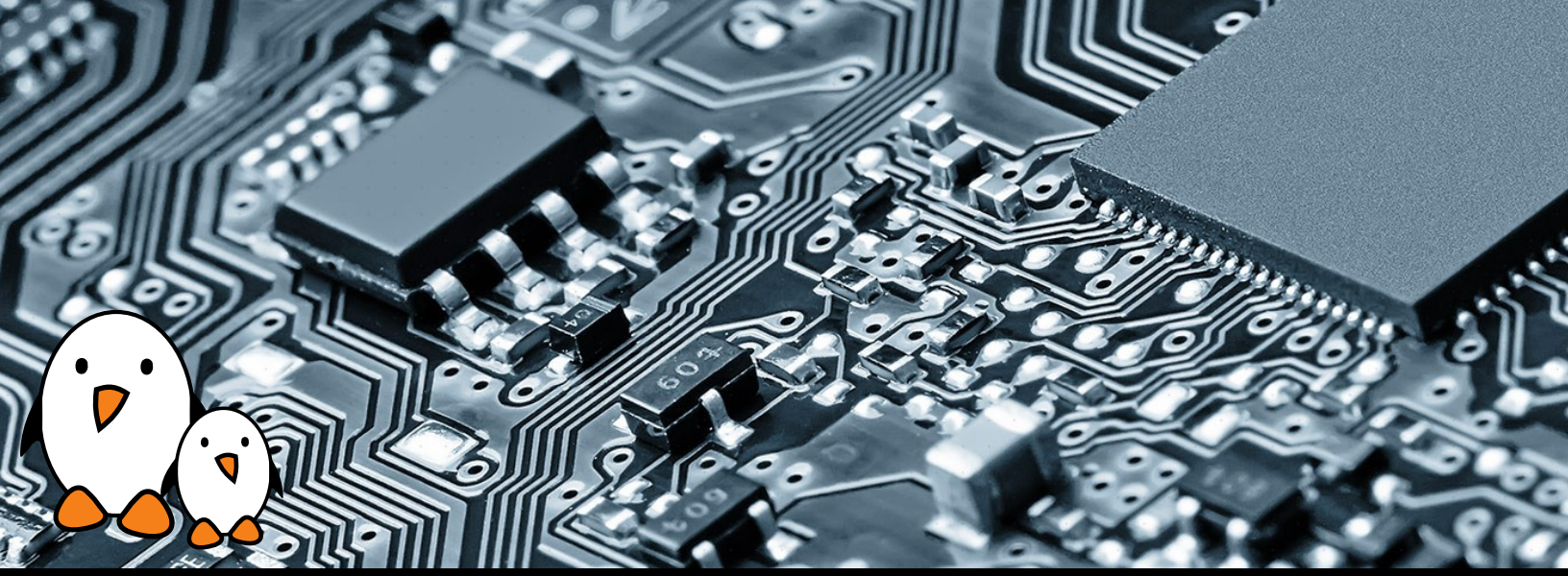

UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

Intégration continue du noyau Linux via KernelCI.org

Rapport de stage ST50 - P2016

Quentin Schulz

Département informatique
Filière Logiciel Embarqué et Informatique Mobile (LEIM)

Free Electrons
25 boulevard Victor Hugo
31770 Colomiers
http://free-electrons.com/

Tuteur en entreprise:
Antoine Ténart

Suiveur UTBM:
Rachid Bouyekhf

utbm
université de technologie
Belfort-Montbéliard

Remerciements

Je tiens à remercier tout particulièrement M. Michael Opdenacker, fondateur de la société Free Electrons SARL, pour avoir accepté de m'accueillir dans leurs locaux et de m'avoir accordé leur confiance dans les différents projets et tout au long du stage.

Je souhaite aussi remercier mon tuteur, M. Antoine Ténart, comme l'ensemble de l'équipe d'ingénieurs pour m'avoir soutenu, guidé, renseigné et aidé tout au long des différentes tâches et projets qui m'ont été confiés au cours de ces 6 mois. Votre pédagogie et votre enthousiasme ont fait de ce stage une expérience fort agréable et instructive.

Contents

1 Introduction **1**

2 Free Electrons **5**
2.1 The company . . . 5
2.2 The team . . . 6
2.3 Strong focus on Free and Open Source software programs . . . 6
2.4 A recognized expertise . . . 6

3 Internship's mission **10**
3.1 Presentation . . . 10
3.2 Schedule . . . 10

4 Linux kernel continuous integration **12**
4.1 Context . . . 12
4.1.1 Linux . . . 12
Contributions . . . 12
Files built when compiling the Linux kernel . . . 14
4.1.2 Free Electrons' use case . . . 15
4.2 State of the art . . . 15
4.3 Project . . . 16
4.3.1 Continuous integration in Linux kernel . . . 16
4.3.2 Building the kernel . . . 17
4.3.3 Testing the kernel . . . 17
Software choices . . . 18
What is LAVA? . . . 18
Control power supply . . . 19
Connect to serial . . . 19
Actual testing . . . 19
4.3.4 Reporting test results . . . 21
4.3.5 Hardware infrastructure . . . 21
Specifications . . . 21
Connect to serial . . . 24
Control power supply . . . 24
Serve files . . . 26
4.3.6 Process to add devices to the lab . . . 29

4.4 Challenges encountered . . . 30
4.5 Conclusion . . . 31
4.5.1 Patches . . . 31

5 Lavabo - LAVA Board Overseer **32**
5.1 Context . . . 32
5.2 Project . . . 32
5.2.1 Implementation . . . 33
Client-server interactions . . . 33
Serial connection . . . 34
Interactions with LAVA . . . 34
Interactions with boards . . . 34
User authentication . . . 36
5.2.2 Typical workflow . . . 36
5.3 Conclusion . . . 36

6 Support for a new board **38**
6.1 Context . . . 38
6.2 Project . . . 39
6.2.1 Add support in the bootloader source code . . . 39
6.2.2 Add support in Linux kernel source code . . . 42
6.3 Conclusion . . . 43

7 Driver for Allwinner SoCs' ADC **44**
7.1 Context . . . 44
7.2 Project . . . 44
Touchscreen driver . . . 46
Temperature driver . . . 46
7.3 Conclusion . . . 46

8 Conclusion **47**

1. Introduction

Dans le cadre de mes études dans le département Informatique de l'UTBM (Université de Technologie de Belfort-Montbéliard), j'ai eu l'opportunité de réaliser un stage de 6 mois comme projet de fin d'études en tant qu'ingénieur. L'objectif de ce stage est d'appliquer les connaissances acquises au cours de notre scolarité dans un contexte professionnel, en acquérir de nouvelles ainsi que de s'adapter au monde professionnel.

Ce stage s'est déroulé lors du dernier semestre de mon cursus ingénieur en 5 années dans les locaux toulousains de l'entreprise Free Electrons.

Dans ce rapport, je présente l'entreprise, le sujet initial du stage et son évolution, les différentes tâches assignées et les résultats fournis au terme des 6 mois.

Note

L'ensemble du travail effectué lors de mon stage chez Free Electrons a été réalisé en anglais, que ce soit le développement de logiciels, drivers ou les interactions avec les différentes communautés. J'ai également été encouragé à écrire une série d'articles sur le sujet écrits en anglais et je présenterai en anglais également mon sujet de stage avec mon tuteur, Antoine Ténart, à l'Embedded Linux Conference Europe à Berlin le 12 octobre 2016. Le sujet de mon stage étant dans un domaine où la langue anglaise est très dominante et où les mots techniques ne peuvent être facilement traduits, il m'a été accordé d'écrire la suite de ce rapport en anglais.

Lexicon

- System on Chip (SoC):

  "A system on a chip or system on chip (SoC or SOC) is an integrated circuit (IC) that integrates all components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio-frequency functions—all on a single chip substrate." - Wikipedia

- Bootloader:

  "The bootloader is a piece of code responsible for basic hardware initialization, loading of an application binary, usually an operating system kernel, potentially decompression of the application binary and execution of the application. Beside 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.)" - Free Electrons' Embedded Linux system development slides

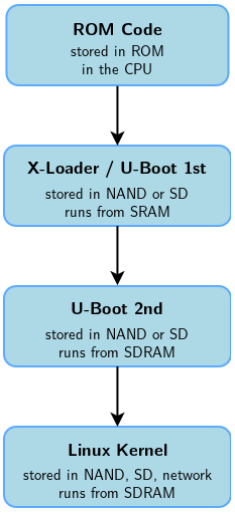

Figure 1.1: One example of boot process from power on to the Linux kernel

In this example, when a board is powered on, the processor loads a small embedded ROM code which will loads a bootloader in the SRAM limited in size. This bootloader, called SPL or Secondary Program Loader, will take care of basic hardware

initialization (mainly DRAM) and the loading of a full size bootloader in DRAM with user interaction. This full size bootloader will then be able to starts the Linux kernel in RAM.

- Root filesystem (rootfs):

  The Linux kernel follows the Filesystem Hierarchy Standard[1] which states that everything is organized in directories in a tree-like hierarchy and all directories are subdirectories of the "/" (root) directory. Unlike Microsoft Windows which creates drives for each filesystem, the Linux kernel uses a subdirectory of the root directory to mount the filesystem. The root filesystem contains the files needed for booting the system and is required for mounting other filesystems, thus, without a root filesystem, the Linux kernel will boot but hang just after, making the board unusable for the user.

- Mainline, upstream:

  Refers to the official current version of a software - the Linux kernel for example - being developed. Mainlining or upstreaming is the process of adding code to the software's mainline version - from http://kernel.org for the Linux kernel. This process includes comments and critics from the community supporting the software and then validation of the proposed code by the developers in charge of the software part impacted by the proposed code. In the Linux kernel, we call these developers maintainers. When the maintainer of a part of the Linux kernel validates a code snippet, it is merged to this part mainline version until it gets added to the Linux kernel mainline version.

- User space:

  The Linux kernel - with every tool used to schedule programs or directly interacts with the hardware - runs in kernel space. Programs executed by users are run in user space and uses tools from the kernel space as an abstraction for the hardware communication.

[1] https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard

2. Free Electrons

2.1 The company

Free Electrons is an engineering company founded in 2004 by Michael Opdenacker, a Linux and Free and Open Source software programs' enthusiast. Since its beginnings, the company offers training services for development in Open Source software programs and currently, offers training for the Linux kernel, the Yocto Project and OpenEmbedded, Buildroot or Android. It also offers its expertise in embedded Linux development for companies willing, for example, to use Linux in their products, to upstream drivers in kernel or build a custom Linux system.

Free Electrons' team is split between two locations in France:

- Orange, for General Managers and the accounting,
- Toulouse, for the engineering team;

Two engineers are also teleworking from Lyon's surroundings.

2.2 The team

Figure 2.1: Free Electrons is proud to have the conviviality of a human-size company with 11 employees and 2 interns. From left to right: Maxime Ripard, Grégory Clément, Mylène Josserand, Antoine Ténart, Maria Llvata (executive director), Quentin Schulz (intern), Thomas Petazzoni (CTO), Romain Perier, Alexandre Belloni, Anja Roubin (executive assistant), Boris Brezillon and Michael Opdenacker (founder and CEO). Florent Revest also joined Free Electrons as an intern later.

2.3 Strong focus on Free and Open Source software programs

Since its creation, Free Electrons is doing its best to contribute to the Free Software community. It does so by releasing all its training materials[1] under free documentation license[2]. The company also strongly encourages clients to share our combined work with the community and thus has a preference for clients willing to interact with and give back to the free software community by sparing some of project's time on upstreaming modifications to free software programs.

2.4 A recognized expertise

Free Electrons' engineers are well-known in communities of free software programs, such as the Linux kernel and Buildroot, thanks to their tremendous number of contributions in

[1] http://free-electrons.com/training/
[2] https://creativecommons.org/licenses/by-sa/3.0/

these different projects and also by attending and presenting talks in famous conferences around the globe, like the Embedded Linux Conference[3] or the FOSDEM[4].

