University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Testing Linux Drivers with Device Emulation Aid

Lucas Pires Stankus

Final Essay

mac 499 — Capstone Project

Supervisor: Paulo Meirelles
Co-supervisor: Marcelo Schmitt

São Paulo
December 5, 2021

Acknowledgments

For supporting me throughout the year, I acknowledge my parents, Renata and Venceslau, and my sister Júlia. They did their best in putting up with my complaints during this intense year of work. I also acknowledge my friends, Augusto, Gabriel, and Guilherme, for being there for me when I needed it the most. A great thanks to Alexandru Ardelean, Andy Shevchenko, Jonathan Cameron and Rob Herring for providing me guidance to develop a high-quality device driver for the Linux kernel; and Darius Berghe, Dragos Bogdan, and Nishant Malpani for mentoring and assisting me through the device driver's development. Last but clearly not least, a special thanks to my supervisors; Marcelo Schmitt, for introducing me to the world of open-source and device driver development and for being a source of inspiration and guidance; Paulo R. M. Meirelles for giving me the opportunity to work with the Linux kernel and for being supportive throughout the year; and both for putting a real effort in reviewing this work. Thank you all.

Resumo

Lucas Pires Stankus. **Testando Drivers do Linux com Auxílio de Emulação de Dispositivos**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Drivers de dispositivos são uma das partes mais importantes do kernel Linux, compondo a maior parcela do seu código fonte. Todavia, ainda falta no kernel um arcabouço para validar e testar drivers. Como consequência, esses acabam concentrando mais erros que qualquer outra de suas partes. Dentro de várias soluções de curto prazo, emulação de dispositivos e utilização desses como alvo de drivers vem sendo uma tática para testá-los adotada por membros relevantes da comunidade do Linux. Este trabalho tem como objetivo analisar o uso de emulação como ferramenta para auxiliar o desenvolvimento de drivers de dispositivos no kernel Linux. Para atingir o objetivo proposto, o modelo ADXL313 foi usado como estudo de caso. Ele é uma especificação de circuito integrado que descreve um acelerômetro digital de três eixos, recomendado para aplicações em alarmes de carros e caixas-pretas. Durante este trabalho, um driver para controlar a operação de dispositivos ADXL313 no kernel Linux e um dispositivo ADXL313 emulado foram desenvolvidos. Ambas implementatações foram testadas contra a outra. Devido ao sucesso deste projeto, o driver desenvolvido foi aceito pela comunidade do Linux e vai estar disponível a partir da versão 5.16 do kernel Linux. O teste do driver contra a emulação também foi bem sucedido. Eles puderam ser testados um contra o outro em um ambiente virtual sem qualquer erro. Contudo, muitas deficiências desse processo foram encontradas em comparação com arcabouços de testes, invalidando o uso da configuração para integração contínua de drivers de dispositivos em ambientes de produção.

Palavras-chave: Linux. Device driver. Free software. QEMU. Emulação. Integração Contínua.

Abstract

Lucas Pires Stankus. **Testing Linux Drivers with Device Emulation Aid**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

Device drivers are one of the essential parts of the Linux kernel, making up the most significant share of its source code. However, the kernel still lacks proper frameworks for validating and testing drivers. As a result, they end up concentrating more bugs than any other of its parts, and among many short-term solutions, emulating devices and using them as the drivers' target has been an approach for testing adopted by relevant members of the Linux community. This work aims to analyse the use of device emulation as a tool for aiding device driver development in the Linux kernel. The ADXL313 device model was taken as a case study to achieve the proposed goal. This design is an integrated circuit specification that describes a 3-axis digital accelerometer recommended for car alarms and black boxes applications. During this work, a driver to control the operation of ADXL313 devices for the Linux kernel and an ADXL313 emulated device was developed. Both implementations were then tested against each other. Due to the success of this project, the driver developed has been accepted by the Linux community and will be available from version 5.16 onwards of the Linux kernel. The test of the driver against emulation was also successful. They were able to be executed against each other in a virtual environment without any errors. However, many shortcomings of the process were found compared to standard testing frameworks, invalidating the use of the setup for continuous integration of device drivers in a production environment.

Keywords: Linux. Device driver. Free software. QEMU. Emulation. Continuous Integration.

Contents

Chapter 1

Introduction

The Linux[1] operating system kernel is one of the most significant open-source software projects to date and one of the most important collaborative projects in history. It is almost ubiquitous in computing, being a predominant force in almost all of its areas. From powering all current top 500 super computers [30], the majority of the cellphone market share with Android [27] and about 70% of devices in IoT applications [20], a strong argument can be made that it is the underlying foundation of modern computing.

Linux is surrounded by vibrant communities of contributors. Developers from around the world dive into its development; either for the challenge, for self-improvement or for fun, people make massive amounts of contributions to the Linux source code. From 2005 to 2017 it has received code patches from roughly 15,600 unique developers, ranking it as one of the most popular projects ever [20]. With this many people actively inspecting and improving the source code, the risk of security issues and user surveillance is minimized. Its open nature foments a sense of community, encouraging knowledge sharing and collaborative development among peers.

An operating system kernel is a software responsible for controlling the hardware operations and managing the machine's resources, providing abstractions for their usage. It is a highly complex software; in particular, Linux's source code has over twenty million lines of code [20]. To handle its complexity and size, Linux separates its codebase into smaller, maintainable parts called subsystems, each responsible for smaller portions of the functionality provided by the kernel. There are components responsible for handling different hardware devices within many of these subsystems. These programs are called device drivers.

Drivers are fundamental to the functioning of the Linux kernel. A driver is responsible for mapping existing kernel abstractions to the functionality provided by a given device. With the abstractions, user-space programs can leverage the device's resources without dealing with any intricacies of the hardware [18]. As this process is heavily dependant on the device specification, each board design must be targeted by a driver to function correctly within the system. With the constant influx of new devices to the market, driver

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

development has become one of the most significant parts of the Linux kernel. Currently, the majority of lines of code in Linux comes from drives, but this brings some issues to light [7].

Code in Linux "hardens" over time, which means older code tends to have fewer bugs and security risks as more developers take time to examine it. However, device drivers are much more recent than core parts of the kernel. Therefore, they tend to be more prone to vulnerabilities. Also, verifying if a given driver works properly is often troublesome. Validation against physical hardware is difficult. A subsystem might contain dozens or even hundreds of drivers, rendering it almost impossible for maintainers to own every device present in it. Validation against Continuous Integration (CI) frameworks is also problematic. CI for device drivers is still an open problem in Linux. Today, there is no built-in kernel infrastructure, and there are very few frameworks for it, most no longer supported and none enforced by subsystems. In summary, it is possible to state that device driver testing is still precarious. As a result, device drivers end up concentrating more bugs than any other part of the kernel [7].

Among the various solutions for the lack of proper testing of device drivers, a short-term possibility is using emulation to assist developers with device testing and development. This approach consists of emulating the target device inside a machine emulation environment. The device driver can then be tested against the emulated peripheral, possibly assisting testing and development of the driver.

