Porting Linux on an ARM board

Bootlin
info@bootlin.com

Alexandre Belloni

- Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Open-source contributor
  - Maintainer for the Linux kernel **RTC subsystem**
  - Co-Maintainer of **kernel support for Atmel ARM processors**
  - Contributing to **kernel support for Marvell ARM (Berlin) processors**

bootlin

embedded Linux and kernel engineering

Mission

Support companies using embedded Linux in their projects.
Promote embedded Linux.

Training courses

Engineering expertise for development and consulting

Bootlin at a glance

- Engineering company created in 2004
  (not a training company!)
- Locations: Orange, Toulouse, Lyon (France)
- Serving customers all around the world
  See `http://bootlin.com/company/customers/`
- Head count: 9
  Only Free Software enthusiasts!
- Focus: Embedded Linux, Linux kernel, Android Free Software / Open Source for embedded and real-time systems.
- Activities: development, training, consulting, technical support.
- Added value: get the best of the user and development community and the resources it offers.

Bootlin on-line resources

- All our training materials:
  `http://bootlin.com/docs/`
- Technical blog:
  `http://bootlin.com/blog/`
- News and discussions (Google +):
  `https://plus.google.com/+Bootlin`
- News and discussions (LinkedIn):
  `https://www.linkedin.com/groups/4501089`
- Quick news (Twitter):
  `http://twitter.com/bootlincom`
- Linux Cross Reference - browse Linux kernel sources on-line:
  `http://lxr.bootlin.com`

Course content

Bootlin
info@bootlin.com

Porting linux

Porting Linux includes a number of steps, starting even before software is involved:

- SoC selection
- SoM, SBC selection or board conception
- Bootloader selection
- Bootloader port
- Linux kernel version selection
- Linux port
- Root filesystem integration

ARM Ecosystem

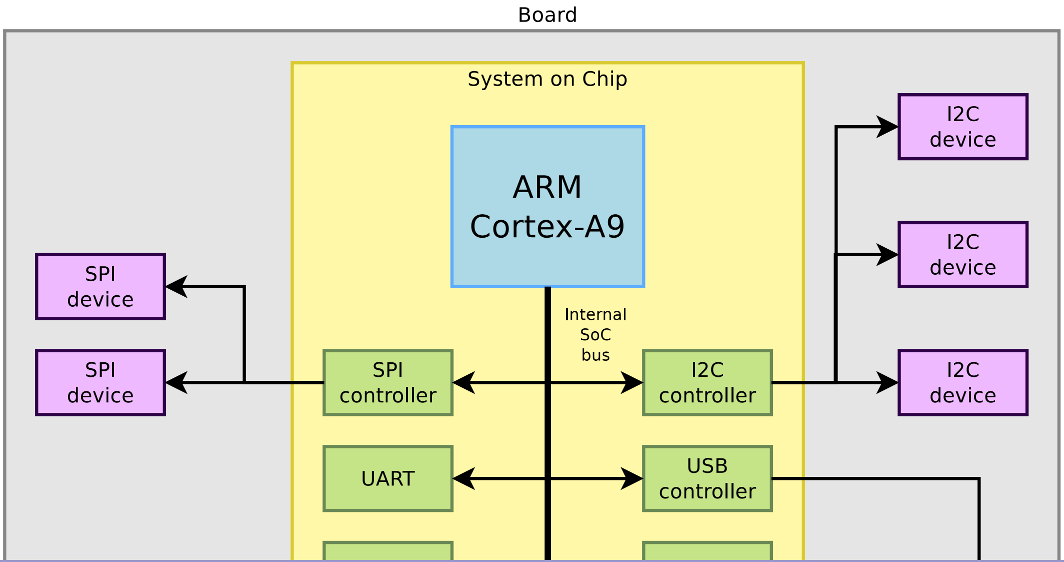
Bootlin
info@bootlin.com

ARM SoCs

ARM platforms

- **ARM** (the company) designs CPU cores: instruction set, MMU, caches, etc.
  - They don't sell any hardware
- **Silicon vendors** buy the CPU core design from ARM, and around it add a number of *peripherals*, either designed internally or bought from third parties
  - Texas Instruments, Atmel, Marvell, Freescale, Qualcomm, Nvidia, etc.
  - They sell *System-on-chip* or *SoCs*
- **System makers** design an actual board, with one or several processors, and a number of on-board peripherals

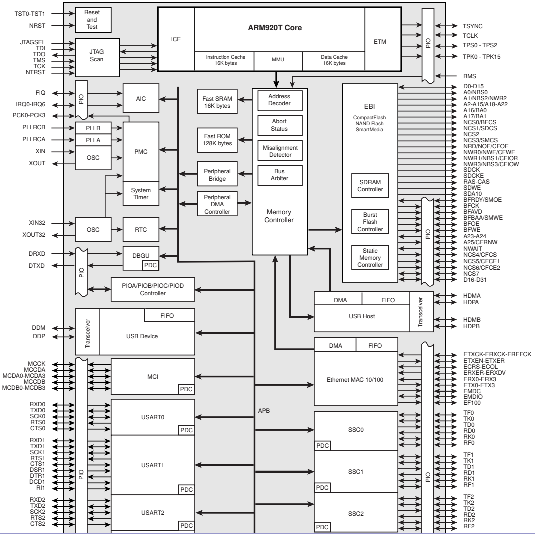
Schematic view of an ARM platform



System on Chip

A System on Chip is typically composed of:

- One or multiple CPU cores
- A bus
- Oscillators and PLL
- Timers
- Memory controller
- Interrupt controller
- Multiple peripherals:
  - UART
  - RTC
  - SPI
  - I2C
  - ADC



ARM cores

The Linux kernel supports a wide range of ARM based architectures, starting with ARMv4T:

ARM family	ARM architecture	ARM Core
ARM7T	ARMv4T	ARM7TDMI ARM720T ARM740T
ARM9T	ARMv4T	ARM9TDMI ARM920T ARM922T
		ARM925T ARM926T ARM940T
ARM9E	ARMv5TE	ARM946E-S
ARM10E	ARMv5TE	ARM1020T ARM1020E ARM1022E
	ARMv5TEJ	ARM1026EJ-S
ARM11	ARMv6Z	ARM1176JZF-S
	ARMv6K	ARM11MPCore
Cortex-M	ARMv7-M	Cortex-M3, Cortex-M4, Cortex-M7
Cortex-A (32-bit)	ARMv7-A	Cortex-A5, Cortex-A7
		Cortex-A8, Cortex-A9, Cortex-A12,
		Cortex-A15, Cortex-A17
Cortex-A (64-bit)	ARMv8-A	Cortex-A53, Cortex-A57, Cortex-A72

ARM cores

Third parties can also license the instruction set and create their own cores:

ARM ISA	Third party core
ARMv4	Faraday FA256, StrongARM SA-110, SA-1100
ARMv5TE	Xscale
ARMv5	Marvell PJ1, Feroceon
ARMv7-A	Broadcom Brahma-B15, Marvell PJ4, PJ4B, Qualcomm Krait, Scorpion
ARMv8-A	Cavium Thunder, Nvidia Denver, Qualcomm Kryo

ARM SoCs

To create an SoC, the silicon vendor integrates:

- one or multiple ARM cores (not necessarily homogeneous, big.LITTLE configurations exist)
- its own peripherals
- third party peripherals (usually from DesignWare, Cadence, PowerVR, Vivante, ...)
- ROM and ROM code
- sometimes one or multiple DSP, FPGA, micro-controller cores

ARM SoCs vendors

ARM SoC vendors with good mainline kernel support include:

- Allwinner
- Atmel
- Freescale
- Marvell
- Rockchip
- Samsung
- ST Micro
- TI (sitara and OMAP families)
- Xilinx

However, be careful when needing certain features like GPU acceleration.

ARM SoMs

System on Module manufacturer then create modules integrating:

- an SoC
- RAM
- Storage
- sometimes the PHYs for some interfaces like Ethernet, HDMI,...
- a connector for the baseboard

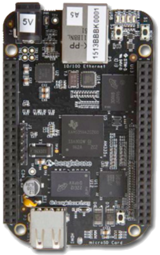
They also often manufacture Single-board computers (SBC) based on those SoM.

ARM SoMs manufacturers

- ACME
- Boundary Devices
- Congatec
- DataModul
- Olimex
- Phytec
- Seco
- Toradex
- Variscite

Community boards

A good way to create a prototype is to use a community board which is usually inexpensive and has expansion headers:



Examples include:

- BeagleBone Black
- Sama5dx Xplained
- OLinuXino boards

Choices

Bootlin
info@bootlin.com

Choices

Hardware

Choosing the right hardware parts

When choosing the hardware, existing software support should be considered.

- A driver exists in the mainline project
  - Does it support all the needed features?
- A driver is provided by the vendor:
  - What version is it compatible with and how difficult is it to port it to another version?
  - Does it use the proper frameworks?
- No driver available:
  - How complex is the hardware?
  - How complex is the framework?

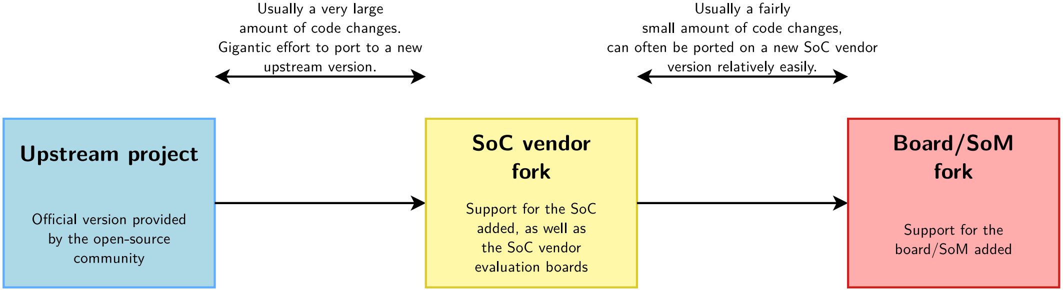
Software

Choosing the right version

Usually, the bootoaders and the Linux kernel are available from the following sources:

- SoM manufacturer
- SoC vendor
- Mainline

Choosing the right version



Upstream project

+ Best code quality
+ Availability of new versions
+ Security and bug fixes
+ Community support

- Not always all features supported (depends on the SoC)

SoC vendor fork

+ Most (if not all) HW features supported

- Often horrible code quality
- Usually unmaintained, no new version

SoC/SoM vendor fork

- Supports most of the Soc features
- May differ significantly from the mainline
- Usually, only a few (1-3) kernel versions are supported for each SoC
  - No security updates
  - No new drivers
  - Version may be ancient and have issues (example: DM368 has 2.6.32.17, from August 2010)
- May not support all the peripherals present on your board.

The SoM manufacturer usually base its BSP on that tree.

Mainline kernel

- Easier to update and benefit from security fixes, bug fixes, new drivers and new features
- Main drawback may be the lack of particular drivers like display, GPU, VPU.
- Maintenance and support from the community

Bootloaders

Bootlin
info@bootlin.com

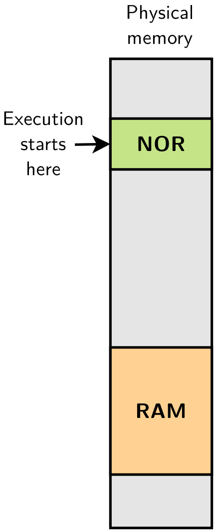
Boot Sequence

Bootloaders

- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel, from flash storage, from the network, or from another type of non-volatile storage.
  - Possibly decompression of the application binary
  - Execution of the application
- Besides these basic functions, most bootloaders provide a shell with various commands implementing different operations.
  - Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc.

Booting on embedded CPUs: case 1

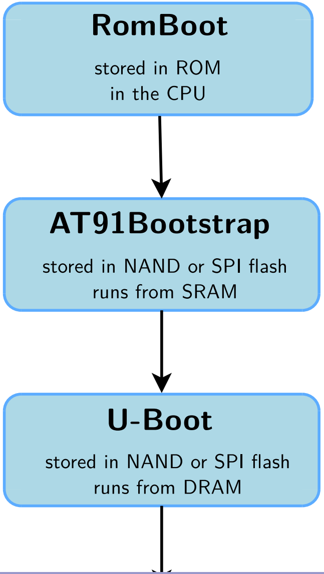
- When powered, the CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions
- The first stage bootloader must be programmed at this address in the NOR
- NOR is mandatory, because it allows random access, which NAND doesn't allow
- **Not very common anymore** (unpractical, and requires NOR flash)



Booting on embedded CPUs: case 2

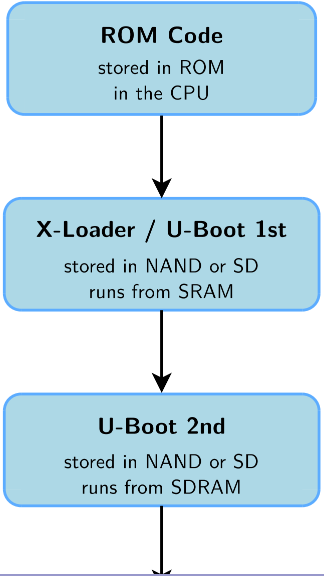
- The CPU has an integrated boot code in ROM
  - BootROM on AT91 CPUs, "ROM code" on OMAP, etc.
  - Exact details are CPU-dependent
- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
  - Storage device can typically be: MMC, NAND, SPI flash, UART (transmitting data over the serial line), etc.
- The first stage bootloader is
  - Limited in size due to hardware constraints (SRAM size)
  - Provided either by the CPU vendor or through community projects
- This first stage bootloader must initialize DRAM and other hardware devices and load a second stage bootloader into RAM

Booting on ARM Atmel AT91



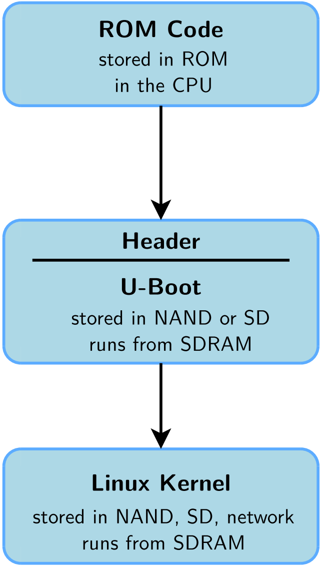
- **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.
- **AT91Bootstrap**: runs from SRAM. Initializes the DRAM, the NAND or SPI controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Booting on ARM TI OMAP3



- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to <64 KB. No user interaction possible.
- **X-Loader** or **U-Boot**: runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible. File called `MLO`.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called `u-boot.bin` or `u-boot.img`.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Booting on Marvell SoC



- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into RAM. The RAM configuration is described in a CPU-specific header, prepended to the bootloader image.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called `u-boot.kwb`.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Generic bootloaders for embedded CPUs

- We will focus on the generic part, the main bootloader, offering the most important features.
- There are several open-source generic bootloaders.
  Here are the most popular ones:
  - **U-Boot**, the universal bootloader by Denx
    The most used on ARM, also used on PPC, MIPS, x86, m68k, NIOS, etc. The de-facto standard nowadays. We will study it in detail.
    `http://www.denx.de/wiki/U-Boot`
  - **Barebox**, a new architecture-neutral bootloader, written as a successor of U-Boot. Better design, better code, active development, but doesn't yet have as much hardware support as U-Boot.
    `http://www.barebox.org`
- There are also a lot of other open-source or proprietary bootloaders, often architecture-specific
  - RedBoot, Yaboot, PMON, etc.

Porting the Bootloader

1st stage

The main goal of the first stage bootloader is to configure the RAM controller. Then it needs to be able to load the second stage bootloader from storage (NAND flash, SPI flash, NOR flash, MMC/eMMC) to RAM. The main porting steps are:

- Finding the proper RAM timings and settings them from the first stage.
- Configuring the storage IP
- Copying the second stage to RAM

Usually, the driver for the storage IP is already present in your first stage bootloader.

