سرفصل آموزشی

پک پیشرفته متخصص اینترنت اشیا

فهرست سرفصل‌های دوره‌های آموزشی

سرفصل‌های دوره آموزشی

YoctoProject

& OpenEmbedded

Introduction to embedded Linux build systems

- Overview of an embedded Linux system architecture
- Methods to build a root filesystem image
- Usefulness of build systems

Overview of the Yocto Project and the Demo - First Yocto Project build

- Organization of the project source tree
- Building a root filesystem image using the Yocto Project
- Downloading the Poky reference build system
- Building a system image

Using Yocto Project – basics

- Organization of the build output
- Flashing and installing the system image

Flashing and booting

- Flashing and booting the image on the board

Using Yocto Project - advanced usage Demo - Using NFS and configuring the build

- Configuring the build system
- Customizing the package selection
- Configuring the board to boot over NFS
- Learn how to use the PREFERRED_PROVIDER mechanism

Writing recipes - basics Demo - Adding an application to the build

- Writing a minimal recipe
- Adding dependencies
- Development workflow with bitbake
- Writing a recipe for nInvaders
- Adding nInvaders to the final image

Writing recipes - advanced features

- Extending and overriding recipes
- Adding steps to the build process
- Learn about classes
- Analysis of examples
- Logging
- Debugging dependencies

Layers Demo - Writing a layer

- What layers are
- Where to find layers
- Creating a layer
- Learn how to write a layer
- Add the layer to the build
- Move nInvaders to the new layer

Writing a BSP Demo - Implementing the kernel changes

- Extending an existing BSP
- Adding a new machine Bootloaders
- Linux and the linux-yocto recipe
- Adding a custom image type
- Extend the kernel recipe to add the MPU6050 accelerometer/gyro driver
- Configure the kernel to compile the MPU6050 accelerometer/gyro driver
- Play nInvaders

Creating a custom image Demo - Creating a custom image

- Writing an image recipe
- Adding users/groups
- Adding custom configuration
- Writing and using package groups recipes
- Writing a custom image recipe
- Adding nInvaders to the custom image

Creating and using an SDK Demo - Experimenting with the SDK

- Understanding the purpose of an SDK for the application developer
- Building an SDK for the custom image
- Building an SDK
- Using the Yocto Project SDK

Questions and Answers

- Questions and answers with the audience about the course topics
- Extra presentations if time is left, according what most participants are interested in

سرفصل‌های دوره آموزشی

Embedded Device Driver

X86 Systems

Make yourself into a Linux kernel specialist, who can

- Configure, compile, and install a Linux kernel
- Do the same for a kernel module
- Navigate and read the Linux kernel sources
- Use the API for internal kernel services
- Design and implement a kernel module
- Modify, or design and implement a device driver
- Measure the performance of your implementation

Introduction to the Linux kernel

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

Kernel sources

- Specifics of Linux kernel development
- Coding standards
- Retrieving Linux kernel sources
- Tour of the Linux kernel sources
- Kernel source code browsers: cscope, Kscope, Elixir
- Making searches in the Linux kernel sources: looking for C definitions, for definitions of kernel configuration parameters, and for other kinds of information
- Using the UNIX command line and then kernel source code browsers

Kernel configuration, compiling and booting on NFS

- Kernel configuration and compilation
- Generated files
- Booting the kernel. Kernel booting parameters
- Mounting a root filesystem on NFS
- Using the qemu software
- Configuring, compiling and booting a Linux
- kernel with NFS boot support

Linux kernel modules writing modules in action

- Linux device drivers
- A simple module
- Programming constraints
- Loading, unloading modules
- Module dependencies
- Kernel symbol table
- Cleanup function

- Adding sources to the kernel tree
- Write a kernel module with several capabilities
- Access kernel internals from your module
- Set up the environment to compile it

Data Types in the Kernel

- Standard C type
- Interface Specific Types
- Linked Lists

Linux device model

- Understand how the kernel is designed to support device drivers
- The device model
- Binding devices and drivers
- Platform devices, Device Tree
- Interface in user space: /sys
- Kobjects, Ksets, and Subsystems
- Low-Level Sysfs Operations
- Hotplug Event Generation
- Buses, Devices, and Drivers
- Classes
- Dealing with Firmware
- Kernel frameworks
- Block vs. Character devices
- Useful data structures
- File and inode Structure
- Char Device registration
- Interaction of user space applications with the kernel
- Details on character devices, file_operations, ioctl(), etc

- Read and write
- Exchanging data to/from user space
- The principle of kernel frameworks

Advanced Char Driver Operations

- Device Control
- Blocking I/O
- Sleeping
- Asynchronous Notification
- Access Control on a device file

Memory management I/O memory and ports

- Linux: memory management - Physical and
- Virtual (kernel and user) address space
- Linux memory management implementation
- Allocating with kmalloc()
- Allocating by pages
- Allocating with vmalloc()
- Caches
- Memory Pools
- Buffers
- I/O register and memory range registration
- I/O register and memory access
- Read / write memory barriers