1.1 Objective

Using emulation to assist driver development is an idea that has already been in the minds of some relevant members of the Linux communities[2]. However, it is still quite novel; hence few real experiments have been made with it and not many reports are available. It's not clear if device emulation is a good solution for device driver testing, but it might provide a great auxiliary tool while real long-term solutions are still in development.

Given that, this work aims to explore the use of emulation to assist the development and testing of device drivers in Linux, taking the ADXL313 model as example. ADXL313 chips were designed by Analog Devices Inc. to measure acceleration primarily in car alarms and black boxes, as they provide several built-in sensing functions and are quite shock resistant.

1.2 Used Conventions

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

- *italic* will be used to express parts of the file system, including files and directories.
- `monospaced` will be used to express parts of the computer source code, such as structures, variables and functions.

[2] https://lore.kernel.org/linux-iio/20210614113507.897732-1-jic23@kernel.org/

- **bold** will be used to highlight definitions and emphasize important concepts.

1.3 Manuscript Structure

Besides this introduction, the manuscript has five other chapters, organized as follows. Chapter 2 introduces the core concepts of a device driver in Linux using the Industrial I/O subsystem's core framework, exemplified by the implementation of the ADXL313 driver in the subsystem. Chapter 3 presents the fundamental concepts behind device emulation using QEMU, going through a bit of its architecture and device model and necessary background. Chapter 4 follows the previous, focusing on concepts about device emulation implementation itself with an in-depth overview of the ADXL313 emulation code. Chapter 5 provides a discussion about the results of the exploratory work and presents the conclusions and possible future works. Finally, Chapter 6 shares the author's personal appreciation about this work.

Chapter 2

Linux Driver Development

One of the essential parts of emulation is understanding the target hardware and how it interacts with the surrounding software from a high level. However, to analyze whether it can aid the development of its surroundings, in-depth knowledge of the interactions and how the software is structured becomes essential. Therefore, as this work aims to study how emulation can assist developers in device driver development, it is essential to break down first how one can be implemented. This chapter aims to cover relevant core concepts of driver development in Linux using the Industrial I/O (IIO) subsystem, focusing on how the device is accessed from the system's perspective and the driver's main routines and data flow.

For such, the ADXL313[1] IIO driver implementation will be used as a role model. The device is a 3-axis digital accelerometer with a handful of features, including configurable automatic sleep mode and activity and inactivity setups for data capture. Additionally, it works with various connection setups, including 4-wire and 3-wire SPI configurations, as well as standard I^{2}C.

This work does not focus on providing a comprehensive guide to Linux driver development. This chapter gives a high-level view of driver structure and the necessary background. Thus, beginners looking to get into driver development might find this work lacking in some respects. For example, topics such as the kernel build system will not be discussed here. If that is the case, one can find more guidance through Schmitt's work [29].

2.1 Regmap

Regmap is a register access mechanism provided by the Linux kernel that mainly targets SPI, I^{2}C, and memory-mapped registers. It was introduced in version 3.1 of the Linux kernel to allow factorizing and unifying access to devices with different protocols. When set, a regmap provides an abstract access layer to device registers which does not require specifying the access method. To set up a regmap, one must initialize the appropriate

[1] https://www.analog.com/en/products/adxl313.html

configuration structures with data suited to the connection capabilities. All accesses to the device are then wrapped into a unified API[2], regardless of protocol [25][26].

Regmaps can leverage driver development for devices with support for multiple protocols. All device access code can be easily abstracted away by setting a regmap at device initialization for the invoked protocol. As access is often the only unique part of each protocol implementation, the driver can be easily split into two conceptual parts, the core code and the protocol-specific initialization and registration. This division leads to cleaner code, with better separation of concerns, and fewer redundancies in general across multiple kernel drivers [25].

```
regmap = devm_regmap_init_i2c(client, &adxl313_i2c_regmap_config);
if (IS_ERR(regmap)) {
        dev_err(&client->dev, "Error initializing i2c regmap: %ld\n",
                PTR_ERR(regmap));
        return PTR_ERR(regmap);
}
```

Figure 2.1: *I^2C regmap initialization for the ADXL313 driver.*

Given that, the ADXL313 driver is split across three source files, *adxl313_core.c*, *adxl313_spi.c*, and *adxl313_i2c.c*, holding, respectively, the core code and the SPI and I^2C specific code. Regmap initialization and configuration are handled only in the latter two. Figure 2.1 exemplifies a regmap initialization with proper error handling. A function call to the appropriate protocol constructor takes as parameters the device being abstracted and the regmap configuration for it. There are many possible properties one can set to a regmap. The most relevant regmap properties for the ADXL313 driver are:

- **reg_bits:** The number of bits required to describe a register address, similar to the 32-bit and 64-bit definitions in modern machines for memory address space.
- **val_bits:** The number of bits in a register value, therefore the size in bits of the value present in a given register address.
- **rd_table:** A pointer to a `regmap_access_table` structure specifying the valid register ranges for read access.
- **wr_table:** Analogous to `rd_table` for write access.
- **max_register:** The maximum valid register address.
- **read_flag_mask:** A mask to be set in the top bytes of the register address when doing a read.

The regmap configuration structures for both SPI and I^2C protocols of the ADXL313 driver can be seen in Figure 2.2 and the register tables referenced by it in Figure 2.3.

Apart from being able to easily catch invalid device register accesses by the driver, specifying which registers can be written to and read from also enables one to use the kernel debug tools provided by the regmap subsystem. Its debug interface can be found in

[2] The complete API can be found at https://github.com/torvalds/linux/blob/master/include/linux/regmap.h

```c
static const struct regmap_config adxl313_spi_regmap_config = {
	.reg_bits	= 8,
	.val_bits	= 8,
	.rd_table	= &adxl313_readable_regs_table,
	.wr_table	= &adxl313_writable_regs_table,
	.max_register	= 0x39,
	 /* Setting bits 7 and 6 enables multiple-byte read */
	.read_flag_mask = BIT(7) | BIT(6),
};

static const struct regmap_config adxl313_i2c_regmap_config = {
	.reg_bits	= 8,
	.val_bits	= 8,
	.rd_table	= &adxl313_readable_regs_table,
	.wr_table	= &adxl313_writable_regs_table,
	.max_register	= 0x39,
};
```

Figure 2.2: *ADXL313 regmap configurations for SPI and I^2C.*

```c
static const struct regmap_range adxl313_readable_reg_range[] = {
	regmap_reg_range(ADXL313_REG_DEVID0, ADXL313_REG_XID),
	regmap_reg_range(ADXL313_REG_SOFT_RESET, ADXL313_REG_SOFT_RESET),
	regmap_reg_range(ADXL313_REG_OFS_AXIS(0), ADXL313_REG_OFS_AXIS(2)),
	regmap_reg_range(ADXL313_REG_THRESH_ACT, ADXL313_REG_ACT_INACT_CTL),
	regmap_reg_range(ADXL313_REG_BW_RATE, ADXL313_REG_FIFO_STATUS),
};

const struct regmap_access_table adxl313_readable_regs_table = {
	.yes_ranges = adxl313_readable_reg_range,
	.n_yes_ranges = ARRAY_SIZE(adxl313_readable_reg_range),
};

static const struct regmap_range adxl313_writable_reg_range[] = {
	regmap_reg_range(ADXL313_REG_SOFT_RESET, ADXL313_REG_SOFT_RESET),
	regmap_reg_range(ADXL313_REG_OFS_AXIS(0), ADXL313_REG_OFS_AXIS(2)),
	regmap_reg_range(ADXL313_REG_THRESH_ACT, ADXL313_REG_ACT_INACT_CTL),
	regmap_reg_range(ADXL313_REG_BW_RATE, ADXL313_REG_INT_MAP),
	regmap_reg_range(ADXL313_REG_DATA_FORMAT, ADXL313_REG_DATA_FORMAT),
	regmap_reg_range(ADXL313_REG_FIFO_CTL, ADXL313_REG_FIFO_CTL),
};

const struct regmap_access_table adxl313_writable_regs_table = {
	.yes_ranges = adxl313_writable_reg_range,
	.n_yes_ranges = ARRAY_SIZE(adxl313_writable_reg_range),
};
```

