Linux Device Driver Development:
a report from the trenches

Marcelo Schmitt

Capstone project

mac 0499

Program: Computer Science
Advisor: Prof. Dr. Paulo Meirelles
Coadvisor: Prof. Dr. Fabio Kon

During this work, the author was supported by the
São Paulo Research Foundation - Brazil (FAPESP)

São Paulo
December 05th, 2019

Acknowledgments

For supporting me throughout this work I acknowledge my mother Marlene and my brother Victor, who were patient in putting up with my complaints over a year of development. A great thank to Rodrigo Siqueira Jordão for teaching me the basics of kernel development and encouraging me to go further; my college mentor Paulo R. M. Meirelles for being understanding and supportive; and to everyone who participated at FLUSP (FLOSS at USP), for their partnership in learning how to contribute to free software projects. Last but not least, a special thanks to Stefan Popa, Alexandru Ardelean, Dragos Bogdan, Jonathan Cameron, and Rob Herring, for providing me guidance to develop a high-quality device driver for the Linux kernel. Thank you all.

Resumo

Marcelo Schmitt. **Linux Device Driver Development**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2019.

O propósito deste trabalho é desenvolver um driver no kernel Linux para controlar a operação de dispositivos AD7292. O AD7292 é uma especificação de circuito integrado que descreve um chip contendo ADC, DACs, sensor de temperatura, e GPIOs, sendo recomendado como um sistema de monitoramento de sinais analógicos e controle de outros dispositivos. Para atingir o objeitvo proposto, foram desempenhadas uma série de atividades coerentes com as práticas de desenvolvimento de software livre. A metodologia adotada contou com a leitura do datasheet do circuito integrado, consulta à documentação do projeto, análise de outros drivers com funcionalidade similar, submissão de versões preliminares do código fonte para a revisão por membros da comunidade, revisão bibliográfica, consulta a sites e blogs. Devido ao êxito nesse processo, o driver desenvolvido foi aceito pela comunidade Linux e estará disponível a partir da versão 5.5 do kernel Linux. Esta primeira versão permite tomar leituras analógicas ao comando do usuário ou de uma aplicação. O término deste trabalho abre caminho para uma série de novos trabalhos na direção de produzir um driver capaz de tirar proveito de outras funcionalidades de dispositivos AD7292.

Palavras-chave: Linux. device driver. free software.

Abstract

Marcelo Schmitt. **Linux Device Driver Development:: *a report from the trenches***. Undergraduate Thesis (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2019.

The purpose of this work is to develop a Linux kernel device driver to control the operation of AD7292 devices. The AD7292 is an integrated circuit specification that describes a chip containing ADC, DACs, temperature sensor, and GPIOs, which is recommended as an analog signal monitoring and control system for other devices. To achieve the proposed goal, a series of activities were performed consistent with the practices of free software development. The methodology adopted included reading the integrated circuit datasheet, consulting the project documentation, analyzing other drivers with similar functionality, submitting preliminary versions of the source code for review by community members, bibliographic review, consulting both websites and blogs. Due to the success of this process, the driver developed has been accepted by the Linux community and will be available from version 5.5 of the Linux kernel. This first version allows one to obtain analog readings through userspace applications. The completion of this work paves the way for a host of new work toward producing a driver that can take advantage of other AD7292 features.

Keywords: Linux. device driver. free software.

Contents

Chapter 1

Introduction

Linux[1] is one of the most significant software projects in history and has been present in the development of various areas of computing. Since November 2017, all top 500 supercomputers use Linux [34]. As of 2017, GNU[2]/Linux operating systems run 90 percent of the public cloud workload [17]. As of April 2018, Linux was used on about 70 percent of devices employed in IoT applications [16].

Around Linux, vibrant free source communities grow. These groups nest many developers who contribute to making people's lives better; by the challenge of dealing with low-level software in a cutting-edge system; or just by the fun and learning provided by the development process. By opening the source code and ensuring it is always free, Linux kernel developers make it possible for the system to be completely auditable, reducing security issues and mitigating user surveillance. Also, free software encourages knowledge sharing and collaborative development, empowering both developers and users.

Linux is an operating system kernel responsible for controlling the operation of computer hardware and granting useful abstractions for resource management. The Linux source code is organized into subsystems, each responsible for smaller portions of the functionality provided by the kernel. Within many of these subsystems, there are software components responsible for managing distinct hardware devices. These components are called device drivers.

Drivers are fundamental to the functioning of Linux. Each hardware component must have an appropriate driver to work correctly. As new devices come to the market, new drivers must be developed to allow Linux to operate with this equipment. Thus, developing drivers for Linux increases the number of tools that can be used with free software, makes the technology more accessible, promotes knowledge production, foster the development of the free software community.

[1] https://www.linuxfoundation.org/projects/linux/
[2] https://www.gnu.org/

1.1 Objective

The purpose of this work is to develop a kernel device driver for AD7292[3] devices. AD7292 chips were designed by Analog Devices Inc. to work as a general-purpose monitoring and control system with an ADC, four DACs, a temperature sensor, and up to twelve GPIOs.

1.1.1 Practices

The development principles that guided this work arise from the combination of several open-source development practices. These practices and their importance are briefly described next:

Driver source code reading

Some device drivers may use similar data structures, mainly when they handle devices with related functionality. Because of that, reading source code from existing drivers may help to develop software for unsupported devices. More generally, studying how other drivers work conduces to a wider understanding of the surrounding subsystem. Since code in the Linux kernel is reviewed by experienced developers, the insights learned from in-tree code may avoid starting a discussion about established solutions. Thus, taking advantage of expertise from other drivers may improve community conversations and shorten development time. Specifically in this work, reading drivers from the Industrial Input Output (IIO) subsystem was very helpful. Some analog-digital converter drivers from which inspiration was drawn are ad7768-1, ad7124, ad7793, ad7923, ad7766, and ad7949.

Datasheet reading

The document that describes the characteristics of an integrated circuit is referred to as the part datasheet. A datasheet often specifies technical details about a product, such as main components, power supply, operation modes, connectivity with other devices, etc. Reading the datasheet is of extreme importance during device driver development since it is necessary to understand hardware details to correctly handle device operation. For instance, the AD7292 datasheet summarizes chip features, explains the overall device operation, list specifications for ADC and DAC components, documents register structure, depicts the communication protocol supported, details other relevant chip features.

Code review

Code reviews offered by the Linux community provide valuable tips on how to improve the software being developed. They also guide development by pointing out the tasks that may be prioritized, which APIs might help, what other drivers may be used as inspiration to improve existing code or to add functionality. Because code quality is a key factor to determine whether code may be included in the Linux kernel, reviewers may worry about many issues. One might check for race conditions, memory leakage,

[3] https://www.analog.com/en/products/ad7292.html

code style conformity, or any other question regarding code quality. Due to this, it is not uncommon for a subsystem maintainer to decline a code contribution. Nevertheless, getting suggestions for improvement is a valuable learning mechanism for newcomers. Throughout the development of the AD7292 driver, the source code was reviewed mainly by Stefan Popa, Alexandru Ardelean, Dragos Bogdan, Jonathan Cameron, and Rob Herring.

Literature review

Books on Linux kernel development are a great source of information but should be read with the assistance of the latest Linux source code. Since Linux is continually changing, literature tends to become outdated as the kernel is updated. Nevertheless, books contain concepts that may not be detailed in the documentation or would otherwise be difficult to grasp from the code. Some remarkable publications are "Linux Device Drivers", from Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman; "Linux Device Drivers Development", from John Madieu; and "Understanding the Linux Kernel", from Daniel P. Bovet, Marco Cesati.

Linux Kernel Documentation Reading

The kernel has extensive documentation available along with the source code, inside the *Documentation* directory, which is also available on the official Linux kernel documentation site at https://www.kernel.org/doc/html/latest/. Documentation is handy as a quick way to search through the internal API since tools like *find* and *grep* allow one to quickly find pages of interest.

Search through websites and blogs

Websites and blogs provide an additional source of information. Within blog posts, it is easier to find content that addresses a specific problem. It is also common to spot pages that explain how a particular kernel part works or how to implement a feature. Some sites and blogs that were helpful through the development of the AD7292 driver are the OS Journey[4], ST wiki[5], Raspberry Pi documentation[6], Linux Journal[7], embedded Linux wiki[8], analog devices wiki[9], FLUSP site[10].

Participate in local developer groups

Sharing experiences may indicate that a problem might be solved by some known implementation. Describing what one wants allows others to help with solving the issue. Expressing yourself is easier when face-to-face with other developers, that is why participating in local developer groups offer a great way of getting assistance. During the

[4]https://oslongjourney.github.io/linux-kernel
[5]https://wiki.st.com/stm32mpu/wiki/Category:IIO
[6]https://www.raspberrypi.org/documentation/
[7]https://www.linuxjournal.com/
[8]https://elinux.org/Main_Page
[9]https://wiki.analog.com/software/linux/docs/iio/iio
[10]https://flusp.ime.usp.br/

development of the AD7292 driver, sharing experiences with the FLUSP (FLOSS at USP) developer group was of significant help in solving development dilemmas.

Analysis of test results

It is good practice to (whenever possible) test every piece of code before submitting it to the kernel community. In the particular case of device drivers, performing hardware test reveals whether the software is working as expected and, otherwise, lets one investigate what is not working as it should be. Throughout the development of the AD7292 driver, numerous tests have done on the AD7292 evaluation board.