2nd stage

- The second stage bootloader has to load the Linux kernel from storage to RAM.
- Depending on the kernel version, it will also set the `ATAGS` or load the Device Tree.
- It may also load an `initramfs` to be used as the `root filesystem`.
- That is also a good place to implement base board or board variant detection if necessary.
- During development, the second stage bootloader also provides more debugging utilities like reading and writing to memory or Ethernet access.

2nd stage

The main porting steps are:

- Configuring the storage IP
- Copying the Linux kernel from storage to RAM
- Optional: copying the Device Tree to RAM
- Optional: implement boot scripts
- Optional: implement base board/board variant detection
- Optional: implement debug tools

Bootloader selection

Bootloaders

Two components to select: 1st stage and 2nd stage. However, it is usually easier to reduce the code base:

- Less code to understand
- Fewer upstream projects to follow
- Reduced maintenance

So, when available, use only one project for the first and the second stage. Example: for OMAP/Sitara, drop X-loader and use u-boot SPL.

Example: at91bootstrap

New board definition

- create a new board directory in `board`
- create `Config.in.board` and reference it from `board/Config.in`
- create `board.mk` and add the proper section to `include/board.h`
- create `board_name.h` and `board_name.c`
- Optional: create a defconfig
- Optional: `Config.in.linux_arg` when loading Linux directly from at91bootstrap.

Note: there will be a new `contrib` directory for non Atmel boards.

Config.in.board

```
config CONFIG_SAMA5D3_XPLAINED
    bool "sama5d3_xplained"
    select SAMA5D3X
    select CONFIG_DDRC
    select ALLOW_NANDFLASH
    select ALLOW_SDCARD
    select ALLOW_CPU_CLK_266MHZ
    select ALLOW_CPU_CLK_332MHZ
    select ALLOW_CPU_CLK_396MHZ
    select ALLOW_CPU_CLK_498MHZ
    select ALLOW_CPU_CLK_528MHZ
    select ALLOW_CRYSTAL_12_000MHZ
    select CONFIG_SUPPORT_PM
    select CONFIG_HAS_EHT0_PHY
    select CONFIG_HAS_EHT1_PHY
    select CONFIG_HAS_PMIC_ACT8865
    select SUPPORT_BUS_SPEED_133MHZ
    select SUPPORT_BUS_SPEED_166MHZ
    help
        Use the SAMA5D3 Xplained development board
```

board/Config.in

```
source "board/sama5d3xek/Config.in.board"
source "board/sama5d3_xplained/Config.in.board"
source "board/sama5d3x_cmp/Config.in.board"
```

board.mk

```
CPPFLAGS += -DCONFIG_SAMA5D3_XPLAINED
ASFLAGS += -DCONFIG_SAMA5D3_XPLAINED
```

include/board.h

```
#ifdef CONFIG_SAMA5D3XEK
#include "sama5d3xek.h"
#endif

#ifdef CONFIG_SAMA5D3_XPLAINED
#include "sama5d3_xplained.h"
#endif

#ifdef CONFIG_SAMA5D3X_CMP
#include "sama5d3x_cmp.h"
#endif
```

sama5d3_xplained.c

```
static void ddramc_reg_config(struct ddramc_register *ddramc_config)
{
    ddramc_config->mdr = (AT91C_DDRC2_DBW_32_BITS
                | AT91C_DDRC2_MD_DDR2_SDRAM);

    ddramc_config->cr = (AT91C_DDRC2_NC_DDR10_SDR9
                | AT91C_DDRC2_NR_13
                | AT91C_DDRC2_CAS_3
                | AT91C_DDRC2_DLL_RESET_DISABLED
                | AT91C_DDRC2_DIS_DLL_DISABLED
                | AT91C_DDRC2_ENRDM_ENABLE
                | AT91C_DDRC2_NB_BANKS_8
                | AT91C_DDRC2_NDQS_DISABLED
                | AT91C_DDRC2_DECOD_INTERLEAVED
                | AT91C_DDRC2_UNAL_SUPPORTED);

#if defined(CONFIG_BUS_SPEED_133MHZ)
    /*
     * The DDR2-SDRAM device requires a refresh every 15.625 us or 7.81 us.
     * With a 133 MHz frequency, the refresh timer count register must to be
     * set with (15.625 x 133 MHz) ~ 2084 i.e. 0x824
     * or (7.81 x 133 MHz) ~ 1039 i.e. 0x40F.
     */
    ddramc_config->rtr = 0x40F;     /* Refresh timer: 7.812us */

    /* One clock cycle @ 133 MHz = 7.5 ns */
    ddramc_config->t0pr = (AT91C_DDRC2_TRAS_(6)    /* 6 * 7.5 = 45 ns */
            | AT91C_DDRC2_TRCD_(2)        /* 2 * 7.5 = 22.5 ns */
            | AT91C_DDRC2_TWR_(2)         /* 2 * 7.5 = 15   ns */
            | AT91C_DDRC2_TRC_(8)         /* 8 * 7.5 = 75   ns */
            | AT91C_DDRC2_TRP_(2)         /* 2 * 7.5 = 15   ns */
            | AT91C_DDRC2_TRRD_(2)        /* 2 * 7.5 = 15   ns */
            | AT91C_DDRC2_TWTR_(2)        /* 2 clock cycles min */
            | AT91C_DDRC2_TMRD_(2));    /* 2 clock cycles */
```

```
static void ddramc_init(void)
{
    struct ddramc_register ddramc_reg;
    unsigned int reg;

    ddramc_reg_config(&ddramc_reg);

    /* enable ddr2 clock */
    pmc_enable_periph_clock(AT91C_ID_MPDDRC);
    pmc_enable_system_clock(AT91C_PMC_DDR);

    /* Init the special register for sama5d3x */
    /* MPDDRC DLL Slave Offset Register: DDR2 configuration */
    reg = AT91C_MPDDRC_S0OFF_1
        | AT91C_MPDDRC_S2OFF_1
        | AT91C_MPDDRC_S3OFF_1;
    writel(reg, (AT91C_BASE_MPDDRC + MPDDRC_DLL_SOR));

    /* MPDDRC DLL Master Offset Register */
    /* write master + clk90 offset */
    reg = AT91C_MPDDRC_MOFF_7
        | AT91C_MPDDRC_CLK90OFF_31
        | AT91C_MPDDRC_SELOFF_ENABLED | AT91C_MPDDRC_KEY;
    writel(reg, (AT91C_BASE_MPDDRC + MPDDRC_DLL_MOR));

    /* MPDDRC I/O Calibration Register */
    /* DDR2 RZQ = 50 Ohm */
    /* TZQIO = 4 */
    reg = AT91C_MPDDRC_RDIV_DDR2_RZQ_50
        | AT91C_MPDDRC_TZQIO_4;
    writel(reg, (AT91C_BASE_MPDDRC + MPDDRC_IO_CALIBR));

    /* DDRAM2 Controller initialize */
    ddram_initialize(AT91C_BASE_MPDDRC, AT91C_BASE_DDRCS, &ddramc_reg);
```

The U-boot bootloader

U-Boot

U-Boot is a typical free software project

- License: GPLv2 (same as Linux)
- Freely available at `http://www.denx.de/wiki/U-Boot`
- Documentation available at `http://www.denx.de/wiki/U-Boot/Documentation`
- The latest development source code is available in a Git repository: `http://git.denx.de/?p=u-boot.git;a=summary`
- Development and discussions happen around an open mailing-list `http://lists.denx.de/pipermail/u-boot/`
- Since the end of 2008, it follows a fixed-interval release schedule. Every three months, a new version is released. Versions are named `YYYY.MM`.

U-Boot configuration

- Get the source code from the website, and uncompress it
- The `include/configs/` directory contains one configuration file for each supported board
  - It defines the CPU type, the peripherals and their configuration, the memory mapping, the U-Boot features that should be compiled in, etc.
  - It is a simple `.h` file that sets C pre-processor constants. See the `README` file for the documentation of these constants. This file can also be adjusted to add or remove features from U-Boot (commands, etc.).
- Assuming that your board is already supported by U-Boot, there should be one entry corresponding to your board in the `boards.cfg` file.
  - Run `./tools/genboardscfg.py` to generate it.
  - Or just look in the `configs/` directory.

U-Boot configuration file excerpt

```
/* CPU configuration */
#define CONFIG_ARMV7 1
#define CONFIG_OMAP 1
#define CONFIG_OMAP34XX 1
#define CONFIG_OMAP3430 1
#define CONFIG_OMAP3_IGEP0020 1
[...]
/* Memory configuration */
#define CONFIG_NR_DRAM_BANKS 2
#define PHYS_SDRAM_1 OMAP34XX_SDRC_CS0
#define PHYS_SDRAM_1_SIZE (32 << 20)
#define PHYS_SDRAM_2 OMAP34XX_SDRC_CS1
[...]
/* USB configuration */
#define CONFIG_MUSB_UDC 1
#define CONFIG_USB_OMAP3 1
#define CONFIG_TWL4030_USB 1
[...]
```

```
/* Available commands and features */
#define CONFIG_CMD_CACHE
#define CONFIG_CMD_EXT2
#define CONFIG_CMD_FAT
#define CONFIG_CMD_I2C
#define CONFIG_CMD_MMC
#define CONFIG_CMD_NAND
#define CONFIG_CMD_NET
#define CONFIG_CMD_DHCP
#define CONFIG_CMD_PING
#define CONFIG_CMD_NFS
#define CONFIG_CMD_MTDPARTS
[...]
```

Configuring and compiling U-Boot

- U-Boot must be configured before being compiled
  - `make BOARDNAME_config`
  - Where `BOARDNAME` is the name of the board, as visible in the `boards.cfg` file (first column).
  - New: you can now run `make menuconfig` to further edit U-Boot's configuration!
- Make sure that the cross-compiler is available in `PATH`
- Compile U-Boot, by specifying the cross-compiler prefix.
  Example, if your cross-compiler executable is `arm-linux-gcc`:
  `make CROSS_COMPILE=arm-linux-`
- The main result is a `u-boot.bin` file, which is the U-Boot image. Depending on your specific platform, there may be other specialized images: `u-boot.img`, `u-boot.kwb`, `MLO`, etc.

Installing U-Boot

- U-Boot must usually be installed in flash memory to be executed by the hardware. Depending on the hardware, the installation of U-Boot is done in a different way:
  - The CPU provides some kind of specific boot monitor with which you can communicate through serial port or USB using a specific protocol
  - The CPU boots first on removable media (MMC) before booting from fixed media (NAND). In this case, boot from MMC to reflash a new version
  - U-Boot is already installed, and can be used to flash a new version of U-Boot. However, be careful: if the new version of U-Boot doesn't work, the board is unusable
  - The board provides a JTAG interface, which allows to write to the flash memory remotely, without any system running on the board. It also allows to rescue a board if the bootloader doesn't work.

U-boot prompt

- Connect the target to the host through a serial console
- Power-up the board. On the serial console, you will see something like:

```
U-Boot 2013.04 (May 29 2013 - 10:30:21)

OMAP36XX/37XX-GP ES1.2, CPU-OPP2, L3-165MHz, Max CPU Clock 1 Ghz
IGEPv2 + LPDDR/NAND
I2C:   ready
DRAM:  512 MiB
NAND:  512 MiB
MMC:   OMAP SD/MMC: 0

Die ID #255000029ff800000168580212029011
Net:   smc911x-0
U-Boot #
```

- The U-Boot shell offers a set of commands. We will study the most important ones, see the documentation for a complete reference or the `help` command.

Information commands

Flash information (NOR and SPI flash)

```
U-Boot> flinfo
DataFlash:AT45DB021
Nb pages: 1024
Page Size: 264
Size= 270336 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C0001FFF (RO) Bootstrap
Area 1: C0002000 to C0003FFF Environment
Area 2: C0004000 to C0041FFF (RO) U-Boot
```

NAND flash information

```
U-Boot> nand info
Device 0: nand0, sector size 128 KiB
  Page size      2048 b
  OOB size         64 b
  Erase size   131072 b
```

Version details

```
U-Boot> version
U-Boot 2013.04 (May 29 2013 - 10:30:21)
```

Important commands (1)

- The exact set of commands depends on the U-Boot configuration
- `help` and `help command`
- `boot`, runs the default boot command, stored in `bootcmd`
- `bootz <address>`, starts a kernel image loaded at the given address in RAM
- `ext2load`, loads a file from an ext2 filesystem to RAM
  - And also `ext2ls` to list files, `ext2info` for information
- `fatload`, loads a file from a FAT filesystem to RAM
  - And also `fatls` and `fatinfo`
- `tftp`, loads a file from the network to RAM
- `ping`, to test the network

Important commands (2)

- `loadb`, `loads`, `loady`, load a file from the serial line to RAM
- `usb`, to initialize and control the USB subsystem, mainly used for USB storage devices such as USB keys
- `mmc`, to initialize and control the MMC subsystem, used for SD and microSD cards
- `nand`, to erase, read and write contents to NAND flash
- `erase`, `protect`, `cp`, to erase, modify protection and write to NOR flash
- `md`, displays memory contents. Can be useful to check the contents loaded in memory, or to look at hardware registers.
- `mm`, modifies memory contents. Can be useful to modify directly hardware registers, for testing purposes.

Environment variables commands (1)

- U-Boot can be configured through environment variables, which affect the behavior of the different commands.
- Environment variables are loaded from flash to RAM at U-Boot startup, can be modified and saved back to flash for persistence
- There is a dedicated location in flash (or in MMC storage) to store the U-Boot environment, defined in the board configuration file

Environment variables commands (2)

Commands to manipulate environment variables:

- `printenv`
  Shows all variables
- `printenv <variable-name>`
  Shows the value of a variable
- `setenv <variable-name> <variable-value>`
  Changes the value of a variable, only in RAM
- `editenv <variable-name>`
  Edits the value of a variable, only in RAM
- `saveenv`
  Saves the current state of the environment to flash

Environment variables commands - Example

```
u-boot # printenv
baudrate=19200
ethaddr=00:40:95:36:35:33
netmask=255.255.255.0
ipaddr=10.0.0.11
serverip=10.0.0.1
stdin=serial
stdout=serial
stderr=serial
u-boot # printenv serverip
serverip=10.0.0.1
u-boot # setenv serverip 10.0.0.100
u-boot # saveenv
```

Important U-Boot env variables

- `bootcmd`, contains the command that U-Boot will automatically execute at boot time after a configurable delay (`bootdelay`), if the process is not interrupted
- `bootargs`, contains the arguments passed to the Linux kernel, covered later
- `serverip`, the IP address of the server that U-Boot will contact for network related commands
- `ipaddr`, the IP address that U-Boot will use
- `netmask`, the network mask to contact the server
- `ethaddr`, the MAC address, can only be set once
- `autostart`, if yes, U-Boot starts automatically an image that has been loaded into memory
- `filesize`, the size of the latest copy to memory (from `tftp`, `fat load`, `nand read`...)