Figure 2.3: *ADXL313 regmap register read and write tables.*

the sysfs virtual file-system, located at */sys/kernel/debug/regmap/entry*, where each entry is an active regmap instance. The features provided by the interface often come in handy for driver development. Among them, it provides ways to peek at register values at any given instant. Thus, one can easily compare raw bit values with the actual return values of the driver, making it a crucial validation tool in the developer's toolkit [26].

2.2 IIO Channels

The most important characteristic of the IIO subsystem is its interface. It abstracts the actual communication with the driver, allowing generic user-space code to interface with a particular device [6]. For such, IIO defines the concept of channels. A channel groups information about a homogeneous data stream that must be exposed to user-space, either for input or output. Besides the raw bits of data, a channel provides relevant metadata for applications, such as the type of physical quantity being measured, sample frequency, scale and offset values to convert data to S.I units, etc. Among the many properties that a channel may have, ADXL313 makes use of the following [16]:

- **type:** The data type of the channel measurement, such as voltage, current, temperature, acceleration.
- **address:** A driver-specific identifier of the channel.
- **modified:** Specifies if a modifier is applied to the channel. The type of this is dependant on the channel's type, and the actual modifier is defined in `channel2`.
- **channel2:** Specifies the modifier to be applied if modified is set. An example modifier is `IIO_MOD_X` to specify the "x" axis in axial sensors.
- **info_mask_separate:** Indicates what information is exposed to the user-space that is specific to the channel.
- **info_mask_shared_by_type:** Indicates what information is exposed to the user-space that is shared by all channels of the same type.
- **info_mask_shared_by_type_available:** Indicates what availability information is exposed to the user-space that is shared by all channels of the same type.
- **scan_type:** A custom structure containing relevant information about the data scan itself, such as the endianness of the data and if it is signed or not. The only field used by the ADXL313 driver is `realbits`, which stores the number of bits used to store the value itself, excluding bits used for padding, for example.

ADXL313 is a 3-axis digital accelerometer that captures data for each axis independently. To expose the data of each axis separately, the device driver implements three unique channels. Per-axis configurable hardware offsets for the acceleration data are also exposed per channel.

The raw data gathered from the accelerometer uses its scaling, defined in the datasheet. However, to be meaningful, the constant for converting the data to standard units[3] must be accessible and thus exposed as channel metadata. Along with that, ADXL313 devices support configuring the polling rate of the acceleration, which can also be provided as channel metadata. The last two channel attributes are set per device type. However, only a specified set of polling rates can be chosen for the accelerometer; these need to be exposed to user-space as well. The complete channel specification of the ADXL313 driver can be seen in Figure 2.4.

[3] Note that not all measurement types use SI units in IIO, one must check the documentation for the standard unit of a given type of data. In the case of acceleration, the standard IIO unit is the same as SI.

```
#define ADXL313_ACCEL_CHANNEL(index, axis) {                            \
        .type = IIO_ACCEL,                                              \
        .address = index,                                               \
        .modified = 1,                                                  \
        .channel2 = IIO_MOD_##axis,                                     \
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) |                  \
                              BIT(IIO_CHAN_INFO_CALIBBIAS),             \
        .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE) |          \
                                    BIT(IIO_CHAN_INFO_SAMP_FREQ),       \
        .info_mask_shared_by_type_available =                           \
                BIT(IIO_CHAN_INFO_SAMP_FREQ),                           \
        .scan_type = {                                                  \
                .realbits = 13,                                         \
        },                                                              \
}

static const struct iio_chan_spec adxl313_channels[] = {
        ADXL313_ACCEL_CHANNEL(0, X),
        ADXL313_ACCEL_CHANNEL(1, Y),
        ADXL313_ACCEL_CHANNEL(2, Z),
};
```

Figure 2.4: *ADXL313 IIO channels specifications.*

Device drivers created by the Linux Device Model, the unified descriptor of devices in the Linux kernel, expose a user-space interface through sysfs and the ones in IIO are no exception [10]. The core framework of the IIO subsystem automatically creates and exposes the device's attributes from the properties of each channel declared by its driver. For an IIO device with index *X*, all its attributes are exposed under the */sys/bus/iio/iio:deviceX* directory of sysfs, where the attributes filenames are also generated given the channel specification. For instance, the `channel2` modifier set in the ADXL313 channels identify what axis the channel represents, reflected by the file's nomenclature. A typical ADXL313 device generates the following files and directories under its sysfs directory:

in_accel_sampling_frequency	*in_accel_y_calibbias*	*of_node*
in_accel_sampling_frequency_available	*in_accel_y_raw*	*power*
in_accel_scale	*in_accel_z_calibbias*	*subsystem*
in_accel_x_calibbias	*in_accel_z_raw*	*uevent*
in_accel_x_raw	*name*	

For each of these files the kernel subsystems processes the access accordingly, eventually calling either *adxl313_read_raw*, *adxl313_write_raw* or *adxl313_read_avail*, given the touched file and the operation, either reading or writing. To convert a raw value to standard units, IIO defines the following formula: value = (raw + offset) * scale. Therefore, reading from any *in_accel_*axis*_raw* returns the raw bit of acceleration data in a given instant, which can then be multiplied by the value read from *in_accel_scale* to get the actual standard unit measurement of the acceleration.

One caveat in the ADXL313 driver is that it does not actually implement offsets from IIO's perspective. By its definition, an offset is applied to the data at the software level, while the ADXL313 offsets are applied at the hardware level. This difference is marked by the different nomenclature of *offset* and *calibbias*. Hence, if a value is written into any *in_accel*_axis_*calibbias*, the offset will be applied automatically to the acceleration data by the device itself, which is not the case for offsets.

Lastly, the device's polling rate can be checked or edited by, respectively, reading from or writing to *in_accel_sampling_frequency*, while the list of valid sampling frequencies can only be retrieved by reading from *in_accel_sampling_frequency_available*.