1.2 Conventions used in this work

In this work, the following typographic convention will be used:

- *italic* will be used to indicate files, directories, environment variables, computer programs, and options inside computer programs.
- `monospaced` will be used to indicate computer programming instructions or commands that are meant to be executed in a shell.
- **bold** will be used to emphasize important concepts or to highlight possible pitfalls.

1.3 Manuscript Structure

This work has five other chapters. Chapter 2 presents the software development process in the Linux kernel and introduces the Linux Device Model framework for the development of device drivers. Then, Chapter 3 introduces the properties of system build configuration needed to understand the details of a device driver implementation. After that, Chapter 4 provides an overview of device driver elements with examples based on the AD7292 driver implementation. Next, Chapter 5 presents the results obtained from the driver development and some final considerations. Lastly, Chapter 6 shares a personal appreciation of the work.

Chapter 2

Linux kernel development

Developing software for Linux requires not only technical skills but also working together with the community. This chapter introduces fundamental concepts about the software development process employed by the Linux kernel community. Next, the core concepts of the device model framework are covered, and finally, some strategies on how new developers can start contributing to the Linux kernel are discussed.

2.1 Linux development model

The Linux development model works through a chain of trust scheme. No one directly changes the widely distributed kernel code, except for leading maintainers Linus Torvalds and Greg Kroah-Hartman. The Linux kernel development process is organized in development cycles composed by a "merge window" and a "stabilization phase". At the beginning of each cycle, Linus receives contributions from other kernel maintainers in the form of patches[1] or pull requests[2] which contain code deemed to be sufficiently stable. During this period of approximately two weeks, Linus merges the contributions into his mainline kernel repository at a rate approaching 1,000 changes per day. At the end of this time, he closes the merge window and announces the first release candidate for the next kernel version [12].

The next part of the release life cycle is the "stabilization phase" in which Linus accepts bug fix patches. Throughout the next six to ten weeks kernel developers shall try to hunt down bugs and regressions[3] while testing system stability. During this stage, Linus publishes a new release candidate ("-rc") kernel roughly once a week. The bug fixes sent on the first week of stabilization get incorporated in the second release candidate (the first candidate is produced when the merge window is closed), for example, mainline-5.5-rc2.

[1] A patch is a change in the source code of the Linux kernel. Patches may change several code lines but ideally, only one single logical change should be made per patch.

[2] A pull request is an invitation to accept code changes (patches) from a particular codebase. When a pull request is accepted, changes from the codebase kept by the requesting developer are incorporated into the repository managed by some responsible maintainer.

[3] Regression is a change that causes something to break for existing users, i.e., code that produces software unable to support previous existing features.

Subsequent candidates (-rc3, -rc4, ...) are generated with fixes sent on the following weeks until the kernel is considered to be sufficiently tested and free of regressions. At this point, Linus declares the last candidate to be the intended kernel version and the development cycle for the mainline kernel starts over again.

When a new mainline kernel is released, it is taken to be the latest stable kernel which is often shipped within GNU/Linux operating systems such as Debian, Ubuntu, Arch, and Manjaro. Each stable kernel receives bugfixes backported from the mainline on an as-needed basis until the next mainline release is available. At this point, support for the current stable kernel is discontinued and the newest kernel release starts to receive updates. The exception for this rule is the longterm maintenance releases, which continue to receive bugfixes for an extended amount of time. Stable kernels are maintained by the stable team lead by Greg Kroah-Hartman [18, 12].

Since the Linux kernel is a huge project receiving thousands of patches every week, it is impossible for one person to review all the contributions sent. To deal with such a problem, the Linux community has organized the kernel into several repositories (trees), each intended to group contributions for a given set of functionality. These kernel trees are publicly available from the Linux kernel page[4] and many of them are intended to receive contributions for a single kernel subsystem.

2.1.1 Kernel Subsystems

"A subsystem is a representation for a high-level portion of the kernel as a whole [20]." It can also be grasped as an abstraction to refer to some part of the kernel responsible for some specific functionality. When Linux kernel version 2.6 was released, it contained a `struct subsystem` composed by a `kset` and an access semaphore. The usual registration of a `kset` result in the creation of a sysfs directory. Then, at that time, a directory at the top of the sysfs hierarchy would also be considered a subsystem. Buses are also considered to be kernel subsystems as well.

There is one kernel tree for each subsystem. Developers designated as responsible for a subsystem kernel tree are then called subsystem maintainers. A subsystem is also often associated with a mailing list and a developer community.

2.1.2 Communities and mailing lists

The Linux community correspond mainly through the kernel mailing lists at vger.kernel.org. The vger.kernel.org domain was created to provide email list services to Linux kernel developers, allowing them to discuss updates, ask for help, submit patches, request comments, get and offer code reviews [36].

Most of kernel mailing lists are listed in the vger-listspage[5]. The subscription process in each of those lists is managed by a mailing list manager called Majordomo, which only takes (correct) actions when triggered by emails in a specific format [2, 28]. Emails sent to Majordomo **must** be in **TEXT/PLAIN** (i.e., must not contain any HTML tags), should

[4]The Linux Kernel main page: https://www.kernel.org/

[5]http://vger.kernel.org/vger-lists.html

have no multipart sections, nor anything "fancy" [37]. These rules apply to any email sent to Majordomo@vger.kernel.org and are also followed as good practice on most of the kernel mailing lists.

It is important to be aware of the development process and how to interact with the developer community by email. However, for driver developers, it is also relevant to understand the fundamental concepts of the device model present in the Linux kernel. Since a basic comprehension of the device model is required to understand most of this work, the next section briefly explains the core concepts associated with it.

2.2 The Linux Device Model

In Linux versions before 2.6, there was no much information about how the parts of the kernel interacted with each other, nor a standard structure that developers could use for driver programming. The absence of such foundation was partly because older hardware exposed very different interfaces, partially because the kernel was working well [21, 6, 29].

However, the appearance of more sophisticated devices, as well as the standardization of some hardware interfaces, have lead developers to build a common infrastructure to handle hardware components. This infrastructure is known as the Linux Device Model (LDM). The LDM provides common abstractions for device driver development and standardizes the implementation of desirable features such as:

Power management: control procedures to shut down, suspend, or resume device operation.

Event handling: Handle device insertion and removal. Do transparent resource allocation when setting up new devices.

Object lifecycle management: Reference counting and resource management Automatic release of resources when objects are no longer referenced. Widely used in managed resources (devres).

Userspace interface: Communication with userspace through a virtual file system called sysfs. LDM object control implementation is heavily used by sysfs.

Device classes: Device grouping by type or functionality. Applications are often interested in what kind of devices are available rather than how they are connected.

To create a framework capable of supporting all these features while encouraging code reuse, the LDM relies on three major abstractions: buses, devices, and drivers.

"A **bus** is a channel between the processor and one or more devices [22]." Within the LDM, the struct `bus_type` represents a bus. Devices and drivers should be registered on the appropriate bus, which maintains a list for each of these object types. Whenever some registration event occurs (a device or a driver register), the bus traverses the complementary list calling the `match` function to check whether the registered asset can handle some device (if its a driver) or can be managed by a driver (if its a device) [7, 23]. Buses are also

responsible for giving support to other features such as power management functions, direct memory access (DMA) configuration routines, event handling.

Devices are represented by struct `device`, which is the lowest abstraction level for devices in the Linux kernel. A **device** holds data that is used by the LDM to model the system. A device instance usually has a name, a reference counter, a pointer to the bus its sitting on, a pointer to the driver managing its operation, a reference to its father device in device hierarchy, a reference to its associated node within the devicetree, a list of managed resources, the class it belongs to, among other attributes.Because they are relatively simple, subsystems typically "extend" the base device by embedding it within some bus-specific device structure.

"A **driver** is a piece of software whose aim is to control and manage a particular hardware device, hence the name device driver [30]." The struct `device_driver` represents device drivers in the LDM. Each driver instance is registered on a bus and can manage multiple device instances. The device model keeps track of drivers to allow new devices to link against them [24]. Drivers have functions for probe initialization, power management (shutdown, suspend, resume), device removal, and debugging. Like devices, drivers also often have more specific representations in subsystems by struct embedding.

Bus, driver, and device attributes have their values exposed to the user space through sysfs. **sysfs** is a memory-based virtual file system intended to provide a means of exporting kernel structures to the user space. By default, all buses, drivers, and devices are represented in sysfs by a directory with their name. Attributes of these objects are exposed as files within the directory associated with the object that owns them. For example, the IIO subsystem bus has its name attribute exposed by the */sys/bus/iio/name* file.

2.2.1 The Device Model and the IIO subsystem

From the LDM perspective, the IIO subsystem takes on the role of a bus (see details in *drivers/iio/industrial-core.c*). Industrial I/O core extends managed resources functionality to allow self-management of buffers, triggers, and data channels. To provide additional functionality, IIO also defines a more device-specific representation, `iio_dev`. Devices `iio_dev` can define data channels that group information about a particular type of reading that hardware can take. The IIO framework automatically sets device attributes to expose channel data in sysfs. In analogy to the concept of OO encapsulation, IIO also has a mechanism for supporting device private attributes.

2.3 How to contribute to the IIO subsystem

The first step to start contributing to the IIO subsystem is to subscribe to the subsystem mailing list. The kernel mailing lists page (http://vger.kernel.org/vger-lists.html) has some useful links for that. Once registered, the developer can participate in the discussions by learning about standard features of temperature sensors, pressure sensors, digital-analog converters, among other devices that have their drivers implemented on the subsystem.