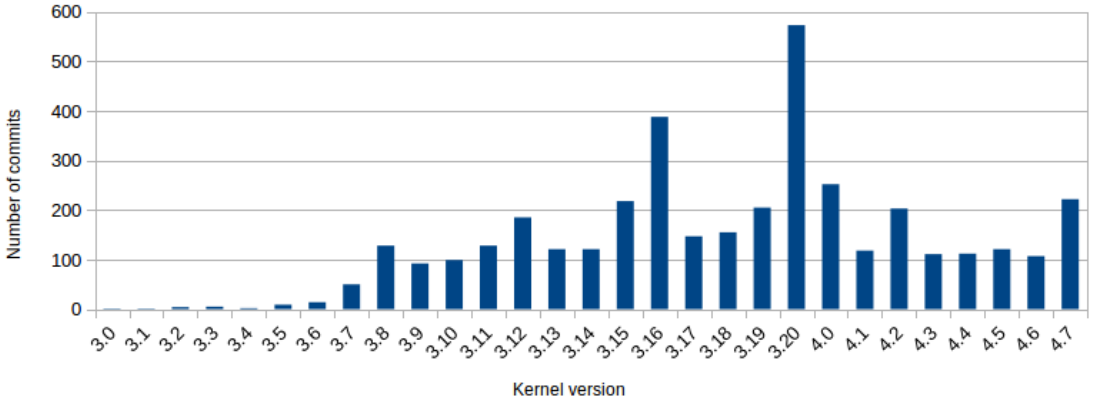

Figure 2.2: Free Electrons' contributions in the Linux kernel

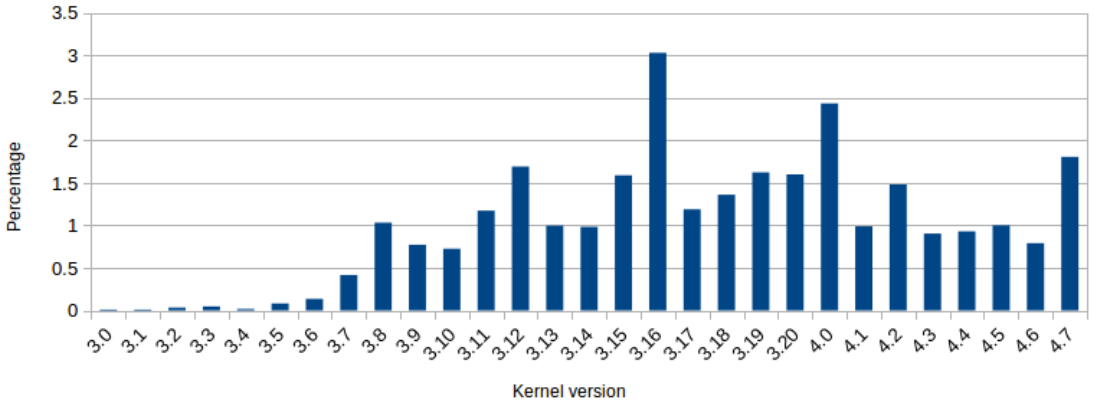

Figure 2.3: Percentage of Free Electrons' contributions on total the Linux kernel's contributions

[3] http://www.embeddedlinuxconference.com/
[4] https://fosdem.org

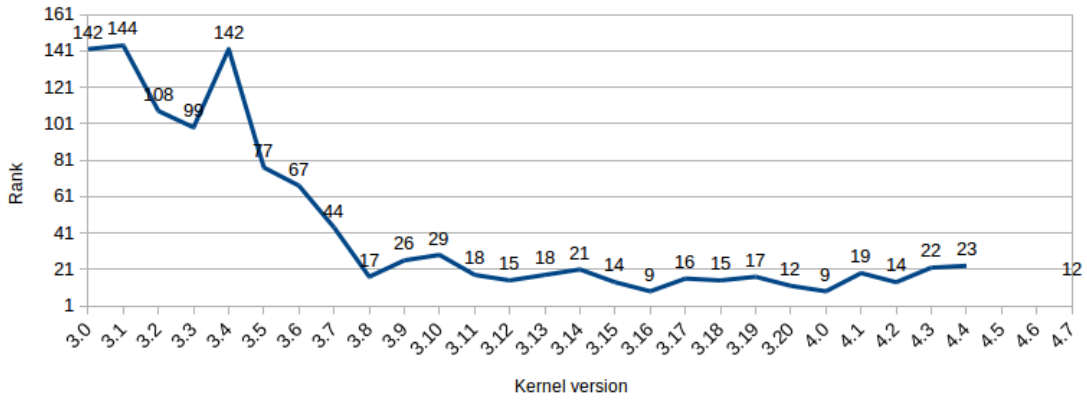

Figure 2.4: Rank in most contributing companies in the Linux kernel

Being a big contributor often opens great opportunities in the Linux kernel. One of these opportunities is to become a maintainer of a subsystem of the kernel. The Linux kernel is such a big project that it is organized in different parts, called subsystems, and also needs maintainers which take care of a subsystem by validating code being merged to the Linux kernel source code.

Free Electrons has currently 6 maintainers[5] in its engineering team:

- Alexandre Belloni, maintainer of ATMEL SoCs, Real-Time Clock,
- Boris Brezillon, maintainer of NAND flash, ATMEL Clocks, Marvell cryptography driver,
- Grégory Clément, maintainer of Marvell SoCs,
- Thomas Petazzoni, maintainer of FBTFT Framebuffer drivers, Marvell MVNETA Ethernet driver, Marvell PCI driver,
- Maxime Ripard, maintainer of Allwinner SoCs, Allwinner A10 DRM drivers, NVMEM framework,
- Antoine Ténart, maintainer of the Annapurna Labs architecture;

The company works with both renown hardware vendors and companies who build custom boards, which asserts Free Electrons' expertise.

We started working with NextThing Co. a year ago on one of the cheapest computer in the world[6]: the C.H.I.P, sold for only 9$ and its dedicated case embedding a touchscreen, a keyboard and a battery pack, the PocketCHIP for 69$. Free Electrons is adding support for

[5] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS
[6] https://getchip.com/

all the hardware components, be it the different video outputs, the camera, the NAND, the Power Management or the screen for example, in the Linux kernel and in the bootloader U-Boot.

Figure 2.5: C.H.I.P.: the 9$ computer

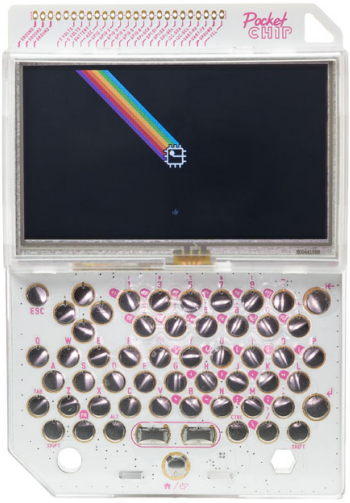

Figure 2.6: PocketCHIP

3. Internship's mission

3.1 Presentation

During my internship, I worked on the following subject: "Linux kernel continuous integration via KernelCI.org".

The KernelCI.org project's goal is to bring continuous integration to the Linux kernel. This involves testing the latest kernel versions on a wide range of systems to detect as soon as possible new regressions. This project is based on a community of labs, each lab having different systems to test.

The internship aims to build such a lab in Free Electrons' offices using the embedded systems the company already owns thanks to its work with some vendors, and to help solving detected problems from continuous integration's tests.

The internship schedule was organized as following:

- find how to boot the latest kernel version on each platform,
- implement a system of remote power control for each platform,
- add each platform to the platforms' pool of KernelCI,
- identify and try to solve the detected regressions,
- interact with the community to integrate the patches for these regressions in the bootloader or the kernel source code,
- create custom tests (e.g.: cryptography) for Free Electrons' systems;

In addition to the main subject, the internship included work on small projects, i.e. the development of drivers for the Linux kernel or help on other projects depending on the needs of clients.

3.2 Schedule

The main subject should take three or four months of the internship total duration and the remaining would be split between drivers' development, contributions in the different projects and improvements of the tests.

I actually worked full-time on building the lab until May, 15th (i.e. 14 weeks) and then worked part-time on the same subject and on different projects.

Finally the internship's duration has been spread as following:

- 17 weeks on building, maintaining, improving the lab (main subject),
- 9 weeks on side projects;

4. Linux kernel continuous integration

4.1 Context

4.1.1 Linux

Linux is an operating system kernel (or simply kernel) created by Linus Torvalds and its source code was released in 1991 under a Free and Open Source license. After requests to remove the limitation on commercialization established by the current license, he ultimately released the source code under the GPL license[1] in 1992.

The Linux kernel is the biggest Open Source software with its 21 millions lines of code, 4000 contributors and several thousands of supported architectures. Such a big project needs people which are guarantors of the quality of the code and the respect of different rules (such as the coding style for example). These people are called maintainers and are specialized in a certain part of the Linux kernel, called a subsystem.

With a project this big and that segmented, it is hard to guarantee the modifications applied to the code source do not break the kernel especially when, in its 4.6 version, it averaged 11600 lines added, 5800 removed, and 2000 modified per day[2].

The Open Source Software biggest strength is in its openness. Everyone can audit the code to find security flaws or bugs and fix them or notify the community of these bugs. Most importantly, companies do not need to reinvent the wheel, they can reuse freely the most tried and tested software programs in the world.

Contributions

When one wants to upstream a code change in the Linux kernel source code, one has to follow a set of rules defined for the Linux kernel. This set of rules[3] asks to:

- extensively describe the code changes,
- separate all changes in small different *patches*,
- check indentation, coding style, comments of patches,
- take time to chose the persons to which the patches will be sent,

[1] https://www.kernel.org/pub/linux/kernel/Historic/old-versions/RELNOTES-0.12
[2] https://www.linux.com/news/greg-kroah-hartman-gives-inside-look-largest-fastest-software-project-all
[3] https://www.kernel.org/doc/Documentation/SubmittingPatches

- respond to review comments,
- sign patches;

Once one has formatted its patches accordingly to the different rules, one has to send its patches for reviews to maintainers of all subsystems impacted by the changes, as well as their mailing lists.

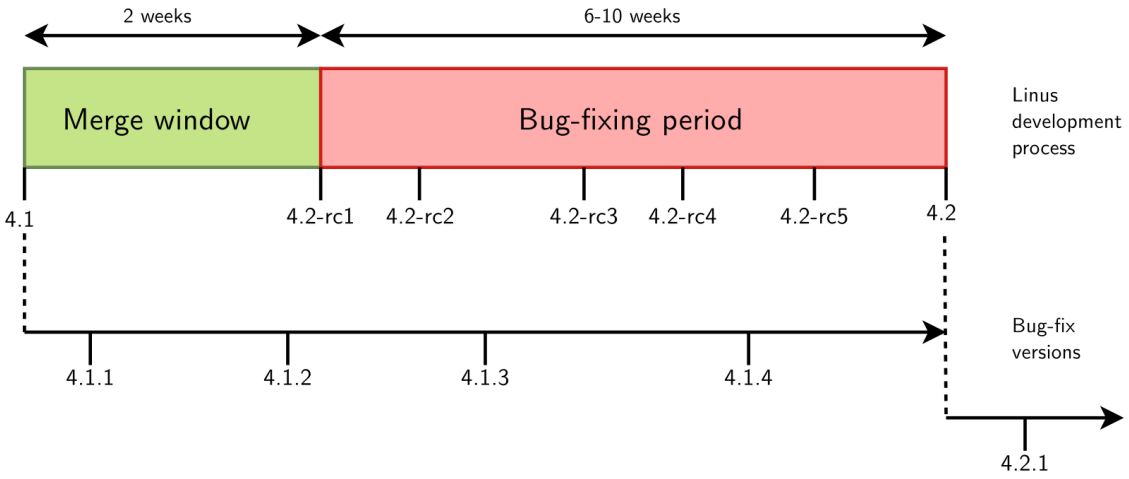

Figure 4.1: Development schedule of the Linux kernel

The Linux kernel has a rather simple development schedule:

- 2 weeks for the merge window,

  During these two weeks, patches validated by the community prior to this period are merged to the upstreaming version of the Linux kernel.

- 6 to 10 weeks for the bug-fixing window,

  During these few weeks, only important bug fixes and bug fixes for new features added during the merge window should be submitted to the community for the next version. This is also the period when the patches for new functionalities for the version after the next version are submitted.

Patches will be discussed via the mailing lists and once they are validated by maintainers, the latter will then merge the patches into their own git repository in branches destined for the Linux kernel version currently in development or the next one. The goal of these branches specific to each subsystem is to ease the merging of patches in the Linux kernel mainline version. Another git repository called linux-next[4], or simply next, is used to make intermediate test releases by automatically merging all subsystems' branches to

[4] https://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git/

the latest version of the Linux kernel. It allows to find merge failures and bugs before merging to the Linux kernel mainline version. During the merge window, Linus Torlads will merge all subsystem maintainers' git branches creating the first Release Candidate, or rc1, version of the next Linux kernel version.

Some vendors work differently with the Linux kernel. They usually forks (make a copy) of the Linux kernel git repository and makes tremendous changes in the source code without sending the changes for reviews to the community. Often, they do not update regularly their version to match the Linux kernel latest version which results in a complete nightmare when the vendor finally decide to upstream their modifications. This induces also that a board might only work with the vendor Linux kernel which does not contain the Linux kernel new functionalities or bug fixes (such as security fixes).

Files built when compiling the Linux kernel

When compiling the Linux kernel, some files are created:

- the kernel image, which can be of different format, for example:
  - Image, which is the format for 64-bits ARM platforms,
  - zImage, which is the format for all other platforms,
  - uImage, which is a format specific to U-Boot bootloader, adding a header to specify the entry and load addresses in RAM of the kernel image;
- Device Tree Blobs, only created for ARM (32 or 64 bits) and PowerPC platforms;

The Device Tree, as its name suggests, is a tree of devices and a representation of hardware devices and buses which are present in a board and are not discoverable at runtime. This file is compiled beside the Linux kernel with the Device Tree Compiler and a change in its code does neither affect the kernel nor requires a rebuild of the kernel. A Device Tree Blob is created once the Device Tree has been compiled and has to be used in conjunction with a compiled kernel or the kernel will not be able to boot. A Device Tree is specific to one board but can inherit components' definition from Device Trees includes.

The Device Tree is a file organized like a tree, with a root node, bus nodes and devices nodes. The root node typically contains a *cpus* node describing each CPU in the system, a *memory* node defining the location and size of the RAM, an *aliases* node specifying shortcuts to some nodes, and nodes describing SoC buses such as USB, i^2c or SPI and on-board devices such as LEDs, eMMC, SD card reader, Ethernet port or power regulators.

Two kernels differ by their configuration file (which drivers to build inkernel or as modules for example), their targeted architecture and the Device Trees compiled with it.

