Embedded Linux

Device driver development

Sébastien Bilavarn
Outline

- Ch1 – Introduction to Linux
- Ch2 – Linux kernel overview
- Ch3 – Linux for Embedded Systems
- Ch4 – Embedded Linux distributions
- Ch5 – Case study: Xilinx PowerPC Linux
- Ch5 bis – Case study: Xilinx Zynq-7000 Linux
- Ch6 – Device driver development
Linux for Embedded Systems

- Introduction to Embedded Linux
  - Key features
  - Embedded Linux development
  - Linux kernel development
What is Embedded Linux?

Strictly speaking

- Embedded Linux is an operating system based on Linux and adapted specifically to the constraints of embedded systems.

- Unlike standard versions of Linux for Personal Computers, embedded Linux is designed for systems with limited resources
  - Memory: few RAM, sometimes no Memory Management Unit (MMU)
  - Storage: Flash instead of hard disk drive

- Since embedded systems are often designed for domain specific purposes and specific target platforms, they use kernel versions that are optimized for a given context of application.

- This results in different Linux kernel variants that are also called Linux distributions
Embedded Linux

What is embedded Linux?
- Embedded Linux is used in embedded systems such as:
  - mobile phones, personal digital assistants (PDA), media players
  - set-top boxes, and other consumer electronics devices
  - networking equipment
  - machine control, industrial automation
  - navigation equipment
  - medical instruments

Key features:
- runs in a small embedded system, typically booting out of Flash, no hard disk drive, no full-size video display, and take far less than 2 minutes to boot up, sometimes supports real-time tasks.
Embedded Linux

What is an embedded Linux system?
- Kernel, system (hardware), distribution
  - The kernel is the core of the system. Among others, it provides users and applications with an interface that allows to communicate with hardware or devices.
  - A specific Linux kernel is not required to create an Embedded Linux System. An Embedded Linux distribution is very similar to a standard Linux distribution, only a few drivers are different.
  - Embedded Linux distributions have been developed to better cope with specific hardware constraints of embedded systems or to better support a specific application domain.

Differences with standard Linux distributions
- Size optimisation regarding limited resources of embedded systems (e.g. reduced memory footprint)
- Support of specific debug tools
- Specific kernel configuration procedure and cross-development for embedded systems.
Embedded Linux

Benefits

- Free, source code fully available without restriction: no royalties
- Open, support of a large community of open source developers
- Different distributions can be used in order to match with the application or domain of application
- Stable
- Increasing number of software available and platform supported
- Multiprocessor support
- Real Time Extension (RTLinux, RTAI, XENOMAI)
Embedded Linux

- Market share
  - the largest OS used in embedded systems
Linux for Embedded Systems

- Introduction to Embedded Linux
  - Key features
- Embedded Linux development
  - Linux kernel development
Constraints of Embedded Systems

- Reduced storage resources
  - Embedded Systems usually have limited memory resources
  - Most Embedded Systems do not have a MMU
    - lower-cost, lower-power
  - No hard disk drive, use of Flash memory instead

- Less peripherals
  - hard disk drive, graphical display, etc.
Key features

- **Specific vs. general purpose**
  - An Embedded Linux distribution can be easily tuned to a target platform or an application domain by enabling / disabling kernel components, e.g.
    - Reducing number of drivers and applications
    - Kernel optimisation in terms of necessary services and functions

- **Other optimisations**
  - Drastic reduction of available commands, use of lightweight and free programs (BusyBox) instead of « desktop applications »
  - Drastic reduction of shared libraries, use of lightweight C libraries (glibc → uClibc)
  - Reduction and simplification of number of configuration files and directories like /etc
Key features

- Supports processors without MMU
  - Linux was originally designed on a processor with a Memory Management Unit.
  - μCLinux version of Linux was specifically designed to run on processors without a MMU.

- Improved Linux boot time
  - Usually less than 10s.