A recommended initial approach is to choose a driver under the *drivers/staging/iio*

directory and then write an email to the list asking for guidance on which features should be implemented to have the driver moved to the main IIO folder under *drivers/iio*. Often, the suggestions provided by the community helps to learn how the driver works, which is essential to improve it. Depending on the driver picked up, the process of enhancing it might lead to a substantial understanding of the IIO subsystem. This strategy suggests working towards moving the driver from staging to the main directory as a natural goal. Unfortunately, this approach hasn't been much applicable in recent times since the number of drivers in staging has dramatically decreased, and the remaining drivers require more complex modifications.

An alternative strategy is to participate in some incentive programs for new open source developers, such as Outreachy and Google Summer of Code. These are outstanding programs that offer mentorship to guide early contributions as well as a cash grant. Many free software organizations submit project proposals planed to aid introducing developers to their communities. With the help of one or more tutors, students have the opportunity to work on these projects developing software meaningful for the growth of the supporting organization. Throughout this experience, newcomers get tightly involved with the community while learning how to produce their first contributions. The Linux Foundation is the organization that submits the projects related to the Linux kernel.

A third route to start contributing is by joining local developer groups. Experienced participants in these communities readily offer help to newcomers, acting as mentors during their first patches. An additional benefit of engaging those groups is the possibility to have a smoother introduction to the development process by participating in events such as hackathons, lectures, workshops, meetings, etc. Examples of the mentioned groups are the FLUSP[6] and LKCAMP[7] student groups.

The most usual way of contributing to the IIO subsystem is to develop new device drivers or to send incremental patches fixing bugs or enhancing functionality on existing drivers. Other tasks are also welcome, though. The IIO core is not an immutable system and often receives contributions to extend its API and make it more efficient. Proper documentation is also a key asset in projects such as the Linux kernel. Improving the documentation has the potential to smooth the initiation of newcomers, presenting good practices, explaining the development model and flow of contributions, providing examples on how to use existent API. Also, reviewing patches from others is a great doing to the community. The more people looking at code, the higher are the chances someone finds out a way of improving it. Code testing is another good practice that can unravel bugs that may be promptly tackled by the community in sequence. All of these leads to improved software quality, pushing Linux to the status of one of the most stable kernels in the world.

The following chapters deal with strictly technical aspects related to the kernel building process and the operation of an IIO driver. Readers who are not interested in these technical details are advised to go directly to Chapter 5.

[6] https://flusp.ime.usp.br/
[7] https://lkcamp.gitlab.io/

Chapter 3

System Build

This chapter presents an introduction to both the Linux kernel build system and devicetree infrastructure. The former is responsible for generating the appropriate kernel modules and images according to stored configuration and compilation flags. The latter provides a way of telling the kernel what the hardware layout is. The basic knowledge of these is necessary to understand further changes a driver implementation does besides its core file.

3.1 The Kernel Build System (kbuild)

The Kernel Build System (kbuild) is a framework based on make, flex, and other GNU tools that allows a highly modular and customizable compilation process for the Linux kernel. Kconfig and Kbuild Makefiles implement most of the kbuild features. The framework grants flexibility by conditional compilation based on configuration options.

3.1.1 Configuration Options

By default, kbuild Makefiles use the configuration options stored in the *.config* file under the kernel root directory. These options hold values for the configuration symbols associated with kernel resources (drivers, tools, features, etc.). Changes in the configuration options reflect on what kbuild generates. Moreover, the configuration options are orderly sensitive. A disabled option may limit the visibility of dependent entries. Thus, directly editing the *.config* file requires caution. Nevertheless, to easy testing with configuration files, an alternative *.config* may be set. Export the path to the *KCONFIG_CONFIG* variable to use a custom configuration source. For instance:

```
1   export KCONFIG_CONFIG=.my_config
```

Alternatively, we may set the *.config* file just when invoking make.

```
1   make KCONFIG_CONFIG=.my_config
```

This option is not much used, though. Many of the thousand values hold by configuration files are common to multiple applications. A cleaner way of storing custom

values for configuration symbols is to use `defconfig`. (We will look at defconfig in Section 3.1.3).

3.1.2 Configuration Symbols

The *Kconfig* files define the configuration symbols associated with the kernel resources. As a general rule, a *Kconfig* file should only declare symbols for resources under the same directory. Nearly all directories inside the kernel source tree have a *Kconfig* file. Top *Kconfig* files include (source) *Kconfig* files from subdirectories thus, creating a tree of configuration symbols. To define a configuration symbol for the AD7292 driver, the following entry was added to the *Kconfig* file at *drivers/iio/adc/*.

```
config AD7292
tristate "Analog Devices AD7292 ADC driver"
depends on SPI
help
  Say yes here to build support for Analog Devices AD7292
  8 Channel ADC with temperature sensor.

  To compile this driver as a module, choose M here: the
  module will be called ad7292.
```

The *config* keyword defines a new configuration symbol, which in turn is presented as a configuration entry in the *.config* file, within tools like *menuconfig*, *nconfig*, or during the compilation process. In particular, the AD7292 configuration symbol has the following attributes:

tristate: the type for the configuration option. It declares that this symbol stands for something that may be compiled as a module (m), built-in compiled (y) (i.e., included in the kernel image), or not compiled at all (n). The type definition also accepts an optional input prompt to set the option name that kernel configuration tools display.

depends on: list of dependency symbols. If its dependencies are not satisfied, this symbol may become non-visible during configuration or compilation time. As an experiment, try to disable SPI[1] support at Device Drivers. The AD7292 will no longer be listed at *Device Drivers → Industrial I/O support → Analog to digital converters*.

help: defines a help text to be displayed as auxiliary info.

Additionally, if a configuration option has no value, the default value is used. If no default is available, the user will be prompted to assign it a value. Keep in mind that a **configuration option** stores the value assigned to a **configuration symbol**. Configuration options have the form *CONFIG_<symbol>*. For instance, *CONFIG_AD7292* stores the value for the AD7292 configuration symbol.

The kernel source code comes with no *.config* file, so one has to be created. Though the compilation process can automatically generate configuration files, it will ask for many

[1]The Serial Peripheral Interface (SPI) is a communication protocol commonly used to exchange data between computers and small peripherals.

configuration values. If some incompatible values are assigned, the resulting image might not work on the desired machine. Programs such as *menuconfig* and *nconfig* present the options available in a menu like interface. The user may use them to find out information about each option as well as to assign values to them. The program then generates a configuration file with the desired values. Non assigned symbols get default values or are prompted for in the compilation process. Alternatively, one can use common values for the platform of interest. These platform-specific values are stored in *defconfig* files.

3.1.3 Defconfig

The purpose of *defconfig* files is to store only specific non-default values for compilation symbols. For instance, one can find *defconfig* files for the ARM architecture under *arch/arm/configs/*. The files *bcm2709_defconfig*, *bcm2835_defconfig*, and *bcmrpi_defconfig* store the configuration values commonly used for Raspberry Pi boards. To add a custom configuration value for the AD7292 symbol, add the following line to a *defconfig* file.

```
CONFIG_AD7292=y
```

The configuration stored at a *defconfig* file may be applied to *.config* using its name as a make target. For instance, to load configuration options from *bcm2709_defconfig*, one may invoke:

```
1   make bcm2709_defconfig
```

Use the `savedefconfig` target to create a *defconfig* file from *.config*.

```
1   make savedefconfig
```

So far, we have seen how to define a configuration symbol and assign it a value. However, we still need to understand how to make kbuild compile a driver source file according to a configuration option. Some knowledge of kbuild makefiles will help us to do that.

3.1.4 Kbuild Makefiles

The main goal of the kbuild Makefiles is to produce the vmlinux (kernel image) and modules. It builds them by recursively descending into subdirectories of the kernel source tree [9]. Akin to *Kconfig* files, kbuild Makefiles are also present in most kernel directories, often working with the values assigned for the symbols defined by the former. According to Javier Canillas:

> The whole build is done recursively — a top Makefile descends into its subdirectories and executes each subdirectory's Makefile to generate the binary objects for the files in that directory. Then, these objects are used to generate the modules and the Linux kernel image [1, 14].

Makefiles in subdirectories should only modify files in their own directory. Thus, we include driver object files in the list of kbuild compilation goals inside the nearby makefile. For instance, the AD7292 driver has its entry inside *drivers/iio/adc/Makefile*:

```
1   obj-$(CONFIG_AD7292) += ad7292.o
```

To summarize the procedure of adding a feature to the Linux kernel, Canillas points out three main steps:

> - Put the source file(s) in a place that makes sense, such as drivers/net/wireless for Wi-Fi devices or fs for a new filesystem.
>
> - Update the Kconfig for the subdirectory (or subdirectories) where you put the files with config symbols that allow you to choose to include the feature.
>
> - Update the Makefile for the subdirectory where you put the files, so the build system can compile your code conditionally [1, 14].

Symbols, options, and makefiles set up, we may now compile the kernel to test our changes.

3.2 Kernel Compilation

To compile the Linux kernel, call GNU make at the root of the source dircetory.

```
1   make
```

The default target (_all) will build the bare kernel image (vmlinux), all of the modules, and other architecture-specific artifacts. By default, the top Makefile sets `$(ARCH)` to be the same as the host system architecture [15]. However, this may not reflect the architecture of the machine one want to run the kernel. Since different computer architectures have distinct instruction sets, code compiled in one computer may not work on another of incompatible design. Nevertheless, it is often desired to use a host machine to generate binaries compatible with a target machine of different architecture. Doing so is called cross-compilation.