2.3 Device Private Data

Most device drivers need to store data particular to the device instance to assist its operation. In IIO, device private data must be encapsulated in a single data structure which is then stored by the generic IIO device at runtime. With this approach, the generic device is used by internal kernel code, which abstracts the internal driver data, and, when needed, the private data structure can be retrieved by a function call. Thus, from an object-oriented perspective, the relationship between the two structures is similar to inheritance, where the `iio_dev` is the superclass, and the private data structure is the subclass. The structure in Figure 2.5 defines the private data of an ADXL313 instance.

```
struct adxl313_data {
        struct regmap   *regmap;
        struct mutex    lock; /* lock to protect transf_buf */
        __le16          transf_buf ____cacheline_aligned;
};
```

Figure 2.5: *Data struct that holds ADXL313 device private data.*

- **regmap:** The regmap used for device register access.
- **lock:** A mutex to lock the cacheline aligned transfer buffer.
- **transf_buf:** A buffer used for multi-byte transmissions, as it needs to be cacheline aligned for direct memory access (DMA) support.

2.4 IIO Operations

The `iio_info` structure defines the static device information used by the IIO subsystem at device registering and while handling any incoming request to the device. Among its fields `iio_info` stores several objects, such as pointers to IIO attribute-linked functions, general-purpose and event attributes, buffers and triggers.

The static information of the ADXl313 driver sets callbacks for the `read_raw`, `write_raw` and `read_avail` functions (see Figure 2.6). In the IIO subsystem, these callbacks are responsible for, respectively, requesting data from the device, pushing incoming data to the device and returning the list of available values for a given type of information. All of them operate based on the device channel definitions, taking as arguments a reference to the device, a reference to the accessed channel specification and the type of information being requested.

```
static const struct iio_info adxl313_info = {
        .read_raw       = adxl313_read_raw,
        .write_raw      = adxl313_write_raw,
        .read_avail     = adxl313_read_freq_avail,
};
```

Figure 2.6: *Struct declaring the static information of the ADXL313 driver.*

Given that the three functions operate on the same parameters, the callbacks have a similar structure between them. First, a lookup table finds the channel type. Then the underlying subroutines are called to perform the desired operation, according to the reference of the channel specification. The action to be performed can be further specialized according to the actual channel attributes. The implementations of those will follow in the following subsections.

2.4.1 read_raw

Among the three callbacks, `read_raw` is the most complex as it deals with the largest amount of distinct types. It handles reads of raw acceleration data (`IIO_CHAN_INFO_RAW`), of the scale constant for standard unit conversion (`IIO_CHAN_INFO_RAW`), of the offset applied to the acceleration (`IIO_CHAN_INFO_CALIBBIAS`) and of the current sampling frequency of the device (`IIO_CHAN_INFO_SAMP_FREQ`).

The subroutines for each data type are quite similar, consisting of reading one or more registers of the device, processing the bits accordingly and returning the expected value. The only exception is the scale constant given by the datasheet and can be returned without peeking at the accelerometer. As regmap abstracts the actual device access, the subroutines will differ the most on data processing. For reference, the full implementation of `adxl313_read_raw` can be seen in Figure 2.7.

Parsing and processing the raw data gathered from the device can be a cumbersome task. However, as these issues are recurrent around the Linux kernel, it provides general implementations treating data to simplify dealing with parsing and processing. For instance, several functions automatically convert little-endian or big-endian to the actual CPU endianness. Another example is converting raw bits, returned from regmap as unsigned chars, to signed integers. These features are widely used across multiple drivers and dramatically improve the driver developer workflow. For such, ADXL313 uses both of these function sets, as its acceleration data is a 13-bit signed integer stored across two 8-bit registers in little-endian.

However, some processing has to be done per device according to its implementation, even with the extensive kernel infrastructure. In the ADXL313 driver, this happens in two instances. Firstly, the offset scale used internally by the accelerometer is four times larger than the acceleration's scale. Thus, the offset must be multiplied by four before it is passed to user-space to be consistent with the acceleration. Secondly, the sampling frequency of ADXL313 is not a value by itself. It is an index for a table of valid polling rates defined at the device-sheet and described by `adxl313_odr_freqs` in Figure 2.8. Therefore, the raw data must be matched against the table, and the result is the returned value.

```c
static int adxl313_read_raw(struct iio_dev *indio_dev,
                            struct iio_chan_spec const *chan,
                            int *val, int *val2, long mask)
{
        struct adxl313_data *data = iio_priv(indio_dev);
        unsigned int regval;
        int ret;

        switch (mask) {
        case IIO_CHAN_INFO_RAW:
                ret = adxl313_read_axis(data, chan);
                if (ret < 0) return ret;

                *val = sign_extend32(ret, chan->scan_type.realbits - 1);
                return IIO_VAL_INT;
        case IIO_CHAN_INFO_SCALE:
                /*
                 * Scale for any g range is given in datasheet as
                 * 1024 LSB/g = 0.0009765625 * 9.80665 = 0.009576806640625 m/s^2
                 */
                *val = 0;
                *val2 = 9576806;
                return IIO_VAL_INT_PLUS_NANO;
        case IIO_CHAN_INFO_CALIBBIAS:
                ret = regmap_read(data->regmap, ADXL313_REG_OFS_AXIS(chan->address), &regval);
                if (ret) return ret;

                /*
                 * 8-bit resolution at +/- 0.5g, that is 4x accel data scale
                 * factor at full resolution
                 */
                *val = sign_extend32(regval, 7) * 4;
                return IIO_VAL_INT;
        case IIO_CHAN_INFO_SAMP_FREQ:
                ret = regmap_read(data->regmap, ADXL313_REG_BW_RATE, &regval);
                if (ret) return ret;

                ret = FIELD_GET(ADXL313_RATE_MSK, regval) - ADXL313_RATE_BASE;
                *val = adxl313_odr_freqs[ret][0];
                *val2 = adxl313_odr_freqs[ret][1];
                return IIO_VAL_INT_PLUS_MICRO;
        default:
                return -EINVAL;
        }
}
```

Figure 2.7: *Function for handling raw read operations for the ADXL313 devices.*

```c
static const int adxl313_odr_freqs[][2] = {
        [0] = { 6, 250000 },
        [1] = { 12, 500000 },
        [2] = { 25, 0 },
        [3] = { 50, 0 },
        [4] = { 100, 0 },
        [5] = { 200, 0 },
        [6] = { 400, 0 },
        [7] = { 800, 0 },
        [8] = { 1600, 0 },
        [9] = { 3200, 0 },
};
```

Figure 2.8: *Sample frequency rate table of the ADXL313 driver.*

2.4.2 write_raw

The ADXL313 driver implementation of the `write_raw` callback only handles writes of two data types `IIO_CHAN_INFO_CALIBBIAS` and `IIO_CHAN_INFO_SAMP_FREQ`, as seen in Figure 2.9. Both of these subroutines are fairly similar to the `read_raw` implementations, but with the inverse data flow. Therefore, a user-space value is received, parsed and then written to the device. For the offset, given the scale difference stated previously, the input must be divided by four before it is written to the device to be consistent. While for the sampling frequency, the input is used for an inverse search on the frequency table to find the index. It is then written onto the device if the search is successful.

```
static int adxl313_write_raw(struct iio_dev *indio_dev,
                             struct iio_chan_spec const *chan,
                             int val, int val2, long mask)
{
        struct adxl313_data *data = iio_priv(indio_dev);

        switch (mask) {
        case IIO_CHAN_INFO_CALIBBIAS:
                /*
                 * 8-bit resolution at +/- 0.5g, that is 4x accel data scale
                 * factor at full resolution
                 */
                if (clamp_val(val, -128 * 4, 127 * 4) != val)
                        return -EINVAL;

                return regmap_write(data->regmap, ADXL313_REG_OFS_AXIS(chan->address), val / 4);
        case IIO_CHAN_INFO_SAMP_FREQ:
                return adxl313_set_odr(data, val, val2);
        default:
                return -EINVAL;
        }
}
```

Figure 2.9: *Function for handling raw write operations for the ADXL313 devices.*

2.4.3 read_avail

Finally, the `read_avail` function is the simplest of the bunch, handling only `IIO_CHAN_INFO_SAMP_FREQ`. As the function is only responsible for returning the available values for the sampling frequency, it must only return the values present in `adxl313_odr_freqs` tables. For that, the subroutine only has to return a pointer to the table, its size and data type, as seen in Figure 2.10.