- Removal of swap system
  - Especially for Flash based file systems
Linux for Embedded Systems

- Introduction to Embedded Linux
- Key features
- Embedded Linux development
- Linux kernel development
Linux kernel

- The kernel is the core of any operating system.
  - It is responsible of the majority of the hardware support.
    → devices, input/output (e.g. network), threads and processes, memory, signals and system management of tasks.
  - Provides an abstraction of hardware and devices for user-level applications (APIs).
  - Consequence: it improves application portability on many architectures supported by the kernel.
  - Low level interfaces are specific to the hardware and devices.
  - Requires a file system to work properly.
Kernel development

- A Linux system is created by configuring and aggregating together appropriate system components
  - Choice of system components
    - Bus/interfaces, input/output, storage, network, system monitoring
    - Selection of kernel version, including software management of the necessary components (drivers).
  - Root Filesystem
    - Native file system including a set of minimal needed applications, librairies and files.
  - Cross-development environment
    - Host / target setup
    - Cross-compilation toolchain
    - Kernel booting utilities (boot loader)
  - Kernel configuration and compilation
    - Produces the kernel image with necessary drivers
Root Filesystem

Need for a Root Filesystem:
- The Linux kernel needs a file system to run properly.
- This file system is called the Root Filesystem because it is contained on the same partition on which the root directory is located.
- The Linux Root Filesystem contains the typical Linux directory structure:
  - /bin, /etc, /lib, /proc, /sys, /usr, /boot, /home, /opt, /sbin, /var, etc.
- The Linux Root Filesystem contains files and executables needed by the kernel.
  - E.g. commands, executables for system administration.
Root Filesystem

Need for a Root Filesystem:

- The Root Filesystem is mounted when the kernel boots.
  - On a desktop workstation installation, the kernel mounts a hard disk on the `/` directory. The following directories exist beneath the `/` directory:
    - `/bin` and `/sbin` contain system executables such as `init`, `ifconfig`, `mount`, `cd`, `mkdir`, and `ping`.
    - `/lib` contains the shared libraries (libc and others) and the Linux dynamic loader
    - `/etc` contains system configuration files and scripts

- There are different options for the Root Filesystem in an embedded system, the choice depends on:
  - The requirements of the system in terms of persistent (non volatile) storage.
  - The stage of development
  - The boot time required
Solutions for the Root Filesystem

- Rewritable data storage resource
  - FLASH, EPROM

- INITial RamDisk (initrd)
  - The Root Filesystem uses an Initial RAM disk (initrd) which is a file system located in RAM memory.
  - Fast solution but a portion of RAM is reserved in this case.
  - Kernel and Root Filesystem are combined in a single compressed image.

- Network File System (NTFS)
  - When using NFS, the Root Filesystem is located on a remote machine accessed via the network.
  - Requires network access and a server.
  - Suited for a system in early stages of development
When doing embedded development, there is always a split between:

- The *host*, the development workstation, which is typically a powerful PC
- The *target*, which is the embedded system under development

They are connected by various means: almost always a serial line for debugging purposes, frequently an Ethernet connection, sometimes a JTAG for low-level debugging.
Cross-development environment

- **PC** (*host*)
  - A Linux workstation is used to host the cross-development environment (usually > 3GB)
    - Linux cross-compiler
    - Debug (gdb, JTAG)

- **Platform** (*target*)
  - Kernel Image, Root Filesystem, bootloader

- **Host/target connexion**
  - Serial console
    - Communication using serial port (RS232)
  - Ethernet
  - JTAG
    - To download and run the kernel

Polytech’Nice Sophia - Département Electronique - Université de Nice Sophia Antipolis - S. Bilavarn - 20 -
Cross-compilation toolchain

- The usual development tools available on a GNU/Linux workstation is a *native toolchain*.
- This toolchain runs on your workstation and generates code for your workstation, usually x86.
- For embedded system development, it is usually impossible or not interesting to use a native toolchain.
  - The target is too restricted in terms of storage and/or memory.
  - The target is very slow compared to a workstation.
  - Developers may not want to install all development tools on the target.
- Therefore, *cross-compilation toolchains* are generally used. They run on your workstation but generate code for your target.
Cross-compilation toolchain

- A cross-compilation toolchain is required for the development of
  - the Linux kernel, to produce
    - the kernel image
    - file system and utilities
  - User applications
    - To compile « user space » C code
  - device drivers
    - To compile « kernel space » C code (modules)
Cross-compilation toolchain

- Compile a Linux kernel for another CPU architecture is
  - much faster than compiling natively, because the target system is much slower than your GNU/Linux workstation.
  - easier as development tools for your GNU/Linux workstation are much easier to find.

- To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system, architecture and sometimes library. Examples:
  - m68k-linux-uclibc-gcc
  - powerpc-linux-gcc
  - arm-linux-gnueabi-gcc
Cross-compilation toolchain

- The CPU architecture and cross-compiler prefix are defined through the ARCH and CROSS_COMPILE variables in the top level Makefile of the kernel source code.