3.2.1 Kernel Cross Compilation

Cross-compilation is assisted by kbuild mainly through the `ARCH` and `CROSS_COMPILE` variables. `ARCH` set the target architecture, which is often the same name as the architecture directory under the *arch* directory [13]. `CROSS_COMPILE` specify part of the cross compiler filename or the path to it. Both variables can be set in the shell environment or during the invocation of make. To compile the Linux kernel for ARM do:

```
1   ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make zImage modules dtbs -j 4
```

The compilation process might take some time (around half an hour on an x86 quad-core machine). The number of CPU cores used for the compilation is set by the `-j` flag. The `dtbs` target makes kbuild compile the devicetree files at *arch/arm/boot/dts* into devicetree blobs that the boot process will pass to kernel. With the `modules` target, kbuild will build code marked as module (m). The `zImage` refers to ARM specific image format. Different from the default *vmlinux* image, the *zImage* is a compressed kernel image. If everything goes well, the last output lines should look like:

```
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
```

```
Kernel: arch/arm/boot/zImage is ready
```

The compilation process will create (or overwrite) the following files:

- *arch/arm/boot/zImage*
- *modules.order*
- *modules.builtin*
- *modules.builtin.modinfo*
- *arch/arm/boot/dts/*.dtb*
- *arch/arm/boot/dts/overlays/*.dtbo*

Many devices integrate systems that run on ARM architecture computers. Though, hardware discovery on the ARM architecture is not as straight forward as in the x86 architecture. In most x86 machines, the system kernel can discover what equipment is available by looking at device tables provided by ACPI or UEFI compliant firmware [8]. Most ARM-based devices are not like that, though.There is no implementation of the ACPI standard nor any means of discovering which hardware is attached to the system [4, 31]. Moreover, ACPI raises many problems regarding performance and security [5]. Due to this, the Linux kernel development has been adopting an alternative way to recognize hardware layout, the Devicetree.

3.3 Devicetree

The **Device Tree (DT)** is a specification on how to describe the hardware on a given system. It is concise, yet very expressive. A devicetree source (DTS) file is a data structure made out of nodes that hold information about each hardware component in a single board computer (SBC) or system on a chip (SoC). Nodes may have property definitions and child node definitions. Properties can hold 32-bit integer cells, strings, hexadecimal bytestrings, references to other nodes, or can be left empty. To simplify node referencing and make the structure more readable, developers may use labels to create aliases to nodes. DTS may also include definitions from devicetree include files (DTSI), which in turn may incorporate definitions from other DTSI as well [27].

The Device Tree Compiler (DTC) turns DTS into Device Tree Binary (DTB) files. A DTB is a flattened binary blob that encodes devicetree data within a compact pointerless structure. A DTB file is passed to the system kernel by the bootloader or wrapped up with the kernel image to support booting on legacy non-DT aware firmware [27, 26, 35]. Early in the boot, the kernel parses the DTB to identify the machine and execute any specific platform setup. At some later point in the kernel initialization, the list of device nodes is obtained from the DTB and used to populate the Linux Device Model with data about the platform [26].

A **device node** is some devicetree node that describes a sensor device. Device nodes have a `compatible` property that holds one or more strings specifying the device model compatibility, from most specific to most general. The order is important to allow devices to

indicate their compatibility with families of device drivers. The recommend format is *"manufacturer,model"*, where *manufacturer* stands for the manufacturer name (or codename), and *model* stands for the device model name or number. For instance:

```
compatible = "samsung,exynos3250-adc", "samsung,exynos-adc-v2";
```

For a device node with such a *compatible* list, the first match attempt will be against a device driver compatible with *samsung,exynos3250-adc*. If no such a driver is found, then there will be a match try against a driver compatible with *samsung,exynos-adc-v2*.

Device node documentation followed a format derived from the Open Firmware (OF) standard that was used mostly in PowerPC and SPARC platforms.These plain text docs were not very restrictive about the structure of device nodes so, it was easy to make mistakes when writing them. Developers are trying to avoid further misconceptions by working in a set of validation tools known as dt-schema.

Devicetree schema files (also known as **DT bindings**) describe how should be the format of data in a DTS. Devicetree schema is written in YAML format and validated by dt-schema to restrict the schema structure to a subset of the DT specification proper for describing device nodes. The purpose of writing devicetree schemas is to provide a way to check whether a device node inside a DTS file is correct and to provide documentation about device binding. A DTS may contain nodes with different properties; therefore, many schema files may be used to validate a single DTS [19]. The most common fields of devicetree schema are:

$id: an UID (unique identifier) for the dt-binding.

$schema: the meta-schema that will be used to validate this binding.

title: documentation for the documentation.

maintainers: enum of maintainers.

description: mandatory property specifying what device binding is being documented and where to find the device datasheet.

properties: list of custom properties for the device node. May include several subproperties describing what a device node may contain and what values each subproperty is expected to have.

required: enum of properties that must be present in a device node that describes such a device.

examples: enum of examples showing how such a device node would appear in a DTS file. It is good practice to provide examples showing the usage of all the documented properties.

A simplified devicetree schema for AD7292 would look like Figure 3.1.

The AD7292 schema uses additional properties to describe an AD7292 device. Let's see what some of them mean.

compatible: string to match with supporting device drivers.

```
$id: http://devicetree.org/schemas/iio/adc/adi,ad7292.yaml#
$schema: http://devicetree.org/meta-schemas/core.yaml#

title: Analog Devices AD7292 10-Bit Monitor and Control System

maintainers:
  - Marcelo Schmitt <marcelo.schmitt1@gmail.com>

description: |
  Analog Devices AD7292 10-Bit Monitor and Control System with ADC, DACs,
  Temperature Sensor, and GPIOs

  Specifications about the part can be found at:
    https://www.analog.com/media/en/technical-documentation/data-sheets/ad7292.pdf

properties:
  compatible:
    enum:
      - adi,ad7292

  reg:
    maxItems: 1

  vref-supply:
    description: |
      The regulator supply for ADC and DAC reference voltage.

  spi-cpha: true

required:
  - compatible
  - reg
  - spi-cpha

examples:
  - |
    spi {
      #address-cells = <1>;
      #size-cells = <0>;

      ad7292: adc@0 {
        compatible = "adi,ad7292";
        reg = <0>;
        spi-max-frequency = <25000000>;
        vref-supply = <&adc_vref>;
        spi-cpha;
      };
    };
```

Figure 3.1: *A simplified devicetree schema for AD7292.*

reg: the location of the device resources within the parent node address space. The specification for *reg* properties was designed to allow the description of (*address*, *length*) pair values that point out the addresses, and how many bytes from each address, are part of the specified device resources. For SPI buses, it is only needed to specify the chip select (CP) lane on which the device is connected. This can be done with a single 32-bit unsigned integer thus, *#address-cells* is set to 1. Moreover, the SPI protocol does not specify any means of a slave device exposing more than its address to the master device. No memory range can be exposed directly on the bus. Thus, *#size-cells* for SPI buses should always be zero.

spi-max-frequency: maximum SPI operating frequency.

vref-supply: AD7292 may be supplied with an external voltage reference. When present, this property value points to the external reference. Otherwise, an internal voltage reference is used.

spi-cpha: whether the chip requires shifted clock phase.

#address-cells: how many 32-bit cells are needed to express the children node addresses. In the above example, the *#address-cells* property at the spi node specifies that the *reg* field of its children (such as the ad7292 node) will have only one 32-bit value for addressing.

#size-cells: how many 32-bit cells are needed to express the children node address sizes. In the above example, the *#size-cells* property at the spi node specifies that the *reg* field of its children nodes will not indicate any address range.

To insert an Analog Devices AD7292 control system into a devicetree, add it as a child node of an SPI bus. A hypothetical simplified DTS that includes an AD7292 device would look like Figure 3.2.

```
1     / {
2         compatible = "hlusp,tommy", "poli,caninos";
3
4         cpus: cpus {
5             <cpu nodes>
6         }
7
8         memory@0 {
9             <memory properties>
10        };
11
12        spi: spi@7e204000 {
13            compatible = "poli,caninos-spi";
14            reg = <0x7e204000 0x200>;
15            #address-cells = <1>;
16            #size-cells = <0>;
17
18            ad7292: adc@0 {
19                compatible = "adi,ad7292";
20                reg = <0>;
21                spi-max-frequency = <25000000>;
22                vref-supply = <&adc_vref>;
23                spi-cpha;
24            };
25        };
26    };
```

Figure 3.2: *An example of how an AD7292 device node would be set into a DTS file.*

Now that we understand the basics of how to provide drivers and devices to the system, we may focus on understanding how a device driver implementation works on Linux.

Chapter 4

Device Driver Implementation

The purpose of this chapter is to provide an overview of how a device driver works by studying the key elements from the AD7292 driver. Every subsection starts by first exploring generic driver aspects and then providing examples based on the AD7292 driver implementation. Each driver element will be addressed in the order in which it appears in the source code and not in the order in which the system would access it. This way, it is expected to be easier for readers to understand the role of each component in the operation of a Linux device driver.

4.1 AD7292 device driver

The first version of the device driver for the AD7292 monitor and control system provides support for single ADC readings. Through synchronous SPI messages, the Linux kernel can communicate with AD7292 devices to issue analog measurements, retrieve, and fathom collected data. Because the source code of a typical device driver in the IIO subsystem can be broken down into snippets of distinct responsibility, the remaining subsections of this chapter reflect the burden of each one in the order they occur in the AD7292 driver. Thus, the subjects addressed by this chapter are the SPDX license identifier, register definitions, bit manipulation macros, IIO channels, device private data, SPI messaging, read_raw operations, driver static information, device probing, managed device resources, properties from devicetree nodes, driver compatibility.