2.5 Device Probing and Setup

When a new device is discovered by the kernel system, the bus-specific routine is dispatched to find a suitable driver for the device in the known drivers' list. The bus routine then calls its registered probe function if one is identified. The probe is responsible for initializing all aspects of the driver necessary to support the newly discovered device. This includes, for example, allocating memory for the private structures, setting up the device's configuration and registering the device itself in the appropriate subsystem [10].

```c
static int adxl313_read_freq_avail(struct iio_dev *indio_dev,
                                   struct iio_chan_spec const *chan,
                                   const int **vals, int *type, int *length,
                                   long mask)
{
        switch (mask) {
        case IIO_CHAN_INFO_SAMP_FREQ:
                *vals = (const int *)adxl313_odr_freqs;
                *length = ARRAY_SIZE(adxl313_odr_freqs) * 2;
                *type = IIO_VAL_INT_PLUS_MICRO;
                return IIO_AVAIL_LIST;
        default:
                return -EINVAL;
        }
}
```

Figure 2.10: *Function for handling read available operations for the ADXL313 devices.*

```c
/**
 * adxl313_core_probe() - probe and setup for adxl313 accelerometer
 * @dev:        Driver model representation of the device
 * @regmap:     Register map of the device
 * @name:       Device name buffer reference
 * @setup:      Setup routine to be executed right before the standard device
 *              setup, can also be set to NULL if not required
 *
 * Return: 0 on success, negative errno on error cases
 */
int adxl313_core_probe(struct device *dev,
                       struct regmap *regmap,
                       const char *name,
                       int (*setup)(struct device *, struct regmap *))
{
        struct adxl313_data *data;
        struct iio_dev *indio_dev;
        int ret;

        indio_dev = devm_iio_device_alloc(dev, sizeof(*data));
        if (!indio_dev) return -ENOMEM;

        data = iio_priv(indio_dev);
        data->regmap = regmap;
        mutex_init(&data->lock);

        indio_dev->name = name;
        indio_dev->info = &adxl313_info;
        indio_dev->modes = INDIO_DIRECT_MODE;
        indio_dev->channels = adxl313_channels;
        indio_dev->num_channels = ARRAY_SIZE(adxl313_channels);

        ret = adxl313_setup(dev, data, setup);
        if (ret) {
                dev_err(dev, "ADXL313 setup failed\n");
                return ret;
        }

        return devm_iio_device_register(dev, indio_dev);
}
```

Figure 2.11: *Probe function registered for the ADXL313 driver.*

As stated in Section 2.1, part of the ADXL313 initialization is protocol specific. Hence, both the I^{2}C and SPI counterparts split the device probe into two separate functions. `adxl313_core_probe` holds the code shared by both protocols, responsible for configuring the device model representation, for allocating and initializing the private device data, for ensuring the device is in a workable state and setting it up for driver use and, finally, for registering it in the IIO subsystem (see Figure 2.11). However, as a device initialization is not complete without the protocol-specific part, the core probe is not registered by any device as its probe function.

On the other hand, the protocol-specific probe function is the first part of the initialization process. They are responsible for setting up the device among the subsystem, configuring and initializing a regmap and, lastly, calling the core probe to finish the device initialization. Since the probe of both protocols does the same procedures thing, but for different device types, their code structure is highly similar, exemplified by `adxl313_spi_probe` in Figure 2.12.

```
static int adxl313_spi_probe(struct spi_device *spi)
{
        const struct spi_device_id *id = spi_get_device_id(spi);
        struct regmap *regmap;
        int ret;

        spi->mode |= SPI_MODE_3;
        ret = spi_setup(spi);
        if (ret) return ret;

        regmap = devm_regmap_init_spi(spi, &adxl313_spi_regmap_config);
        if (IS_ERR(regmap)) {
                dev_err(&spi->dev, "Error initializing spi regmap: %ld\n", PTR_ERR(regmap));
                return PTR_ERR(regmap);
        }

        return adxl313_core_probe(&spi->dev, regmap, id->name, &adxl313_spi_setup);
}
```

Figure 2.12: *SPI specific probe function of the ADXL313 driver.*

2.6 Summary

As stated in the introduction of this chapter, this is not a comprehensive guide to Linux driver development. As one may have noticed, a significant part of the intricacies of the code snippets presented throughout this chapter are purposely ignored in favor of a broader view of the code behaviour. Also, several parts of the driver implementation are not present in this chapter, as they would not add much value in terms of exemplifying the driver data flow; even though they are essential for its proper functioning. However, given this broad overview of an IIO device driver, one can have a better understanding of how data is moved to and from the devices by the driver. This is relevant for emulation, as it is now plausible for one to map an abstract action in the operating system, such as reading from a file in sysfs, to its actual consequence in the hardware-level, such as an access to a certain device register, following with the example. With this in mind, we can now follow up with the emulation itself.

Chapter 3

Device Emulation with QEMU

With the basic understanding of a driver in mind, the following two chapters provide an overview of the code required to emulate a sensor device in QEMU. For such, they will cover the necessary fundamentals and key concepts behind device emulation using QEMU, including relevant code snippets retrieved from an example targeting the ADXL313 accelerometer. The emulation itself is heavily dependant on the target device. However, while the expected behaviour of the code should drastically change between devices, the general guidelines and key concepts presented here should be generalizable for most cases.

This chapter, in particular, will cover the necessary foundations of the QEMU's virtual machine architecture and its device model, and how one can modify the virtual machine in order to add a new device and its dependencies, without going through its implementation.

3.1 ADXL313 Overview

As seen in Chapter 2, the current ADXL313 Linux driver only implements standard read/write functionality. It lacks triggers and triggered buffers; thus, no feature that uses interrupt signals is supported at the moment. At this state, the accelerometer in Linux behaves like a polling system for getting the current acceleration, with the added option of setting offsets to the retrieved data. Nevertheless, this simple implementation requires complex interfacing with the IIO core framework and shares the same core concepts as more advanced device drivers.

Considering the current driver state and that the emulation code is as complex as the device it is trying to emulate, the code to support such features is quite simple but still presents some meaningful challenges. From such challenges, it is possible to analyze the viability of emulation with QEMU to aid driver development and testing, making the ADXL313 a proper case study.