4.1.2 Free Electrons' use case

Some of Free Electrons' clients ask help to upstream a driver to the Linux kernel source code. Since Linux is FOSS (Free and Open Source Software), everyone can modify the code of this driver to add functionality or to fix bugs but it can also introduce new bugs because developers only test their code on the boards they own, often not the ones we own. Free Electrons' engineers need to continuously keep an eye on part of codes used by or developed for a client. Several clients also ask to follow the whole life-cycle of the project (from its early development stages to its discontinuity) so engineers have to keep track of all modifications of the kernel that might break the product.

4.2 State of the art

Since the kernel supports several thousands of different platforms, when a developer writes new code for the kernel, he cannot test his code on all the platforms and usually tests only on the platforms he owns. Before merging its code to the kernel source code, some users or other developers might test it on other platforms or in other configurations but it is rather rare. Furthermore, the only way to guarantee a modification works on all impacted platforms is to build the kernel in all its configurations and test on all these platforms if the kernel boots. This is extremely tedious and time consuming.

As explained above, there is currently no way to automatically discover a regression in the kernel beside Intel's 0-day[5] for x86 platforms. The only way to detect a regression for other platforms is when a user of the kernel performs an audit or is using the kernel in a certain configuration so that he discovers a bug. The software developed by Intel called 0-day which is a building test robot provides little help by taking patches from different mailing lists, applying them one by one to different branches and testing the Linux kernel can be properly built with the modifications included in the patch.

It is the role of a maintainer to guarantee the stability and the working of the subsystem he's in charge. However, since he hasn't all the platforms impacted by his subsystem, the only thing he can do is to look for common errors in the code and try to find corner cases.

The main goal of KernelCI project is to implement a continuous integration in the kernel so build or boot failures are detected before reaching the end users. There already are several labs contributing to the project: several from Linaro and few from KernelCI founders. However, they are often expensive (i.e. the Linaro's lab uses several 500$ APC PDU[6] systems to control the power of boards) and we do not want to put that much money in a lab so one of the challenges is to have working, cheap (but not hackish) solutions.

At Free Electrons, we also love to contribute to free software programs and we are very

[5] https://01.org/lkp
[6] Power Distribution Unit

proud to be part of the effort to bring continuous integration to the kernel.

4.3 Project

4.3.1 Continuous integration in Linux kernel

Continuous integration is the process of automated building and testing of a software when one or multiple changes have been made to its source code, and then reporting of the results. The main benefit of continuous integration is to detect as soon as possible regressions in the code so they could be identified and corrected quickly, ideally before being in a production environment.

Because of Linux's well-known ability to run on numerous different platforms and the obvious impossibility for developers to test changes on all these platforms, continuous integration has a big role to play in Linux kernel development and maintenance.

Continuous integration is made up of three different steps:

- building the software which in our case is the Linux kernel,
- testing the software,
- reporting the tests results;

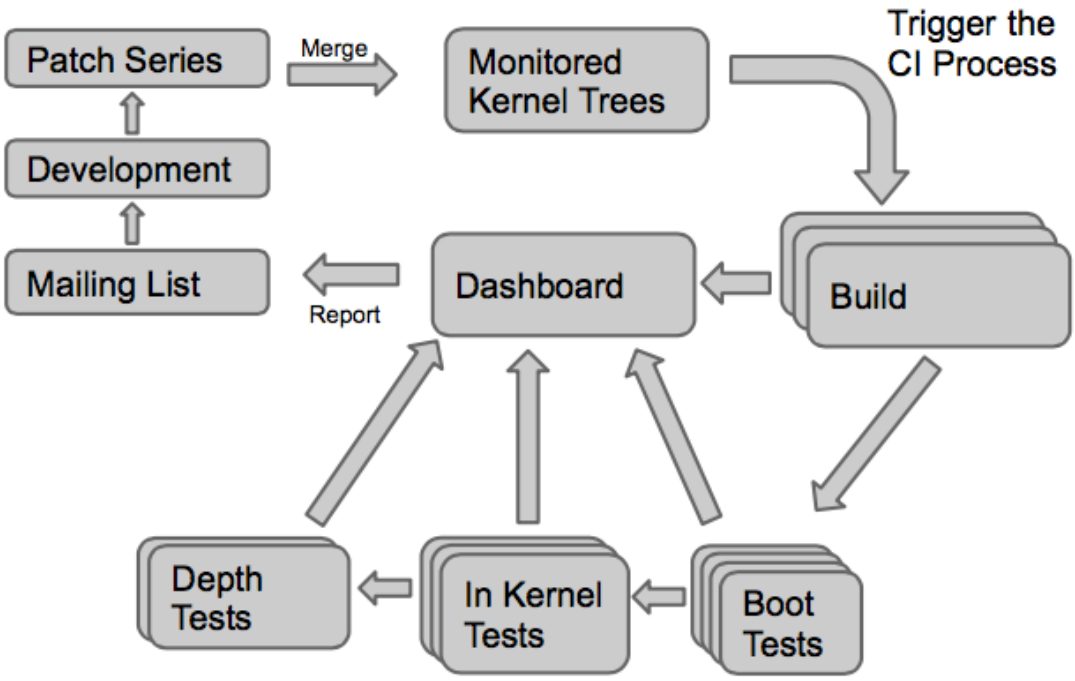

Figure 4.2: KernelCI complete process

4.3.2 Building the kernel

Something has to trigger the build of the Linux kernel. It could be either changes in a git branch or in a whole git repository or patches sent to mailing lists. Most of these tracking means are not automatic and have to be scheduled to check for changes once in a while. We can tweak the granularity of the tracking by controlling the period of time between two scheduled checks.

KernelCI is permanently tracking around a dozen of kernel git repositories. Each hour, KernelCI checks if the git repositories have been updated and if so, KernelCI builds from the last commit the kernel for ARM, ARM64 and x86 platforms in many configurations and their associated Device Trees. Then it stores all these builds in a publicly available storage[7].

4.3.3 Testing the kernel

KernelCI is not in charge of this part of continuous integration but is rather delegating this step to the software installed in organization's or individual's labs.

[7] https://storage.kernelci.org/

Software choices

At this moment, two software programs are known to work with KernelCI: Linaro Automated Validation Architecture (LAVA)[8] and pyboot[9]. pyboot is going deprecated so LAVA is highly preferred for new labs. LAVA was created early 2012 to meet Linaro's developers' needs to continuously test their work before delivering it to their client. Note that KernelCI does not solely interact with pyboot or LAVA since it offers an API[10], so if none meets your needs, go ahead and make your own!

What is LAVA?

LAVA is a self-hosted software, organized in a server-dispatcher model, for controlling boards to automate boot, bootloader and user-space testing. The server receives *jobs* specifying what to test, how and on which boards to run those tests, and transmits those jobs to the dispatcher linked to the specified board. The dispatcher applies all modifications on the kernel image needed to make it boot on the said board and then fully interacts with it through the serial. There is only one LAVA server but there could be one or more dispatchers, each one having its own boards. Since we plan to have around 50 boards in our lab, we have only one computer which is used as both LAVA server and dispatcher.

Since LAVA has to fully and autonomously control boards, it needs to:

- interact with the board through serial connection,
- control the power supply to reset the board (in case of a crashed kernel or when a job has finished to limit useless power consumption),
- know the commands needed to boot the kernel from the bootloader,
- serve files (kernel, Device Tree Blob (DTB), rootfs) to the board.

The first three means are given to LAVA by per-board configuration files. The latter is done by the LAVA dispatcher in charge of the board which downloads files specified in the job and copy them to a directory accessible by the board through TFTP.

LAVA organizes the lab in *device types* and in *devices*. All identical devices are from the same device type and share the same configuration file. A device type configuration file contains the set of instructions to run in the bootloader to boot the kernel (e.g.: how and where to load files) and the bootloader configuration (e.g.: can it boot zImages or only uImages). A device configuration file stores the commands run by a dispatcher to interact with the device: how to connect to serial, how to power it on and off. LAVA interacts with

[8] http://www.linaro.org/initiatives/lava/
[9] https://github.com/khilman/pyboot
[10] https://api.kernelci.org/

devices via external tools: it has support for conmux or telnet to communicate via serial and power commands can be executed by custom scripts or pdudaemon[11] for example.

Control power supply

Some labs use expensive (∼500$) systems to control the power supply of each board but we do not want to put that much money into a small part of our lab. Therefore, we went for Ethernet-controlled relays. We could have used USB-controlled relays as well but there are already a lot of USB connections going on with the serial connection for each board so we chose the former. It is possible to communicate with these Ethernet-controlled relays using TCP frames.

I first developed our own software to control boards' power supplies but I figured out it would be better to contribute to pdudaemon. pdudaemon is developed by Linaro and is widely tried and tested for almost 3 years now. It works in a server-client fashion: client sends the request (power on or off a board) to the server which queues it if there is more than one power-control pending operation and then executes the requested operation. To control boards' power, we use several Devantech ETH008 Ethernet-controlled relays for which I had to add support in pdudaemon[12], which was made straight-forward thanks to pdudaemon's drivers model implementation. The server is handled by a daemon while the client is called by LAVA dispatchers to send power-control requests to pdudaemon's server.

Connect to serial

As advised in LAVA's installation guide[13], we went with telnet and ser2net[14] to connect to boards' serial. ser2net basically opens a Linux device and allows to interact with it through TCP sockets on a defined port. A LAVA dispatcher will then launch a telnet client to connect to boards' serial. Because of the well-known fact that Linux devices' name might change between reboots, I had to use udev rules in order to guarantee the serial we connect to is the one we want to connect to.

Actual testing

Now that LAVA knows how to handle devices, it has to run jobs on those devices. LAVA jobs contain which images to boot (kernel, DTB, rootfs), what kind of tests to run when in user space and where to find them. A job is strongly linked to a device type since it contains the kernel and DTB specifically built for this device type. LAVA has the ability to run jobs simultaneously, meaning jobs can be run on different device types at the same

[11] https://github.com/Linaro/pdudaemon
[12] https://github.com/Linaro/pdudaemon/pull/9
[13] https://validation.linaro.org/static/docs/v1/installation.html#setting-up-serial-connections-to-lava-devices
[14] http://linux.die.net/man/8/ser2net

time. Since devices in the same device type are assumed to behave like each other, when a job requests a certain device type, it will only run on one device of this device type.

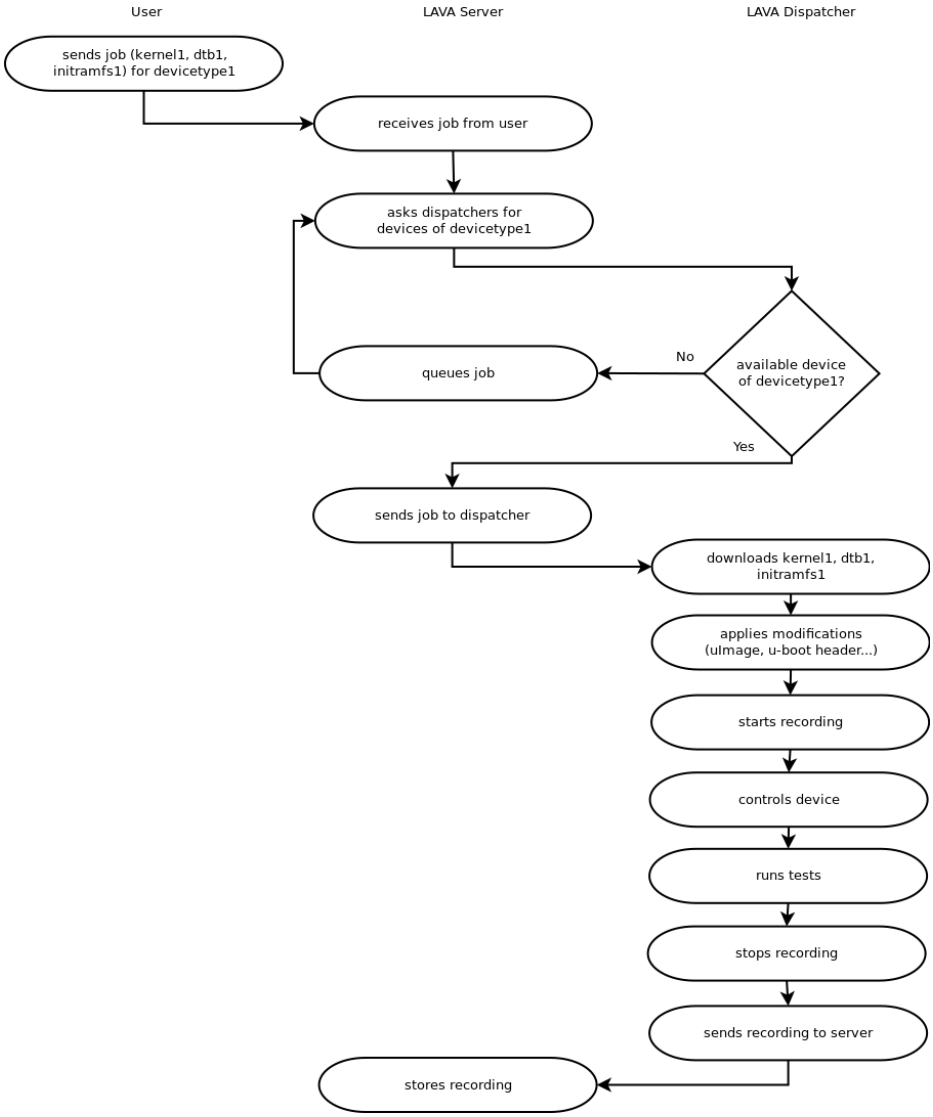

Figure 4.3: LAVA server and dispatcher complete process

As you can see, everything is automated only after the user has sent the job. Yet, to achieve full automation, we have to automate the sending of jobs too which is made possible thanks to LAVA's API. Now, the only remaining barrier is to build kernels on git notifications or crontabs for example, create tests and send them to LAVA via its API.