Scripts in environment variables

- Environment variables can contain small scripts, to execute several commands and test the results of commands.
  - Useful to automate booting or upgrade processes
  - Several commands can be chained using the `;` operator
  - Tests can be done using `if command ; then ... ; else ... ; fi`
  - Scripts are executed using `run <variable-name>`
  - You can reference other variables using `${variable-name}`
- Example
  - `setenv mmc-boot 'if fatload mmc 0 80000000 boot.ini; then source; else if fatload mmc 0 80000000 zImage; then run mmc-boot; fi; fi'`

Transferring files to the target

- U-Boot is mostly used to load and boot a kernel image, but it also allows to change the kernel image and the root filesystem stored in flash.
- Files must be exchanged between the target and the development workstation. This is possible:
  - Through the network if the target has an Ethernet connection, and U-Boot contains a driver for the Ethernet chip. This is the fastest and most efficient solution.
  - Through a USB key, if U-Boot supports the USB controller of your platform
  - Through a SD or microSD card, if U-Boot supports the MMC controller of your platform
  - Through the serial port

TFTP

- Network transfer from the development workstation to U-Boot on the target takes place through TFTP
  - *Trivial File Transfer Protocol*
  - Somewhat similar to FTP, but without authentication and over UDP
- A TFTP server is needed on the development workstation
  - `sudo apt-get install tftpd-hpa`
  - All files in `/var/lib/tftpboot` are then visible through TFTP
  - A TFTP client is available in the `tftp-hpa` package, for testing
- A TFTP client is integrated into U-Boot
  - Configure the `ipaddr` and `serverip` environment variables
  - Use `tftp <address> <filename>` to load a file

Porting u-boot

Adding a new board

- Create a new board directory in `board/vendor`
- Write your board specific code. It can be split across multiple headers and C files.
- Create a `Makefile` referencing your code.
- Create a configuration header file
- Create a `Kconfig` file defining at least `SYS_BOARD`, `SYS_VENDOR` and `SYS_CONFIG_NAME`
- Add a target option for your board and source your `Kconfig` either from `arch/arm/<soc>/Kconfig` or `arch/arm/Kconfig`
- Optional: create a defconfig
- Optional: create a `MAINTAINERS` file

board/ti/am335x/

```
board.c
board.h
Kconfig
MAINTAINERS
Makefile
mux.c
README
u-boot.lds
```

board/ti/am335x/Makefile

```
#
# Makefile
#
# Copyright (C) 2011 Texas Instruments Incorporated - http://www.ti.com/
#
# SPDX-License-Identifier:        GPL-2.0+
#

ifeq ($(CONFIG_SKIP_LOWLEVEL_INIT),)
obj-y         := mux.o
endif

obj-y         += board.o
```

board/ti/am335x/Kconfig

```
if TARGET_AM335X_EVM

config SYS_BOARD
        default "am335x"

config SYS_VENDOR
        default "ti"

config SYS_SOC
        default "am33xx"

config SYS_CONFIG_NAME
        default "am335x_evm"

config CONS_INDEX
        int "UART used for console"
        range 1 6
        default 1
        help
          The AM335x SoC has a total of 6 UARTs (UART0 to UART5 as referenced
          in documentation, etc) available to it.  Depending on your specific
          board you may want something other than UART0 as for example the IDK
          uses UART3 so enter 4 here.
[...]
endif
```

arch/arm/Kconfig

```
[...]
config TARGET_AM335X_EVM
        bool "Support am335x_evm"
        select CPU_V7
        select SUPPORT_SPL
        select DM
        select DM_SERIAL
        select DM_GPIO
[...]
source "board/ti/am335x/Kconfig"
[...]
```

include/configs/am335x_evm.h

```
#ifndef __CONFIG_AM335X_EVM_H
#define __CONFIG_AM335X_EVM_H

#include <configs/ti_am335x_common.h>

/* Don't override the distro default bootdelay */
#undef CONFIG_BOOTDELAY
#include <config_distro_defaults.h>

#ifndef CONFIG_SPL_BUILD
#ifndef CONFIG_FIT
# define CONFIG_FIT
#endif
# define CONFIG_TIMESTAMP
# define CONFIG_LZO
#endif

[...]
```

include/configs/ti_am335x_common.h

```
#ifndef __CONFIG_TI_AM335X_COMMON_H__
#define __CONFIG_TI_AM335X_COMMON_H__

#define CONFIG_AM33XX
#define CONFIG_ARCH_CPU_INIT
#define CONFIG_SYS_CACHELINE_SIZE       64
#define CONFIG_MAX_RAM_BANK_SIZE        (1024 << 20)        /* 1GB */
#define CONFIG_SYS_TIMERBASE                0x48040000         /* Use Timer2 */
#define CONFIG_SPL_AM33XX_ENABLE_RTC32K_OSC

#include <asm/arch/omap.h>

/* NS16550 Configuration */
#ifdef CONFIG_SPL_BUILD
#define CONFIG_SYS_NS16550_SERIAL
#define CONFIG_SYS_NS16550_REG_SIZE        (-4)
#endif
#define CONFIG_SYS_NS16550_CLK                48000000
[...]
/*
 * SPL related defines.  The Public RAM memory map the ROM defines the
 * area between 0x402F0400 and 0x4030B800 as a download area and
 * 0x4030B800 to 0x4030CE00 as a public stack area.  The ROM also
 * supports X-MODEM loading via UART, and we leverage this and then use
 * Y-MODEM to load u-boot.img, when booted over UART.
 */
#define CONFIG_SPL_TEXT_BASE                0x402F0400
#define CONFIG_SPL_MAX_SIZE                (0x4030B800 - CONFIG_SPL_TEXT_BASE)
#define CONFIG_SYS_SPL_ARGS_ADDR        (CONFIG_SYS_SDRAM_BASE + \
                                         (128 << 20))
```

include/configs/ti_am335x_common.h

```
/* Enable the watchdog inside of SPL */
#define CONFIG_SPL_WATCHDOG_SUPPORT

/*
 * Since SPL did pll and ddr initialization for us,
 * we don't need to do it twice.
 */
#if !defined(CONFIG_SPL_BUILD) && !defined(CONFIG_NOR_BOOT)
#define CONFIG_SKIP_LOWLEVEL_INIT
#endif

/*
 * When building U-Boot such that there is no previous loader
 * we need to call board_early_init_f.  This is taken care of in
 * s_init when we have SPL used.
 */
#if !defined(CONFIG_SKIP_LOWLEVEL_INIT) && !defined(CONFIG_SPL)
#define CONFIG_BOARD_EARLY_INIT_F
#endif
[...]
```

board/ti/am335x/board.c

```
[...]
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
[...]
static const struct ddr_data ddr3_beagleblack_data = {
        .datardsratio0 = MT41K256M16HA125E_RD_DQS,
        .datawdsratio0 = MT41K256M16HA125E_WR_DQS,
        .datafwsratio0 = MT41K256M16HA125E_PHY_FIFO_WE,
        .datawrsratio0 = MT41K256M16HA125E_PHY_WR_DATA,
};
[...]
static const struct cmd_control ddr3_beagleblack_cmd_ctrl_data = {
        .cmd0csratio = MT41K256M16HA125E_RATIO,
        .cmd0iclkout = MT41K256M16HA125E_INVERT_CLKOUT,

        .cmd1csratio = MT41K256M16HA125E_RATIO,
        .cmd1iclkout = MT41K256M16HA125E_INVERT_CLKOUT,

        .cmd2csratio = MT41K256M16HA125E_RATIO,
        .cmd2iclkout = MT41K256M16HA125E_INVERT_CLKOUT,
};
[...]
static struct emif_regs ddr3_beagleblack_emif_reg_data = {
        .sdram_config = MT41K256M16HA125E_EMIF_SDCFG,
        .ref_ctrl = MT41K256M16HA125E_EMIF_SDREF,
        .sdram_tim1 = MT41K256M16HA125E_EMIF_TIM1,
        .sdram_tim2 = MT41K256M16HA125E_EMIF_TIM2,
        .sdram_tim3 = MT41K256M16HA125E_EMIF_TIM3,
        .zq_config = MT41K256M16HA125E_ZQ_CFG,
        .emif_ddr_phy_ctlr_1 = MT41K256M16HA125E_EMIF_READ_LATENCY,
};
[...]
#endif
```

```c
void sdram_init(void)
{
	__maybe_unused struct am335x_baseboard_id header;

	if (read_eeprom(&header) < 0)
		puts("Could not get board ID.\n");

	if (board_is_evm_sk(&header)) {
		/*
		 * EVM SK 1.2A and later use gpio0_7 to enable DDR3.
		 * This is safe enough to do on older revs.
		 */
		gpio_request(GPIO_DDR_VTT_EN, "ddr_vtt_en");
		gpio_direction_output(GPIO_DDR_VTT_EN, 1);
	}

	if (board_is_evm_sk(&header))
		config_ddr(303, &ioregs_evmsk, &ddr3_data,
			   &ddr3_cmd_ctrl_data, &ddr3_emif_reg_data, 0);
	else if (board_is_bone_lt(&header))
		config_ddr(400, &ioregs_bonelt,
			   &ddr3_beagleblack_data,
			   &ddr3_beagleblack_cmd_ctrl_data,
			   &ddr3_beagleblack_emif_reg_data, 0);
	else if (board_is_evm_15_or_later(&header))
		config_ddr(303, &ioregs_evm15, &ddr3_evm_data,
			   &ddr3_evm_cmd_ctrl_data, &ddr3_evm_emif_reg_data, 0);
	else
		config_ddr(266, &ioregs, &ddr2_data,
			   &ddr2_cmd_ctrl_data, &ddr2_emif_reg_data, 0);
}
```

```
/*
 * Basic board specific setup.  Pinmux has been handled already.
 */
int board_init(void)
{
#if defined(CONFIG_HW_WATCHDOG)
        hw_watchdog_init();
#endif

        gd->bd->bi_boot_params = CONFIG_SYS_SDRAM_BASE + 0x100;
#if defined(CONFIG_NOR) || defined(CONFIG_NAND)
        gpmc_init();
#endif
        return 0;
}

#ifdef CONFIG_BOARD_LATE_INIT
int board_late_init(void)
{
#ifdef CONFIG_ENV_VARS_UBOOT_RUNTIME_CONFIG
        char safe_string[HDR_NAME_LEN + 1];
        struct am335x_baseboard_id header;

        if (read_eeprom(&header) < 0)
                puts("Could not get board ID.\n");

        /* Now set variables based on the header. */
        strncpy(safe_string, (char *)header.name, sizeof(header.name));
        safe_string[sizeof(header.name)] = 0;
        setenv("board_name", safe_string);

        /* BeagleBone Green eeprom, board_rev: 0x1a 0x00 0x00 0x00 */
        if ( (header.version[0] == 0x1a) && (header.version[1] == 0x00) &&
```

arch/arm/cpu/armv7/am33xx/board.c

```
[...]
#ifdef CONFIG_SPL_BUILD
void board_init_f(ulong dummy)
{
        board_early_init_f();
        sdram_init();
}
#endif

void s_init(void)
{
        /*
         * The ROM will only have set up sufficient pinmux to allow for the
         * first 4KiB NOR to be read, we must finish doing what we know of
         * the NOR mux in this space in order to continue.
         */
#ifdef CONFIG_NOR_BOOT
        enable_norboot_pin_mux();
#endif
        watchdog_disable();
        set_uart_mux_conf();
        setup_clocks_for_console();
        uart_soft_reset();
#if defined(CONFIG_SPL_AM33XX_ENABLE_RTC32K_OSC)
        /* Enable RTC32K clock */
        rtc32k_enable();
#endif
}
#endif
```

Linux kernel

Bootlin
info@bootlin.com

Corrections, suggestions, contributions and translations are welcome!

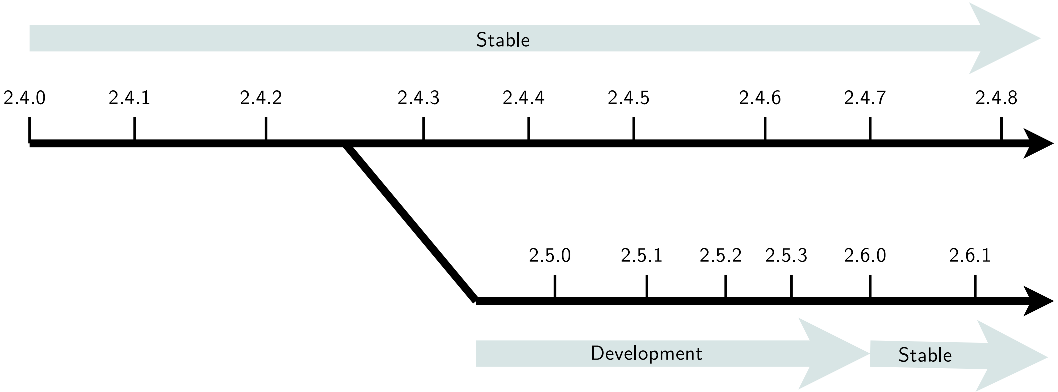
embedded Linux and kernel engineering

Linux versioning scheme and development process

Until 2.6 (1)

- One stable major branch every 2 or 3 years
  - Identified by an even middle number
  - Examples: `1.0.x`, `2.0.x`, `2.2.x`, `2.4.x`
- One development branch to integrate new functionalities and major changes
  - Identified by an odd middle number
  - Examples: `2.1.x`, `2.3.x`, `2.5.x`
  - After some time, a development version becomes the new base version for the stable branch
- Minor releases once in while: `2.2.23`, `2.5.12`, etc.

Until 2.6 (2)


Stable
2.4.0
2.4.1
2.4.2
2.4.3
2.4.4
2.4.5
2.4.6
2.4.7
2.4.8
2.5.0
2.5.1
2.5.2
2.5.3
2.6.0
2.6.1
Development
Stable

Changes since Linux 2.6

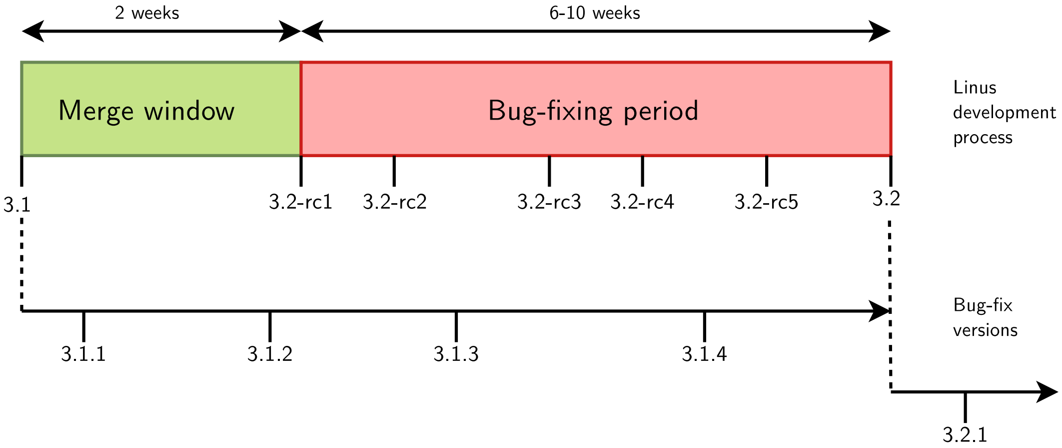
- Since `2.6.0`, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make disruptive changes to existing subsystems.
- Since then, there has been no need to create a new development branch massively breaking compatibility with the stable branch.
- Thanks to this, **more features are released to users at a faster pace**.

3.x stable branch

- From 2003 to 2011, the official kernel versions were named `2.6.x`.
- Linux `3.0` was released in July 2011
- This is only a change to the numbering scheme
  - Official kernel versions are now named `3.x` (`3.0, 3.1, 3.2`, etc.)
  - Stabilized versions are named `3.x.y` (`3.0.2, 3.4.3`, etc.)
  - It effectively only removes a digit compared to the previous numbering scheme

New development model

Using merge and bug fixing windows



New development model - Details

- After the release of a `3.x` version (for example), a two-weeks merge window opens, during which major additions are merged.
- The merge window is closed by the release of test version `3.(x+1)-rc1`
- The bug fixing period opens, for 6 to 10 weeks.
- At regular intervals during the bug fixing period, `3.(x+1)-rcY` test versions are released.
- When considered sufficiently stable, kernel `3.(x+1)` is released, and the process starts again.