3.2 Devicetree Infrastructure

Most modern machines rely on auto-configuration protocols, commonly ACPI, to automatically identify connected devices and build a model of the available hardware. They require minimal to no pre-configuration from users and make the computers accessible for non-technical people with 'plug-and-play' style features. However, ACPI raises several questions regarding system performance and security, which are strong roadblocks for some hardware implementations [9]. System on a Chip (SoC) and Single Board Computers (SBC) designs usually lack ACPI implementations or any other means of discovering devices, which leaves them with an open problem for modelling the available hardware [8][28]. To solve this, Linux is now favouring the adoption of Devicetrees.

The **Devicetree (DT)** is a data structure specially designed to describe the available hardware in a device concisely. A Devicetree Source (DTS) file describes the hardware in an expressive and human-readable way. It contains a node-tree data structure where each node represents a single component of an SBC or SoC and may contain child nodes and properties definitions. A node property can hold integer cells, strings, raw bytestrings, hexadecimals, references to other nodes, or it can be left empty. To make DTS development friendlier and make its overall structure more readable, labels can be used as aliases to reference nodes, and external definitions can be included from Devicetree Include Files (DTSI), which may also include definitions from other DTSI files [23].

A node that describes a particular sensor device is called **device node**. Since these sensors require device drivers to work properly, device nodes are set to have a `compatible` property that holds one or more strings specifying the device model compatibility. The order of the strings is important as it sets a matching priority, from most specific to most general, and thus allows for setting compatibility with families of device drivers. The recommended format for the compatible strings is *"manufacturer,model"*, where *manufacturer* identifies the manufacturer of the chip and *model* stands for the device model name or number, as the following example:

```
compatible = "mediatek,mt2701-cirq", "mediatek,mtk-cirq";
```

For device nodes with this `compatible` list, the first match attempt will be against a device driver compatible with *mediatek,mt2701-cirq*. If no such driver is found, then the list follows on, and a match will be tried against drivers compatible with *mediatek,mtk-cirq*. If again no driver is found, the node can be ignored. Figure 3.1 shows a device node example for an ADXL313 digital accelerometer.

The raw DTS format is great for human readability but is prohibitively expensive to be stored and processed at boot time, so it needs to be compiled to a better-suited format first by the Devicetree Compiler (DTC). The compilation output is a flat binary encoding of the Devicetree called Flattened Devicetree or Devicetree Blob (DTB). The binary encodes the entire Devicetree in a single, linear, pointerless data structure that is small enough to fit in the bootloader and fast enough to process at boot time [23].

On Linux, the blob is passed to the kernel at boot time by the bootloader, or it is wrapped up with the kernel image, the latter case for supporting non-DT aware firmware [22]. Early in the boot, the kernel parses the DTB to identify the machine and execute the

```
#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/interrupt-controller/irq.h>
spi {
    #address-cells = <1>;
    #size-cells = <0>;

    /* Example for a SPI device node */
    accelerometer@0 {
        compatible = "adi,adxl313";
        reg = <0>;
        spi-max-frequency = <5000000>;
        interrupt-parent = <&gpio0>;
        interrupts = <0 IRQ_TYPE_LEVEL_HIGH>;
        interrupt-names = "INT1";
    };
};
```

Figure 3.1: *A DTS device node example for the ADXL313 digital accelerometer using the SPI protocol.*

platform-specific code required. Later in the kernel initialization, the blob is utilized again to retrieve the list of all device nodes, which are then used to populate the Linux Device Model with data about the platform.

3.3 QEMU

QEMU is a generic and open-source machine emulator and virtualizer. An emulator enables a computer, called host, to run programs targeted for another computer, called guest. Following the trail of thought, a machine emulator aims to emulate an entire computer, which enables running a whole operating system and its applications in a virtual machine (VM). QEMU mainly implements full system emulation for several computer architectures, including x86, arm, and PowerPC. It also supports computer virtualization and user-space emulation for Linux machines [3].

With the architectures, QEMU includes complete configuration for many boards, such as various Raspberry Pi models. These configurations are virtual machines modelled based on real-world board specifications and thus can emulate the intricacies of the board, which would save a developer from having to test code on real hardware. Documentation for the supported machine specifications and architectures can be found inside the system documentation[1] directory at QEMU's source code repository.

One of the emulated specifications is *virt*, a generic virtual machine that, on the contrary, does not target any real-world hardware specification. Most architectures in QEMU have a *virt* machine, and they tend to be very minimal, including only the necessary to get a system running. Such machines are targeted for use cases where reproducing the particularities and limitations of a bit of hardware is not required [12].

Using QEMU as a tool for assisting developers is not a novel idea. Given its capabilities and open-source nature, there has been a growing interest in using QEMU to build tools and frameworks in complex use cases [24][17][19]. It enables programmers to have low-level

[1] https://gitlab.com/qemu-project/qemu/-/tree/master/docs/system

control of the hardware and gives implementations for most typical computer architectures, making it great for code inspection and hand-crafted simulated environments. With that, QEMU presents itself as a perfect candidate for any exploratory work using virtual machines.

As this work aims to analyze how emulation can aid the development and testing of device drivers using virtual machines, the particularities of any device other than the actual test target is out of scope. There are fewer moving parts in virtual architectures' hardware because of their lean configuration. Thus they tend to be easier to modify. Given that, *virt* specs shine as the perfect candidates for emulating sensors. The best option would be an arm-based architecture since the ADXL313 accelerometer is mainly targeted at SoCs and SBCs based on it. Out of the bunch, *aarch64* is the easiest to configure custom kernel versions and, because of that, is the one used in the code examples. One caveat to note, by default arm *virt* uses a 32-bit cortex-a15 CPU, so a custom one with 64-bit support must be explicitly set to use the *aarch64* architecture [12].

For *virt* machines, the hardware is built at VM startup, and the virtual machine specs are hard-coded in the source; hence changing the hardware configuration requires editing its source code. This forces users to understand some concepts of its architecture to make simple modifications to the machine [12]. Since this chapter aims to explain the code required for device emulation, the required changes for adding a virtual device will also be covered in later sections. However, QEMU is a highly complex software with many features, so going in-depth about them is significantly outside of scope, and the snippets will focus closely only on the required concepts.

QEMU's documentation has gone a long way in recent years, but it still lacks in some ways. Platforms tend to have their overall structure well documented with information about their devices and particularities of the board implementation, while QEMU's internal details are often overlooked. The lack of official information about such details blurs the line of what is architecture or board specific to what is global to QEMU. Therefore, even though the intricacies of QEMU covered in the following sections are mostly global, some will be exclusive to arm *virt*.

3.4 Qdev

In QEMU, the virtual devices are organized using a device model abstraction called **qdev** (exemplified in Figure 3.2). They are managed in a tree-like data structure of devices connected by buses. Qdev offers a common API for virtual devices with generic configuration and control for different use cases while keeping the overall model conceptually simple [2].