  - ARCH is the name of the architecture. It is defined by the name of the subdirectory in arch/ in the kernel sources.
  - CROSS_COMPILE is the prefix of the cross compilation tools.
    - Example: arm-linux- if your compiler is arm-linux-gcc
Cross-compilation toolchain

- Three solutions to set ARCH and CROSS_COMPILE:
  - Force these two variables in the main kernel Makefile
    ```
    ARCH ?= arm
    CROSS_COMPILE ?= arm-linux
    ```
  - Pass ARCH and CROSS_COMPILE on the make command line
    ```
    make ARCH=arm CROSS_COMPILE=arm-eabi- uImage
    ```
  - Define ARCH and CROSS_COMPILE as environment variables
    ```
    export CROSS_COMPILE=ppc_4xx-
    export ARCH=powerpc
    ```

- Don't forget to have the values properly set at all steps, otherwise the kernel configuration and build system gets confused.
Cross-compilation toolchain

The configuration and generation of a cross-compilation toolchain is a complex and difficult operation which is based on:

- **gcc**
  - GNU Compiler Collection
  - Collection of free integrated software able to compile many programming languages (C, C++, Java, Ada, Fortran)

- **Binutils**
  - GNU binary utilities, a set of tools to generate and manipulate binaries for a given CPU architecture. Ex: GNU assembler (as) and linker (ld)

- **Glibc**
  - Standards C libraries of the GNU project, used by many system programs. To save space, these libraries are shared.
  - Ex: standard C library, standard math library.
  - Light versions for embedded systems (newlib, uClibc, sglibc, dietlibc)
Cross-compilation toolchain

- The generation of a cross-compilation toolchain is a significant amount of work when porting a Linux kernel on a new platform.
  - It is preferable to reuse an existing cross-compiler as much as possible.

- Toolchain examples:
  - PowerPC
    - Denx software engineering: [http://www.denx.de/](http://www.denx.de/)
  - ARM
Linux for Embedded Systems

- Introduction to Embedded Linux
- Key features
- Embedded Linux development
- Linux kernel development
Kernel development overview

- Linux is a C project.
- A Makefile is present at the root directory to compile the kernel source code.
- This Makefile needs to know some information to be able to compile the kernel.
  - Kernel configuration (hardware, drivers, filesystem, etc.)
- The compilation process produces an executable called the kernel image
  - The name can be different from a distribution to another
    - zImage
    - bzImage
    - uImage
Linux source code overview

- **Official distribution:**

![Directory Structure of Linux Kernel Source Code](image_url)
Linux source code overview

/arch                      Architecture dependent code
/crypto                   Library of cryptographic functions
/Documentation            Official kernel documentation
/drivers                  Device drivers (drivers/usb/, etc.)
/fs                       Filesystem (fs/ext3/, etc)
/include                  General kernel headers
/include/asm-<arch>       Architecture dependent kernel headers
/include/linux            Headers of the core of the Linux kernel
/init                     Linux initialization (includes main.c)
/IPC                      Inter process communication
/kernel                   Core of the kernel
/lib                      Various libraries (zlib, crc32, etc.)
/mm                       Memory management
Linux source code overview

/net  Network support (not including drivers)
/scripts  Internal / external scripts
/security  Security model implementation (selinux, ...)
/sound  Sound and sound drivers support
/usr  Utilities: gen_init_cpio, initramfs_data.S
COPYING  Conditions for Linux copying (GNUGPL)
CREDITS  Main Linux contributors
MAINTAINERS  People in charge with kernel parts
README  Introduction and compilation instructions
REPORTING-BUGS  Registering for bug reporting
Makefile  Main file for kernel compilation
Kernel development

- Kernel selection
- Kernel configuration
- Kernel compilation
- Kernel installation
Kernel development

- Kernel selection
  - The official distribution kernel.org is not necessarily the best choice
    - Linux development is split into different teams, each specialised in a specific target platform (e.g. ARM, OMAP, PowerPC, etc).
    - Synchronisation (update) with the official distribution is not immediate.
  - The choice of a Linux distribution depends first on the target platform and the associated development team. Examples:
    - x86: http://www.kernel.org
    - ARM: http://www.arm.linux.org.uk
      - Patch of the official distribution
    - PowerPC: http://penguinppc.org
      - Xilinx: http://git.xilinx.com
    - MIPS: http://www.linux-mips.org
    - M68k: http://www.linux-m68k.org
    - Microblaze: http://www.petalogix.com/
Kernel development

- Kernel selection
- Kernel configuration
- Kernel compilation
- Kernel installation
Kernel development

- Kernel configuration
  - The kernel contains thousands of device drivers, file system drivers, network protocols and other configurable items.
  - Thousands of options are available, that are used to selectively compile parts of the kernel source code.
  - The kernel configuration is the process of defining the set of options with which you want your kernel to be compiled.
  - The set of options depends:
    - on your hardware (for device drivers, etc.)
    - on the capabilities you would like to give to your kernel (network capabilities, file systems, real-time, etc.)
Kernel development

- The configuration is stored in a `.config` file at the root of the kernel source code
  - Simple text file, `key=value` style

- There are dependencies between kernel options
  - For example, enabling a network driver requires the network stack to be enabled.