4.1.1 SPDX License Identifiers

The Linux kernel is distributed under the GNU GPL-2.0 license, with a single exception on the syscall interface that lies between the kernel and user-space programs. Device driver source code should be compatible with the permission used by Linux, i.e., it should comply with the GPL-2.0 or a combination of GPL-2.0 and other permissive licenses such as MIT or BSD [3]

The common way to specify the license of a software is to add the license text in the top of each source code file. This, however, makes license tracking over kernel files difficult since developers may add slightly different excerpts for the same license, within distinct

comment style and formatting. To simplify the specification of kernel source files, the Software Package Data Exchange (SPDX) license identifiers are being used. SPDX provides an alternative for license tagging that is easier to be machine parsed and thus, helpful for the license tracking of the many Linux kernel source files [3].

The ad7292 driver starts with the GPL-2.0 license identifier to indicate it is distributed as free software.

```
1    // SPDX-License-Identifier: GPL-2.0
```

4.1.2 Register definitions

SPI and I2C devices have registers responsible for storing internal state, alert signals, measurement results, operating mode configuration, etc. Though it is possible to address registers with decimal integers, hexadecimal is preferred since most datasheets list register addresses with hexadecimal numbers. Also, to improve code readability, register addresses are often aliased by names related to their functionality. It is not different in the AD7292 driver (see Figure 4.1).

```
1     /* AD7292 registers definition */
2     #define AD7292_REG_VENDOR_ID            0x00
3     #define AD7292_REG_CONF_BANK            0x05
4     #define AD7292_REG_CONV_COMM            0x0E
5     #define AD7292_REG_ADC_CH(x)            (0x10 + (x))
6
7     /* AD7292 configuration bank subregisters definition */
8     #define AD7292_BANK_REG_VIN_RNG0        0x10
9     #define AD7292_BANK_REG_VIN_RNG1        0x11
10    #define AD7292_BANK_REG_SAMP_MODE       0x12
```

Figure 4.1: *Definitions for some AD7292 internal registers.*

The usual formatting for register aliasing is:

`<device model>_REG_<functionality>`

Starting aliases with <device model> shall ensure they do not conflict with any other alias within the kernel. `REG` makes clear that the alias is about a device register. Lastly, a <functionality> mnemonic helps to rapidly figure out what the register is about.

The first alias stands for the register address that holds the vendor ID number. The second alias point to the configuration register bank address. The configuration register bank is the base address for many subregisters that hold operating configuration, such as the VIN range and sampling mode subregisters. The VIN range and sampling mode subregisters will be described in later sections. Next, there is an alias for the conversion command register address. To start ADC conversions, a conversion command must be written to that address. Finally, an alias definition for the ADC conversion result registers. These hold the result of ADC conversions on each of the AD7292 eight channels.

The AD7292 has other registers than the ones listed above. However, for the functionality implemented so far, these registers are enough.

4.1.3 Bit manipulation macros

Bits contained in the same register may have different meanings. For example, in a 16-bit register containing bits D_0 through D_{15}, the bits D_0 and D_1 may indicate alerts, the bits D_2, D_3, D_4 and D_5 may indicate the ID of a read channel, while the bits D_6 to D_{15} may contain the result of an analog read. This is exactly the case of AD7292 ADC conversion result registers.

To help dealing with such bit ranges, the kernel defines some useful macros:

BIT(nr): generates an unsigned long with only one bit set at index `nr`.

GENMASK(h, l): generates a contiguous bitmask setting the bits `l` through `h` (inclusive).

FIELD_GET(mask, reg): extracts the field specified by `mask` from the bitfield passed in as `reg` by masking and shifting it down.

A **bitmask** is a bitstring that describes which bits are of interest within another numeric value. Bitmasks are often set as a fixed value which has the bits at the desired indexes set to 1. For instance, to get only the 6 rightmost bits of a value, one would define the following bitmask:

```
1   #define SOME_BITMASK        0x3F
```

The above bitmask would then be applied to some variable using the bitwise and operator, like:

```
1   read_bits = x & SOME_BITMASK;
```

Bitmasks can be applied one over another allowing very specific data formatting. If a developer would like to also have bit 7 set, then he/she would use the `BIT` macro as bitmask as well. The following diagram shows the bits set into each of `x`, `0x3F`, `BIT(7)` and the result of applying the bitmasks.

Bit index		7	6	5	4	3	2	1	0
	x	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
&	0x3F	0	0	1	1	1	1	1	1
|	BIT(7)	1	0	0	0	0	0	0	0
BIT(7) | (x & 0x3F)		1	0	x_5	x_4	x_3	x_2	x_1	x_0

Thus, the previous macro formats a data to contain the bit at index 7 set, the bit at index 6 unset, and the bits at indexes 0 to 5 equal to the bits of some input `x` address.

Back to the ADC conversion result registers example, the `GENMASK` and `FIELD_GET` macros may be used to extract the ADC conversion result bits from those registers. The bits of interest are D_6 through D_{15} so, a `GENMASK(6, 15)` would express those bits. However, just applying the bitmask would leave the value with trailing zeros which doesn't came out from ADC output. To shift the returned value so that the ADC result bits becomes the right most ones, the `FIELD_GET` macro might be used. Actually, `FIELD_GET` does both things. It applies the bitmask defined by its mask argument, and does the bit shifting. Hence, to obtain the bits from the result of an ADC reading, one should apply `FIELD_GET(GENMASK(6,`

15), x), where x stands for the value obtained from the conversion result register of a channel.

Figure 4.2 shows macros defined to deal with AD7292 chip-specific registers and their layout.

```
1   #define AD7292_RD_FLAG_MSK(x)          (BIT(7) | ((x) & 0x3F))
2
3   /* AD7292_REG_ADC_CONVERSION */
4   #define AD7292_ADC_DATA_MASK           GENMASK(15, 6)
5   #define AD7292_ADC_DATA(x)             FIELD_GET(AD7292_ADC_DATA_MASK, x)
6
7   /* AD7292_CHANNEL_SAMPLING_MODE */
8   #define AD7292_CH_SAMP_MODE(reg, ch)   (((reg) >> 8) & BIT(ch))
9
10  /* AD7292_CHANNEL_VIN_RANGE */
11  #define AD7292_CH_VIN_RANGE(reg, ch)   ((reg) & BIT(ch))
```

Figure 4.2: *AD7292 bit manipulation macros.*

AD7292_RD_FLAG_MSK and AD7292_ADC_DATA work exactly as described above. The former is applied to format register addresses into bitstrings for reading their values, the latter is applied to get only the ADC conversion result bits from an ADC conversion result register. AD7292_CH_SAMP_MODE and AD7292_CH_VIN_RANGE, respectively, help to obtain the sampling mode and the voltage range of input channels from subregisters under the configuration register bank.

4.1.4 IIO channels

An IIO channel groups information about some sort of data a device can provide or can be provided to. Not only the raw bits of data, but also important metadata that is meaningful to application is grouped by IIO channels. It may include scale factor, offset, data direction (input/output), type of physical quantity being measured, etc. The specification of an IIO channel is defined by an iio_chan_spec struct. Among the many properties a channel can have, the most important ones for the AD7292 operation are:

type: physical quantity being measured (voltage, current, temperature, accelerance, etc.)

indexed: some devices may have multiple channels of the same type in which case they need to be indexed to avoid mistakes. This property indicates whether a channel is indexed.

channel: if the channel is indexed, this defines what index it has.

info_mask_separate: indicates which information should be unique to the channel. For each type of information a device attribute will be exported for exclusive use by the defining channel.

info_mask_shared_by_type: indicates which information the channel will share with other channels of the same type. One single attribute will be exported to serve all of the channels of the same type that share the indicated information.

info_mask_shared_by_dir: indicates which information the channel will share with other channels of the same direction. One single attribute will be exported to serve all the channels of the same direction that share the indicated information.

info_mask_shared_by_all: indicates which information the channel will share with any other channel. One single attribute will be exported to serve all the channels that share the indicated information.

differential: whether the channel is for differential analog read.

channel2: if differential is set then this is the second channel to operate as differential pair. If a modifier is applied to the channel then this value defines which modifier.

output: the direction of the channel. 0 for input, 1 for output.

The analog input channels of AD7292 devices map to the IIO channels defined in the device driver as can be seen in Figure 4.3.

```
1   #define AD7292_VOLTAGE_CHAN(_chan)                                  \
2   {                                                                   \
3           .type = IIO_VOLTAGE,                                        \
4           .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) |              \
5                                 BIT(IIO_CHAN_INFO_SCALE),             \
6           .indexed = 1,                                               \
7           .channel = _chan,                                           \
8   }
9
10  static const struct iio_chan_spec ad7292_channels[] = {
11          AD7292_VOLTAGE_CHAN(0),
12          AD7292_VOLTAGE_CHAN(1),
13          AD7292_VOLTAGE_CHAN(2),
14          AD7292_VOLTAGE_CHAN(3),
15          AD7292_VOLTAGE_CHAN(4),
16          AD7292_VOLTAGE_CHAN(5),
17          AD7292_VOLTAGE_CHAN(6),
18          AD7292_VOLTAGE_CHAN(7)
19  };
20
21  static const struct iio_chan_spec ad7292_channels_diff[] = {
22          {
23                  .type = IIO_VOLTAGE,
24                  .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
25                  .indexed = 1,
26                  .differential = 1,
27                  .channel = 0,
28                  .channel2 = 1,
29          },
30          AD7292_VOLTAGE_CHAN(2),
31          AD7292_VOLTAGE_CHAN(3),
32          AD7292_VOLTAGE_CHAN(4),
33          AD7292_VOLTAGE_CHAN(5),
34          AD7292_VOLTAGE_CHAN(6),
35          AD7292_VOLTAGE_CHAN(7)
36  };
```

Figure 4.3: *AD7292 analog input channels.*

AD7292 devices have 8 analog input channels (VIN0 to VIN7) multiplexed to a single digital-analog converter. Each channel can provide different measurements, and therefore, the raw bits of the digital-analog conversion result should be made available separately for

each channel. Also, each analog channel can be configured with different input ranges. The input range directly influences the scaling factor that must be applied to convert the bits from the ADC into real values (voltage measurement). Therefore, scale factor information must also be made available separately for each AD7292 channel. Channel definitions are stored in the vector `ad7292_channels`.

