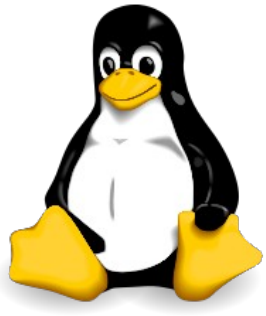
V2019.03

A.L.S.E
Advanced **L**ogic **S**ynthesis for **E**lectronics
FPGA & HDL Experts
http://www.FPGA.fr



Embedded Linux Kernel & Driver development

Formation adaptée aux SoC-FPGAs

Construire des applications complètes autour des Processeurs d'Application multi-cœurs ARM-A9-MP devient très vite un **challenge majeur**. L'utilisation d'une distribution **Linux Embarqué** est pratiquement incontournable, mais construire, déployer, maîtriser l'*Operating System* et prendre en compte les spécificités de l'application (Device Tree : périphériques HPS et périphériques FPGA custom) exige une compréhension profonde du **noyau Linux** ainsi que de savoir précisément comment concevoir, coder, mettre au point et déployer des ***Device Drivers* custom**.

Dans un service ou une équipe de développement, il n'est pas nécessaire que *tous* les ingénieurs maîtrisent les connaissances enseignées dans cette formation, mais il est **indispensable** qu'il y en ait **au moins un** !

Au moins UN Ingénieur logiciel
d'une équipe de conception Soc-FPGA
doit maîtriser ces connaissances !

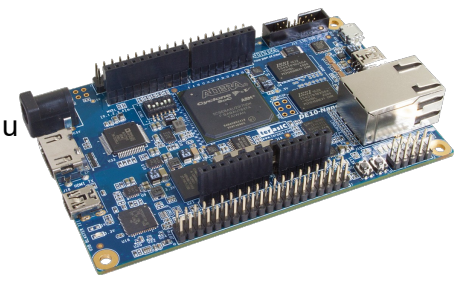
Pour offrir le niveau de qualité auquel nos clients sont habitués, nous nous sommes appuyés sur un contenu reconnu par l'industrie. Nous avons ensuite développé et adapté ce contenu théorique pour coller au mieux aux besoins des utilisateurs de SoC-FPGAs Intel, et tous nos exercices se font sur une plateforme Intel Soc-FPGA **ARM-A9-MP**.

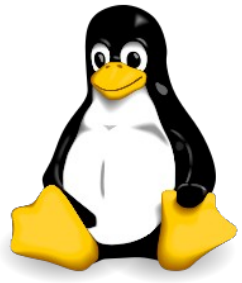
Le succès des formations ALSE est en partie lié à la qualité des très **nombreux exercices pratiques** (qui représentent environ la moitié du temps). Ils mettent en pratique la théorie pas-à-pas et progressivement, sur un **kit SoC-FPGA** largement disponible et peu coûteux. Ils serviront aussi de modèles directement réutilisables dans les projets réels.

Exercices pratiques sur Kit SoC-FPGA abordable !

Nous avons porté tous les exercices sur un Kit SoC-FPGA abordable (DE10-Nano, incluant un Cyclone V SoC -voir photo à droite-).

Bien entendu, **ce que vous apprendrez est indépendant de la cible** mais vous pourrez facilement reproduire les exercices après la formation en faisant l'acquisition de ce kit qui peut également servir de modèle hardware u même de prototype pour vos propres projets.





Embedded Linux Kernel and driver development

Formation de 5 jours

A.L.S.E

Objectifs

Cette formation permet (entre autres) de répondre aux besoins suivants :

- ✔ Comprendre le Noyau (Kernel) Linux
- ✔ Configurer, compiler et debugger le Noyau Linux
- ✔ Porter le Noyau Linux sur une autre cible ou sur une autre architecture
- ✔ Concevoir des Linux Device Drivers pour périphériques embarqués ou custom
- ✔ Utiliser les drivers pour contrôler depuis Linux les périphériques HPS et FPGA
- ✔ Transférer des données entre côtés Programmable & HPS
- ✔ Introduction à l'API DMA Linux, le DMA ARM et les solutions alternatives.

Prérequis

- **Connaissance et pratique de la programmation en langage C.**
- Connaissance et pratique des commandes Unix ou GNU/Linux. Les candidats manquant d'expérience sur ce sujet peuvent préalablement s'auto-former avec les supports disponibles gratuitement sur le Web.

Agenda détaillé

Voir pages suivantes

Langue

Anglais ou Français. En principe, les sessions publiques à Paris sont en Français.
Si le choix de la langue est important pour vous, vérifiez avec A.L.S.E en vous enregistrant.