- As options have dependencies, typically never edited by hand, but through graphical or text interfaces:
  - `make xconfig`, `make gconfig` (graphical)
  - `make menuconfig`, `make nconfig` (text)
  - You can switch from one to another, they all load/save the same `.config` file, and show the same set of options
Kernel development

- Kernel configuration
  - Kernel options
    - In general, the configuration process is guided by a Graphic User Interface.
    - Kernel configuration produces a .config file, symbolic links and header files needed to proceed to compilation.

- Description of available options:
  - Code maturity level options
    - Use of components or drivers in test phase.
  - General setup
    - General options for kernel configuration.
  - Loadable module support
    - Use of kernel non permanent modules (instead of permanent drivers).
  - Memory technology devices
    - Generic support for the management of storage devices.
Kernel development

- Kernel configuration
  - Kernel options
    - Block layer
      - Support of block type devices (ramdisk and loopback).
    - Processor
      - Configuration options related to the target processor.
    - Platform options
      - Configuration options related to the target platform.
    - Bus options
      - Options for bus support (PCI)
    - Networking
      - Support of network fonctionnalities.
    - Device drivers
      - Configuration options for device drivers.
    - File systems
      - Configuration options for different types of file system (ext2, ext3, fat, …)
Kernel development

Kernel configuration
- Kernel options
  - Kernel hacking
  - Kernel debug facilities
- Security options
  - Support of many security options at different levels (kernel, network)

Configuration
- Each item in the configuration menu can be:
  - integrated to the kernel – permanently (driver)
  - integrated to the kernel – not permanently (kernel module)
  - excluded from the kernel
Configuration methods

- make config
  - Command line configuration

- make menuconfig
  - GUI based on Ncurses library

- make xconfig
  - GUI based on Xwindows

→ this step produces a .config file.
Kernel development

- There is a variety of kernel options
  - difficult to understand / select all options
  - their choice is critical for the success of a kernel compilation.

- To help this choice, default configuration files are provided, per board or per-CPU family:
  - They are stored in arch/<target>/configs/, and are just minimal .config files
  - This is the most common way of configuring a kernel for embedded platforms
  - Run `make help` to find if one is available for your platform
  - To load a default configuration file, just run for example:
    - `make xilinx_zynq_defconfig`
  - This will overwrite your existing .config!
Kernel development

- Kernel selection
- Kernel configuration
- Kernel compilation
- Kernel installation
Kernel development

Kernel compilation

Compilation (make) involves several steps

- Dependency generation
  - make dep
  - Because of the presence of many header files, the main Makefile must be informed of dependencies between files to compile the kernel correctly.

- Build the kernel image
  - make zImage (or bzImage or uImage)
  - Informs the Makefile that it must compile a kernel image that will be further compressed using gzip (<512KB).
  - If an option (selected at previous step) is not supported, compilation fails.
  - The compilation operation is usually greater than 5 minutes (target and host dependent).

- Build the kernel modules
  - make modules
  - Kernel modules are elements that are external to the kernel, by definition.
  - They can be inserted to the kernel at run time. Kernel modules allow reducing the memory footprint (size) of the kernel.
  - *Kernel modules has to be cross-compiled using the same cross compiler used to build the kernel, and requires the kernel headers.*
Kernel development

- Kernel compilation
  - Other make utilities
    - Clean
      - `make clean`
      - Removes all of the object files and some other things that an old version leaves behind.
      - In any case, *do not* forget this step before attempting to recompile a kernel.
    - More cleaning
      - `make mrproper`
      - Will do a more extensive cleaning. It is sometimes necessary; you may wish to do it at every patch. Removes all generated files (needed when switching from one architecture to another).
      - Will also delete your configuration file, so you might want to make a backup of it (`.config`).
    - Clean distribution
      - `make distclean`
      - To return your linux kernel source directory to its original (unconfigured) condition. Similar to `mrproper`, but removes also editor backup and patch reject files.
Kernel development