KernelCI is taking care of sending jobs to contributing labs to run minimal testing on latest versions of the kernel. To send jobs to LAVA's API, it uses lava-ci[15] for which we

[15] https://github.com/kernelci/lava-ci/

added and will continue to add the support for all boards. When KernelCI has finished to build a kernel, it creates a job for all boards represented by a Device Tree in the different configurations of this kernel.

4.3.4 Reporting test results

After KernelCI has built the kernel, sent jobs to contributing labs and LAVA (or another software) has run the jobs, KernelCI will then get the tests results from the labs, aggregate them on its website and notify maintainers of errors via a mailing list.

4.3.5 Hardware infrastructure

Specifications

Our lab is supposed to take place in Free Electrons' offices in a 100cm wide space. The room in which the lab will take place is separated from the office by a 200cm high and 75cm wide door and we thought it would be a good idea to build the lab a little smaller so if we move out of this office a day, we do not have to completely disassemble it but rather slide it through the doors.

Finally, Free Electrons' engineers filled a spreadsheet with all boards they wanted to have in the lab and their particularities regarding which connectors are used for the serial communication and the power supply. We reached a total of 48 boards to put into the lab. Among those boards, we can distinguish two different types:

- boards which are powered by an ATX power supply,
- boards which are powered by chargers;

In addition to needing a full ATX power supply, the first type of boards often has a huge motherboard. These boards sometimes also embed extension ports such as PCI-e, making their height bigger. The biggest height we could find was 25cm for a board with a PCI-e device plugged in.

We also wanted the lab to be easy to use and evolve. Sometimes, engineers will need to take a board out of the lab or to add one. The easier the process is, the better the lab is. Building the lab with drawers seemed to be the easiest, cheapest and quickest way to make our lab modular. Since we have two types of boards, one taking a lot of room in a drawer, we thought it would be a good idea to separate both types in different drawers. *Big drawers* would host the boards which need a separate ATX power supply while the *small drawers* would host the smallest boards.
Typically, considering the size limitations of the lab, our drawers would have a maximal

size of 95x70cm. We determined we could put 8 small boards on a small drawer and up to 4 big boards on a big drawer. Since we went with drawers, we had to think of a system to limit the number of cables connecting the server and the hardware components on each drawer so it is easy to slide the drawer.
Given the maximal height of the lab to be a bit less than 200cm and the minimal height allowed for a drawer to be 25cm, we knew we could have 8 drawers in our lab. From the spreadsheet containing all the boards supposed to be in the lab, we eventually decided there would be 3 *big drawers* for up to 12 *big* boards and 5 *small drawers* for up to 40 *small* or *medium-sized* boards.

To organize our drawers and ease their evolution, we needed to cleanly guide the cables to each board while making it easy to remove them in any occasion. Some boards are also so tiny and light, the rigid cables connected to the board imposed the place where the board will stay. Also, we needed to keep the power supplies and other hardware materials in place. Therefore, we needed an easy-to-remove way to guide cables and also maintain boards and heavy materials at a given place. We thought scratch bands were an excellent choice:

- easy to reposition,
- unharmful to cables and boards (neither electrostatic nor made of metal),
- strong enough to hold boards and heavy materials in place,
- cheap and easy to find;

Therefore, we needed a scratch band with one side to stick to the drawer and the other in either hooks or loops material, and a second scratch band with one side in hook material and the other in loop material. The second scratch band would circle cables and boards hooking to itself and to the first scratch thus keeping them in place. The first scratch would also be used to stick the power supplies, the switches, USB hubs and Ethernet-controlled relays to the drawer.

Figure 4.5: Scratch hooks (CC-BY-3.0 Alexander Klink)

Figure 4.6: Scratch loops (CC-BY-3.0 Alexander Klink)

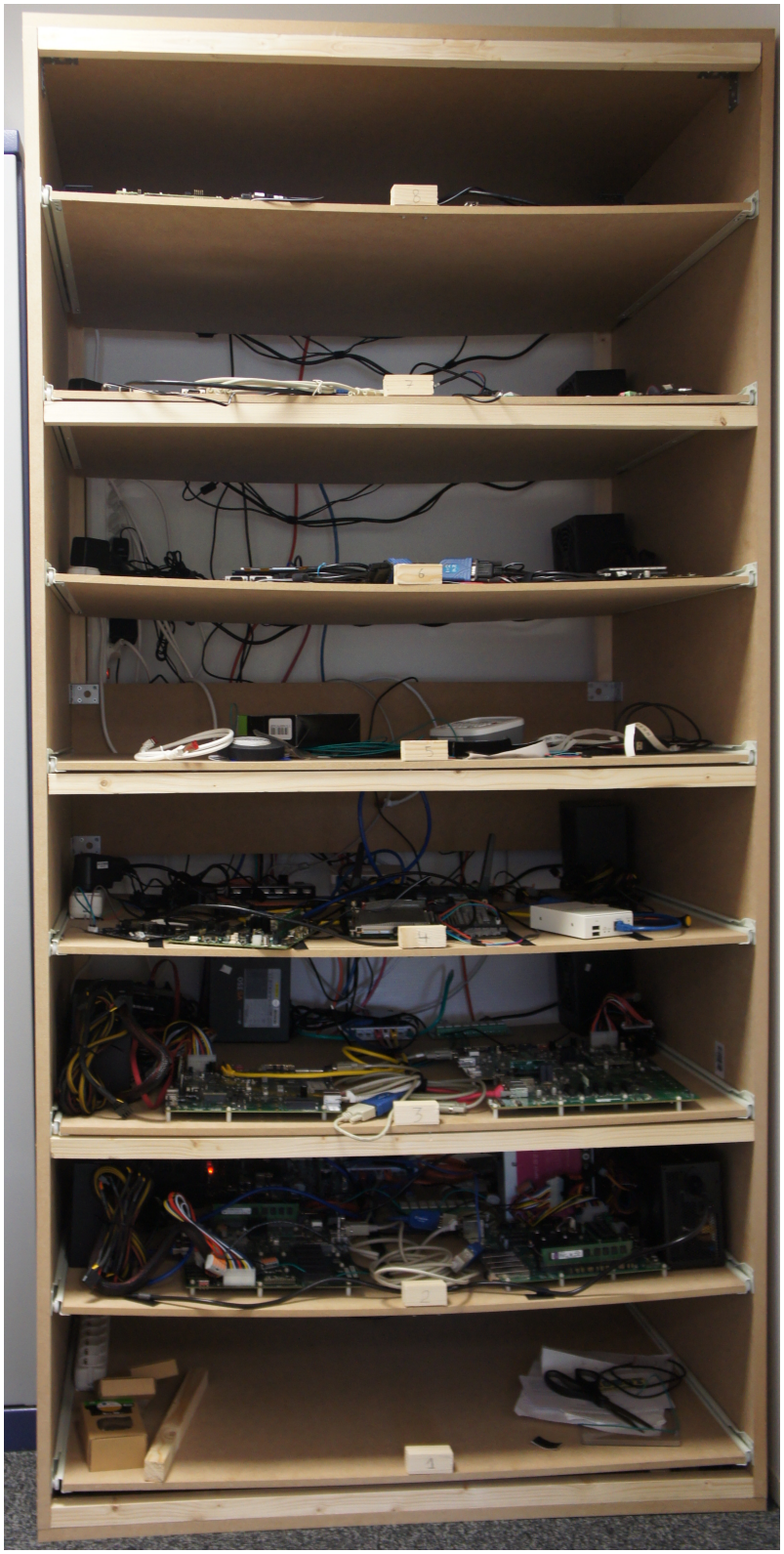

Figure 4.4: Free Electrons' 8 drawers lab

Furthermore, since the lab would host a server and a lot of boards and power supplies, potentially producing a lot of heat, we had to keep the lab as open as it can be while making sure it is strong enough to hold the drawers.

After that, we had to meet LAVA hardware requirements.

Connect to serial

Serial connections are mostly handled via USB on host side but there are many different connectors on target side (in our lab, we have 6 different connectors: DE9, microUSB, miniUSB, separate male pins, separate female pins and USB-B).

In a LAVA setup with several dispatchers, each dispatcher has to be physically connected to all of the boards it's in charge of. In our case, the server hosts the LAVA master and dispatcher all at once. Therefore, our server had to have a physical connection with each of the 50 boards present in the lab. The need for USB hubs was then obvious.

Since we wanted as less cables connecting the server and the drawers as possible, we decided to have one USB hub per drawer, be it a *big drawer* or a *small drawer*. In a *small drawer*, up to 8 boards can be present, meaning the hub needs at least 8 USB ports. In a *big drawer*, up to 4 serial connections can be needed so smaller and more common USB hubs can do the work. However, since the serial connection may draw some current on the USB port, we wanted all of our USB hub to be powered with a dedicated power supply.

All USB hubs are then connected to a main USB hub which in turn is connected to our server.

Control power supply

LAVA needs to control each board's power to be able to automatically power on or off a board. It will power on the board when it needs to test a new kernel on it and power it off at the end of the test or when the kernel has frozen or could not boot at all.

We thought of two different ways to control power supplies:

- having a power supply for each board connected to an Ethernet-controlled multi-socket which can power on or off each socket individually,
- having one big power supply for all the boards, split the output wire between all boards and connect it to Ethernet-controlled relays which open or close the connection between the power supply and the board;

In *small drawers*, up to 8 boards need a separate power supply. Professional controllable multi-sockets are too expensive, too big and too heavy (sometimes rackable) and often

offer US sockets or *kettle cords*[16] only which means we have to buy adapters to switch from European to US sockets or to connect the kettle cord to the board's charger. More affordable controllable multi-sockets are available on the market but they often include only 6 sockets which is not enough for our *small drawers*. The first option seems out of our scope.

The second option would need a power supply with enough amperage to power all boards at the same time. In the future, we may want to attach hard drives to some boards for cryptography or RAID tests for example. Spin-up of hard drives draws a lot of current so we have to add some amps to the minimal needed amperage.
Amongst our boards some need 5 Volts inputs while other need 12 Volts inputs. This requirement would induce to have two different power supplies (one delivering 5V and the other 12V) or a power supply which can deliver both at the same time. When gathering information and experiences from different labs, Kevin Hilman, one of KernelCI's founders, told us he uses ATX power supplies to power both 5V and 12V boards. He splits both outputs and plugs them in an Ethernet-controlled relay to individually open or close the power circuit to each board. This was the best product we could think of to power the boards since desktop ATX power supplies often deliver a lot of current to power multiple graphic cards, hard drives and the processor. So we could use one ATX power supply per *small drawer* to handle all the power for all boards in this drawer.

Figure 4.7: Splitting ATX power supply 5V, 12V and ground outputs

However, there was still one problem: an ATX power supply is always powered but does not always deliver power. This is a way to spare electricity when the desktop is not used thus not needing the ATX power supply to deliver power. When we press the power

[16] https://en.wikipedia.org/wiki/IEC_60320#C13.2FC14_coupler

button on a desktop, it sends a signal to the ATX power supply to start delivering power. The boards in our *small drawers* are not supposed to be powered with an ATX power supply so they do not know how to send this signal to the power supply. Moreover, the signal has to be sent on a separate wire which is not connected to any of these boards. This signal is called *PS_ ON#*[17] and is supposed to be applied to the 16th pin (normally green wire) of the 24-pins connector of the power supply. When this pin is connected to the ground, the power supply turns on.

Since the Ethernet-controlled relays make sure the power circuit to the boards is open when the boards are powered off, we can permanently turn on the power supply by putting the pin to the ground for *small drawers*.

However, it is different for boards which need a separate ATX power supply, i.e. in *big drawers*. In this case, the board is powered when the switch on the board is in the *ON* position. This switch basically takes the ground from the power supply and connects it to the 16th pin when it is in the *ON* position. We cannot manually switch the board's switch each time we need to reset the power of the boards since LAVA needs to be autonomous. The solution was to split the ground, cut the green wire from the ATX power supply before the connector and plug both in the same Ethernet-controlled relay which will connect them together only when needed, thus powering the ATX power supply.

By definition, a power supply does not deliver a stable voltage but rather guarantees the delivered voltage will be within a certain range. However, the power supply might still malfunction and deliver a voltage too high for the board thus burning it. This is a big problem, besides fire hazard, because boards are expensive and sometimes owned only once. To try to prevent such problems, we added TVS[18] diodes for each voltage output in each drawer. These diodes will absorb all the voltage when it exceeds the maximum authorized value and explode and are connected in parallel in the circuit to protect[19].

Serve files

LAVA has two ways of serving files to a board: via TFTP[20] or via fastboot[21]. Since fastboot is using the USB protocol, it is by definition slower than TFTP which uses the UDP protocol. Thus, when possible, we use TFTP over fastboot for speed sake. Moreover, fastboot is often not supported due to an old version of the bootloader for example.

Since TFTP is using UDP protocol to serve files to each board, we needed to connect the boards and the server to the same network. Because we wanted the least possible cables between drawers and the server and also because an at-least 52-ports switch is unbelievably expensive, we decided to have one switch per drawer which will be connected to a main

[17] http://www.formfactors.org/developer/specs/atx2_2.pdf
[18] Transient-voltage-suppression diode
[19] https://en.wikipedia.org/wiki/Transient-voltage-suppression_diode
[20] https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
[21] https://wiki.cyanogenmod.org/w/Doc:_fastboot_intro

switch to which the server will be connected.

