سرفصل آموزشی

پک حرفه ای مهندسی Embedded Linux

فهرست سرفصل‌های دوره‌های آموزشی

سرفصل‌های دوره آموزشی

Real-Time with PREEMPT-RT

Introduction to Real-Time behavior and determinism

- Definition of a Real-Time Operating System
- Specificities of multi-task systems
- Common locking and prioritizing patterns
- Overview of existing Real-Time Operating Systems
- Approaches to bring Real-Time capabilities to Linux

The PREEMPT_RT patch Demo & Building a mainline Linux Kernel with

- The PREEMPT_RT patch
- History and future of the PREEMPT_RT patch
- Real-Time improvements from PRE-EMPT_RT in mainline Linux
- The internals of PREEMPT_RT
- Interrupt handling: threaded interrupts, softirqs
- Locking primitives: mutexes and spinlocks, sleeping spinlocks
- Preemption models
- Downloading the Linux Kernel, and apply-ing the patch
- Configuring the Kernel
- Booting the Kernel on the target hardware

Hardware configuration and limitations for Real-Time

- Interrupts and deep firmware's
- Interaction with power management features: CPU frequency scaling and sleep states
- DMA

Tools: Benchmarking, Stressing and Analyzing

- Benchmarking with cyclictest
- System stressing with stress-ng and hack-bench
- The Linux Kernel tracing infrastructure
- Latency and scheduling analysis with ftrace, kernelshark or LTTng
- Usage of benchmarking and stress tools
- Common benchmarking techniques
- Benchmarking and configuring the hard-ware platform

Kernel infrastructures and configuration

- Good practices when writing Linux kernel drivers
- Scheduling policies and priorities: SCHED_FIFO, SCHED_RR, SCHED_DEADLINE
- CPU and IRQ Affinity
- Memory management
- CPU isolation with isolcpus

Real-Time Applications programming patterns

- Debugging a demo application
- POSIX real-time API
- Thread management and configuration
- Memory management: memory allocation and memory locking, stack
- Locking patterns: mutexes, priority inheritance
- Inter-Process Communication
- Signaling
- Make a demo userspace application deterministic
- Use the tracing infrastructure to identify the cause of a latency
- Learn how to use the POSIX API to manage threads, locking and memory
- Learn how to use the CPU affinities and configure the scheduling policy

Hard real-time solutions Demo - Xenomai latency tests

- Xenomai, a hard real-time solution for Linux: features, concepts, implementation and examples.
- Setting up Xenomai.
- Latency tests with Xenomai.
- Comparing the results with PREEMPT_RT

سرفصل‌های دوره آموزشی

Embedded Linux

Boot-Time Optimization

Principles

- How to measure boot time
- Main ideas
- Downloading bootloader, kernel and Build-root source code
- Board setup, setting up serial communication
- Configure Buildroot and build the system
- Configure and build the U-Boot bootloader.
- Prepare an SD card and boot the bootloader from it
- Configure and build the kernel. Boot the system

Measuring time

- Generic software techniques
- Hardware techniques
- Specific solutions for each stage
- Modify the system to measure time at various steps
- Timing messages on the serial console
- Timing the launching of the application

Toolchain optimizations

- Introduction to toolchains
- C libraries
- Size information
- Measuring executable performance with time
- Using strace and ltrace
- Other profiling techniques
- Finding unnecessary configuration options in applications
- Modifying configuration options through Buildroot
- Experiments with strace to trace program execution

Application optimization

- Using strace and ltrace
- Other profiling techniques
- Finding unnecessary configuration options in applications
- Modifying configuration options through Buildroot
- Experiments with strace to trace program execution

Optimizing system initialization

- Using BusyBox bootchartd
- Optimizing init scripts
- Possibility to start your application directly
- Using Buildroot to remove unnecessary scripts and commands
- Access-time based technique to identify unused files
- Simplifying BusyBox
- Starting the application as the init program

Filesystem optimizations

- Available filesystems, performance and boot time aspects
- Making UBIFS faster
- Tweaks for reducing boot time
- Booting on an initramfs
- Using static executables: licensing constraints
- Trying and measuring two block filesystems: ext4 and SquashFS
- Trying and measuring the initramfs solution. Constraints due to this solution

Kernel optimizations

- Using Initcall debug to generate a boot graph
- Compression and size features
- Reducing or suppressing console output
- Multiple tweaks to reduce boot time
- Generating and analyzing a boot graph for the kernel
- Find and eliminate unnecessary kernel features
- Find the best kernel compression solution for our system

Bootloader optimizations

- Generic tips for reducing U-Boot's size and boot time
- Optimizing U-Boot scripts and kernel loading
- Skipping the bootloader - How to modify U-Boot to enable its Falcon mode
- The Device Tree preparation work that U-Boot does to prepare Linux booting
- Using the spl export command to do this work in advance
- Modifying U-Boot's source code and configuring it for directly booting Linux and

Skipping the U-Boot second stage.

- Example instructions and setups for booting from MMC and NAND flash
- How to debug Falcon mode
- How to fall back to U-Boot
- Limitations

Bootloader optimizations

- Using the above techniques to make the bootloader as quick as possible
- Switching to faster storage
- Configuring U-Boot for Falcon mode booting, skipping U-Boot's second stage

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