More stability for the kernel source tree

- Issue: bug and security fixes only released for most recent stable kernel versions.
- Some people need to have a recent kernel, but with long term support for security updates.
- You could get long term support from a commercial embedded Linux provider.
- You could reuse sources for the kernel used in Ubuntu Long Term Support releases (5 years of free security updates).
- The `http://kernel.org` front page shows which versions will be supported for some time (up to 2 or 3 years), and which ones won't be supported any more ("EOL: End Of Life")

mainline:	**3.14-rc8**	2014-03-25
stable:	**3.13.7**	2014-03-24
stable:	**3.11.10 [EOL]**	2013-11-29
longterm:	**3.12.15**	2014-03-26
longterm:	**3.10.34**	2014-03-24
longterm:	**3.4.84**	2014-03-24
longterm:	**3.2.55**	2014-02-15
longterm:	**2.6.34.15 [EOL]**	2014-02-10
longterm:	**2.6.32.61**	2013-06-10
linux-next:	**next-20140327**	2014-03-27

What's new in each Linux release?

- The official list of changes for each Linux release is just a huge list of individual patches!

```
commit aa6e52a35d388e730f4df0ec2ec48294590cc459
Author: Thomas Petazzoni <thomas.petazzoni@bootlin.com>
Date:   Wed Jul 13 11:29:17 2011 +0200

    at91: at91-ohci: support overcurrent notification

    Several USB power switches (AIC1526 or MIC2026) have a digital output
    that is used to notify that an overcurrent situation is taking
    place. This digital outputs are typically connected to GPIO inputs of
    the processor and can be used to be notified of these overcurrent
    situations.

    Therefore, we add a new overcurrent_pin[] array in the at91_usbh_data
    structure so that boards can tell the AT91 OHCI driver which pins are
    used for the overcurrent notification, and an overcurrent_supported
    boolean to tell the driver whether overcurrent is supported or not.

    The code has been largely borrowed from ohci-da8xx.c and
    ohci-s3c2410.c.

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@bootlin.com>
    Signed-off-by: Nicolas Ferre <nicolas.ferre@atmel.com>
```

  - Very difficult to find out the key changes and to get the global picture out of individual changes.
- Fortunately, there are some useful resources available
  - `http://wiki.kernelnewbies.org/LinuxChanges`
  - `http://lwn.net`
  - `http://linuxfr.org`, for French readers

Porting

Porting

Porting the kernel involves:

- Adding support for the CPU core
- Writing drivers for the SoC peripherals and SoC specific features (SMP, power management)
- Writing drivers for the board peripherals
- Integrating all the drivers and describing how the peripherals are connected on the board.

Hopefully, only the last step is needed.

Board support

Bootlin
info@bootlin.com

Discoverable vs. non-discoverable hardware

- Certain busses have dynamic discoverability features
  - USB, PCI
  - Allow to enumerate devices on the bus, query their characteristics, at runtime.
  - No need to know in advance what's on the bus
- But many busses do not have such features
  - Memory-mapped devices inside SoC, I2C, SPI, SDIO, etc.
  - The system has to know in advance "where" the different devices are located, and their characteristics
  - Such devices, instead of being dynamically detected, must be statically described in either:
    - The kernel source code
    - The *Device Tree*, a hardware description file used on some architectures.

ARM code organization in the Linux kernel

- `arch/arm/{kernel,mm,lib,boot}/`
  The core ARM kernel. Contains the code related to the ARM core itself (MMU, interrupts, caches, etc.). Relatively small compared to the SoC-specific code.
- `arch/arm/mach-<foo>/`
  The SoC-specific code, and board-specific code, for a given SoC family (clocks, pinmux, power management, SMP, and more.)
  - `arch/arm/mach-<foo>/board-<bar>.c`.
    The board-specific code.

Platform drivers

Platform devices

- Amongst the non-discoverable devices, a huge family are the devices that are directly part of a system-on-chip: UART controllers, Ethernet controllers, SPI or I2C controllers, graphic or audio devices, etc.
- In the Linux kernel, a special bus, called the **platform bus** has been created to handle such devices.
- It supports **platform drivers** that handle **platform devices**.
- It works like any other bus (USB, PCI), except that devices are enumerated statically instead of being discovered dynamically.

Implementation of a Platform Driver

- The driver implements a `struct platform_driver` structure (example taken from `drivers/serial/imx.c`)

```
static struct platform_driver serial_imx_driver = {
    .probe = serial_imx_probe,
    .remove = serial_imx_remove,
    .driver = {
        .name = "imx-uart",
        .owner = THIS_MODULE,
    },
};
```

- And registers its driver to the platform driver infrastructure

```
static int __init imx_serial_init(void) {
    ret = platform_driver_register(&serial_imx_driver);
}

static void __exit imx_serial_cleanup(void) {
    platform_driver_unregister(&serial_imx_driver);
}
```

Platform Device Instantiation: old style (1/2)

- As platform devices cannot be detected dynamically, they are defined statically
  - By direct instantiation of `struct platform_device` structures, as done on some ARM platforms. Definition done in the board-specific or SoC specific code.
  - By using a *device tree*, as done on Power PC (and on some ARM platforms) from which `struct platform_device` structures are created
- Example on ARM, where the instantiation is done in `arch/arm/mach-imx/mx1ads.c`

```
static struct platform_device imx_uart1_device = {
    .name = "imx-uart",
    .id = 0,
    .num_resources = ARRAY_SIZE(imx_uart1_resources),
    .resource = imx_uart1_resources,
    .dev = {
        .platform_data = &uart_pdata,
    }
};
```

Platform device instantiation: old style (2/2)

- The device is part of a list

```
static struct platform_device *devices[] __initdata = {
    &cs89x0_device,
    &imx_uart1_device,
    &imx_uart2_device,
};
```

- And the list of devices is added to the system during board initialization

```
static void __init mx1ads_init(void)
{
    [...]
    platform_add_devices(devices, ARRAY_SIZE(devices));
}

MACHINE_START(MX1ADS, "Freescale MX1ADS")
    [...]
    .init_machine = mx1ads_init,
MACHINE_END
```

The Resource Mechanism

- Each device managed by a particular driver typically uses different hardware resources: addresses for the I/O registers, DMA channels, IRQ lines, etc.
- Such information can be represented using `struct resource`, and an array of `struct resource` is associated to a `struct platform_device`
- Allows a driver to be instantiated for multiple devices functioning similarly, but with different addresses, IRQs, etc.

Declaring resources

```
static struct resource imx_uart1_resources[] = {
    [0] = {
        .start = 0x00206000,
        .end = 0x002060FF,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = (UART1_MINT_RX),
        .end = (UART1_MINT_RX),
        .flags = IORESOURCE_IRQ,
    },
};
```

Using Resources

- When a `struct platform_device` is added to the system using `platform_add_device()`, the `probe()` method of the platform driver gets called
- This method is responsible for initializing the hardware, registering the device to the proper framework (in our case, the serial driver framework)
- The platform driver has access to the I/O resources:

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
sport->rxirq = platform_get_irq(pdev, 0);
```

platform_data Mechanism

- In addition to the well-defined resources, many drivers require driver-specific information for each platform device
- Such information can be passed using the `platform_data` field of `struct device` (from which `struct platform_device` inherits)
- As it is a `void *` pointer, it can be used to pass any type of information.
  - Typically, each driver defines a structure to pass information through `struct platform_data`

platform_data example 1/2

- The i.MX serial port driver defines the following structure to be passed through `struct platform_data`

```
struct imxuart_platform_data {
    int (*init)(struct platform_device *pdev);
    void (*exit)(struct platform_device *pdev);
    unsigned int flags;
    void (*irda_enable)(int enable);
    unsigned int irda_inv_rx:1;
    unsigned int irda_inv_tx:1;
    unsigned short transceiver_delay;
};
```

- The MX1ADS board code instantiates such a structure

```
static struct imxuart_platform_data uart1_pdata = {
    .flags = IMXUART_HAVE_RTSCTS,
};
```

platform_data Example 2/2

- The uart_pdata structure is associated to the struct platform_device structure in the MX1ADS board file (the real code is slightly more complicated)

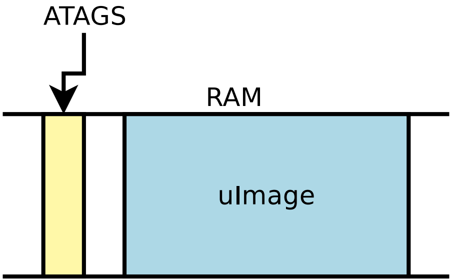
```
struct platform_device mx1ads_uart1 = {
    .name = "imx-uart",
    .dev {
        .platform_data = &uart1_pdata,
    },
    .resource = imx_uart1_resources,
    [...]
};
```

- The driver can access the platform data:

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imxuart_platform_data *pdata;
    pdata = pdev->dev.platform_data;
    if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
        sport->have_rtscts = 1;
    [...]
```

Booting

- The kernel contains the entire description of the hardware.
- The bootloader loads a single binary, the kernel image, and executes it.
  - `uImage` or `zImage`
- The bootloader prepares some additional information, called `ATAGS`, which address is passed to the kernel through register `r2`
  - Contains information such as memory size and location, kernel command line, etc.
- The bootloader tells the kernel on which board it is being booted through a *machine type* integer, passed in register `r1`.
- U-Boot command: `bootm <kernel img addr>`
- Barebox variable: `bootm.image`




```
r1 = <machine type>
r2 = <pointer to ATAGS>
```

Board file

- The machine type is matched with the ones defined using `struct machine_desc`
- Those definitions are done using the `MACHINE_START` and `MACHINE_END` macros.

```
MACHINE_START(MX1ADS, "Motorola MX1ADS")
        /* Maintainer: Sascha Hauer, Pengutronix */
        .phys_io        = 0x00200000,
        .io_pg_offst    = ((0xe0000000) >> 18) & 0xfffc,
        .boot_params    = 0x08000100,
        .map_io         = mx1ads_map_io,
        .init_irq       = imx_init_irq,
        .timer          = &imx_timer,
        .init_machine   = mx1ads_init,
MACHINE_END
```

Device Tree

Device Tree

- On many embedded architectures, manual instantiation of platform devices was considered to be too verbose and not easily maintainable.
- Such architectures are moving, or have moved, to use the *Device Tree*.
- It is a **tree of nodes** that models the hierarchy of devices in the system, from the devices inside the processor to the devices on the board.
- Each node can have a number of **properties** describing various properties of the devices: addresses, interrupts, clocks, etc.
- At boot time, the kernel is given a compiled version, the **Device Tree Blob**, which is parsed to instantiate all the devices described in the DT.
- On ARM, they are located in `arch/arm/boot/dts/`.

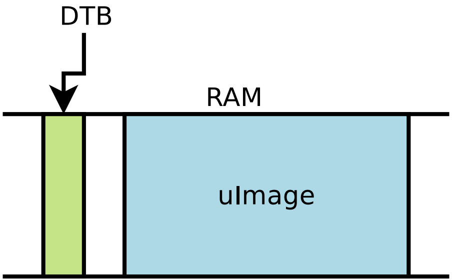
What is the Device Tree ?

- Quoted from the *Power.org Standard for Embedded Power Architecture Platform Requirements (ePAPR)*
  - The ePAPR specifies a concept called a device tree to describe system hardware. A boot program loads a device tree into a client program's memory and passes a pointer to the device tree to the client.
  - A device tree is a tree data structure with nodes that describe the physical devices in a system.
  - An ePAPR-compliant device tree describes device information in a system that cannot be dynamically detected by a client program.

Booting

- The kernel no longer contains the description of the hardware, it is located in a separate binary: the *device tree blob*
- The bootloader loads two binaries: the kernel image and the *DTB*
  - Kernel image remains `uImage` or `zImage`
  - DTB located in `arch/arm/boot/dts`, one per board
- The bootloader passes the DTB address through `r2`. It is supposed to adjust the DTB with memory information, kernel command line, and potentially other info.
- No more *machine type*.
- U-Boot command: `boot[mz] <kernel img addr> - <dtb addr>`
- Barebox variables: `bootm.image`, `bootm.oftree`

Booting




```
r1 = don't care
r2 = <pointer to DTB>
```

Compatibility mode for DT booting

- Some bootloaders have no specific support for the Device Tree, or the version used on a particular device is too old to have this support.
- To ease the transition, a *compatibility* mechanism was added: `CONFIG_ARM_APPENDED_DTB`.
  - It tells the kernel to look for a DTB right *after* the kernel image.
  - There is no built-in Makefile rule to produce such kernel, so one must manually do:

```
cat arch/arm/boot/zImage arch/arm/boot/dts/myboard.dtb > my-zImage
mkimage ... -d my-zImage my-uImage
```

- In addition, the additional option `CONFIG_ARM_ATAG_DTB_COMPAT` tells the kernel to read the *ATAGS* information from the bootloader, and update the DT using them.

Basic Device Tree syntax

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];

        child-node@0 {
            first-child-property;
            second-child-property = <1>;
            a-reference-to-something = <&node1>;
        };

        child-node@1 {
        };
    };

    node1: node@1 {
        an-empty-property;
        a-cell-property = <1 2 3 4>;

        child-node@0 {
        };
    };
};
```

- Node name
- Unit address
- Property name
- Property value
- Properties of node@0
- Bytestring
- A phandle (reference to another node)
- Label
- Four cells (32 bits values)

From source to binary

- On ARM, all **Device Tree Source** files (DTS) are for now located in `arch/arm/boot/dts`
  - `.dts` files for board-level definitions
  - `.dtsi` files for included files, generally containing SoC-level definitions
- A tool, the **Device Tree Compiler** compiles the source into a binary form.
  - Source code located in `scripts/dtc`
- The **Device Tree Blob** is produced by the compiler, and is the binary that gets loaded by the bootloader and parsed by the kernel at boot time.
- `arch/arm/boot/dts/Makefile` lists which DTBs should be generated at build time.

```
dtb-$(CONFIG_ARCH_MVEBU) += armada-370-db.dtb \
        armada-370-mirabox.dtb \
...
```

Exploring the DT on the target

- In /sys/firmware/devicetree/base, there is a directory/file representation of the Device Tree contents

```
# ls -l /sys/firmware/devicetree/base/
total 0
-r--r--r--    1 root     root       4 Jan  1 00:00 #address-cells
-r--r--r--    1 root     root       4 Jan  1 00:00 #size-cells
drwxr-xr-x    2 root     root       0 Jan  1 00:00 chosen
drwxr-xr-x    3 root     root       0 Jan  1 00:00 clocks
-r--r--r--    1 root     root      34 Jan  1 00:00 compatible
[...]
-r--r--r--    1 root     root       1 Jan  1 00:00 name
drwxr-xr-x   10 root     root       0 Jan  1 00:00 soc
```

- If dtc is available on the target, possible to "unpack" the Device Tree using:
  dtc -I fs /sys/firmware/devicetree/base

A simple example, DT side

```
auart0: serial@8006a000 {
```

Defines the "programming model" for the device. Allows the operating system to identify the corresponding device driver.

```
compatible = "fsl,imx28-auart", "fsl,imx23-auart";
```

Address and length of the register area.

```
reg = <0x8006a000 0x2000>;
```

Interrupt number.

```
interrupts = <112>;
```

DMA engine and channels, with names.

```
dmas = <&dma_apbx 8>, <&dma_apbx 9>;
dma-names = "rx", "tx";
```

Reference to the clock.

```
clocks = <&clks 45>;
```

The device is not enabled.

```
status = "disabled";
```

A simple example, driver side (1)

The compatible string used to bind a device with the driver

```
static struct of_device_id mxs_auart_dt_ids[] = {
        {
                .compatible = "fsl,imx28-auart",
                .data = &mxs_auart_devtype[IMX28_AUART]
        }, {
                .compatible = "fsl,imx23-auart",
                .data = &mxs_auart_devtype[IMX23_AUART]
        }, { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, mxs_auart_dt_ids);
[...]
static struct platform_driver mxs_auart_driver = {
        .probe = mxs_auart_probe,
        .remove = mxs_auart_remove,
        .driver = {
                .name = "mxs-auart",
                .of_match_table = mxs_auart_dt_ids,
        },
};
```

Code from `drivers/tty/serial/mxs-auart.c`

A simple example, driver side (2)

- `of_match_device` allows to get the matching entry in the `mxs_auart_dt_ids` table.
- Useful to get the driver-specific `data` field, typically used to alter the behavior of the driver depending on the variant of the detected device.