In this abstraction, a bus is responsible for handling data transfers. All devices must have a bus as their parent, and buses must be implemented by devices. This relationship between buses and devices builds the overall structure of the Qdev tree. Taking an example from Figure 3.2, the PIIX4_PM device implements an I^{2}C bus, which in turn holds several other devices. The only exception to the general rule is the `main-system-bus`, the root bus for the device model tree in QEMU. A single bus can have multiple children devices, and

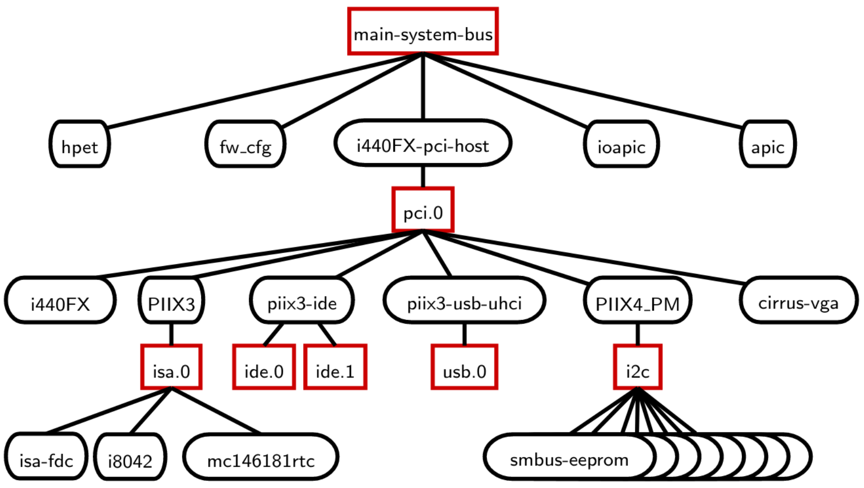

Figure 3.2: *An example tree of a device structure using qdev [2].*

all communication done by them is handled by the direct parent bus. Usually, the children devices would also have local addresses within the scope of their parent [13][2].

To have a particular transfer protocol inside a VM in QEMU, a bus must be implemented by one of the machine's devices, usually an emulated bus controller. Despite that, to increase the security against attacks from the guest machine and to reduce the risk of the virtual machine being broken by any kernel updates, *virt* machines in QEMU tend to avoid having many controllers. To emulate sensor devices, this comes up as an issue as most of them use either the SPI or I^{2}C transfer protocols, both lacking in the default configuration of most *virt* machines. Therefore, to add a sensor to the virtual architecture, one device that implements the correct bus protocol must be added first.

3.5 System Bus Device

Even though a bus and a device are conceptually distinct, in practice, devices that implement a bus are wrapped into one single entity called **system bus device** (sysbus). As sysbus devices include both standard devices and buses, their initialization is more complex. Along with the standard device emulation code, they require their mapped region of memory in the I/O address space and an I/O pin to signal interrupt requests (IRQ).

There would be little to no difference choosing either SPI or I^{2}C as the data transfer protocol to emulate a sensor device; both are plausible to work within the VM. Also, QEMU already has implementations of sysbus devices for either of them, leaving no need to write the emulation code from scratch. Therefore, there should not be much difference between each protocol in terms of VM structure. The main decider between them should be what does the target device support. If a device supports both, as ADXL313 does, either protocol suffice. Given that, the code examples will use QEMU's implementation of the pl022[2] bus microcontroller to handle SPI data transfers. The following subsections will get in-depth about memory-mapped regions for device I/O and interrupt generation.

[2] https://datasheets.globalspec.com/ds/4439/ARM/8A846544-6043-483B-BF18-579EE30A8CC1

3.5.1 Memory Map

One common abstraction of peripheral device communication is memory-mapped I/O (MMIO). Some regions of the addressable memory space are allocated for a particular I/O device. Instead of a typical read/write to memory, when these addresses are accessed, the hardware generates a data access request and sends it to the device. The request is then processed by the peripheral, and the response is given as a regular memory operation from the user's perspective. As it changes the usable memory layout, the BIOS usually creates MMIO regions at boot time and rarely changes during regular operation. The operating system also has to interface with them to properly manage the system RAM without any conflict [5][26].

QEMU uses MMIO for most I/O device communication, with memory addresses defined during the VM initialization. Buses are bound to function under MMIO operation, requiring a memory address region at creation, validated and reserved for it. A memory region in QEMU is defined by the `MemMapEntry` structure, which includes the base memory address and the region's size. One important distinction is that an `MemMapEntry` only holds the necessary information to identify the area of the memory, but by itself, the structure does not interact with it [12][14].

QEMU enforces safety measures while registering the address in the memory. Unless explicitly set, allocated MMIO areas cannot overlap; for such, it would cause request duplication between the devices on writes and possibly undefined behaviour on reads. Likewise, mappings can be set to follow general memory region rules according to their application, defined by the architecture and board specification. For instance, the arm *virt* defines the following region map for the memory addresses[3]:

- Between 0 and 128MB: reserved for flash devices (includes boot device)
- Between 128MB and 256MB: reserved for miscellaneous device I/O
- Between 256MB and 1GB: reserved for PCI support, but yet to be implemented
- Over 1GB: reserved for normal RAM

On some virtual architectures, usually the ones with more complex device structures, it is common to centralize the `MemMapEntry` structures in a map of some sort. Since the map only holds the entry structures, it does not provide any checks or validation of the regions. However, moving them all to a central location helps the overall code organization and simplifies dealing with the memory layout. A map is usually set by an array of `MemMapEntry`, where the index serves as the identifier for the region itself. Macros can cover the indexes to avoid magic numbers in the code.

In arm *virt* two arrays are used to map the memory, one for the lower memory, for addresses up to RAM and used for most cases, and one for the higher memory, after RAM and used for some exceptional use cases. Figure 3.3 exhibits the low memory map of the architecture, in it `VIRT_SPI` stores the index of the entry for the SPI controller.