Optionally, the first two analog input channels (VIN0 and VIN1) can operate in differential mode, allowing one to attenuate noise in ADC readings. This way, the first two entries characterize a single IIO channel. An exclusive vector (`ad7292_channels_diff`) stores channel definitions when VIN0 and VIN1 are configured to operate in differential mode.

Devices created by the Linux Device Model expose a userspace interface through files and directories in the sysfs file system. On older interfaces, it was common for records to contain various information stored in different formats. Userspace programs were then required to parse those files to get device information. However, this was discouraging for interface maintenance because when kernel changes were needed, there was a high risk of breaking applications in the user space if the parsing algorithms had to be changed.

To develop smoother integration between kernel and userspace, new drivers tend to store only a few data in sysfs files (a number, a pair of integers, a string, a boolean, etc.). Thus, if the kernel changes internally and no longer provides some data, the file that would contain such value will no longer be created, and userspace programs may realize so by checking the existence of such a record. Consequently, a change in the interface is less likely to break applications.

An IIO device of index *X*, exposes its attributes in several files under */sys/bus/iio/iio:deviceX*. For channel-related data, file names are constructed following a pattern defined in the subsystem API at *drivers/iio/drivers/iio/industrial-core.c*.

Files with data provided by the AD7292 analog channels are named after the format *<output>_<type><channel>_<mask>* where *<output>* is the channel direction, *<type>* is the type, *<channel>* is the index, and *<mask>* is the type of information the file holds. Differential read channels are exposed by files whose name follows the format *<output>_<type<channel>-<type><channel2>_<mask>*. A typical AD7292 device contains the following files and directories under its sysfs directory:

```
dev                      in_voltage3_scale  in_voltage6_raw    of_node
in_voltage0-voltage1_raw in_voltage4_raw    in_voltage6_scale  power
in_voltage2_raw          in_voltage4_scale  in_voltage7_raw    subsystem
in_voltage2_scale        in_voltage5_raw    in_voltage7_scale  uevent
in_voltage3_raw          in_voltage5_scale  name
```

In general, IIO channel names reflect the characteristics that define them. More examples of channels can be found in the IIO subsystem elements documentation [10].

4.1.5 Device private data

It is common for a device driver to define custom data structures for the operation of supported devices. Often, a “state” struct group (encapsulate) the device private data. With

macros and functions exported by the LDM together with the IIO API, one can retrieve the generic devices associated with a state struct instance to gain access to more generic functionality. Thus, looking from an object-oriented programming (OOP) point of view, the relationship between the iio_dev (superclass) and the state (subclass) structs is much like an inheritance. Thanks to it, some of the benefits from OOP may be enjoyed by Linux kernel developers.

The struct `ad7292_state` in Figure 4.4, defines the attributes that are private for each AD7292 device.

```
1    struct ad7292_state {
2            struct spi_device *spi;
3            struct regulator *reg;
4            unsigned short vref_mv;
5
6            __be16 d16 ____cacheline_aligned;
7            u8 d8[2];
8    };
```

Figure 4.4: *State struct that holds AD7292 device private data.*

spi : parent SPI device.

reg : voltage regulator device. Used in case the AD7292 device has been configured to use an external voltage reference.

vref_mv : voltage reference im milivolts.

d16 : buffer used for reading device registers.

d8 : buffer used to write in device registers.

4.1.6 SPI messaging

The kernel SPI subsystem provides two useful structs for organizing data transmission, `spi_message` and `spi_transfer`. A `spi_transfer` represents a simple transfer of data from the computer to the device or vice versa. This struct specifies a data read and write buffers, the number of bytes to be read and written, the delay between transfers, among other transmission options. A `spi_message` represents a transaction formed by a sequence of `spi_transfer`. It groups multiple SPI transfers, ensuring that each transfer occurs in the order specified by its transfer vector.

To perform an analog read on AD7292 devices, you must first write to the conversion command register, signaled to the device that initiates a digital-analog conversion process. The result of the conversion is stored in one of the ADC conversion result registers. Hence, two SPI transfers must be done sequentially. The first writes the bits to order the start of an ADC conversion, which results will be fetched by a second one that reads from one of the result registers. An SPI message encapsulates both transfers in the correct order so the SPI subsystem can do the transmission as desired.

Figure 4.5 shows how an `spi_transfer` is filled in the AD7292 driver. The `spi_sync_transfer` function assigns the `t` vector to an `spi_message` and then dispatches it

to the device.

```
1     static int ad7292_single_conversion(struct ad7292_state *st,
2                                         unsigned int chan_addr)
3     {
4             int ret;
5
6             struct spi_transfer t[] = {
7                     {
8                             .tx_buf = &st->d8,
9                             .len = 4,
10                            .delay_usecs = 6,
11                    }, {
12                            .rx_buf = &st->d16,
13                            .len = 2,
14                    },
15            };
16
17            st->d8[0] = chan_addr;
18            st->d8[1] = AD7292_RD_FLAG_MSK(AD7292_REG_CONV_COMM);
19
20            ret = spi_sync_transfer(st->spi, t, ARRAY_SIZE(t));
21
22            if (ret < 0)
23                    return ret;
24
25            return be16_to_cpu(st->d16);
26    }
```

Figure 4.5: *Function developed to take analog readings form AD7292 devices.*

4.1.7 read_raw operations

The IIO system allows the definition of call back functions for reading and writing on the device. The *read_raw* is responsible for requesting values from the device while the *write_raw* is accounted for pushing data to it. Both operate based on the channel definitions assigned to the device channel list. They take as argument a reference to the device, a reference to the specification of the channel, the type of information to access, and then call underlying routines to provide requested data or perform desired operations.

So far, the current implementation of the AD7292 `read_raw` function deals only with two types of information: the simple analog conversion reading (`IIO_CHAN_INFO_RAW`), and the scale associated with a channel (`IIO_CHAN_INFO_SCALE`).

When an analog read is requested, the first thing to do is to call the `AD7292_REG_ADC_CH` macro to return the conversion result register address that is going to be read. Next, the `ad7292_single_conversion` function takes care of issuing an ADC conversion command and returning the bits stored in the result register. The SPI communication may fail, so the return code is checked. After that, `AD7292_ADC_DATA` extracts only the bits that represent the result of the analog conversion, which are then assigned to `*val`. Lastly, `ad7292_read_raw` returns `IIO_VAL_INT` to indicate to the IIO subsystem to print the result value (`*val`) in the sysfs file whose reading resulted in a call to this function (see Figure 4.6).

When a scale value is requested, `ad7292_vin_range_multiplier` reads from both ADC sampling mode and VIN range subregisters to calculate the correct range multiplier. Each

AD7292 ADC channel may have its input range adjusted according to the settings at the ADC sampling mode and VIN range subregisters. For any channel, the input range is equal to the voltage reference multiplied by a factor of 1, 2 or 4, according to the following rule:

- If the channel is being sampled with respect to AGND:
  - factor = 4 if VIN range0 and VIN range1 equal 0
  - factor = 2 if only one of VIN ranges equal 1
  - factor = 1 if both VIN range0 and VIN range1 equal 1
- If channel is being sampled with respect to AVDD:
  - factor = 4 if VIN range0 and VIN range1 equal 0
  - Behavior is undefined if any of VIN range doesn't equal 0

To convert a raw value to standard units, the IIO defines this formula: Scaled value = (raw + offset) * scale. For the scale to be a correct multiplier for (raw + offset), it must be calculated as the input range divided by the number of possible distinct input values. Given the ADC data is 10 bit long, it may assume 2^{10} distinct values. Hence, scale = range / 2^{10}. The `IIO_VAL_FRACTIONAL_LOG2` return type indicates to the IIO subsystem to divide `*val` by 2 to the power of `*val2` when returning from `read_raw`.

```
1   static int ad7292_read_raw(struct iio_dev *indio_dev,
2                              const struct iio_chan_spec *chan,
3                              int *val, int *val2, long info)
4   {
5           struct ad7292_state *st = iio_priv(indio_dev);
6           unsigned int ch_addr;
7           int ret;
8
9           switch (info) {
10          case IIO_CHAN_INFO_RAW: /* Read raw ADC data */
11                  ch_addr = AD7292_REG_ADC_CH(chan->channel);
12                  ret = ad7292_single_conversion(st, ch_addr);
13                  if (ret < 0)
14                          return ret;
15
16                  *val = AD7292_ADC_DATA(ret);
17
18                  return IIO_VAL_INT;
19          case IIO_CHAN_INFO_SCALE: /* Calculate ADC scale factor */
20                  ret = ad7292_vin_range_multiplier(st, chan->channel);
21                  if (ret < 0)
22                          return ret;
23
24                  *val = st->vref_mv * ret;
25                  *val2 = 10;
26                  return IIO_VAL_FRACTIONAL_LOG2;
27          default:
28                  break;
29          }
30          return -EINVAL;
31  }
```

Figure 4.6: *Function that handles read_raw operation for AD7292 devices.*

4.1.8 Driver static information

The struct `iio_info` defines which static device information should be considered by the IIO subsystem when registering the device and answering requests from the sysfs interface. General-purpose attributes, pointers to IIO attribute-linked functions, buffers, and triggers are among the objects stored by `iio_info`.