- Kernel selection
- Kernel configuration
- Kernel compilation
- Kernel installation
Kernel development

Kernel installation

Files

- The kernel image (zImage or bzImage or uImage) is in the directory arch/<target>/images/
- In a cross-development context, this image must be uploaded on the target
  - JTAG: the kernel image can be uploaded and executed using JTAG based debug tools (e.g. XMD for Xilinx). This is useful for fast kernel testing, but not convenient for systems in advanced stage of development.
  - Flash: a Flash based system is typically based on two partitions: one for the Root Filesystem, one for the kernel image. The image kernel thus has to be copied on the correct Flash partition. Then usually, switching on the development board boots from the Flash content.
Kernel development

- **Bootloader**
  - The bootloader combines the kernel image and the Root Filesystem in the kernel boot process.
    - It is the first program that runs when booting the kernel
    - It depends on the type of persistent storage resource used for the Root Filesystem (RAM, Flash or network)
    - There are different types of bootloaders, the choice strongly depends on the target platform.

- **Kernel startup process**
  - Hardware pre-initialisation
    - Memory and serial console have to be setup before any further operation is possible.
  - Execution of kernel startup code: `start_kernel()`
    - Mounts the Root Filesystem and runs the initialisation process (`init`)
Kernel development

Overall booting process

- ROM startup code
  - Initializes CPU, registers, memory controller, on chip devices (serial console).
  - Configures kernel memory mapping
  - Runs the bootloader

- Bootloader
  - Uncompress the kernel in RAM memory
  - executes `start_kernel` (architecture independent).

ROM startup code
Architecture dependent initialization

Bootloader
Loads kernel to RAM and starts it

Kernel execution
Initializes hardware devices and kernel subsystems
Mounts the Root Filesystem
Starts the `init` process

/sbin/init
starts other userspace services and applications

shell
other applications

Root Filesystem
Overall booting process

Kernel execution

- Initializes caches, interrupts and various hardware devices
- Mounts the Root Filesystem indicated by `root=`
- Starts the `init` process
  - `/sbin/init` by default
  - Executing `init` loads shared runtime libraries
  - `init` reads its configuration file `/etc/inittab` and execute scripts
  - Typical script: `/etc/rc.d/rcS/` which configures and starts networking and other system services
  - `init` enters a run level where system duties can be performed or the login process can start, allowing for user sessions

ROM startup code
Architecture dependent initialization

Bootloader
Loads kernel to RAM and starts it

Kernel execution
Initializes hardware devices and kernel subsystems
Mounts the Root Filesystem
Starts the `init` process

/sbin/init
starts other userspace services and applications

shell
other applications

Root Filesystem
Kernel development

- Modules installation
  - Simple case: host = target
    - Modules compilation
      - make modules
      - Dependencies between modules may occur and must be solved.
      - Results in module files with .ko extension
  - Module execution
    - insmod/lsmod/rmmod to run, list and remove modules
  - Modules installation
    - make modules_install
    - Modules are installed by default in /lib/modules/
Kernel development

- Modules installation
  - Cross-development: host ≠ target
    - Modules compilation
      - `make -C /path/to/kernel-source M=$(pwd) modules`
      - Results in module files with `.ko` extension
      - `insmod/lsmod/rmmod` to run, list and remove modules
    - Modules installation
      - Compiled modules must be copied in the Root Filesystem of the target.
Cross-compile « Hello World » module for ARM using your laptop

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

int hello_init(void);
void hello_exit(void);

module_init(hello_init);
module_exit(hello_exit);

int hello_init(void) {
    printk("<1> Hello world!\n");
    return 0;
}

void hello_exit(void) {
    printk("<1> Bye, cruel world\n");
}
```
Cross-compile « Hello World » module for ARM on your laptop

- Install ARM cross-compiler on Ubuntu
  - Linaro cross-toolchain version 7.4.0
    - `sudo apt-get install gcc-arm-linux-gnueabi`
  - Test Linaro cross-toolchain by checking the cross-compiler version
    - `arm-linux-gnueabi-gcc -v`
- Configure environment variables
  - `export ARCH=arm`
  - `export CROSS_COMPILE=arm-linux-gnueabi-`

Compile modules

- Requires the path to the kernel headers
  - For a Xilinx Linux distribution on ZedBoard
  - Untar archive in directory `~/ZedBoard/`
  - Path to kernel headers: `~/ZedBoard/linux-xilinx`