subsystem

Symbolic debugging for Linux kernel and userspace

Learn about using traditional stop-the-world debuggers on Linux systems.

- Fundamentals
    - Kernel features for multi-process debugging
    - Compiler options for debugging
    - Debugging with gdb
    - Cross-debugging with gdbremote
- Case study: IDE integration using Eclipse
- 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 only covers the "social" aspects of upstream kernel development. It is best suited to trainees who 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.

Part I: Mechanics

- 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

Part II: Tips, tools and techniques

- 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.

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
- Arm memory map
- Kernel stack organisation
- Using automatic tools to fail early
- Case studies
    - "I can't reproduce but my customer can"
    - "My XYZ missed its deadline"
    - "My board just stopped dead"
    - "I'm sure this used to work"
- Use case: "My board just randomly failed"
- Lab/homework
    - initcall_debug experiments
    - DYNAMIC_DEBUG experiments
    - Explore DebugFS

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

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

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 behaviour 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 behaviour of real-time Linux systems.

Part I: Managing real time activity

- 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

Part II: Real time implementation and analysis

- 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 behaviours: 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 four module course equivalent to approximately two 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.

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

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 organisation
  - Crates, cargo and crates.io
  - Building with Cargo
- Idiomatic Rust
  - Introducing traits
  - Example: wc
  - Handling errors and panics in Rust

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?

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

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 FFI interfacing. 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
- Rust4Linux
    - Minimal example
    - Case study: PL061 driver
    - Kernel flavoured Rust

A Practical Introduction to OpenEmbedded/Yocto

This course is an introduction to working with typical OpenEmbedded distributions. It can be used as a quick introduction to working with OpenEmbedded systems or it can act as a primer for Building Custom Systems with OpenEmbedded/Yocto to support trainees without prior OpenEmbedded experience.

A Practical Introduction to OpenEmbedded/Yocto is a one day course when delivered face-to-face. It can also be delivered in remote format as a two 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). Trainees will learn how to build existing OpenEmbedded images and augment them with a custom kernel. They will also learn about the tools used to write custom applications.

Getting started

- Your first build
- Describing what happened during your first build

Anatomy of a typical Linux distribution

- Bootloader
- Kernel
- Init system and device manager
- Libraries, Applications and Services
- Package management

Kicking the tires

- Installing and booting
- Filesystem Hierarchy Standard
- Adding packages

Developing applications

- Building an SDK
- Cross-compiling your own applications

Updating the Linux kernel

- Configuring the kernel build
- Devicetree
- The right way to copy

Building Custom Systems with OpenEmbedded/Yocto

Building Custom Systems with OpenEmbedded/Yocto is a complete introduction to developing and maintaining Linux distributions using OpenEmbedded tools and Yocto Project releases.

Building Custom Systems with OpenEmbedded/Yocto is a two and a half day course when delivered face-to-face. It can also be delivered in remote format as a six module course.

Trainees are expected either to have some existing experience that they would like to enhance or to have taken A Practical Introduction to OpenEmbedded/Yocto *as a primer. Trainees will learn how to package software and integrate it into existing OpenEmbedded distributions and how to use the OpenEmbedded tools to create complete turn-key embedded Linux systems.*

Introduction to OpenEmbedded and the Yocto Project

- Introduction to (embedded) Linux distros
- Introduction and history of OpenEmbedded and the Yocto Project
- Linaro OpenEmbedded releases
- Lab/homework
    - Building OpenEmbedded/Yocto systems

OpenEmbedded main concepts

- Metadata
- Build environment
- Recipes, packages, dependencies
- Configuration files
- Bitbake
- Lab/homework
    - Learn how to write recipes

Build workflow

- Anatomy of the build folder
- Build workflow

OpenEmbedded advanced concepts

- Virtual packages
- Classes

- Layers, machines and distros configuration, images
- Packageconfig
- Application development and SDK
- Lab/homework
    - Learn how to use layers

Debug the build

- Buildhistory
- How to modify source code locally and externalsrc
- Run scripts and log files
- Cleaning
- Runtime debug

Toolchain

- OpenEmbedded toolchain
- Using Linaro toolchain
- Using binary toolchain
- Support for LLVM/clang
- Lab/homework
    - Build a customized turn-key GNU/Linux image

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.

LAVA for users - Part I

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

LAVA for users - Part II

- 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

LAVA for administrators

- 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).

Introduction for energy aware scheduling

- Review power management evolution for big.LITTLE
- Basic concept for PELT and signals used by EAS
- Discuss detailed implementation for EAS
- Lab/homework
    - Enable CPUFreq driver
    - Enable CPU capacity and energy model
    - Run workload with scheduler statistics

Practical Power modeling

- Background for power modeling
- Review CPU power state
- Measurement method on Hikey
- Generate power model for EAS
- Generate power model for IPA: simplified method and normal method
- Lab/homework
    - CPU capacity modeling
    - Generating CPU and cluster power data
    - CPU power modeling

Tools and Techniques

- Brief introduce for tools
- Examples driven to breakdown detailed scheduler signals
- Lab/homework
    - Enable EAS trace points
    - Enabling LISA on QEMU

SchedTune and CPUFreq

- Introduce SchedTune implementation and pragmatic usage
- Review how SchedTune co-operate with CPUFreq governor
- Lab/homework
  - Using LISA to observe UTIL_EST
  - Using uclamps for task
  - Using Cgroup controller

KVM and Virtual I/O for Armv8 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 Armv8 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 for Armv8

- 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

Device access using virtio and VFIO

- 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 Armv8-A systems

This is an introductory course helping developers quickly learn about Trusted firmware A and how it can be used to implement bootloaders, PSCI and secure world switching on 64-bit Armv8 systems.

Trusted firmware A is a two module course equivalent to approximately a day 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 systems and how to integrate the reference bootloaders into existing and future systems.

ARMv8 exception model and boot

- ARMv8 exception model
  - Review development of ARMv6, ARMv7 and ARMv8 exception models
- Bootloaders in ARM trusted firmware
- Trusted boot
- Firmware update
- Crash reporting
- Lab/homework
  - Explore Trusted Firmware-A boot flow

Secure monitor and power management

- Handling Secure Monitor Call
- Context management
- Interrupt handling
- Secure payload dispatch
- Power State Coordination Interface
- Lab/homework
  - Debugging PSCI 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.

Introduction to OP-TEE

- 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

OP-TEE concepts and TA development

- 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

OP-TEE porting and interfaces

- 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

OP-TEE advanced concepts and debug

- 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.

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

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