The current version of the AD7292 driver specifies only its `read_raw` function as a callback function for reading files tied to its channels (see Figure 4.7).

```
1    static const struct iio_info ad7292_info = {
2            .read_raw = ad7292_read_raw,
3    };
```

Figure 4.7: *Struct that holds static information about the AD7292 device driver.*

4.1.9 Device probing

When the system kernel discovers a device (e.g., a plug and play device gets attached), bus specific subroutines traverse the list of known drivers seeking for some that can handle the new device. If a driver states it can operate the device, the bus routine calls the probe function registered for the selected driver. The probe function then does all the initialization needed to handle the device properly [25]. It allocates memory for private driver structures, initializes essential device attributes, gets ancillary devices, sets up device configuration, does anything else that is needed before the device is made available to userspace, and registers the device with the appropriate subsystem.

For instance, the probe function registered by the AD7292 driver performs the following tasks:

- requests to the IIO subsystem a new `iio_dev` device.
- get a pointer to the memory address that holds private device data.
- stores a pointer to its SPI parent device.
- puts a pointer to the initializing IIO device into the parent device.
- gets a voltage reference either from an external or internal voltage regulator.
- initializes name, operation mode, iio_info, on its parent IIO device.
- sets up the input channels based on device node properties.
- registers the initialized device with the IIO subsystem.

Figure 4.8 presents a reduced version of the AD7292 `probe` function.

The following sections bring more details on how to get voltage regulators as well as to obtain properties defined on device nodes.

```
1   static int ad7292_probe(struct spi_device *spi)
2   {
3           struct ad7292_state *st;
4           struct iio_dev *indio_dev;
5           struct device_node *child;
6           bool diff_channels = 0;
7           int ret;
8
9           indio_dev = devm_iio_device_alloc(&spi->dev, sizeof(*st));
10          if (!indio_dev)
11                  return -ENOMEM;
12
13          st = iio_priv(indio_dev);
14          st->spi = spi;
15
16          spi_set_drvdata(spi, indio_dev);
17
18          /* Get managed regulator device or use internal voltage reference */
19
20          indio_dev->dev.parent = &spi->dev;
21          indio_dev->name = spi_get_device_id(spi)->name;
22          indio_dev->modes = INDIO_DIRECT_MODE;
23          indio_dev->info = &ad7292_info;
24
25          /* Set channels according to devicetree node description */
26
27          return devm_iio_device_register(&spi->dev, indio_dev);
28  }
```

Figure 4.8: *A reduced version of the probe function registered by the AD7292 driver.*

4.1.10 Managed device resources

The device core has functions that allow self-managed use of resources. Devices booted and registered with these functions have their resources automatically freed when removed from the system. By letting developers associate `release` functions to give off each asset reserved for a device, the subsystem mitigates resource leakage and errors when detaching drivers [11]. IIO extends the resource management interface by also allowing self-management of channels, buffers, and triggers.

During the initialization of AD7292 devices, the driver requests the LDM a voltage regulator to get a voltage reference. If served, it registers `ad7292_regulator_disable` as release function (see Figure 4.9). Since the LDM manages the voltage regulator instance, all the AD7292 driver has to do in its release function is to say that it will no longer use the regulator. Otherwise, the driver uses the device internal voltage regulator to operate.

The `devm_regulator_get_optional` function is used to obtain a voltage regulator device. It takes the regulator name and a device node to look within. The voltage regulator framework molds a property name by appending "-supply" to the regulator's name. Then a recursive search from the consumer device to its children looks for the regulator property name inside each device node. If found, a reference to the specified voltage regulator (`regulator_dev`) is returned and used as a supply for the requesting device.

For AD7292, the property name that specifies the voltage regulator is `vref-supply` (see Figure 3.1).

```
1     st->reg = devm_regulator_get_optional(&spi->dev, "vref");
2     if (!IS_ERR(st->reg)) {
3             ret = regulator_enable(st->reg);
4             if (ret) {
5                     dev_err(&spi->dev,
6                             "Failed to enable external vref supply\n");
7                     return ret;
8             }
9
10            ret = devm_add_action_or_reset(&spi->dev,
11                                           ad7292_regulator_disable, st);
12            if (ret) {
13                    regulator_disable(st->reg);
14                    return ret;
15            }
16
17            ret = regulator_get_voltage(st->reg);
18            if (ret < 0)
19                    return ret;
20
21            st->vref_mv = ret / 1000;
22    } else {
23            /* Use the internal voltage reference. */
24            st->vref_mv = 1250;
25    }
```

Figure 4.9: *Code snippet responsible for acquiring external voltage regulators for AD7292 devices.*

4.1.11 Properties from devicetree nodes

It is sometimes desirable to obtain information about the device hardware configuration. When supplied with a devicetree, the kernel allows rescuing nodes from it. There are consolidated procedures for searching, browsing, reading, updating, and removing device nodes implemented as part of the open firmware (OF) framework.

For the ad7292 driver, it is meaningful to find out the configuration of the first two analog input channels. The OF framework helps to retrieve this information from the DTS. The `for_each_available_child_of_node` macro expands in a loop that traverses the child nodes of ad7292 to check whether any of them contain a property called diff-channels. If so, the driver sets VIN0 and VIN1 as differential; otherwise, it sets all channels as single-ended (see Figure 4.10).

```
1     for_each_available_child_of_node(spi->dev.of_node, child) {
2             diff_channels = of_property_read_bool(child, "diff-channels");
3             if (diff_channels)
4                     break;
5     }
6
7     if (diff_channels) {
8             indio_dev->num_channels = ARRAY_SIZE(ad7292_channels_diff);
9             indio_dev->channels = ad7292_channels_diff;
10    } else {
11            indio_dev->num_channels = ARRAY_SIZE(ad7292_channels);
12            indio_dev->channels = ad7292_channels;
13    }
```

Figure 4.10: *AD7292 channel set selection.*

4.1.12 Driver compatibility

The `MODULE_DEVICE_TABLE` macro creates aliases that are further processed and included in tables referenced by the driver core to match devices against drivers. Since the driver core has multiple methods of discovering devices (bus address space, devicetree, ACPI tables), some drivers export more than one compatibility table. To express compatibility with devices initialized by the SPI subsystem, drivers must provide a `spi_device_id` array containing the names of the supported devices. Similarly, to indicate compatibility with devices described in devicetree, a vector of type `of_device_id` must be initialized.

SPI drivers must also initialize a `spi_driver` struct to specify what are the available attributes and functions. The `module_spi_driver` macro generates both an `_init`, and an `_exit` function for registering and unregistering the driver within the SPI subsystem. The SPI subsystem, in turn, registers the device with the driver core to enable it to carry on underlying initialization such as reference counting, event handling, sysfs integration.

The AD7292 driver is an SPI driver, so it defines a struct `spi_driver` assigning the driver name, devicetree compatibility table, SPI subsystem compatibility table, and probe function (see Figure 4.11).

```
1   static const struct spi_device_id ad7292_id_table[] = {
2           { "ad7292", 0 },
3           {}
4   };
5   MODULE_DEVICE_TABLE(spi, ad7292_id_table);
6
7   static const struct of_device_id ad7292_of_match[] = {
8           { .compatible = "adi,ad7292" },
9           { },
10  };
11  MODULE_DEVICE_TABLE(of, ad7292_of_match);
12
13  static struct spi_driver ad7292_driver = {
14          .driver = {
15                  .name = "ad7292",
16                  .of_match_table = ad7292_of_match,
17          },
18          .probe = ad7292_probe,
19          .id_table = ad7292_id_table,
20  };
21  module_spi_driver(ad7292_driver);
```

Figure 4.11: *AD7292 driver compatibility table definition.*

4.2 Linux maintainers

Many of Linux artifacts have its maintainers listed in the MAINTAINERS file. At the root of the kernel source tree, this file contains the list of developers accounted for developing or giving support for some resource. To include information about the AD7292 driver, the following entry was added:

```
ANALOG DEVICES INC AD7292 DRIVER
M: Marcelo Schmitt <marcelo.schmitt1@gmail.com>
```

```
L: linux-iio@vger.kernel.org
W: http://ez.analog.com/community/linux-device-drivers
S: Supported
F: drivers/iio/adc/ad7292.c
F: Documentation/devicetree/bindings/iio/adc/adi,ad7292.yaml
```

This specifies that:

- the Linux now have a driver for the AD7292 monitoring system.
- the maintainer of the AD7292 driver is Marcelo Schmitt.
- the mailing list in which patches to this part should be discussed is linux-iio@vger.kernel.org.
- the web-page with status information about this part is http://ez.analog.com/community/linux-device-drivers.
- the status of the driver is supported, which means that someone is actually paid to look after it.
- the resource consists of two files:
  - *drivers/iio/adc/ad7292.c*
  - *Documentation/devicetree/bindings/iio/adc/adi,ad7292.yaml*

A list with all accepted tags and their meanings can be found in the MAINTAINERS file.