```
static int mxs_auart_probe(struct platform_device *pdev)
{
        const struct of_device_id *of_id =
                        of_match_device(mxs_auart_dt_ids, &pdev->dev);

        if (of_id) {
                /* Use of_id->data here */
                [...]
        }
        [...]
}
```

A simple example, driver side (3)

- Getting a reference to the clock
  - described by the `clocks` property
  - `s->clk = clk_get(&pdev->dev, NULL);`
- Getting the I/O registers *resource*
  - described by the `reg` property
  - `r = platform_get_resource(pdev, IORESOURCE_MEM, 0);`
- Getting the interrupt
  - described by the `interrupts` property
  - `s->irq = platform_get_irq(pdev, 0);`
- Get a DMA channel
  - described by the `dmas` property
  - `s->rx_dma_chan = dma_request_slave_channel(s->dev, "rx");`
  - `s->tx_dma_chan = dma_request_slave_channel(s->dev, "tx");`
- Check some custom property
  - `struct device_node *np = pdev->dev.of_node;`
  - `if (of_get_property(np, "fsl,uart-has-rtscts", NULL))`

Device Tree inclusion

- Device Tree files are not monolithic, they can be split in several files, including each other.
- `.dtsi` files are included files, while `.dts` files are *final* Device Trees
- Typically, `.dtsi` will contain definition of SoC-level information (or sometimes definitions common to several almost identical boards).
- The `.dts` file contains the board-level information.
- The inclusion works by **overlaying** the tree of the including file over the tree of the included file.
- Inclusion using the DT operator `/include/`, or since a few kernel releases, the DTS go through the C preprocessor, so `#include` is recommended.

Device Tree inclusion example

Definition of the AM33xx SoC

```
/ {
	compatible = "ti,am33xx";
	[...]
	ocp {
		uart0: serial@44e09000 {
			compatible = "ti,omap3-uart";
			reg = <0x44e09000 0x2000>;
			interrupts = <72>;
			status = "disabled";
		};
	};
};
```

am33xx.dtsi

+

Definition of the BeagleBone board

```
#include "am33xx.dtsi"

/ {
	compatible = "ti,am335x-bone", "ti,am33xx";
	[...]
	ocp {
		uart0: serial@44e09000 {
			pinctrl-names = "default";
			pinctrl-0 = <&uart0_pins>;
			status = "okay";
		};
	};
};
```

am335x-bone.dts

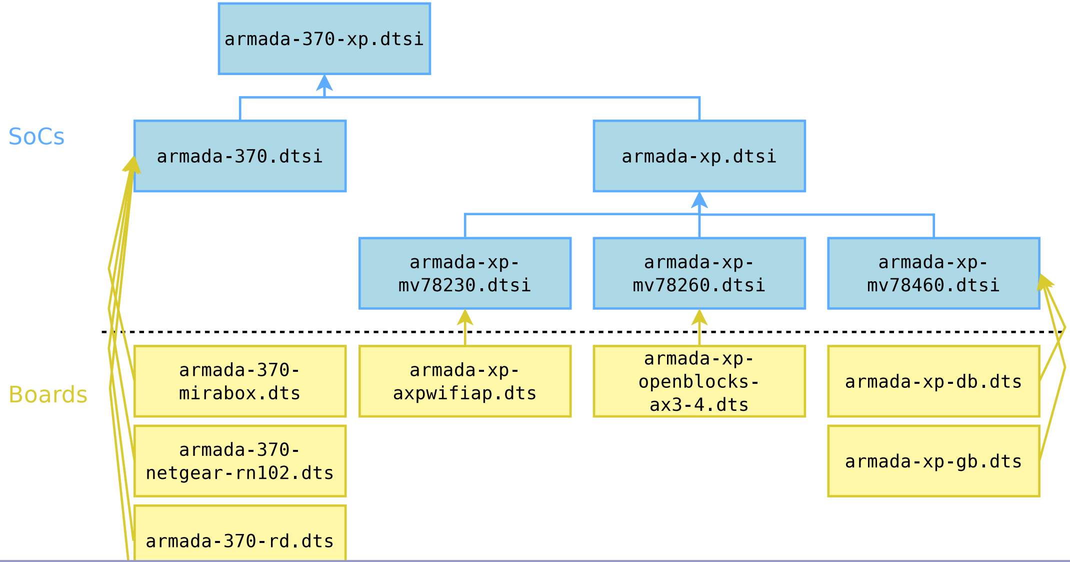
=

Compiled DTB

```
/ {
	compatible = "ti,am335x-bone", "ti,am33xx";
	[...]
	ocp {
		uart0: serial@44e09000 {
			compatible = "ti,omap3-uart";
			reg = <0x44e09000 0x2000>;
			interrupts = <72>;
			pinctrl-names = "default";
```

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

Device Tree inclusion example (2)



Concept of Device Tree binding

- Quoting the *ePAPR*:
  - This chapter contains requirements, known as **bindings, for how specific types and classes of devices are represented in the device tree**.
  - The `compatible` property of a device node describes the specific binding (or bindings) to which the node complies.
  - When creating a new device tree representation for a device, a **binding should be created that fully describes the required properties and value of the device**. This set of properties shall be sufficiently descriptive to provide device drivers with needed attributes of the device.

Documentation of Device Tree bindings

- All Device Tree bindings recognized by the kernel are documented in `Documentation/devicetree/bindings`.
- Each binding documentation described which properties are accepted, with which values, which properties are mandatory vs. optional, etc.
- All new Device Tree bindings must be reviewed by the *Device Tree maintainers*, by being posted to `devicetree@vger.kernel.org`. This ensures correctness and consistency across bindings.

```
OPEN FIRMWARE AND FLATTENED DEVICE TREE BINDINGS
M:      Rob Herring <rob.herring@calxeda.com>
M:      Pawel Moll <pawel.moll@arm.com>
M:      Mark Rutland <mark.rutland@arm.com>
M:      Stephen Warren <swarren@wwwdotorg.org>
M:      Ian Campbell <ijc+devicetree@hellion.org.uk>
L:      devicetree@vger.kernel.org
```

Device Tree binding documentation example

```
* Freescale MXS Application UART (AUART)

Required properties:
- compatible : Should be "fsl,<soc>-auart". The supported SoCs include
  imx23 and imx28.
- reg : Address and length of the register set for the device
- interrupts : Should contain the auart interrupt numbers
- dmas: DMA specifier, consisting of a phandle to DMA controller node
  and AUART DMA channel ID.
  Refer to dma.txt and fsl-mxs-dma.txt for details.
- dma-names: "rx" for RX channel, "tx" for TX channel.

Example:
auart0: serial@8006a000 {
        compatible = "fsl,imx28-auart", "fsl,imx23-auart";
        reg = <0x8006a000 0x2000>;
        interrupts = <112>;
        dmas = <&dma_apbx 8>, <&dma_apbx 9>;
        dma-names = "rx", "tx";
};

Note: Each auart port should have an alias correctly numbered in "aliases"
node.

Example:
[...]
```

Documentation/devicetree/bindings/tty/serial/fsl-mxs-auart.txt

Device Tree organization: top-level nodes

Under the root of the Device Tree, one typically finds the following top-level nodes:

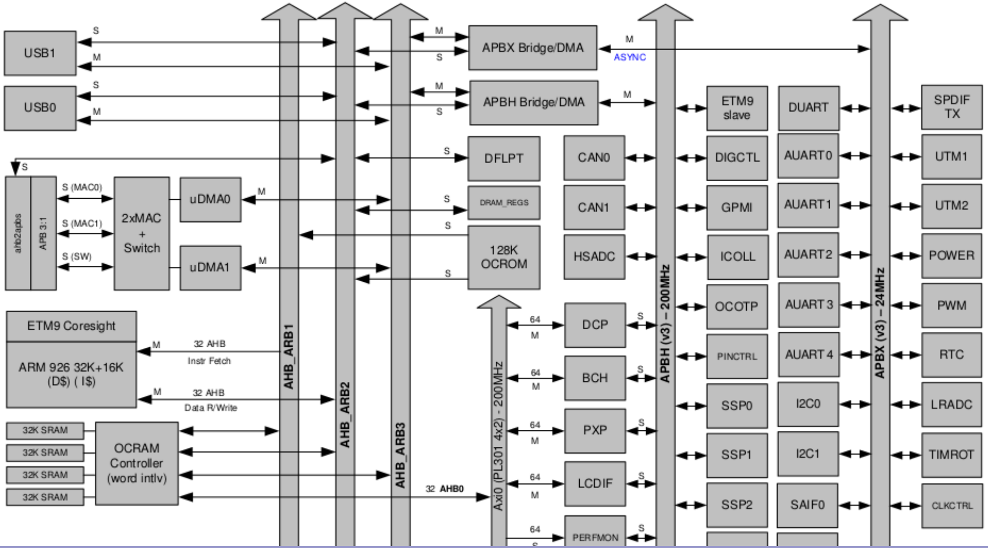
- A `cpus` node, which sub-nodes describing each CPU in the system.
- A `memory` node, which defines the location and size of the RAM.
- A `chosen` node, which defines *parameters chosen or defined by the system firmware at boot time*. In practice, one of its usage is to pass the kernel command line.
- A `aliases` node, to define shortcuts to certain nodes.
- One or more nodes defining the *buses* in the SoC.
- One or mode nodes defining on-board devices.

Device Tree organization: imx28.dtsi

arch/arm/boot/dts/imx28.dtsi

```
/ {
        aliases { ... };
        cpus { ... };

        apb@80000000 {
                apbh@80000000 {
                        /* Some devices */
                };

                apbx@80040000 {
                        /* Some devices */
                };
        };

        ahb@80080000 {
                /* Some devices */
        };
};
```

i.MX28 buses organization



Device Tree organization: imx28-evk.dts

arch/arm/boot/dts/imx28-evk.dts

```
/ {
        model = "Freescale i.MX28 Evaluation Kit";
        compatible = "fsl,imx28-evk", "fsl,imx28";

        memory {
                reg = <0x40000000 0x08000000>;
        };


        apb@80000000 {
                apbh@80000000 { ... };
                apbx@80040000 { ... };
        };

        ahb@80080000 { ... };

        sound { ... };
        leds { ... };
        backlight { ... };
};
```

Top-level compatible property

- The top-level `compatible` property typically defines a compatible string for the board, and then for the SoC.
- Values always given with the most-specific first, to least-specific last.
- Used to match with the `dt_compat` field of the `DT_MACHINE` structure:

```
static const char *mxs_dt_compat[] __initdata = {
        "fsl,imx28",
        "fsl,imx23",
        NULL,
};

DT_MACHINE_START(MXS, "Freescale MXS (Device Tree)")
        .dt_compat      = mxs_dt_compat,
        [...]
MACHINE_END
```

- Can also be used within code to test the machine:

```
if (of_machine_is_compatible("fsl,imx28-evk"))
        imx28_evk_init();
```

Bus, address cells and size cells

Inside a bus, one typically needs to define the following properties:

- A `compatible` property, which identifies the bus controller (in case of I2C, SPI, PCI, etc.). A special value `compatible = "simple-bus"` means a simple memory-mapped bus with no specific handling or driver. Child nodes will be registered as *platform devices*.
- The `#address-cells` property indicate how many cells (i.e 32 bits values) are needed to form the base address part in the `reg` property.
- The `#size-cells` is the same, for the size part of the `reg` property.
- The `ranges` property can describe an *address translation* between the child bus and the parent bus. When simply defined as `ranges;`, it means that the translation is an identity translation.

simple-bus, address cells and size cells

```
apbh@80000000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        reg = <0x80000000 0x3c900>;
        ranges;

        [...]

        hsadc: hsadc@80002000 {
                reg = <0x80002000 0x2000>;
                interrupts = <13>;
                dmas = <&dma_apbh 12>;
                dma-names = "rx";
                status = "disabled";
        };

        [...]
};
```

I2C bus, address cells and size cells

```
i2c0: i2c@80058000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx28-i2c";
        reg = <0x80058000 0x2000>;
        interrupts = <111>;
        [...]

        sgtl5000: codec@0a {
                compatible = "fsl,sgtl5000";
                reg = <0x0a>;
                VDDA-supply = <&reg_3p3v>;
                VDDIO-supply = <&reg_3p3v>;
                clocks = <&saif0>;
        };

        at24@51 {
                compatible = "at24,24c32";
                pagesize = <32>;
                reg = <0x51>;
        };
};
```

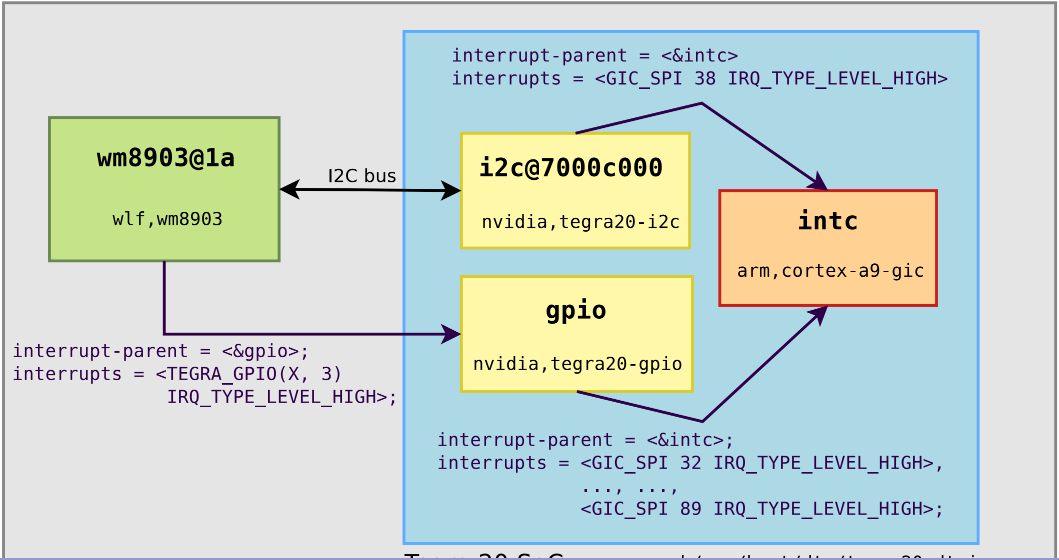
Interrupt handling

- `interrupt-controller;` is a boolean property that indicates that the current node is an interrupt controller.
- `#interrupt-cells` indicates the number of cells in the `interrupts` property for the interrupts managed by the selected interrupt controller.
- `interrupt-parent` is a *phandle* that points to the interrupt controller for the current node. There is generally a top-level `interrupt-parent` definition for the main interrupt controller.

Interrupt example: imx28.dtsi

```
/ {
    interrupt-parent = <&icoll>;
    apb@80000000 {
        apbh@80000000 {
            icoll: interrupt-controller@80000000 {
                compatible = "fsl,imx28-icoll", "fsl,icoll";
                interrupt-controller;
                #interrupt-cells = <1>;
                reg = <0x80000000 0x2000>;
            };

            ssp0: ssp@80010000 {
                [...]
                interrupts = <96>;
            };
        };
    };
};
```

A more complicated example on Tegra 20



Interrupt example: tegra20.dtsi

```
/ {
  interrupt-parent = <&intc>;

  intc: interrupt-controller {
    compatible = "arm,cortex-a9-gic";
    reg = <0x50041000 0x1000 0x50040100 0x0100>;
    interrupt-controller;
    #interrupt-cells = <3>;
  };

  i2c@7000c000 {
    compatible = "nvidia,tegra20-i2c";
    reg = <0x7000c000 0x100>;
    interrupts = <GIC_SPI 38 IRQ_TYPE_LEVEL_HIGH>;
    #address-cells = <1>;
    #size-cells = <0>;
    [...]
  };

  gpio: gpio {
      compatible = "nvidia,tegra20-gpio";
      reg = <0x6000d000 0x1000>;
      interrupts = <GIC_SPI 32 IRQ_TYPE_LEVEL_HIGH>, <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>,
           [...], <GIC_SPI 89 IRQ_TYPE_LEVEL_HIGH>;
      #gpio-cells = <2>;
      gpio-controller;
      #interrupt-cells = <2>;
      interrupt-controller;
  };
};
```

Interrupt example: tegra20-harmony.dts