Each board needs its own Ethernet cable, meaning 8 ports are reserved on a *small drawer*'s switch for boards' network connection while 4 ports are reserved on a *big drawer*'s switch for its own boards. Each drawer also needs to reserve one port for the Ethernet-controlled relays and one port used to connect to the main switch. Therefore, switches present on *small drawers* will need at least 10 ports while switches on *big drawers* will need 6 ports.
We might want in the future to perform speed tests on network interfaces of boards and thus, we decided to go only with gigabit switches. With this in mind, there were two choices for switches on *small drawers*: one 16-ports switch or two 8-ports switches. Since we already needed one 8-ports switch per *big drawer* and the fact that two 8-ports switches were cheaper than one 16-ports switch, we chose to have only 8-ports switches for drawers.

We settled for a 16-ports gigabit switch as the main switch since it connects 8 switches, the server and the ISP[22] box.

For network cables, we quickly calculated the maximal length of a link in a drawer is 1m while we need a minimum of 2m for cables connecting drawer's switches and the main switch and multiple very short cables for connecting switches in *small drawers* and the Ethernet-controlled relays. Therefore we bought around 80 meters of network cables.

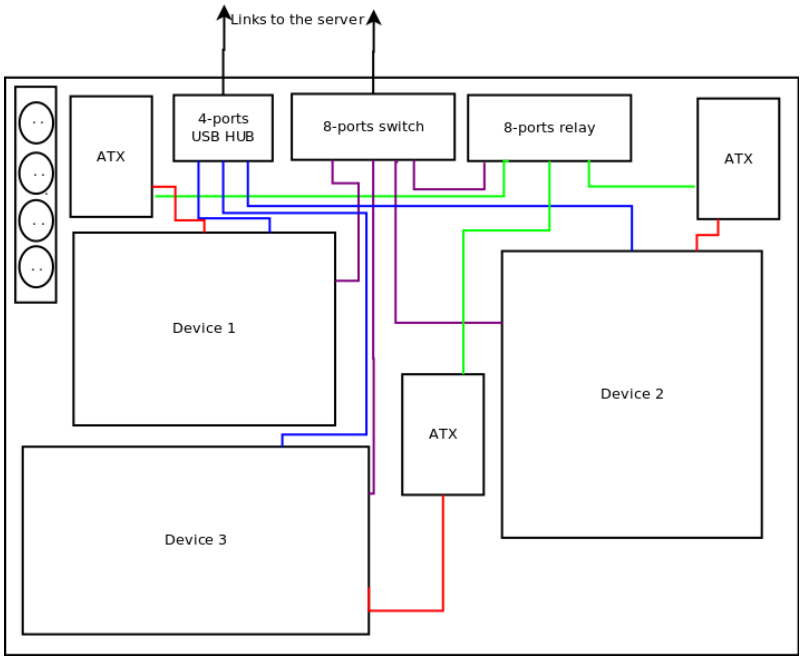

Figure 4.8: *Big drawer* example scheme

[22]Internet Service Provider

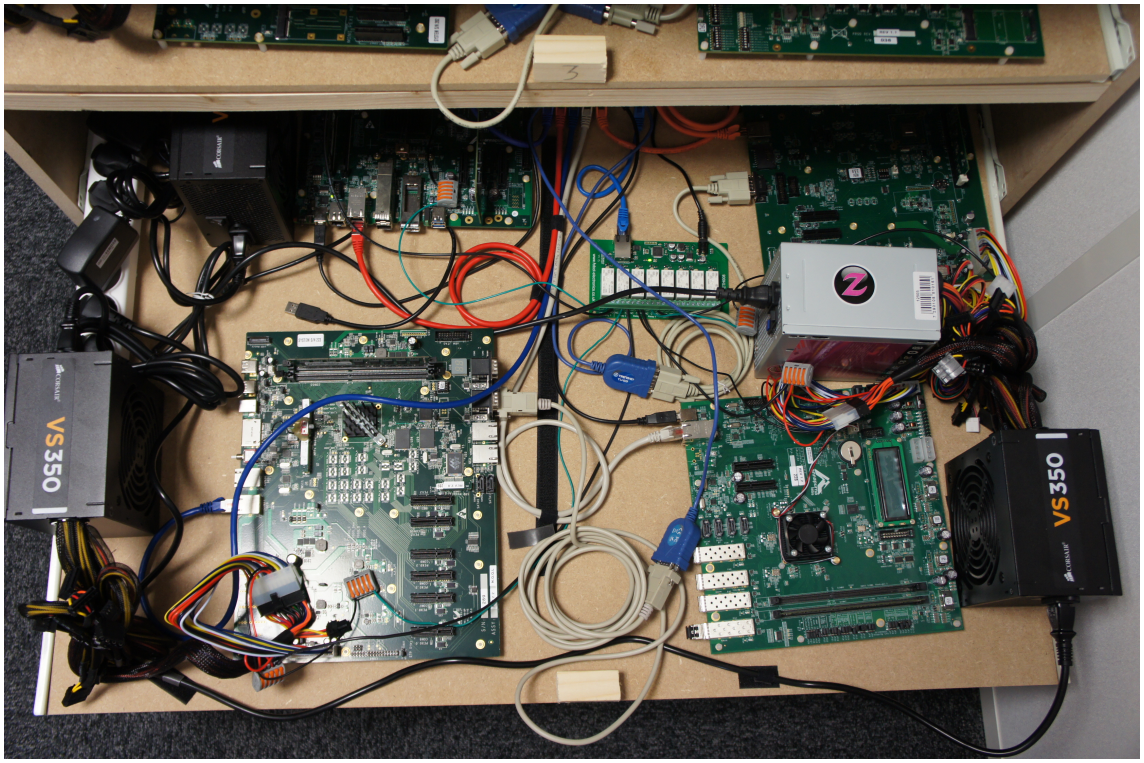

Figure 4.9: *Big drawer* in the lab

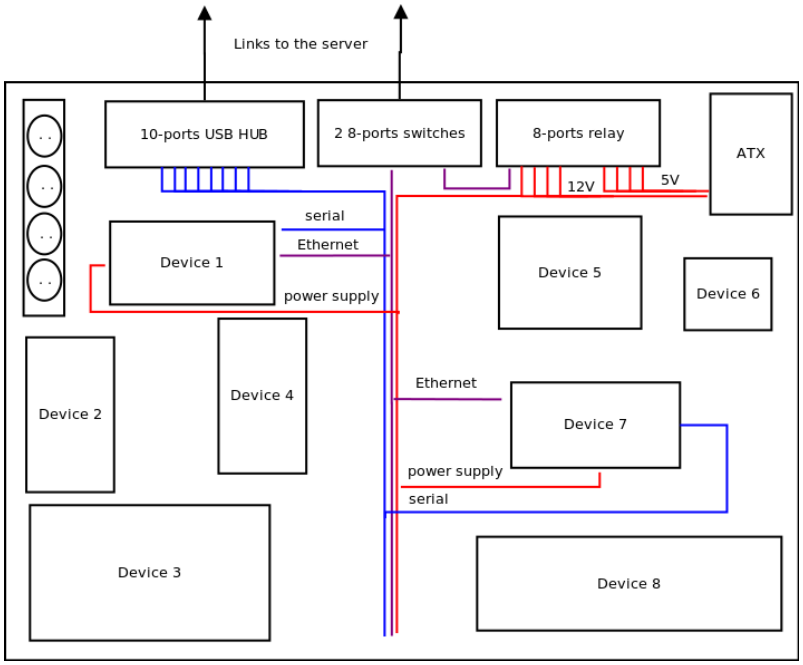

Figure 4.10: *Small drawer* example scheme

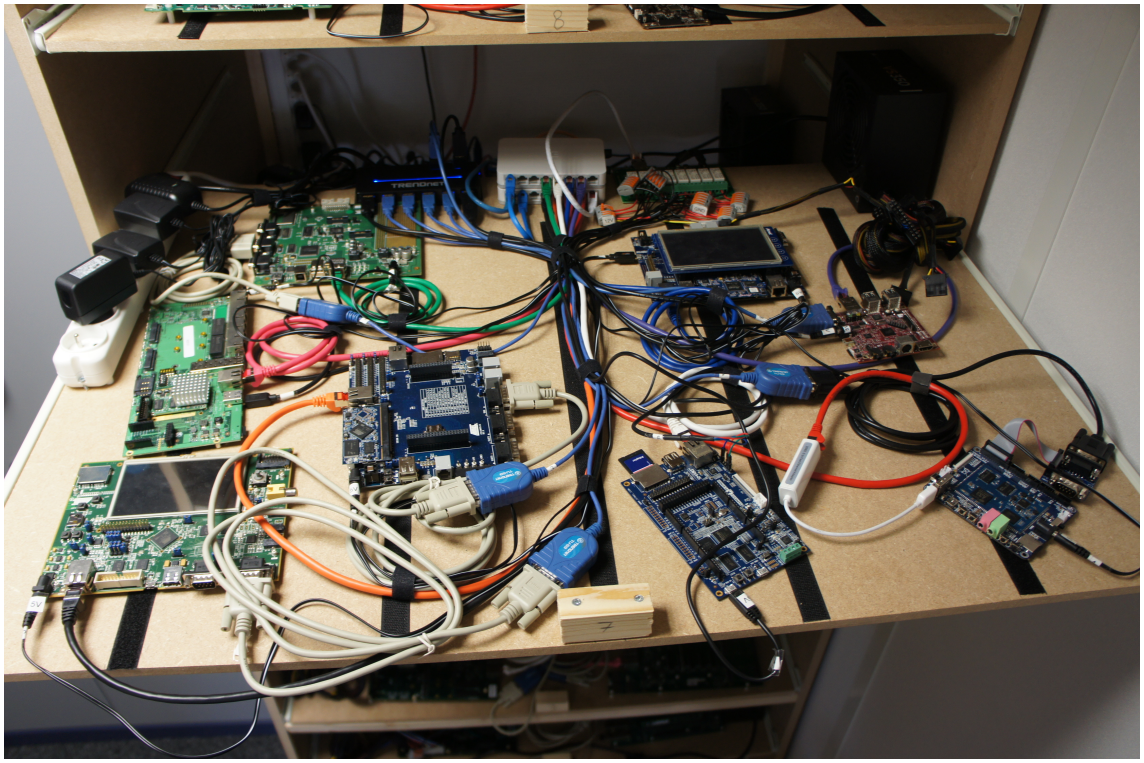

Figure 4.11: *Small drawer* in the lab

4.3.6 Process to add devices to the lab

We first make sure to have a fully working bootloader on the board. This bootloader has to be accessible "hands-free" when the board has its power reset (no button should have to be pushed) or we cannot add the board to the lab. The bootloader needs TFTP (or fastboot) support to get files from the server. We then need to know where to load the kernel, DTB and initramfs in the RAM of the board. Most of the time, it is in the environment of the bootloader but we might have to take a look at the User Manual or Datasheet of our board.

Then we have to physically add the board to the lab: wire the power supply to the relay and the board, connect the serial cable to the USB hub and the board and plug an Ethernet cable in both the switch and the board. After that, we test that the serial connection pops up in our LAVA dispatcher under the device we expect thanks to our udev rules. However, some boards might expose multiple serial interfaces in Linux and we have to make sure the one we connect to is the one used to debug the board and in this case, we need to modify our generic udev rules to take into account these board-specific settings.

The last requirement before adding our board to the lab is to find a working kernel to make sure the board has booted at least once, or we would not know if it is our board's

configuration or the kernel that is faulty. After a successful boot, we can install the board in the lab, add the board in LAVA and edit the scripts used by KernelCI to add support for the board[23]. Then, we make sure the modifications are valid by running the script locally and sending the created jobs to our lab.

For the ATMEL AT91SAMA5D2 Xplained:

- This creates jobs associated to the board in ./jobs

```
$ ./lava-kernel-ci-job-creator.py
↪  http://storage.kernelci.org/next/next-20150114/ --plans boot
↪  --targets at91-sama5d2_xplained
```

- and this will send the jobs to our LAVA server.

```
$ ./lava-job-runner.py --username LAVA_USERNAME --token LAVA_TOKEN
↪  --server LAVA_SERVER --stream /anonymous/mybundle/
```

- Then we need to follow the jobs' status and make sure they succeed before sending a patch for the added support of the board to KernelCI.

4.4 Challenges encountered

The first and most challenging problem encountered was to install LAVA whereas LAVA was migrating to a new model. For backward compatibility, LAVA has both models in the current code until it gets definitely removed in a few years. However, since LAVA developers are actively developing the new version, they did not take the time (at the time) to separate the old and new model's documentations. There were no way to distinguish which model the documentation was refering to. The main problem is KernelCI using only the old model. This cost me a lot of time to figure out why LAVA was not working in certain configurations or when following tutorials.

Among the other challenges, I had to deal with bootloaders. One of the tasks was to find the correct addresses where to load the kernel, Device Tree Blob and rootfs images in the RAM. It is often in bootloader's default configuration but sometimes I had to read the User Manual of the SoC to find at which address starts the RAM. Sometimes, even this data is not available so I had to empirically guess the address of the RAM. For some boards, the bootloader was outdated and impossible to update. Some of these bootloaders did not have bootz support, the boot method used to boot the images created when building the kernel. Instead, they have bootm support which is used to boot uImages, a format which includes the kernel and Device Tree Blob in the same image and specifies to which address in the RAM to load the uImage. Some bootloaders did not have DHCP, so we had to set fixed IP ranges for bootloaders without DHCP support.