Chapter 5

Final remarks

The driver for AD7292 devices is the main result of this work. It was developed following the best practices of open source development in the Linux kernel which included:

- Community review steps and implementation of proposed improvements.
- Hardware testing under various configurations.
- Development of respective documentation.
- Validation of proposed source code format and documentation.

As a result of the success of this process, the driver has been accepted by the community and will be available worldwide from Linux kernel version 5.5 [32, 33]. This first release allows you to use AD7292 devices to take analog read measurements at the command of the user or an application.

Among the lessons learned throughout the development of this work, it seems fair to stress the importance of interacting constructively with the community. Many developers participating in the Linux kernel community are volunteers, even those who do are paid to work on it may not be available at any time. Therefore, patience during the review process is very appreciated. Also, kernel developers occasionally get bothered when one asks questions previously discussed. Thus, the tip to avoid blunt answers is to look for information in the kernel documentation before asking in the mailing list. This attitude can help saving time of more experienced developers so they can spend more hours contributing with code review, bug fixing, or software development. By the way, always answer the reviewers thanking for their help and addressing any question raised. Developers who ignore reviewers increase their chance of getting ignored in turn. On the other hand, showing gratitude may captivate people to continue supporting the work being done. Notwithstanding, kernel developers are human beings like everyone else. As it is in real life, people make mistakes and when things go wrong, we apologize and try to work toward a solution. An additional hint for non-native English speakers is to carefully read the messages in the mailing lists. A few minutes of a second reading may avoid wasting more time clarifying misunderstandings. Finally, as general advice, always try to be friendly. Most of the time, good faith can be assumed by the developers working in the Linux kernel.

Over a year and a half of kernel development, my experience is that people always try to help.

As a last piece of advice, we suggest reading the official documentation[1]. It contains valuable tips on how to interact with the community, what to do and what not to do, and answers to frequently asked questions.

As a suggestion for future work, one may tackle some of the AD7292 features that are yet to be supported by a Linux device driver:

- check the BUSY pin state after ADC conversion commands to ensure conversion correctness before reading data from the result register.
- add support for internal temperature readings.
- add support for operating the DACs.
- add support for customizing configurations at the register bank.
- add support for alarm and custom GPIO features.
- implement continuous readings within a buffer triggered mode.

Finally, the conclusion of this work paves the way for a series of work toward producing a driver that can exploit all of the features of AD7292 devices.

[1] https://www.kernel.org/doc/html/latest/

Chapter 6

Personal Appreciation

I really enjoyed developing software for the Linux kernel. It was a challenging and rewarding experience at the same time. It was very different from the majority of college work. I interacted with developers from different parts of the world (England, Romania, the United States of America, and China).

I had the opportunity to work in an area of intersection between hardware and software that kept me motivated all the time to learn more about both sides. At the same time, I participated in many community activities on Hardware Livre USP[1] and FLUSP[2] that prompted me to continue encouraging others in the community to also engage in open source hardware and/or software projects. Overall I think this year was very tiring, but also very rewarding.

[1] http://hardwarelivreusp.org/
[2] https://flusp.ime.usp.br/

References

[1] Javier Martinez Canillas. *Kbuild: the Linux Kernel Build System*. Dec. 2012. URL: https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system (visited on 10/02/2019) (cit. on pp. 13, 14).

[2] Piers Cawley. *Majordomo*. 1995. URL: https://www.linuxjournal.com/article/1067 (visited on 12/01/2019) (cit. on p. 6).

[3] The kernel development community. *Linux kernel licensing rules*. URL: https://www.kernel.org/doc/html/latest/process/license-rules.html (visited on 10/25/2019) (cit. on pp. 19, 20).

[4] Jonathan Corbet. *An alternative device-tree source language*. Aug. 2017. URL: https://lwn.net/Articles/730217/ (visited on 10/16/2019) (cit. on p. 15).

[5] Jonathan Corbet. *Kernel development*. 2001. URL: https://lwn.net/2001/0704/kernel.php3 (visited on 10/16/2019) (cit. on p. 15).

[6] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, Nov. 2005, p. 16 (cit. on p. 7).

[7] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, Nov. 2005, p. 657 (cit. on p. 7).

[8] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, Nov. 2005, p. 16 (cit. on p. 15).

[9] The Linux Kernel documentation. *Building External Modules*. 2019. URL: https://www.kernel.org/doc/html/latest/kbuild/makefiles.html (visited on 10/08/2019) (cit. on p. 13).

[10] The Linux Kernel documentation. *Core elements*. 2019. URL: https://www.kernel.org/doc/html/latest/driver-api/iio/core.html#iio-device-channels (visited on 11/12/2019) (cit. on p. 24).

[11] The Linux Kernel documentation. *Devres - Managed Device Resource*. 2019. URL: https://www.kernel.org/doc/html/latest/driver-api/driver-model/devres.html (visited on 11/15/2019) (cit. on p. 29).

[12] The Linux Kernel documentation. *HOWTO do Linux kernel development*. 2020. URL: https://www.kernel.org/doc/html/latest/process/howto.html (visited on 01/12/2020) (cit. on pp. 5, 6).

[13] The Linux Kernel documentation. *Kbuild*. 2019. URL: https://www.kernel.org/doc/html/latest/kbuild/kbuild.html#arch (visited on 10/09/2019) (cit. on p. 14).

[14] The Linux Kernel documentation. *Kbuild: the Linux Kernel Build System*. 2019. URL: http://delivery.acm.org/10.1145/2400000/2392897/11333.html?ip=143.107.45.1&id=2392897&acc=ACTIVE%20SERVICE&key=344E943C9DC262BB%2E0DBCED839AA5AFE8%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1574540392_5702b62683a92e05b622a0c25521dcd4 (visited on 11/23/2019) (cit. on pp. 13, 14).

[15] The Linux Kernel documentation. *Linux Kernel Makefiles*. 2019. URL: https://www.kernel.org/doc/html/latest/kbuild/makefiles.html#kbuild-variables (visited on 10/08/2019) (cit. on p. 14).

[16] Inc Eclipse Foundation. *IoT Developer Survey Results*. Apr. 2018. URL: https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2018.pdf (visited on 11/28/2019) (cit. on p. 1).

[17] The Linux Foundation. *2017 Linux Kernel Development Report*. 2019. URL: https://go.pardot.com/l/6342/2017-10-24/3xr3f2/6342/188781/Publication_LinuxKernelReport_2017.pdf (visited on 11/29/2019) (cit. on p. 1).

[18] The Linux Foundation. *Active kernel releases*. 2020. URL: https://www.kernel.org/category/releases.html (visited on 01/12/2020) (cit. on p. 6).

[19] Rob Herring. *Device-tree schemas*. URL: https://github.com/robherring/dt-schema (visited on 10/24/2019) (cit. on p. 16).

[20] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, Feb. 2005, p. 370 (cit. on p. 6).

[21] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, Feb. 2005, p. 362 (cit. on p. 7).

[22] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, Feb. 2005, p. 377 (cit. on p. 7).

[23] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, Feb. 2005, p. 379 (cit. on p. 7).

[24] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, Feb. 2005, p. 385 (cit. on p. 8).

[25] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, Feb. 2005, p. 394 (cit. on p. 28).

[26] Grant Likely. *Linux and the Device Tree*. URL: https://github.com/torvalds/linux/blob/master/Documentation/devicetree/usage-model.txt (visited on 11/25/2019) (cit. on p. 15).

[27] Linaro Ltd Linaro Ltd. *Devicetree Specification*. Dec. 2017. URL: https://github.com/devicetree-org/devicetree-specification/releases/download/v0.2/devicetree-specification-v0.2.pdf (visited on 10/16/2019) (cit. on p. 15).

[28] LinuxLinks. *Majordomo - mailing list manager*. 2019. URL: https://www.linuxlinks.com/majordomo/ (visited on 12/01/2019) (cit. on p. 6).

[29] John Madieu. *Linux Device Drivers Development*. Packt Publishing, 2017, p. 332 (cit. on p. 7).

[30] John Madieu. *Linux Device Drivers Development*. Packt Publishing, 2017, p. 16 (cit. on p. 8).

[31] Marta Rybczyńska. *Device-tree schemas*. Nov. 2018. URL: https://lwn.net/Articles/771621/ (visited on 10/16/2019) (cit. on p. 15).

[32] Marcelo Schmitt. *Industrial Input / Output Subsytem tree*. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/jic23/iio.git/commit/?h=testing&id=d898f9ac542f9c60c5760cfe4b9cb10c635feb38 (visited on 12/05/2019) (cit. on p. 33).

[33] Marcelo Schmitt. *Industrial Input / Output Subsytem tree*. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/jic23/iio.git/commit/?h=testing&id=506d2e317a0a02631a74bbc4c508334c29e26eae (visited on 12/05/2019) (cit. on p. 33).

[34] TOP500 team. *Operating system Family / Linux*. 2019. URL: https://www.top500.org/statistics/details/osfam/1 (visited on 11/28/2019) (cit. on p. 1).

[35] Unknown. *Device Tree Compiler Manual*. URL: https://git.kernel.org/pub/scm/utils/dtc/dtc.git/plain/Documentation/manual.txt?id=HEAD (visited on 11/25/2019) (cit. on p. 15).

[36] vger.kernel.org. *VGER.KERNEL.ORG*. 2019. URL: http://vger.kernel.org/ (visited on 12/01/2019) (cit. on p. 6).

[37] vger.kernel.org. *VGER.KERNEL.ORG Majordomo Info*. 2019. URL: http://vger.kernel.org/majordomo-info.html (visited on 12/01/2019) (cit. on p. 7).