```
i2c@7000c000 {
  status = "okay";
  clock-frequency = <400000>;

  wm8903: wm8903@1a {
    compatible = "wlf,wm8903";
    reg = <0x1a>;
    interrupt-parent = <&gpio>;
    interrupts = <TEGRA_GPIO(X, 3) IRQ_TYPE_LEVEL_HIGH>;

    gpio-controller;
    #gpio-cells = <2>;

    micdet-cfg = <0>;
    micdet-delay = <100>;
    gpio-cfg = <0xffffffff 0xffffffff 0 0xffffffff 0xffffffff>;
  };
};
```

DT is hardware description, not configuration

- The Device Tree is really a hardware description language.
- It should **describe the hardware layout**, and how it works.
- But it should **not describe which particular hardware configuration** you're interested in.
- As an example:
  - You may describe in the DT whether a particular piece of hardware supports DMA or not.
  - But you may not describe in the DT if you *want* to use DMA or not.

Device Tree Resources

- The drivers will use the same mechanism that we saw previously to retrieve basic information: interrupts numbers, physical addresses, etc.
- The available resources list will be built up by the kernel at boot time from the device tree, so that you don't need to make any unnecessary lookups to the DT when loading your driver.
- Any additional information will be specific to a driver or the class it belongs to, defining the *bindings*

sysfs

- The bus, device, drivers, etc. structures are internal to the kernel
- The `sysfs` virtual filesystem offers a mechanism to export such information to user space
- Used for example by `udev` to provide automatic module loading, firmware loading, device file creation, etc.
- `sysfs` is usually mounted in `/sys`
  - `/sys/bus/` contains the list of buses
  - `/sys/devices/` contains the list of devices
  - `/sys/class` enumerates devices by class (`net`, `input`, `block`...), whatever the bus they are connected to. Very useful!
- Take your time to explore `/sys` on your workstation.

References

- Power.orgTM Standard for Embedded Power Architecture Platform Requirements (ePAPR), `http://www.power.org/resources/downloads/Power_ePAPR_APPROVED_v1.0.pdf`
- DeviceTree.org website, `http://www.devicetree.org`
- Device Tree documentation in the kernel sources, `Documentation/devicetree`
- The Device Tree kernel mailing list, `http://dir.gmane.org/gmane.linux.drivers.devicetree`

Linux device and driver model

Bootlin
info@bootlin.com

Introduction

The need for a device model?

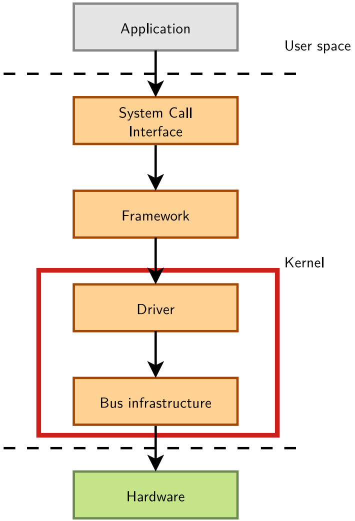
- The Linux kernel runs on a wide range of architectures and hardware platforms, and therefore needs to **maximize the reusability** of code between platforms.
- For example, we want the same *USB device driver* to be usable on a x86 PC, or an ARM platform, even though the USB controllers used on these platforms are different.
- This requires a clean organization of the code, with the *device drivers* separated from the *controller drivers*, the hardware description separated from the drivers themselves, etc.
- This is what the Linux kernel **Device Model** allows, in addition to other advantages covered in this section.

Kernel and Device Drivers

In Linux, a driver is always interfacing with:

- a **framework** that allows the driver to expose the hardware features in a generic way.
- a **bus infrastructure**, part of the device model, to detect/communicate with the hardware.

This section focuses on the *device model*, while *kernel frameworks* are covered later in this training.



Device Model data structures

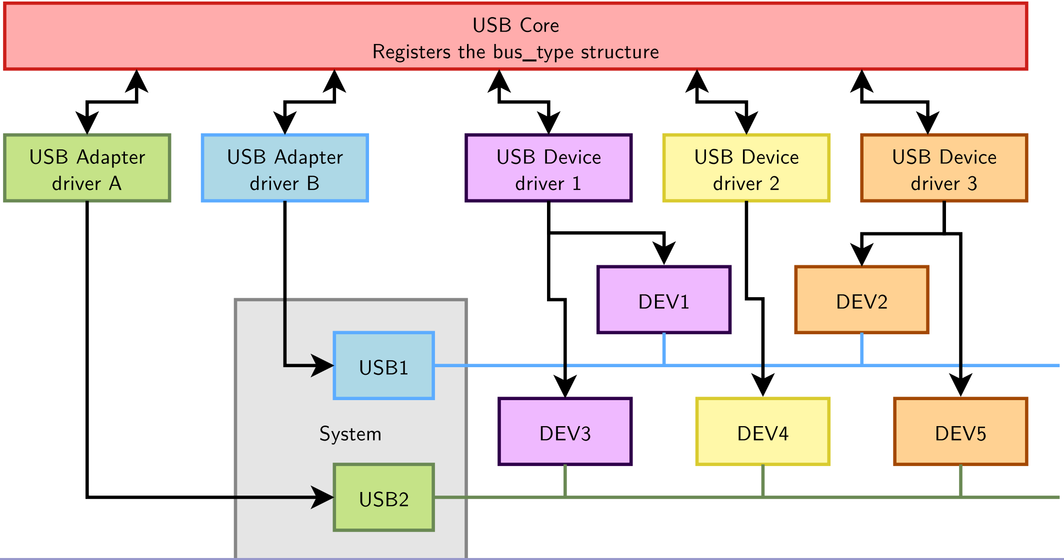
- The *device model* is organized around three main data structures:
  - The `struct bus_type` structure, which represent one type of bus (USB, PCI, I2C, etc.)
  - The `struct device_driver` structure, which represents one driver capable of handling certain devices on a certain bus.
  - The `struct device` structure, which represents one device connected to a bus
- The kernel uses inheritance to create more specialized versions of `struct device_driver` and `struct device` for each bus subsystem.
- In order to explore the device model, we will
  - First look at a popular bus that offers dynamic enumeration, the *USB bus*
  - Continue by studying how buses that do not offer dynamic enumerations are handled.

Bus Drivers

- The first component of the device model is the bus driver
  - One bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.
- It is responsible for
  - Registering the bus type (`struct bus_type`)
  - Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able to detect the connected devices, and providing a communication mechanism with the devices
  - Allowing the registration of device drivers (USB devices, I2C devices, PCI devices, etc.), managing the devices
  - Matching the device drivers against the devices detected by the adapter drivers.
  - Provides an API to both adapter drivers and device drivers
  - Defining driver and device specific structures, mainly `struct usb_driver` and `struct usb_interface`

Example of the USB bus

Example: USB Bus 1/2



Example: USB Bus 2/2

- Core infrastructure (bus driver)
  - `drivers/usb/core/`
  - `struct bus_type` is defined in `drivers/usb/core/driver.c` and registered in `drivers/usb/core/usb.c`
- Adapter drivers
  - `drivers/usb/host/`
  - For EHCI, UHCI, OHCI, XHCI, and their implementations on various systems (Atmel, IXP, Xilinx, OMAP, Samsung, PXA, etc.)
- Device drivers
  - Everywhere in the kernel tree, classified by their type

Example of Device Driver

- To illustrate how drivers are implemented to work with the device model, we will study the source code of a driver for a USB network card
  - It is USB device, so it has to be a USB device driver
  - It is a network device, so it has to be a network device
  - Most drivers rely on a bus infrastructure (here, USB) and register themselves in a framework (here, network)
- We will only look at the device driver side, and not the adapter driver side
- The driver we will look at is `drivers/net/usb/rtl8150.c`

Device Identifiers

- Defines the set of devices that this driver can manage, so that the USB core knows for which devices this driver should be used
- The `MODULE_DEVICE_TABLE()` macro allows `depmod` to extract at compile time the relation between device identifiers and drivers, so that drivers can be loaded automatically by `udev`. See `/lib/modules/$(uname -r)/modules.{alias,usbmap}`

```
static struct usb_device_id rtl8150_table[] = {
    { USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150) },
    { USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX) },
    { USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR) },
    { USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX) },
    [...]
    {}
};
MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Instantiation of usb_driver

- struct usb_driver is a structure defined by the USB core. Each USB device driver must instantiate it, and register itself to the USB core using this structure
- This structure inherits from struct device_driver, which is defined by the device model.

```
static struct usb_driver rtl8150_driver = {
    .name = "rtl8150",
    .probe = rtl8150_probe,
    .disconnect = rtl8150_disconnect,
    .id_table = rtl8150_table,
    .suspend = rtl8150_suspend,
    .resume = rtl8150_resume
};
```

Driver (Un)Registration

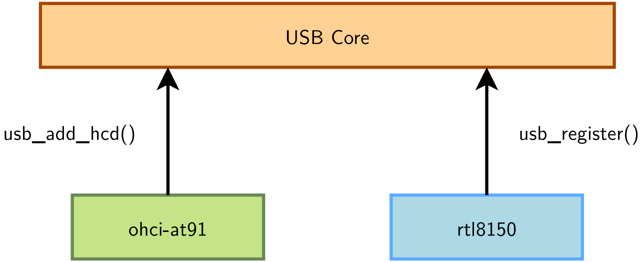
- When the driver is loaded or unloaded, it must register or unregister itself from the USB core
- Done using `usb_register()` and `usb_deregister()`, provided by the USB core.

```
static int __init usb_rtl8150_init(void)
{
    return usb_register(&rtl8150_driver);
}

static void __exit usb_rtl8150_exit(void)
{
    usb_deregister(&rtl8150_driver);
}

module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

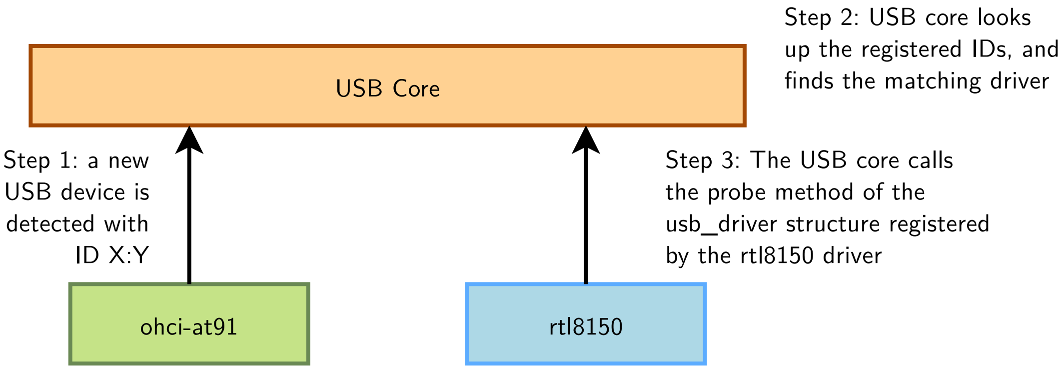
- Note: this code has now been replaced by a shorter `module_usb_driver()` macro call.

At Initialization

- The USB adapter driver that corresponds to the USB controller of the system registers itself to the USB core
- The `rtl8150` USB device driver registers itself to the USB core



- The USB core now knows the association between the vendor/product IDs of `rtl8150` and the `struct usb_driver` structure of this driver

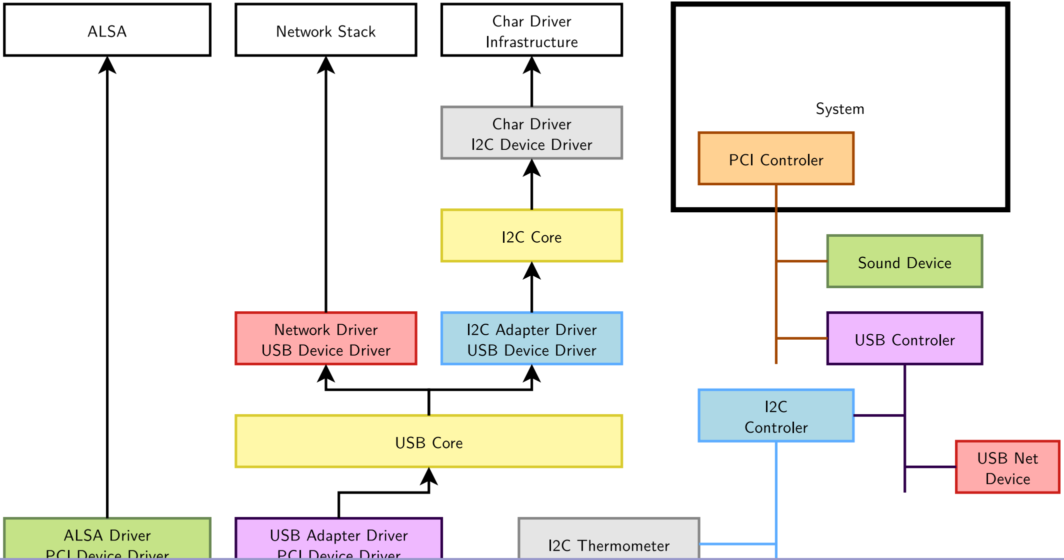

USB Core
Step 2: USB core looks up the registered IDs, and finds the matching driver
Step 1: a new USB device is detected with ID X:Y
Step 3: The USB core calls the probe method of the usb_driver structure registered by the rtl8150 driver
ohci-at91
rtl8150

- The `probe()` method receives as argument a structure describing the device, usually specialized by the bus infrastructure (`struct pci_dev`, `struct usb_interface`, etc.)
- This function is responsible for
  - Initializing the device, mapping I/O memory, registering the interrupt handlers. The bus infrastructure provides methods to get the addresses, interrupt numbers and other device-specific information.
  - Registering the device to the proper kernel framework, for example the network infrastructure.

Probe Method Example