The misc kernel subsystem

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

Time, Delays, Processes, scheduling, sleeping and interrupts sleeping and handling interrupts in a device driver in a real example

- Time measurement
- Process management in the Linux kernel
- Process Specific registers
- The Linux kernel scheduler and how processes sleep
- Interrupt handling in device drivers
- Interrupt handler registration and programming, scheduling deferred work.
- IRQ Number
- Interrupt Sharing
- Interrupt Driven I/O
- Kernel Timers
- Delaying Execution
- Tasklets
- Workqueues
- Adding read capability to the character driver developed earlier
- Register an interrupt handler
- Waiting for data to be available in the read() file operation
- Waking up the code when data is available from the device

Concurrency and Race Conditions Locking in action

- Issues with concurrent access to shared resources
- Locking primitives: mutexes, semaphores, spinlocks
- Atomic operations
- Typical locking issues
- Using the lock validator to identify the sources of locking problems
- Observe problems due to concurrent accesses to the device
- Add locking to the driver to fix these issues

USB Drivers

- USB Device Basics
- Writing a USB Driver
- probe and disconnect
- USB Transfers Without Urbs

Driver debugging techniques investigating kernel faults in action

- Debugging with printk
- Using Debugfs
- Analyzing a kernel oops
- Using kgdb, a kernel debugger
- Using the Magic SysRq commands
- Studying a broken driver
- Analyzing a kernel fault message and locating the problem in the source code

The Linux kernel development process

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

سرفصل‌های دوره آموزشی

Embedded Device Drivers

Introduction to the Linux kernel:

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

Kernel sources:

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

Kernel source code:

- Making searches in the Linux kernel sources: looking for C definitions, for definitions of kernel configuration parameters, and for other kinds of information.
- Using the Unix command line and then kernel source code browsers

Configuring, compiling and booting the Linux kernel:

- Kernel configuration
- Native compiling. Generated files.
- Booting the kernel
- Kernel booting parameters

NFS booting and cross-compiling:

- Booting on a directory on your GNU/Linux workstation, through NFS
- Kernel cross-compiling

Kernel configuration, cross-compiling and booting on NFS

- Using the ARM board
- Configuring, cross-compiling and booting a Linux kernel with NFS boot support

Linux kernel modules:

- Linux device drivers
- A simple module
- Programming constraints
- Loading, unloading modules
- Module parameters
- Module dependencies
- Adding sources to the kernel tree
- Generating patches to share them with others

Writing modules:

- Write a kernel module with several capabilities, including module parameters.
- Access kernel internals from your module
- Setup the environment to compile it

Memory management:

- Linux: memory management - Physical and virtual (kernel and user) address spaces
- Linux memory management implementation
- Allocating with kmalloc()
- Allocating by pages
- Allocating with vmalloc()

I/O memory and ports:

- I/O register and memory range registration
- I/O register and memory access
- Read / write memory barriers
- Make a remote connection to your board through ssh
- Access the system console through the network
- Reserve the I/O memory addresses used by the serial port
- Read device registers and write data to them, to send characters on the serial port

Character drivers:

- Device numbers
- Getting free device numbers
- Implementing file operations: read, write, open, close, ioctl...
- Exchanging data between kernel-space and user-space
- Character driver registration
- Using the ARM board
- Writing a simple character driver, to write data to the serial port
- On your workstation, checking that transmitted data is received correctly
- Exchanging data between userspace and kernel space
- Practicing with the character device driver API
- Using kernel standard error codes

Processes, scheduling, sleeping and interrupts:

- Process management in the Linux kernel
- The Linux kernel scheduler and how processes sleep

- Interrupt handling in device drivers: interrupt handler registration and programming
- Scheduling deferred work
- Adding read capability to the character driver developed earlier
- Register an interrupt handler
- Waiting for data to be available in the read file operation
- Waking up the code when data is available from the device

Driver debugging techniques:

- Debugging with printk
- proc and debugfs entries
- Analyzing a kernel oops
- Using kgdb, a kernel debugger
- Using the Magic SysRq commands
- Debugging through a JTAG probe
- SystemTap and demonstration

Investigating kernel faults:

- Using the ARM board
- Studying a broken driver
- Analyzing a kernel fault and locating the problem in the source code

Kernel boot-up details:

- Detailed description of the kernel boot-up process, from execution by the bootloader to the execution of the first userspace program
- Initcalls: how to register your own initialization routines

Working with the community:

- How to get help from the community
- Report bugs
- Generate and send patches
- Useful resources about the kernel

Managing kernel sources with git:

- Very useful to manage your changes to the Linux kernel (drivers, board support code), staying in sync with mainstream updates
- Cloning an existing git tree
- Creating your own branch with your own changes
- Generating patches against the reference tree
- Review of useful git commands
- Understanding the work flow used by kernel developers, through the study of typical scenarios
- Create your own git branch from the mainline tree
- Get changes from trees and generate your own patch-set
- Keep your branch updated with the changes in your reference tree