Durée : **5 jours** (total)

Prix (session publique A.L.S.E Training Center / Paris France)

Comprend les supports de cours (Manuel + et livret d'Exercices), repas de midi et pauses.

Date / Lieu / Enregistrement

Prochaine session à Paris 13ème (A.L.S.E Training Center) : consultez le calendrier sur note site web.

Pour vous enregistrer :

contactez ALSE (info@alse-fr.com)
8 Passage Barrault – 75013 PARIS
Tel +33 (0)1 84 16 32 32

Advanced Logic Synthesis for Electronics
A.L.S.E. The FPGA Experts www.FPGA.fr

Embedded Linux Kernel and Driver development

Programme *

Jour 1

- **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, Kscope, Linux Cross Reference (LXR)

  Practical Exercise: 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 kernel source code browsers.

- **Configuring, compiling and booting the Linux kernel**

  Kernel configuration. Native and cross-compilation. Generated files.
  Booting the kernel. Kernel booting parameters. Booting the kernel using NFS.

  Practical Exercise (on the Altera DE10-Nano): Configuring, cross-compiling and booting a Linux kernel with NFS boot support.

Jour 2

- **Linux kernel modules**

  Linux device drivers. A simple module. Programming constraints. Loading, unloading modules
  Module dependencies. Adding sources to the kernel tree.

  Practical Exercise (on DE10-Nano):
  Write a kernel module with several capabilities. Access kernel internals from your module.
  Setup the environment to compile it.

- **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 userspace: /sys

  Practical Exercise (on DE10-Nano): Linux device model for an I²C driver.
  Implement a driver that registers as an I2C driver. Modify the Device Tree to list an I2C device.
  Get the driver called when the I2C device is enumerated at boot time.

---

* : Le contenu est fourni à titre indicatif et peut être modifié et/ou adapté lors du training.
** : Supports de cours en partie Copyright 2004-2017, Bootlin. Creative Commons BY-SA 3.0 license.

Jour 3

- **Introduction to the I²C API**

  The I²C subsystem of the kernel.
  Details about the API provided to kernel drivers to interact with I2C devices

- **Pin Muxing**

  Understand the *pinctrl* framework of the kernel.
  Understand how to configure the muxing of pins

- *Practical Exercise (on DE10-Nano):*
  Extend the I2C driver started in the previous lab to communicate with the accelerometer.

- **Kernel frameworks**

  Block vs. Character devices. Interaction of userspace applications with the kernel
  Details on character devices, file_operations, ioctl(), etc.
  Exchanging data to/from userspace. The principle of kernel frameworks

  Practical Exercise (on DE10-Nano):
  Extend the I²C driver to expose the accelerometer features to userspace applications, as an input device.
  Test the operation of the accelerometer using sample userspace applications

Jour 4

- **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.

  Practical Exercise (on DE10-Nano):
  Implement a minimal platform driver.
  Modify the Device Tree to instantiate the new serial port device (Jtag UART on FPGA side).
  Reserve the I/O memory addresses used by the serial port.
  Read device registers and write data to them, to send characters on the Jtag UART.

- **The misc kernel subsystem**

  What the misc kernel subsystem is useful for.
  API of the misc kernel subsystem, both the kernel side and the userspace side

  Practical Exercise (on DE10-Nano):
  Extend the driver started in the previous lab by registering it into the misc subsystem
  Implement serial port output functionality through the misc subsystem
  Test Jtag UART output from userspace.

- **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.

  Practical Exercise (on DE10-Nano):
  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.

Jour 5

- **Locking**

  Issues with concurrent access to resources. Locking primitives: mutexes, semaphores, spinlocks. Atomic operations. Typical locking issues.
  Using the lock validator to identify the sources of locking problems.

  Practical Exercise (on DE10-Nano):
  Observe problems due to concurrent accesses to the device.
  Add locking to the driver to fix these issues.

- **Direct Memory Access (DMA) principles, Linux API and mmap.**

  Overview. DMA integration and constraints. Coherent and streaming mappings. Mmap usage.
  Using devmem2 for raw memory R/W access (great for debugging purpose).

- **Driver debugging techniques**

  Debugging with printk. Debugfs entries. Analyzing a kernel oops.
  Using kgdb, a kernel debugger. Using the Magic SysRq commands
  Debugging through a JTAG probe

  Practical Exercise (on DE10-Nano):
  Studying a broken driver. Analyzing a kernel fault and locating the problem in the source code.

- **ARM board support and SoC support**

  Understand the organization of the ARM support code.
  Understand how the kernel can be ported to a new hardware board.

- **The Linux kernel development process** (Optional, if time available)

  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.

-=oOo=-