[23] https://github.com/kernelci/lava-ci#add-board-to-kernelci

The biggest problems I had to take care of were board-specific problems.
Some boards like the Marvell RD-370 need a rising edge on a pin to boot meaning we cannot avoid pressing the reset button between each boot. To work ou this problem, we had to customize the board (mainly swap resistors) to bypass this limitation.
Some other boards lose their serial connection. This is the heart of the continuous integration since it is the only way to communicate with the board. Some lose it when resetting their power, problem we found acceptable to solve by infinitely reconnecting to the serial. However, we still have a problem with few boards which randomly close their serial connection without any reason. After that, we are able to connect to the serial connection again but it does not send any character. The only way to get it to work again is to physically re-plug the cable used by the serial connection. Unfortunately, we did not find yet a way to solve this bug.

As detailed before, we need a serial connection for each board (up to 50) in the lab. However, the Linux kernel of our server refused to bind more than 13 USB devices when it was time to create a second drawer of boards. After some research, I found out that the culprit was the xHCI[24] driver. In modern computers, it is possible to deactivate the support of xHCI in the BIOS but this option was not present in our server's BIOS. The solution to this problem was to rebuild and install a kernel for the server without the xHCI driver compiled. From that day, the number of USB devices is limited to 127 as in USB specification[25].

4.5 Conclusion

At the end of my internship at Free Electrons, the company had a lab since May 2016 contributing to KernelCI with a pool of 20 boards (on a maximum of 50). Custom tests were unfortunately not implemented because I started to work on the subjects presented in next chapters. However, our lab performed more than 60.000 boot tests in a three months span and our engineers and more globally maintainers are using the boot reports to help themselves for support.

4.5.1 Patches

- 32 new LAVA device types configuration files,
- 33 new supported devices in KernelCI,
- lava-ci documentation updates #1, #2,

[24] https://en.wikipedia.org/wiki/Extensible_Host_Controller_Interface
[25] http://www.usb.org/developers/docs/usb_31_052016.zip

5. Lavabo - LAVA Board Overseer

5.1 Context

Since our lab works in a completely autonomous manner thanks to LAVA, we wondered if we could take control of the lab remotely so anyone from Free Electrons could access the boards in the lab from anywhere. Lavabo is born from this idea and its goal is to take full remote control of the boards as it is done in LAVA: interface with the serial, control the power supply and serve files to the board. This software also has to be fully compatible with LAVA.

This also find a meaning in Free Electrons sometimes owning only one copy of a board. We still want to automatically test new kernels via LAVA and KernelCI on this board but we might also have to work once in a while on the said board. Before lavabo was developed, engineers had had to physically remove the board from the lab and re-plug all the cables on their desk. This is not an ideal process because, while the board is outside of the lab, the new kernels are not tested on it. At the moment, we have two engineers teleworking far from our Toulouse offices. These engineers might as well work on the board we own only once meaning we have to send them by postal way. Some other engineers also telework from time to time due to business travel and in that case, they would have to take the board (and the needed cables and power supply) with them.

5.2 Project

LAVA takes full control of a board but unfortunately, does not let users interact with it. When LAVA receives a job, LAVA looks among its dispatchers for online and available devices and if no device is online, drops the jobs it wanted to perform on this device. If there is a device online but is busy, LAVA queues the job. The software to develop should interact with LAVA so the same board can be used alternatively by users or by LAVA without interferences. Thus, thanks to LAVA's API, we need to tell LAVA to mark a device offline when a user controls it via our software and mark it online when the user has finished working on it.

Since we want our software to be as simple as possible and since LAVA already has all means to control the board, we chose to use the same tools used by LAVA and the configuration file of each board in lavabo to control the board.

5.2.1 Implementation

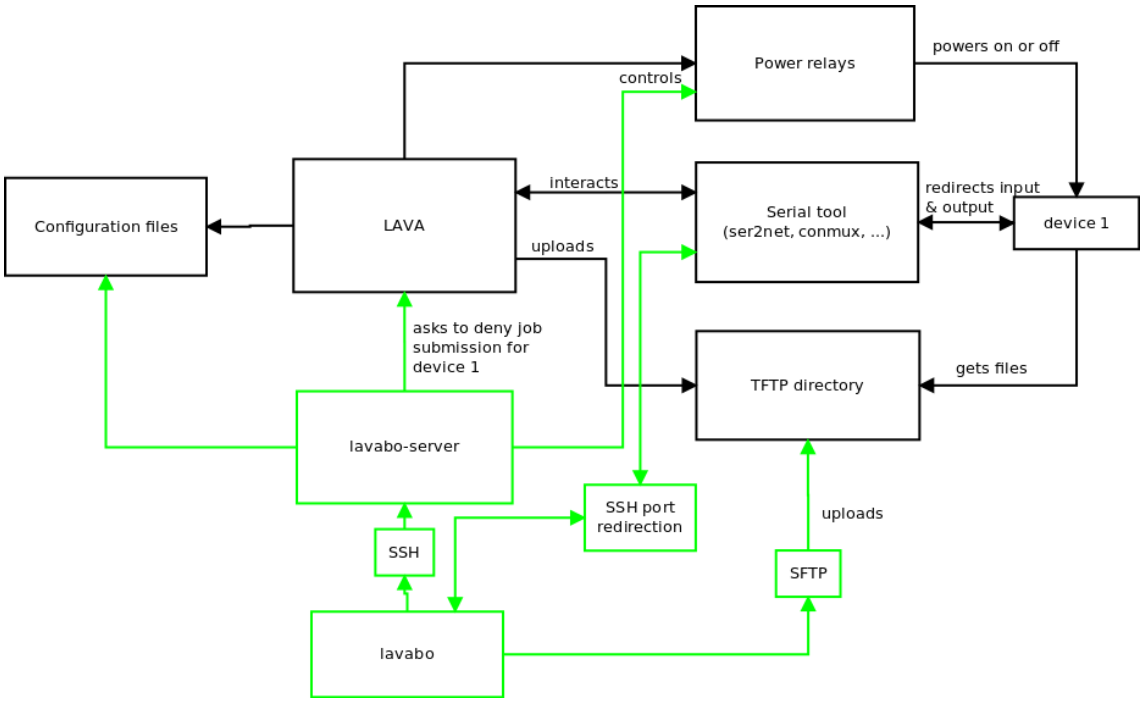

Figure 5.1: lavabo reuses LAVA tools and configuration files

Client-server interactions

Lavabo is a two parts software: a client installed on all engineers' laptop and a server hosted on the same machine as LAVA. This allows us to use the server part as a bridge to the host to reuse the same tools as LAVA.

All lavabo's logic is in the server part which takes care of calling the right tools directly on the server machine and making the right calls to LAVA's API. It controls the boards and interacts with the LAVA instance to reserve and release a board. The server side of lavabo will be executed every time someone logs in the server machine with a certain user via SSH.

The client is basically connecting via SSH on the server machine with a given user and sending the orders to the server which translates it into commands to execute or calls to make to LAVA's API. Since the client will be connected via SSH on the server with the right user, the server part of lavabo will be executed and the client will interact with the server by writing on server's stdin and reading on server's stdout. This behavior is borrowed from Attic[1]. The communication is done via JSON dictionaries so it is easy to parse.

[1] https://attic-backup.org/

Serial connection

This is perfect for all interactions except the ones that need a continuous stream of data like the serial connection. Writing a terminal functioning through TCP sockets is extremely difficult, but since we already have ser2net exposing serial connections on telnet ports on the server machine, we could do SSH port-forwarding to redirect these telnet ports to ports on engineers' laptop and open a local telnet process connecting to this local port.

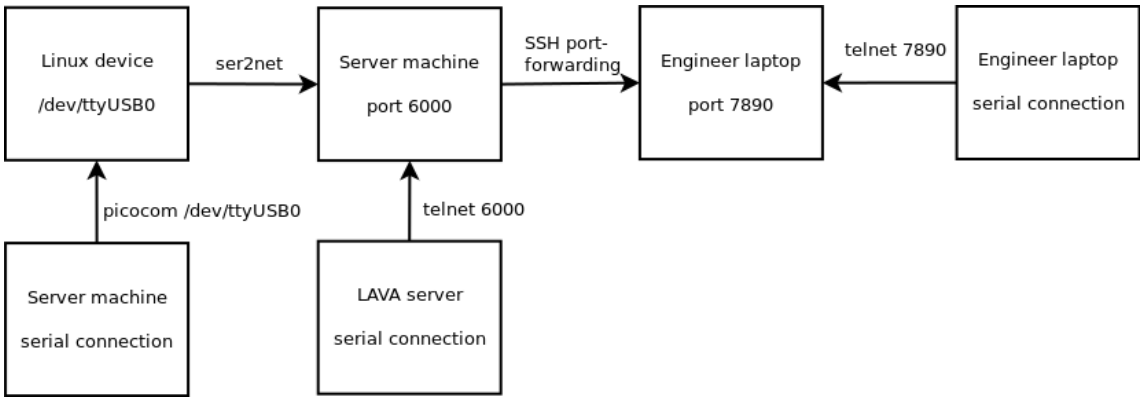

Figure 5.2: Different ways to connect to the serial

Interactions with LAVA

To use a board outside of LAVA, we have to interact with LAVA to tell him it cannot be used anymore. I therefore had to work with LAVA developers to add endpoints for *putting online* (release)[2] and for *putting offline* (reserve)[3] boards and an endpoint to get the current status of a board (busy, idle or offline)[4] in LAVA's API.

Interactions with boards

Now that we know how the client and the server interact and also how the server communicates with LAVA, we need a way to know which boards are in the lab, on which port the serial connection of a board is exposed and what are the commands to control the board's power. Fortunately, all this configuration has already been done in LAVA. To get the list of the available devices, one only needs to list all the files in LAVA devices' configuration file directory. In these configuration file, we can also find all we need to know: what is the port on which the serial connection is exposed, which command to execute to power off and on a board's power. Therefore, lavabo's server part lists and reads LAVA configuration files to avoid duplication of code or configuration.

[2] https://github.com/Linaro/lava-server/commit/2091ac9c3f9305f5a4e083d156c04d1f098ac2fa
[3] https://github.com/Linaro/lava-server/commit/6da3b4abf0c8a66f746f5a9264fc6ae4ad01f9fc
[4] https://github.com/Linaro/lava-server/commit/0d7742db6462fa5c8117556c46b1ce7ec41eda7e

The last requirement is to serve files to the board. Again, LAVA already has a directory accessible through TFTP from the boards which is one of the mechanisms used to serve files to boards. Therefore, the easiest and most obvious way is to send files from the client to the server and move the files to this directory. We could use either scp or SFTP for this mission. We went with the latter for reasons we expose in the next paragraph.

As explained earlier, we want the server part of lavabo to be automatically launched when a user connects via SSH to the server machine and there are only two means for automatic execution of a code when connecting in SSH:

- `$ ssh user@lavabo.server.com "my_command"`

  This has to be done on client side and specifies to SSH to execute `my_command` when connected to the remote. This is a serious security hazard because we do not control what kind of command can be executed in the remote (even if it is not run from root).

- use *authorized_keys* configuration file of ssh with a fixed command to execute

  This is set on server side but allows only one command to be set and needs SSH keys of the computer from which the user is trying to connect to the server machine. This command will be executed every time a user is connected via SSH.

We obviously chose the second option for security reasons but it also brings an interesting feature: authentication. However, this restricts the number of possible commands to be run to only one. Scp is not possible when a command is set in authorized_keys but SFTP is possible when the subsystem sftp is enabled in the configuration of OpenSSH. Trying to connect to the server machine via sftp will actually use the following command to connect: `ssh user@lavabo.server.com internal-sftp`. Fortunately, in the *authorized_keys* configuration file, it is possible to retrieve everything that is written after `ssh user@lavabo.server.com` with the `$SSH_ORIGINAL_COMMAND` variable[5]. Still, the SFTP server needs to be started but we can only have one command to run. Therefore, we pass `$SSH_ORIGINAL_COMMAND` to lavabo's server part as a parameter and spawns the SFTP server in a subprocess of lavabo's server.
There is a small subtlety for the SSH port-forwarding needed to connect to the serial of a board: the command is called even when willing to do only port-forwarding. After some investigation, it turned out the command is run only when the SSH connection is spawning a shell. Fortunately, we do not need a shell on the SSH remote to do port-forwarding so we just had to avoid to spawn a shell with the *-N* argument[6] when executing the SSH command in the client.

[5] http://unixwars.blogspot.fr/2014/12/getting-sshoriginalcommand.html
[6] http://linux.die.net/man/1/ssh

User authentication

Since the serial cannot be shared among several sessions, it is essential to guarantee a board can only be used by one engineer at a time. With the SSH keys stored in the *authorized_keys*, we can identify who is connected to the server machine and customize the executed command depending on the user. This allows to have a multi-user management. However, we also need a way to keep track of the user currently using the board to deny access to any new request from other users on this board. To perform such mechanism, we use a small SQLite database. When several engineers work with lavabo, they also have a separate TFTP upload directory so it is easy to know which files to serve to the board.