```
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    [...]
    dev = netdev_priv(netdev);
    tasklet_init(&dev->tl, rx_fixup, (unsigned long)dev);
    spin_lock_init(&dev->rx_pool_lock);
    [...]
    netdev->netdev_ops = &rtl8150_netdev_ops;
    alloc_all_urbs(dev);
    [...]
    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);
    register_netdev(netdev);

    return 0;
}
```

The Model is Recursive



Introduction to the I2C subsystem

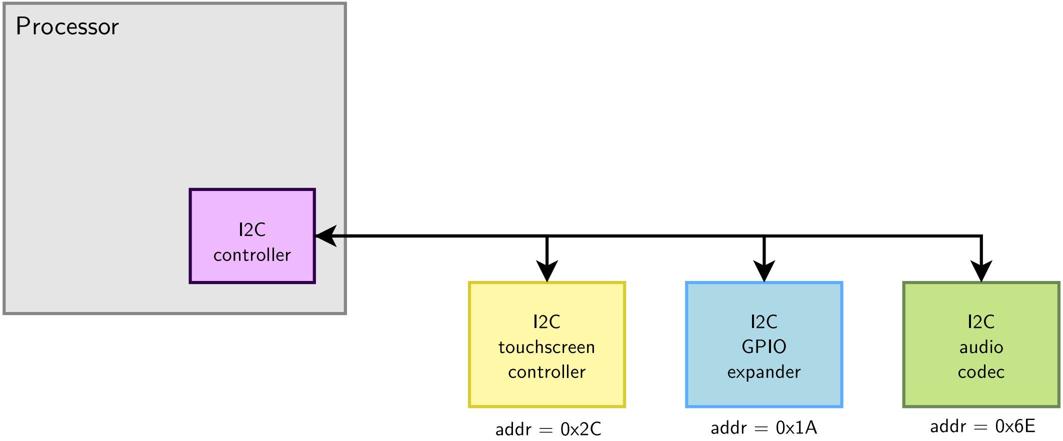
Bootlin
info@bootlin.com

embedded Linux and kernel engineering

What is I2C?

- A very commonly used low-speed bus to connect on-board devices to the processor.
- Uses only two wires: SDA for the data, SCL for the clock.
- It is a master/slave bus: only the master can initiate transactions, and slaves can only reply to transactions initiated by masters.
- In a Linux system, the I2C controller embedded in the processor is typically the master, controlling the bus.
- Each slave device is identified by a unique I2C address. Each transaction initiated by the master contains this address, which allows the relevant slave to recognize that it should reply to this particular transaction.

An I2C bus example



The I2C subsystem

- Like all bus subsystems, the I2C subsystem is responsible for:
  - Providing an API to implement I2C controller drivers
  - Providing an API to implement I2C device drivers, in kernel space
  - Providing an API to implement I2C device drivers, in user space
- The core of the I2C subsystem is located in `drivers/i2c/`.
- The I2C controller drivers are located in `drivers/i2c/busses/`.
- The I2C device drivers are located throughout `drivers//`, depending on the type of device (ex: `drivers/input/` for input devices).

Registering an I2C device driver

- Like all bus subsystems, the I2C subsystem defines a `struct i2c_driver` that inherits from `struct device_driver`, and which must be instantiated and registered by each I2C device driver.
  - As usual, this structure points to the `->probe()` and `->remove()` functions.
  - It also contains an `id_table` field that must point to a list of *device IDs* (which is a list of tuples containing a string and some private driver data). It is used for non-DT based probing of I2C devices.
- The `i2c_add_driver()` and `i2c_del_driver()` functions are used to register/unregister the driver.
- If the driver doesn't do anything else in its `init()`/`exit()` functions, it is advised to use the `module_i2c_driver()` macro instead.

Registering an I2C device driver: example

```
static const struct i2c_device_id <driver>_id[] = {
        { "<device-name>", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, <driver>_id);

#ifdef CONFIG_OF
static const struct of_device_id <driver>_dt_ids[] = {
        { .compatible = "<vendor>,<device-name>", },
        { }
};
MODULE_DEVICE_TABLE(of, <driver>_dt_ids);
#endif

static struct i2c_driver <driver>_driver = {
        .probe          = <driver>_probe,
        .remove         = <driver>_remove,
        .id_table       = <driver>_id,
        .driver = {
                .name   = "<driver-name>",
                .owner  = THIS_MODULE,
                .of_match_table = of_match_ptr(<driver>_dt_ids),
        },
};

module_i2c_driver(<driver>_driver);
```

Registering an I2C device: non-DT

- On non-DT platforms, the `struct i2c_board_info` structure allows to describe how an I2C device is connected to a board.
- Such structures are normally defined with the `I2C_BOARD_INFO()` helper macro.
  - Takes as argument the device name and the slave address of the device on the bus.
- An array of such structures is registed on a per-bus basis using `i2c_register_board_info()`, when the platform is initialized.

Registering an I2C device, non-DT example

```
static struct i2c_board_info <board>_i2c_devices[] __initdata = {
        {
                I2C_BOARD_INFO("cs42l51", 0x4a),
        },
};

void board_init(void)
{
        /*
         * Here should be the registration of all devices, including
         * the I2C controller device.
         */

        i2c_register_board_info(0, <board>_i2c_devices,
                                ARRAY_SIZE(<board>_i2c_devices));

        /* More devices registered here */
}
```

Registering an I2C device, in the DT

- In the Device Tree, the I2C controller device is typically defined in the `.dtsi` file that describes the processor.
  - Normally defined with `status = "disabled"`.
- At the board/platform level:
  - the I2C controller device is enabled (`status = "okay"`)
  - the I2C bus frequency is defined, using the `clock-frequency` property.
  - the I2C devices on the bus are described as children of the I2C controller node, where the `reg` property gives the I2C slave address on the bus.

Registering an I2C device, DT example (1/2)

Definition of the I2C controller, .dtsi file

```
i2c@7000c000 {
        compatible = "nvidia,tegra20-i2c";
        reg = <0x7000c000 0x100>;
        interrupts = <GIC_SPI 38 IRQ_TYPE_LEVEL_HIGH>;
        #address-cells = <1>;
        #size-cells = <0>;
        clocks = <&tegra_car TEGRA20_CLK_I2C1>,
                 <&tegra_car TEGRA20_CLK_PLL_P_OUT3>;
        clock-names = "div-clk", "fast-clk";
        status = "disabled";
};
```

Registering an I2C device, DT example (2/2)

Definition of the I2C device, .dts file

```
i2c@7000c000 {
        status = "okay";
        clock-frequency = <400000>;

        alc5632: alc5632@1e {
                compatible = "realtek,alc5632";
                reg = <0x1e>;
                gpio-controller;
                #gpio-cells = <2>;
        };
};
```

probe() and remove()

- The ->probe() function is responsible for initializing the device and registering it in the appropriate kernel framework. It receives as argument:
  - A struct i2c_client pointer, which represents the I2C device itself. This structure inherits from struct device.
  - A struct i2c_device_id pointer, which points to the I2C device ID entry that matched the device that is being probed.
- The ->remove() function is responsible for unregistering the device from the kernel framework and shut it down. It receives as argument:
  - The same struct i2c_client pointer that was passed as argument to ->probe()

Probe/remove example

```
static int <driver>_probe(struct i2c_client *client,
                          const struct i2c_device_id *id)
{
        /* initialize device */
        /* register to a kernel framework */

        i2c_set_clientdata(client, <private data>);
        return 0;
}

static int <driver>_remove(struct i2c_client *client)
{
        <private data> = i2c_get_clientdata(client);
        /* unregister device from kernel framework */
        /* shut down the device */
        return 0;
}
```

Communicating with the I2C device: raw API

The most **basic API** to communicate with the I2C device provides functions to either send or receive data:

- `int i2c_master_send(struct i2c_client *client, const char *buf, int count);`
  Sends the contents of `buf` to the client.
- `int i2c_master_recv(struct i2c_client *client, char *buf, int count);`
  Receives `count` bytes from the client, and store them into `buf`.

Communicating with the I2C device: message transfer

The message transfer API allows to describe **transfers** that consists of several **messages**, with each message being a transaction in one direction:

- `int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg, int num);`
- The `struct i2c_adapter` pointer can be found by using `client->adapter`
- The `struct i2c_msg` structure defines the length, location, and direction of the message.

I2C: message transfer example

```
struct i2c_msg msg[2];
int error;
u8 start_reg;
u8 buf[10];

msg[0].addr = client->addr;
msg[0].flags = 0;
msg[0].len = 1;
msg[0].buf = &start_reg;
start_reg = 0x10;

msg[1].addr = client->addr;
msg[1].flags = I2C_M_RD;
msg[1].len = sizeof(buf);
msg[1].buf = buf;

error = i2c_transfer(client->adapter, msg, 2);
```

SMBus calls

- SMBus is a subset of the I2C protocol.
- It defines a standard set of transactions, for example to read or write a register into a device.
- Linux provides SMBus functions that *should be used* instead of the raw API, if the I2C device supports this standard type of transactions. The driver can then be used on both SMBus and I2C adapters (can't use I2C commands on SMBus adapters).
- Example: the `i2c_smbus_read_byte_data()` function allows to read one byte of data from a device register.
  - It does the following operations:
    `S Addr Wr [A] Comm [A] S Addr Rd [A] [Data] NA P`
  - Which means it first writes a one byte data command (*Comm*), and then reads back one byte of data (*[Data]*).
- See `Documentation/i2c/smbus-protocol` for details.

List of SMBus functions

- Read/write one byte
  - `s32 i2c_smbus_read_byte(const struct i2c_client *client);`
  - `s32 i2c_smbus_write_byte(const struct i2c_client *client, u8 value);`
- Write a command byte, and read or write one byte
  - `s32 i2c_smbus_read_byte_data(const struct i2c_client *client, u8 command);`
  - `s32 i2c_smbus_write_byte_data(const struct i2c_client *client, u8 command, u8 value);`
- Write a command byte, and read or write one word
  - `s32 i2c_smbus_read_word_data(const struct i2c_client *client, u8 command);`
  - `s32 i2c_smbus_write_word_data(const struct i2c_client *client, u8 command, u16 value);`
- Write a command byte, and read or write a block of data (max 32 bytes)
  - `s32 i2c_smbus_read_block_data(const struct i2c_client *client, u8 command, u8 *values);`
  - `s32 i2c_smbus_write_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values);`
- Write a command byte, and read or write a block of data (no limit)
  - `s32 i2c_smbus_read_i2c_block_data(const struct i2c_client *client, u8 command, u8 length, u8 *values);`
  - `s32 i2c_smbus_write_i2c_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values);`

I2C functionality

- Not all I2C controllers support all functionalities.
- The I2C controller drivers therefore tell the I2C core which functionalities they support.
- An I2C device driver must check that the functionalities they need are provided by the I2C controller in use on the system.
- The `i2c_check_functionality()` function allows to make such a check.
- Examples of functionalities: `I2C_FUNC_I2C` to be able to use the raw I2C functions, `I2C_FUNC_SMBUS_BYTE_DATA` to be able to use SMBus commands to write a command and read/write one byte of data.
- See `include/uapi/linux/i2c.h` for the full list of existing functionalities.

References

- http://en.wikipedia.org/wiki/I2C, general presentation of the I2C protocol
- Documentation/i2c/, details about the Linux support for I2C
  - writing-clients, how to write I2C device drivers
  - instantiating-devices, how to instantiate devices
  - smbus-protocol, details on the SMBus functions
  - functionality, how the functionality mechanism works
  - and many more documentation files
- http://bootlin.com/pub/video/2012/elce/elce-2012-anders-board-bringup-i2c.webm, excellent talk: *You, me and I2C* from David Anders at ELCE 2012.

Kernel frameworks for device drivers

Bootlin
info@bootlin.com

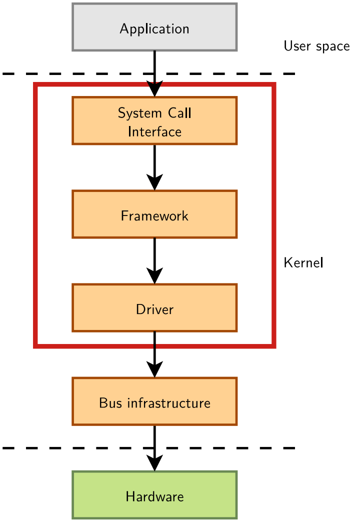
bootlin
embedded Linux and kernel engineering

Kernel and Device Drivers

In Linux, a driver is always interfacing with:

- a **framework** that allows the driver to expose the hardware features to user space applications.
- a **bus infrastructure**, part of the device model, to detect/communicate with the hardware.

This section focuses on the *kernel frameworks*, while the *device model* was covered earlier in this training.



User space vision of devices

Types of devices

Under Linux, there are essentially three types of devices:

- **Network devices**. They are represented as network interfaces, visible in user space using `ifconfig`.
- **Block devices**. They are used to provide user space applications access to raw storage devices (hard disks, USB keys). They are visible to the applications as *device files* in `/dev`.
- **Character devices**. They are used to provide user space applications access to all other types of devices (input, sound, graphics, serial, etc.). They are also visible to the applications as *device files* in `/dev`.

→ Most devices are *character devices*, so we will study these in more details.

Major and minor numbers

- Within the kernel, all block and character devices are identified using a *major* and a *minor* number.
- The *major number* typically indicates the family of the device.
- The *minor number* typically indicates the number of the device (when they are for example several serial ports)
- Most major and minor numbers are statically allocated, and identical across all Linux systems.
- They are defined in `admin-guide/devices`.

Devices: everything is a file

- A very important Unix design decision was to represent most of the "system objects" as files
- It allows applications to manipulate all "system objects" with the normal file API (`open`, `read`, `write`, `close`, etc.)
- So, devices had to be represented as files to the applications
- This is done through a special artifact called a **device file**
- It is a special type of file, that associates a file name visible to user space applications to the triplet *(type, major, minor)* that the kernel understands
- All *device files* are by convention stored in the `/dev` directory

Device files examples

Example of device files in a Linux system

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda1 /dev/sda2 /dev/zero
brw-rw---- 1 root disk    8,  1 2011-05-27 08:56 /dev/sda1
brw-rw---- 1 root disk    8,  2 2011-05-27 08:56 /dev/sda2
crw------- 1 root root    4,  1 2011-05-27 08:57 /dev/tty1
crw-rw---- 1 root dialout 4, 64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root    1,  5 2011-05-27 08:56 /dev/zero
```

Example C code that uses the usual file API to write data to a serial port

```c
int fd;
fd = open("/dev/ttyS0", O_RDWR);
write(fd, "Hello", 5);
close(fd);
```

Creating device files

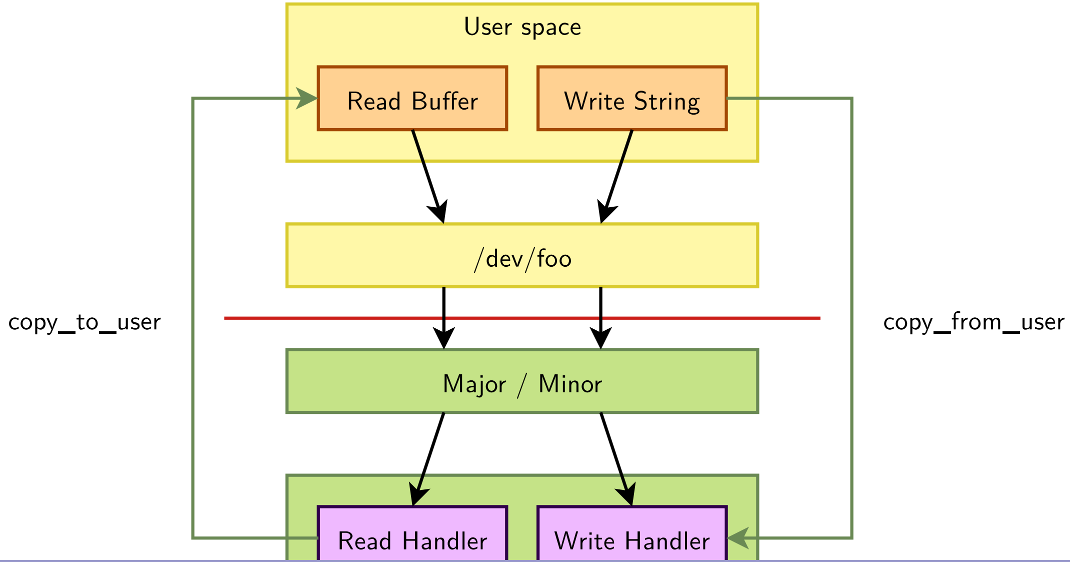
- On a basic Linux system, the device files have to be created manually using the `mknod` command
  - `mknod /dev/<device> [c|b] major minor`
  - Needs root privileges
  - Coherency between device files and devices handled by the kernel is left to the system developer
- On more elaborate Linux systems, mechanisms can be added to create/remove them automatically when devices appear and disappear
  - `devtmpfs` virtual filesystem
  - `udev` daemon, solution used by desktop and server Linux systems
  - `mdev` program, a lighter solution than `udev`

Character drivers

A character driver in the kernel

- From the point of view of an application, a *character device* is essentially a **file**.
- The driver of a character device must therefore implement **operations** that let applications think the device is a file: `open`, `close`, `read`, `write`, etc.
- In order to achieve this, a character driver must implement the operations described in the `struct file_operations` structure and register them.
- The Linux filesystem layer will ensure that the driver's operations are called when a user space application makes the corresponding system call.