5.2.2 Typical workflow

1. `$ lavabo list`, to get the list of boards in the lab,
2. `$ lavabo reserve am335x-boneblack_01`, to put the board named *am335x-boneblack_01* virtually outside of the lab (offline in LAVA),
3. `$ lavabo upload kernel`, to upload the file name *kernel* in the user's TFTP directory,
4. `$ lavabo serial am335x-boneblack_01`, to connect to the board named *am335x-boneblack_01*,
5. `$ lavabo reset am335x-boneblack_01`, to reset power of the board named *am335x-boneblack_01*,
6. `$ lavabo power-off am335x-boneblack_01`, to power off the board named *am335x-boneblack_01*,
7. `$ lavabo release am335x-boneblack_01`, to virtually put back the board named *am335x-boneblack_01* in the lab (online in LAVA);

It is possible to use auto-completion to get the name of a board or to store the name of the board in an environment variable name `LAVABO_BOARD` with the following line: `$ export LAVABO_BOARD=am335x-boneblack_01`

5.3 Conclusion

This tool is available on the company's GitHub[7] under the GNU GPLv2 license and has already been used a fair number of times by the Free Electrons' engineers.

However, there are some possible improvements[8]:

[7] https://github.com/free-electrons/lavabo
[8] https://github.com/free-electrons/lavabo/issues

- control boards which boots only with a NFS rootfs,

  In the lab, there are some boards which can boot only when using a root filesystem mounted over NFS. However, there is no possibility yet to add an NFS directory to the server machine from lavabo. This problem prevents the use of these boards via lavabo.

- automatically close serial connection when releasing a board;

  When following the typical workflow, engineers often forgot to stop `$ lavabo serial am335x-boneblack_01` which means they keep the serial connection open and nobody can use it even if the board is marked as available.

6. Support for a new board

6.1 Context

One of our clients wanted to release a new version of its product in a midterm future. They sent an evaluation board so we could work on adding support for this board in both Linux and U-Boot.

An evaluation board is a board which is offered by some vendors to build a beta version of the new product they want to release. The main advantage is the possibility to test the product before building the hardware. This means they can test the software stack before building the multiple beta iterations of their product's hardware and test the hardware to validate it actually meet their needs. After the hardware has been validated, they can change its form-factor and adjust the software to reflect those changes. In short, the evaluation board is a must-have in the creation process of a product: it saves both money and time by not having to actually build your product to be able to test it.

The new version of the product will share most of its components with the Allwinner Parrot EVB R16, the evaluation board we have to add support for. This board is shipped with one of the newest SoCs of Allwinner, the R16 which is assumed to be very close to an Allwinner A33 SoC. We assume this closeness with the A33 because there is no publicly available documentation yet for this SoC but the few information we have let us guess it is sharing a lot of similarities with the A33.

Yet, there is no support for the Allwinner Parrot EVB R16 in either the Linux kernel or U-Boot, but fortunately, the A33 is supported in both projects. Since they are assumed to be pretty much the same SoC, we can start from the files needed by the A33 and tweak them bit by bit to make all the components work.

The goals of this project are first to be able to interact with the U-Boot bootloader in order to then boot a Linux kernel.

6.2 Project

正面/反面：

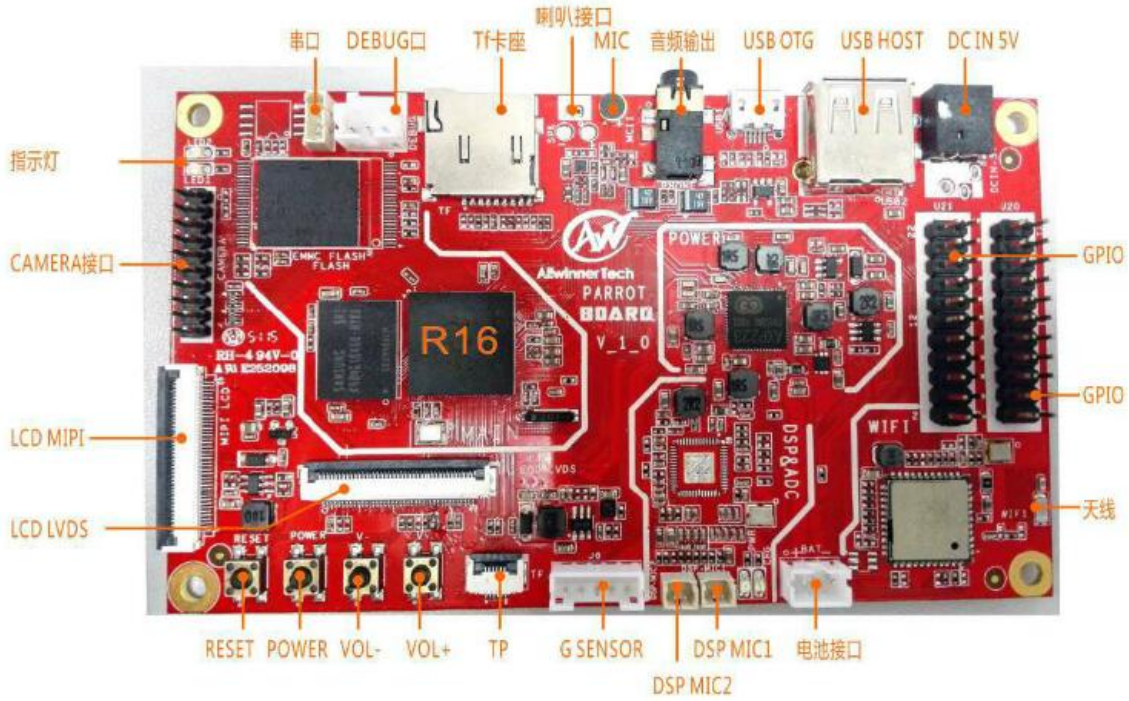

Figure 6.1: The Allwinner Parrot EVB R16 and all its components

The Parrot Board is an evaluation board with an Allwinner R16 (assumed to be close to an Allwinner A33), 4GB of eMMC[1], 512MB of RAM, USB host and OTG[2], a WiFi/Bluetooth combo chip, a micro SD card reader, 2 controllable buttons, two controllable LEDs, an LVDS port (mainly for use with an LCD) with separated backlight and capacitive touch panel ports, an audio/microphone jack, a camera CSI port, 2 sets of 22 GPIOs and an accelerometer.

6.2.1 Add support in the bootloader source code

To support a new board, the first requirement is to have a working bootloader from which we will later be able to boot the Linux kernel. There are multiple open source bootloaders[3] available but only a few are widely used. At Free Electrons, we work substantially with

[1] https://en.wikipedia.org/wiki/MultiMediaCard#eMMC
[2] https://en.wikipedia.org/wiki/USB_On-The-Go
[3] https://en.wikipedia.org/wiki/Comparison_of_boot_loaders

U-Boot[4] and in few occasions with Barebox[5]. U-Boot is today the most widely used bootloader in embedded systems.

When an Allwinner SoC (named sunxi) is powered on, it loads a small ROM code embedded in the processor which in turn will load a bootloader in SRAM. In Allwinner SoCs, the SRAM is too small to contain a bootloader with all features so we need a first stage bootloader (SPL) which will mainly initialize the DRAM and load a full bootloader in it.

The next step is to find on which pins of the board the serial communication is working so we can communicate with the bootloader of the board (if there is one).
Once that's done, we have to figure a way to place the bootloader we'll compile in one of the locations the ROM code will try to load the bootloader from. On Allwinner SoCs, we can either load images in RAM by using the FEL mode[6] or create a bootable SD card[7]. Entering the FEL mode is a board specific manipulation and, unfortunately, I did not have enough documentation on the board to enter in such mode. Therefore, every time I compiled a new bootloader, I had to recreate a bootable SD card with the newest SPL and full versions of U-Boot.

Here is the process of creating a bootable SD card for Allwinner SoCs:

1. compile U-Boot,

```
$ make CROSS_COMPILE=arm-linux-gnueabihf- my_config
$ make CROSS_COMPILE=arm-linux-gnueabihf- -j$(nproc)
```

2. identify the SD card device name,

```
$ dmesg
[59358.008977] mmc0: new high speed SDHC card at address b368
[59358.019617] mmcblk0: mmc0:b368 NCard 3.72 GiB
```

3. erase the part of the SD card with the SPL bootloader,

```
# dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=1
```

4. write the SPL version of U-Boot to the SD card,

```
# dd if=u-boot-sunxi-with-spl.bin of=/dev/mmcblk0 bs=1024 seek=8
```

5. partition the SD card,

[4] http://www.denx.de/wiki/U-Boot/
[5] http://www.barebox.org/
[6] http://linux-sunxi.org/FEL
[7] http://linux-sunxi.org/Bootable_SD_card

```
# blockdev --rereadpt ${card}
cat <<EOT | sfdisk ${card}
1M,16M,c
,,L
EOT
```

We'll then have two partitions: mmcblk0p1 for data accessible by U-Boot SPL and full bootloader and mmcblk0p2 for external storage.

6. format partitions,

```
# mkfs.vfat /dev/mmcblk0p1
# mkfs.ext4 /dev/mmcblk0p2
```

This will format the first partition in FAT and the second in EXT4.

7. copy the full bootloader to the SD card,

```
# mount /dev/mmcblk0p1 /mnt
# cp u-boot.bin /mnt
# umount /mnt
```

The U-Boot bootloader needs to be configured with the right settings so it is compiled for the correct architecture, and can find how to configure the RAM, the USB ports, the internal storage and so on.

The first settings to get right are the serial, the architecture (ARM), the platform (Allwinner SoC) and the DRAM initialization. These settings' values can be found in vendor documents. Once we have a bootloader which boots without errors, we can add bit by bit new settings to enable more components. Between each compilation, it is important to test the bootloader and all its components on the board to see if we are not introducing regressions.

The core components of the bootloader are the ones which can be used to load files from external storage, internal storage and other devices. Without these, it is impossible to make a kernel boot since we cannot get the file in the bootloader. U-Boot can get files from NAND, eMMC, SD card, USB from USB sticks or fastboot, the network via TFTP and serial. On the Allwinner Parrot EVB R16, we have 4GB of eMMC, an USB host port, a micro USB OTG port and a micro SD card reader which can help to retrieve files. We could use the USB OTG port to simulate a network interface in the bootloader and get files over the network via TFTP.

Once we get all those important components, we can send the new files and modifications to the U-Boot community so they can review it, comment it, maybe ask for some modifications and validate it so it can be added to the source code.

6.2.2 Add support in Linux kernel source code

The client wanted an initial support in the kernel. The aim is to make the kernel detect hardware components and map them to existing drivers. If a component does not have yet an existing or working driver, there is no need to develop one at the moment.

As presented before, the R16 is assumed to be close to the A33 and most components of the Parrot EVB R16 are also in the Sinlinx SinA33 (which embeds an A33). The easiest and most straight-forward process is to start from the files supporting the architecture, defining the Device Tree and drivers needed for the Sinlinx SinA33, strip all its components leaving only the ones required to boot and add component by component until everything is included.

There would have been more files required if the SoC was not actually close to a known one, but fortunately, in this case, the only needed file to make the kernel boot is a Device Tree file. Each node (even the root node) has a *compatible* property which is used by an authority, the platform framework, when reading the whole Device Tree to match registered drivers' compatibles against this node's compatible. When a driver compatibles match the node's compatible, the authority probes the driver, making the driver able to access all settings defined in these nodes. The goal is to give as many relevant hardware information as possible in the Device Tree nodes so the drivers have enough data to know how to handle the device. These information can be links to other nodes (power regulators or pin muxing for example), to hardware interrupts or address of registers.

One important thing to add is the node in charge of the UART device which is used to communicate with the board via serial. If this node is wrongly set, we would not be able to interact with the kernel which is absolutely useless. Once this device is correctly set, we can add more components such as the USB port, the SD card reader, WiFi and Bluetooth chip, power regulators and the micro USB OTG port.

Two of the things to keep in mind when writing a Device Tree file are pin muxing and power regulators. In a SoC, there are a lot of hardware components but there is only a limited number of pins on it. These two statements seem to be completely opposite but pin muxing exists to make them possible at the same time. Pin muxing is a hardware workaround which allows a sam pin of the SOC to be wired to different hardware components. The SoC decides which controller is activated and has the right to communicate with it through this pin. This obviously needs some thinking when designing the board because two (or more) components on the same pin cannot work at the same time.
On a board, components are not all powered with the same voltage and sometimes accept voltages in a certain range or only a fixed voltage. If we mess up the power regulators in the Device Tree, we can burn components. Having a voltages range in some power regulators is useful when trying to reduce consumption of a component or make it temporarily more powerful.

6.3 Conclusion

Adding support for the Allwinner Parrot EVB R16 in both U-Boot and Linux kernel were my first interactions with the Linux and U-Boot communities. These interactions made me realize how important it is to ask other persons to comment, criticize, discuss and validate my code.

For this project, the different struggles were the lack of proper documentation (only electrical schematics and a FEX file[8]) which were sometimes contradicting each other and the need to rewrite the SD card each time a change was made to U-Boot. The rewriting of the SD card was painfully slow when adding support for the board in U-Boot.

To sum up, the initial support for this board has been added to U-Boot and Linux kernel. However, the support for the accelerometer was not added since the documentation does not mention any brand or model name and I could not find it by trying to guess it.

Patches

- Support in U-Boot for Allwinner Parrot R16 EVB,
- Support in the Linux kernel for Allwinner Parrot R16 EVB,

[8] http://linux-sunxi.org/Fex_Guide

7. Driver for Allwinner SoCs' ADC

7.1 Context

An Analog to Digital Converter or ADC is a device which converts a signal often representing a voltage or a current to a digital value. This component is really useful in data processing in embedded systems because most of them only understand discrete values. For example, a variable resistor such as a thermistor, a photoresistor, a variator or even a touchscreen controller. These resistors reduce current flow depending how they react to temperatures or light for example. The ADC works by comparing the input signal with a reference signal which is either ground or from another input (thus called differential ADC). The known function representing their behavior can then be applied to the value returned by the ADC to get a "human-readable" value. An ADC can be used as a touchscreen controller because of the inner working of the touchscreen[1].
In the Linux kernel, an input of an ADC is called a channel.

The Allwinner SoCs all have an ADC that can also act as a touchscreen controller and a thermal sensor. The first four channels can be used either for the ADC or the touchscreen and the fifth channel is used for the thermal sensor. However some ADCs only embed the thermal sensor.

There is currently a driver[2] which handles the ADC in touchscreen mode and the thermal sensor but there is no access to the ADC feature at all. This driver also only works for the SoCs which can act as an ADC and a touchscreen controller.

The goal is to write a generic driver for all Allwinner SoCs, even the one acting only as thermal sensor, which can handle all features of the ADC, be it the touchscreen controller, the ADC controller or the thermal sensor and replace the current driver.

7.2 Project

Allwinner A13, A20 and A31 ADCs have all features (ADC and touchscreen controllers and thermal sensor) whereas the A23 and A33 ones only have the temperature sensor. They all share the same registers addresses and behavior except for the A31 which has a slightly different address for one of its register but they have different functions for computing the real value in Celsius of the thermal sensor. In the long term, the H3 could also use this driver but with some modifications since there are a lot of changes in registers addresses but its behavior is globally the same as the aforementioned SoCs.

The ADC cannot operate at the same time as an ADC controller and as a touchscreen

[1] https://en.wikipedia.org/wiki/Resistive_touchscreen
[2] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/input/touchscreen/sun4i-ts.c

controller since the four channels of the Allwinner SoCs' ADC are used by either of them. Therefore, there must be an exclusion mechanism. Moreover, the thermal sensor is only returning valid values when the ADC operates as a touchscreen controller.
To switch between touchscreen and ADC modes, we need to poke a few bits in registers and (de)activate an interrupt which notifies when a press is released from the touchscreen.

Since a touchscreen may not be attached to the ADC as well as the ADC may only have one channel for the thermal sensor, the driver has to be modular, deactivating touchscreen or ADC controllers when they are not present physically in the board. The perfect system would be to have different drivers in different Linux kernel subsystems to handle the different parts of the ADC, one for the temperature, one for the ADC controller and one for the touchscreen controller. Since they share almost all the code and are part of the same component, they have to cooperate before communicating with the it. This seems complex but there is one type of driver which handles this perfectly: the Multi Function Device or MFD driver.

As presented in the previous chapter, drivers are probed if one of their compatibles is found in one of the Device Tree nodes. However, it would be tedious to edit the Device Tree each time we want a Linux kernel with or without a touchscreen controller for example. Fortunately, MFD drivers are capable of probing what are called subdrivers or MFD cells without using the Device Tree. The subdrivers are probed according to which compatible they match and are passed resources such as interrupts and properties as if they were probed from matching compatibles in a Device Tree.

Regmap functions are able to map memory and share it among different drivers which suits perfectly sharing memory between subdrivers of an MFD driver. It offers shared mechanisms for memory writing and reading as well as interrupt handling which are everything we need for our subdrivers.

The global principle is the following:

- the MFD driver is probed thanks to compatible present in the Device Tree,
- the MFD driver maps the memory and interrupts,
- the MFD driver probes subdrivers with shared memory and interrupts;

The Allwinner SoCs' ADC driver handles most of the logic of all drivers because most of it is reading data from the ADC registers or switching modes. The other drivers, be it the touchscreen or the temperature driver, interact with the ADC driver for switching modes and request the ADC driver for data. The ADC driver has to explicitly share its channels with drivers and then and only then the other drivers will be able to request data from it. It is also possible to request data from the ADC driver from the system as a user by opening some special files (sysfs) which are handled by the driver and are basically triggers for data requests.

Touchscreen driver

The touchscreen driver is expecting data structured as an array of touch coordinates (X and Y) to notify the input framework of the Linux kernel where a touch has occurred. When the touchscreen is activated, it will trigger the mode switch in the ADC driver to change to touchscreen mode and ask the ADC driver to start to continuously read data from the hardware component. In the ADC driver, a buffer of touch coordinates representing the data in the hardware FIFO of the ADC is filled in with coordinates until the FIFO is emptied and then sent to the touchscreen driver which knows how to process the data. The latter driver also has to notify the input framework when the touch has ended, basically when the finger released all pressure from the touchscreen. It does so by registering a handler for an hardware interrupt which occurs when there is an "up" event, understand a touch being released.

Temperature driver

On the other side, there is already a driver named iio_hwmon which exposes the temperature of the SoC by reading channels from different ADCs. Since our ADC driver exposes the channel used by the thermal sensor, we can use our ADC driver in conjunction with iio_hwmon to let Linux users request the internal temperature of the SoC.

7.3 Conclusion

After a few weeks, I already had a working project with all the needed drivers but then I had to make my code *mergeable* in the Linux kernel source code by proposing a patch to the Linux kernel community. This was my first intensive interaction with this community and it is still going on at the moment of the redaction. I learned a lot from all the comments coming from different developers from different subsystems about code readability and factorization as well as when to explicitly comment code and respect rules set up in each subsystem.

After four versions of my patches, which is fairly common, I am confident of getting them validated by the community and thus being merged to the Linux source code. The patches for the touchscreen still need a lot of work with the community before it is determined to be acceptable.

Most of changes in the four versions were cosmetic but there were also few major mistakes such as NULL pointers dereferencing or functions used not as they should be. In addition, my patches require modifications in some frameworks and opened discussions on how we should do it. The difficult part is to separate the requirements for my patches to work and *would-be-great-to-have* features in framework, which is basically learning how to say *no* or *later* to maintainers or developers.

8. Conclusion

Les différents projets que j'ai réalisés au cours de mon stage effectué au sein de l'entreprise Free Electrons, ont porté sur plusieurs aspects.

Le sujet principal de mon stage était la création d'une ferme (ou laboratoire) de systèmes embarqués qui doit contribuer au projet KernelCI permettant d'apporter l'intégration continue au noyau Linux. L'objectif de ce projet est de tester le noyau Linux dans ses différentes configurations sur l'ensemble des systèmes embarqués mis à disposition par des laboratoires partenaires. Depuis ses plus jeunes années, Free Electrons est un fervent contributeur aux initiatives Open Source et particulièrement quand elles touchent leur cœur de métier, à savoir le développement et la formation pour Linux. La réalisation de cette ferme en conjonction avec le projet KernelCI permet aux ingénieurs de l'entreprise de faire de la veille sur les produits des clients sans perdre le temps de recompiler et tester les différentes configurations du noyau Linux. C'est un gain de temps considérable en plus d'un suivi des produits de meilleure qualité. De plus, certains ingénieurs de Free Electrons sont également mainteneurs et profitent donc de ce projet pour s'aider dans leurs tâches. Enfin, c'est également une excellente opportunité d'aider la communauté du noyau Linux en mettant à disposition des cartes de développement d'anciens clients qui ne sont pas maintenues par nos ingénieurs. Le site du projet[1] est consulté régulièrement par la communauté des développeurs Linux et c'est une bonne vitrine pour l'entreprise de faire partie des 10 laboratoires partenaires du projet. Cependant, il reste une partie du sujet de stage à faire et qui pourrait bénéficier grandement aux ingénieurs de Free Electrons : implémenter une copie du projet KernelCI pour faire de l'intégration continue de nos noyaux Linux avec nos tests plus approfondis. Free Electrons va également recevoir de la visibilité parmi la communauté Open Source avec la publication de plusieurs articles ainsi qu'une présentation à l'Embedded Linux Conference Europe 2016 sur mon sujet de stage, présentant les différentes démarches et processus de création du laboratoire.

Parmi les sujets secondaires figurait le développement d'un outil permettant de prendre contrôle à distance les différents systèmes embarqués qui sont présents dans le laboratoire. Ce nouvel outil permet déjà depuis quelques mois aux ingénieurs de Free Electrons de travailler sur des systèmes embarqués auxquels ils ne peuvent pas accéder, que ce soit pour cause de participation à des conférences, à des formations ou de télétravail. Ce logiciel permet également de mieux partager le travail sur les différentes cartes de développement ajoutées au laboratoire. En effet, il est désormais beaucoup plus simple d'"échanger" une carte puisqu'il n'y a plus à la débrancher et la rebrancher, il suffit de se déconnecter de l'outil pour permettre à une autre personne de prendre le contrôle de cette même carte.
J'ai également ajouté le support d'une carte de développement à la fois dans le noyau Linux et le bootloader U-Boot pour un client qui souhaiterait faire une nouvelle version de son produit basé cette fois-ci sur la-dite carte de développement. Enfin, j'ai travaillé sur le développement d'un driver de Convertisseur Analogique-Numérique ou CAN qui sert à la fois de CAN, de contrôleur d'écran tactile et de sonde thermique du SoC. Avant que je ne développe ce driver, un autre driver existait déjà mais celui-ci ne permettait pas

[1] https://kernelci.org/

l'utilisation du CAN comme simple CAN, uniquement comme contrôleur d'écran tactile et sonde thermique.

Ce stage a été pour moi une première expérience très enrichissante dans le monde de l'informatique embarquée et a confirmé mon désir de travailler dans ce domaine. Il m'a permis de découvrir une bonne partie des différentes étapes de développement pour systèmes embarqués, allant du flashage du bootloader, de la compilation du noyau Linux, du développement de drivers et Device Tree à la création d'un root filesystem contenant les outils nécessaires au client. Une des plus grandes difficultés a été l'absence de documentation et quand il y en avait, de son incohérence avec d'autres.
Ce stage a également été cause de mes premières interactions avec les communautés Open Source et m'a permis de me rendre compte de l'utilité de partager son code pour les faire commenter, critiquer, discuter et valider par les développeurs de la communauté. Malgré une totale autonomie sur le sujet principal du stage qui a des fois été frustrante et difficile à gérer, la réussite des différents sujets ainsi que les multiples retours des différentes communautés et les nombreuses aides des ingénieurs de l'entreprise sur les autres sujets me permettent de tirer un bilan très positif de cette expérience à Free Electrons.

Bibliography

- *Device Tree for Dummies* slides from Thomas Petazzoni presented at Embedded Linux Conference Europe 2013 and Embedded Linux Conference 2014, https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf
- Linux Cross Reference, used to search for variables or functions in Linux kernel source code, http://lxr.free-electrons.com
- Linux kernel documentation, https://git.kernel.org/cgit/linux/kernel/git/tomba/linux.git/tree/Documentation
- *IIO, a new kernel subsystem* slides from Maxime Ripard presented at FOSDEM 2012, https://archive.fosdem.org/2012/schedule/event/693/127_iio-a-new-subsystem.pdf
- *Supporting multi-function devices in the Linux kernel: a tour of mfd, regmap and syscon APIs* slides from Alexandre Belloni presented at Embedded Linux Conference Europe 2015, http://events.linuxfoundation.org/sites/events/files/slides/belloni-mfd-regmap-syscon_0.pdf
- Wiki for Allwinner SoCs (sunxi) based boards, http://linux-sunxi.org
- Wiki for ATMEL SoCs (at91) based boards, http://www.at91.com/linux4sam/bin/view/Linux4SAM/
- Documentation of LAVA, https://validation.linaro.org/static/docs/
- ATX power supply specifications, http://www.formfactors.org/developer/specs/atx2_2.pdf
- Allwinner SoCs documentation, https://github.com/allwinner-zh/documents
- Article on how one of LAVA developers built his own lab, https://linux.codehelp.co.uk/home-server-rack.html

- Linux Development process,
  https://www.kernel.org/doc/Documentation/development-process/2.Process
- *#linaro-lava* chan on http://irc.freenode.net,
- *#kernelci* chan on http://irc.freenode.net,

Mots clefs

(17) SSII
(07) Informatique
(11) Développements logiciels
(18) Logiciel – système d'exploitation

Quentin Schulz

Rapport de stage ST50 - A2016

Résumé

Dans ce rapport, je présente les différents projets que l'on m'a confiés durant mon stage. J'ai construit l'armoire qui allait accueillir une ferme de boards, équipé la ferme des différents moyens de prendre entièrement contrôle de systèmes embarqués de manière autonome et installé un logiciel permettant de faire de l'intégration continue sur ces systèmes embarqués. J'ai ensuite développé un outil de prise de contrôle à distance de ces systèmes sans utiliser le logiciel susmentionné. Enfin, j'ai eu l'opportunité d'ajouter le support d'une carte dans Linux et U-Boot ainsi que de développer un driver de CAN et d'écran tactile pour les SoCs Allwinner.

Ce stage a été particulièrement enrichissant et m'a permis de développer des compétences dans l'informatique embarquée, un domaine qui m'était relativement nouveau, ainsi que dans la gestion de projets Open Source et les différentes interactions avec leurs communautés et m'a également permis d'apprendre comment lire et comprendre différents documents techniques (datasheets et manuels utilisateurs).

Free Electrons
25 boulevard Victor Hugo
31770 Colomiers
http://free-electrons.com/

utbm
université de technologie
Belfort-Montbéliard