From user space to the kernel: character devices



File operations

- Here are the most important operations for a character driver. All of them are optional.

```
#include <linux/fs.h>

struct file_operations {
    ssize_t (*read) (struct file *, char __user *,
        size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
        size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int,
        unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
};
```

open() and release()

- `int foo_open(struct inode *i, struct file *f)`
  - Called when user space opens the device file.
  - `struct inode` is a structure that uniquely represents a file in the system (be it a regular file, a directory, a symbolic link, a character or block device)
  - `struct file` is a structure created every time a file is opened. Several file structures can point to the same `inode` structure.
    - Contains information like the current position, the opening mode, etc.
    - Has a `void *private_data` pointer that one can freely use.
    - A pointer to the `file` structure is passed to all other operations
- `int foo_release(struct inode *i, struct file *f)`
  - Called when user space closes the file.

read()

- `ssize_t foo_read(struct file *f, char __user *buf, size_t sz, loff_t *off)`
  - Called when user space uses the `read()` system call on the device.
  - Must read data from the device, write at most `sz` bytes in the user space buffer `buf`, and update the current position in the file `off`. `f` is a pointer to the same file structure that was passed in the `open()` operation
  - Must return the number of bytes read.
  - On UNIX, `read()` operations typically block when there isn't enough data to read from the device

write()

- `ssize_t foo_write(struct file *f, const char __user *buf, size_t sz, loff_t *off)`
  - Called when user space uses the `write()` system call on the device
  - The opposite of `read`, must read at most `sz` bytes from `buf`, write it to the device, update `off` and return the number of bytes written.

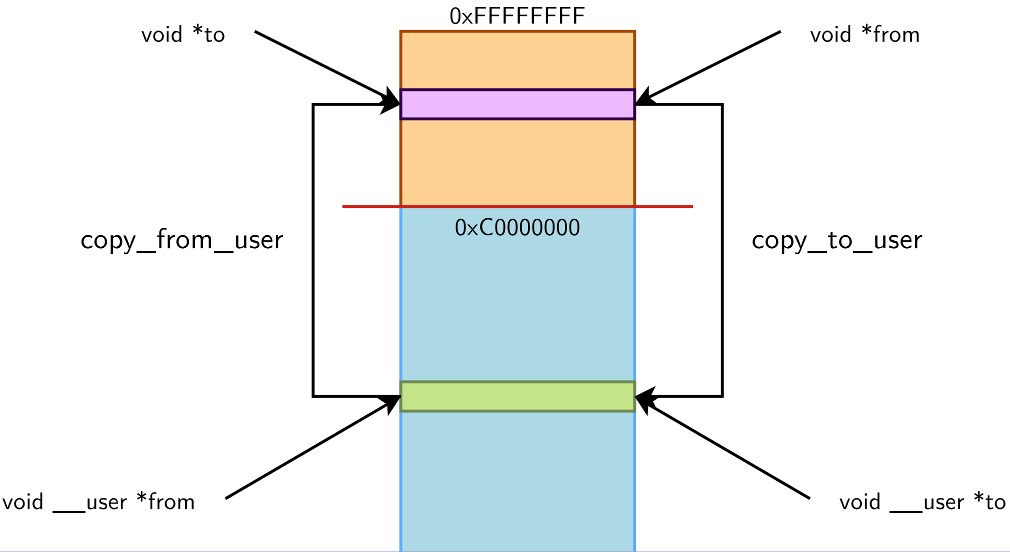
Exchanging data with user space 1/3

- Kernel code isn't allowed to directly access user space memory, using `memcpy()` or direct pointer dereferencing
  - Doing so does not work on some architectures
  - If the address passed by the application was invalid, the application would segfault.
- To keep the kernel code portable and have proper error handling, your driver must use special kernel functions to exchange data with user space.

Exchanging data with user space 2/3

- A single value
  - `get_user(v, p);`
    - The kernel variable `v` gets the value pointed by the user space pointer `p`
  - `put_user(v, p);`
    - The value pointed by the user space pointer `p` is set to the contents of the kernel variable `v`.
- A buffer
  - ```
    unsigned long copy_to_user(void __user *to,
    const void *from, unsigned long n);
    ```
  - ```
    unsigned long copy_from_user(void *to,
    const void __user *from, unsigned long n);
    ```
- The return value must be checked. Zero on success, non-zero on failure. If non-zero, the convention is to return `-EFAULT`.

Exchanging data with user space 3/3



Zero copy access to user memory

- Having to copy data to or from an intermediate kernel buffer can become expensive when the amount of data to transfer is large (video).
- *Zero copy* options are possible:
  - `mmap()` system call to allow user space to directly access memory mapped I/O space. See our `mmap()` chapter.
  - `get_user_pages_fast()` to get a mapping to user pages without having to copy them. See `http://j.mp/1sML71P` (Kernel API doc). This API is more complex to use though.

unlocked_ioctl()

- `long unlocked_ioctl(struct file *f, unsigned int cmd, unsigned long arg)`
  - Associated to the `ioctl()` system call.
  - Called unlocked because it didn't hold the Big Kernel Lock (gone now).
  - Allows to extend the driver capabilities beyond the limited read/write API.
  - For example: changing the speed of a serial port, setting video output format, querying a device serial number...
  - `cmd` is a number identifying the operation to perform
  - `arg` is the optional argument passed as third argument of the `ioctl()` system call. Can be an integer, an address, etc.
  - The semantic of `cmd` and `arg` is driver-specific.

ioctl() example: kernel side

```c
static long phantom_ioctl(struct file *file, unsigned int cmd,
    unsigned long arg)
{
    struct phm_reg r;
    void __user *argp = (void __user *)arg;

    switch (cmd) {
    case PHN_SET_REG:
        if (copy_from_user(&r, argp, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    case PHN_GET_REG:
        if (copy_to_user(argp, &r, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    default:
        return -ENOTTY;
    }

    return 0; }
```

Selected excerpt from `drivers/misc/phantom.c`

Ioctl() Example: Application Side

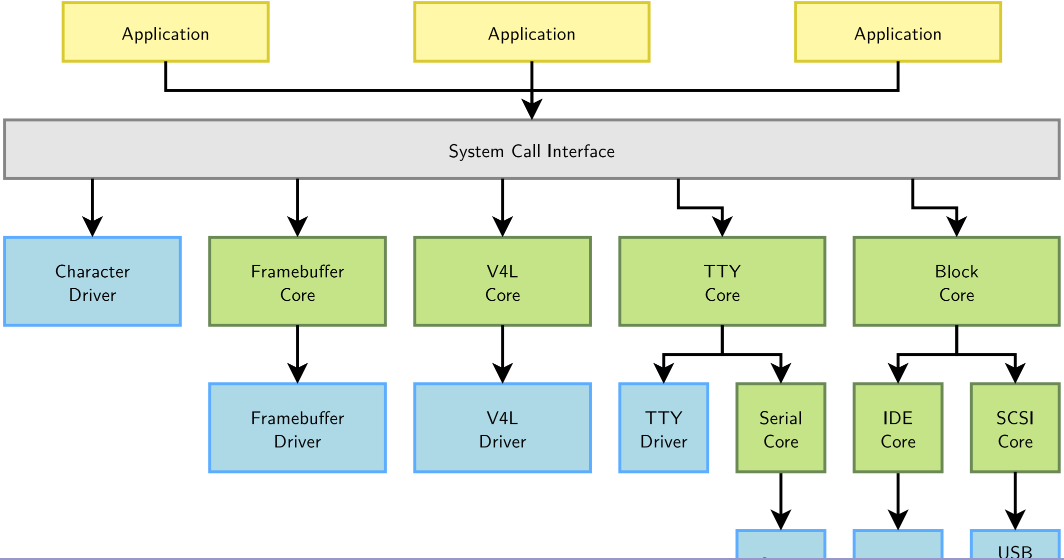
```
int main(void)
{
    int fd, ret;
    struct phm_reg reg;

    fd = open("/dev/phantom");
    assert(fd > 0);

    reg.field1 = 42;
    reg.field2 = 67;

    ret = ioctl(fd, PHN_SET_REG, & reg);
    assert(ret == 0);

    return 0;
}
```

The concept of kernel frameworks

Beyond character drivers: kernel frameworks

- Many device drivers are not implemented directly as character drivers
- They are implemented under a *framework*, specific to a given device type (framebuffer, V4L, serial, etc.)
  - The framework allows to factorize the common parts of drivers for the same type of devices
  - From user space, they are still seen as character devices by the applications
  - The framework allows to provide a coherent user space interface (`ioctl`, etc.) for every type of device, regardless of the driver

Kernel Frameworks



Driver-specific Data Structure

- Each *framework* defines a structure that a device driver must register to be recognized as a device in this framework
  - `struct uart_port` for serial ports, `struct netdev` for network devices, `struct fb_info` for framebuffers, etc.
- In addition to this structure, the driver usually needs to store additional information about its device
- This is typically done
  - By subclassing the appropriate framework structure
  - By storing a reference to the appropriate framework structure
  - Or by including your information in the framework structure

Driver-specific Data Structure Examples 1/2

- i.MX serial driver: `struct imx_port` is a subclass of `struct uart_port`

```
struct imx_port {
    struct uart_port port;
    struct timer_list timer;
    unsigned int old_status;
    int txirq, rxirq, rtsirq;
    unsigned int have_rtscts:1;
    [...]
};
```

- ds1305 RTC driver: `struct ds1305` has a reference to `struct rtc_device`

```
struct ds1305 {
    struct spi_device       *spi;
    struct rtc_device       *rtc;
    [...]
};
```

Driver-specific Data Structure Examples 2/2

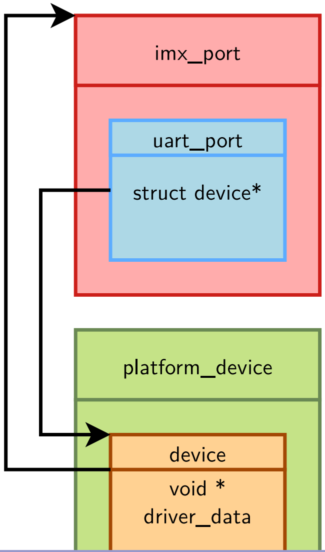
- rtl8150 network driver: `struct rtl8150` has a reference to `struct net_device` and is allocated within that framework structure.

```
struct rtl8150 {
    unsigned long flags;
    struct usb_device *udev;
    struct tasklet_struct tl;
    struct net_device *netdev;
    [...]
};
```

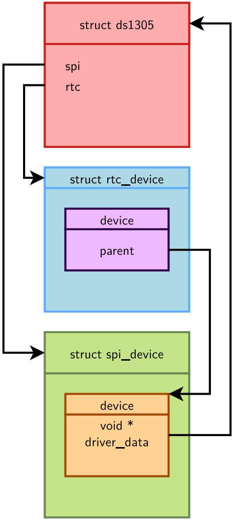
Link Between Structures 1/4

- The framework typically contains a `struct device *` pointer that the driver must point to the corresponding `struct device`
  - It's the relation between the logical device (for example a network interface) and the physical device (for example the USB network adapter)
- The device structure also contains a `void *` pointer that the driver can freely use.
  - It's often used to link back the device to the higher-level structure from the framework.
  - It allows, for example, from the `struct platform_device` structure, to find the structure describing the logical device

```c
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport;
    [...]
    /* setup the link between uart_port and the struct
     * device inside the platform_device */
    sport->port.dev = &pdev->dev;
    [...]
    /* setup the link between the struct device inside
     * the platform device to the imx_port structure */
    platform_set_drvdata(pdev, sport);
    [...]
    uart_add_one_port(&imx_reg, &sport->port);
}

static int serial_imx_remove(struct platform_device *pdev)
{
    /* retrieve the imx_port from the platform_device */
    struct imx_port *sport = platform_get_drvdata(pdev);
    [...]
    uart_remove_one_port(&imx_reg, &sport->port);
    [...]
}
```

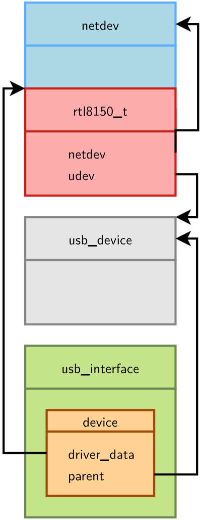


```
static int ds1305_probe(struct spi_device *spi)
{
    struct ds1305                   *ds1305;

    [...]

    /* set up driver data */
    ds1305 = devm_kzalloc(&spi->dev, sizeof(*ds1305), GFP_KERNEL);
    if (!ds1305)
            return -ENOMEM;
    ds1305->spi = spi;
    spi_set_drvdata(spi, ds1305);

    [...]

    /* register RTC ... from here on, ds1305->ctrl needs locking */
    ds1305->rtc = devm_rtc_device_register(&spi->dev, "ds1305",
                    &ds1305_ops, THIS_MODULE);

    [...]
}

static int ds1305_remove(struct spi_device *spi)
{
    struct ds1305 *ds1305 = spi_get_drvdata(spi);

    [...]
}
```



Link Between Structures 4/4

```
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(intf);
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    dev = netdev_priv(netdev);

    [...]

    dev->udev = udev;
    dev->netdev = netdev;

    [...]

    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);

    [...]
}

static void rtl8150_disconnect(struct usb_interface *intf)
{
    rtl8150_t *dev = usb_get_intfdata(intf);

    [...]
}
```



Example

`drivers/rtc/rtc-abx80x.c`

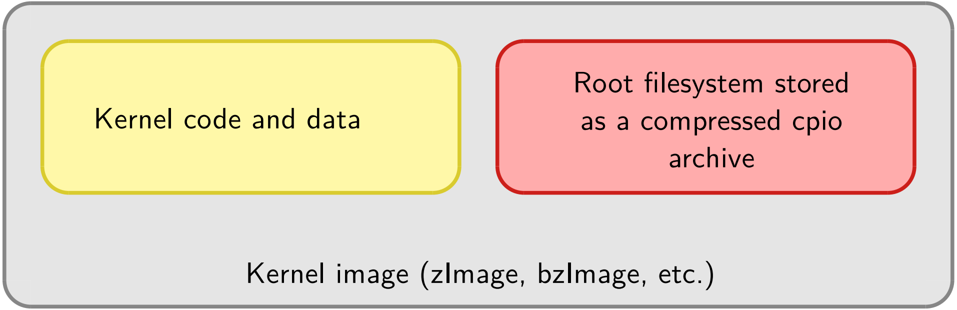
Board bringup tips

Bootlin
info@bootlin.com

Useful tips

- Use tftp
  - reduces the test cycle
  - requires Ethernet support in u-boot, it can be worth it to use an USB to Ethernet dongle.
- Use an initramfs
  - the root filesystem then reside in memory
  - it is loaded alongside the kernel by the bootloader
  - allows to boot Linux and test devices before getting proper storage support.
- Use NFS once networking is available

initramfs embedded in the kernel



initramfs embedded in the kernel

- The contents of an initramfs are defined at the kernel configuration level, with the `CONFIG_INITRAMFS_SOURCE` option
  - Can be the path to a directory containing the root filesystem contents
  - Can be the path to a cpio archive generated by your buildsystem
  - Can be a text file describing the contents of the initramfs (see documentation for details)
- The kernel build process will automatically take the contents of the `CONFIG_INITRAMFS_SOURCE` option and integrate the root filesystem into the kernel image
- Details (in kernel sources):
  `Documentation/filesystems/ramfs-rootfs-initramfs.txt`
  `Documentation/early-userspace/README`

standalone initramfs

- Use a cpio archive build using a buildsystem
- Load it from storage or network, like the kernel
- Pass the address from the boootloader to the kernel. For example using u-boot:
  `bootz 0x22000000 0x24000000 0x21000000`

Useful tools

- devmem - allows to read/write memory, in particular SoC registers
- i2c-tools - I2C utilities to probe, read and write I2C devices
- evtest - input devices debugging
- alsa-utils - sound utilities
- tslib - Touchscreen utilities, calibration and debugging
- debugfs - `sudo mount -t debugfs none /sys/kernel/debug`