[3] From code at: https://gitlab.com/qemu-project/qemu/-/blob/master/hw/arm/virt.c#L120

```c
static const MemMapEntry base_memmap[] = {
    /* Space up to 0x8000000 is reserved for a boot ROM */
    [VIRT_FLASH] =              { 0x00000000, 0x08000000 },
    [VIRT_CPUPERIPHS] =         { 0x08000000, 0x00020000 },
    /* GIC distributor and CPU interfaces sit inside the CPU peripheral space */
    [VIRT_GIC_DIST] =           { 0x08000000, 0x00010000 },
    [VIRT_GIC_CPU] =            { 0x08010000, 0x00010000 },
    [VIRT_GIC_V2M] =            { 0x08020000, 0x00001000 },
    [VIRT_GIC_HYP] =            { 0x08030000, 0x00010000 },
    [VIRT_GIC_VCPU] =           { 0x08040000, 0x00010000 },
    /* The space in between here is reserved for GICv3 CPU/vCPU/HYP */
    [VIRT_GIC_ITS] =            { 0x08080000, 0x00020000 },
    /* This redistributor space allows up to 2*64kB*123 CPUs */
    [VIRT_GIC_REDIST] =         { 0x080A0000, 0x00F60000 },
    [VIRT_UART] =               { 0x09000000, 0x00001000 },
    [VIRT_RTC] =                { 0x09010000, 0x00001000 },
    [VIRT_FW_CFG] =             { 0x09020000, 0x00000018 },
    [VIRT_GPIO] =               { 0x09030000, 0x00001000 },
    [VIRT_SECURE_UART] =        { 0x09040000, 0x00001000 },
    [VIRT_SMMU] =               { 0x09050000, 0x00020000 },
    [VIRT_PCDIMM_ACPI] =        { 0x09070000, MEMORY_HOTPLUG_IO_LEN },
    [VIRT_ACPI_GED] =           { 0x09080000, ACPI_GED_EVT_SEL_LEN },
    [VIRT_NVDIMM_ACPI] =        { 0x09090000, NVDIMM_ACPI_IO_LEN},
    [VIRT_PVTIME] =             { 0x090a0000, 0x00010000 },
    [VIRT_SECURE_GPIO] =        { 0x090b0000, 0x00001000 },
    [VIRT_MMIO] =               { 0x0a000000, 0x00000200 },
    [VIRT_SPI] =                { 0x0b100000, 0x00001000 },
    /* ...repeating for a total of NUM_VIRTIO_TRANSPORTS, each of that size */
    [VIRT_PLATFORM_BUS] =       { 0x0c000000, 0x02000000 },
    [VIRT_SECURE_MEM] =         { 0x0e000000, 0x01000000 },
    [VIRT_PCIE_MMIO] =          { 0x10000000, 0x2eff0000 },
    [VIRT_PCIE_PIO] =           { 0x3eff0000, 0x00010000 },
    [VIRT_PCIE_ECAM] =          { 0x3f000000, 0x01000000 },
    /* Actual RAM size depends on initial RAM and device memory settings */
    [VIRT_MEM] =                { GiB, LEGACY_RAMLIMIT_BYTES },
};
```

Figure 3.3: *Memory map with a VIRT_SPI entry.*

3.5.2 Interrupt Request Map

An interrupt request (IRQ) is conceptually simple. It is a hardware signal passed to an interrupt controller, which temporarily stops the running program on the CPU and launches another, the interrupt handler. Interrupts are fundamental to computing because they allow hardware devices to signal when they need to be processed without constantly polling the device for updates, for example, checking when there is a keypress on a keyboard. Computers nowadays have multiple interrupt lines where IRQs can be sent through with specific functionalities, allowing more intelligent decisions by the handler [5].

QEMU implements interrupt after the hardware counterpart. They are just signals sent through an emulated I/O pin, usually a General Purpose Input/Output (GPIO) pin. Consequently, setting up an IRQ for a sysbus device consists of choosing the correct pin to use for its interrupts. The interrupt controllers are modelled after their real-life counterparts, including their set of intricacies, so the chosen interrupt line would vary according to each controller and require case by case analysis. There is a function called `qdev_get_gpio_in` to get a GPIO pin from the controller, which takes as arguments a sysbus device to take the pin from and an integer with the pin id within the bus. For the case of interrupt controllers, the pin id is usually the lane number.

For arm *virt*, the interrupt controller is the Generic Interrupt Controller (GIC) v3[4]. GIC v3 handles a large number of configurable interrupt lanes and, with the limited amount of devices in arm *virt*, most of them end up free for use. A similar array to the memory map is set for the interrupt lanes identifiers to organize the controllers' pins, identifying the taken lanes (see Figure 3.4). The `qdev_get_gpio_in` function should be called with the interrupt controller followed by the identifier as parameters to get the pin for a particular lane identifier.

```
static const int a15irqmap[] = {
    [VIRT_UART] = 1,
    [VIRT_RTC] = 2,
    [VIRT_PCIE] = 3, /* ... to 6 */
    [VIRT_GPIO] = 7,
    [VIRT_SECURE_UART] = 8,
    [VIRT_ACPI_GED] = 9,
    [VIRT_SPI] = 10,
    [VIRT_MMIO] = 16, /* ...to 16 + NUM_VIRTIO_TRANSPORTS - 1 */
    [VIRT_GIC_V2M] = 48, /* ...to 48 + NUM_GICV2M_SPIS - 1 */
    [VIRT_SMMU] = 74,    /* ...to 74 + NUM_SMMU_IRQS - 1 */
    [VIRT_PLATFORM_BUS] = 112, /* ...to 112 + PLATFORM_BUS_NUM_IRQS -1 */
};
```

Figure 3.4: *Interrupt request map (irqmap) with a VIRT_SPI entry.*

With an I/O pin for interrupts and a memory region, adding a device to the VM is just a function call of `sysbus_create_simple`. It takes as arguments, in order, a string to identify the sysbus device, a `MemMapEntry` for the allocated region of memory and a pin for IRQ requests, as in Figure 3.5, where `vms` represents the virtual machine state of the current architecture. For arm *virt* the structure is defined in */include/hw/arm/virt.h*. Going in-depth about the internal representation of a VM's state is outside of scope, although, for context, the `memmap` field is an array similar to the one in Figure 3.3, holding the entries of the memory map of the VM, and `gic` is the interrupt controller device.

```
DeviceState *dev = sysbus_create_simple(
    "pl022",
    vms->memmap[VIRT_SPI].base,
    qdev_get_gpio_in(vms->gic, vms->irqmap[VIRT_SPI])
);
```

Figure 3.5: *Creation of a pl022 sysbus device.*

This whole process is equivalent to physically connecting the device to the computer. Since only a few arm architectures implement auto-discovery protocols for devices and arm *virt* is not one of them, the operating system cannot identify the newly connected device. The device would also need to be described in the devicetree of the VM to achieve that.

[4] https://developer.arm.com/ip-products/system-ip/system-controllers/interrupt-controllers

3.5.3 Devicetree

In the virtual architectures of QEMU, instead of using a pre-compiled DTB file, the flatted device tree (fdt) is generated at VM initialization. QEMU implements a custom framework to generate the fdt from the C source code, which is then passed on to the operating system. Having the devicetree generated in the code simplifies the maintenance of the machines and gives developers more tools at their disposal, such as functions for getting guaranteed unique node handles at runtime [12]. The fdt manipulation API follows quite closely the structure of the devicetree and can be transpiled almost directly to a DTS equivalent. It has been made this way to improve the developers' workflow for adding new devices to the architecture since almost certainly they would already have a DTS example from which they could use as a model.

The code and API of the framework can be found at the */dtc/libfdt/* directory in QEMU's source code repository. Its functions are all prefixed with `qemu_fdt` and most of them perform modifications to the proper fdt, for instance `qemu_fdt_add_subnode`, adds a subnode to the tree, or `qemu_fdt_setprop_cell`, sets a property of a node with a cell. The functions also have the same argument structure, where the first one is the target fdt, the second is a string with the path of nodes from the root to the node in question, and the latter arguments are related to the operation itself. An example of code and the DTS equivalent can be seen at Figures 3.6 and 3.7, respectively.

```
qemu_fdt_add_subnode(fdt, "/parent");
qemu_fdt_setprop(fdt, "/parent", "property-a", "a string", sizeof("a string"));
qemu_fdt_add_subnode(fdt, "/parent/child");
qemu_fdt_setprop_cell(fdt, "/parent/child", "property-b", 0x10);
```

Figure 3.6: *Example code of fdt manipulation.*

```
parent {
    property-a = "a string";
    child {
        property-b = <0x10>;
    };
};
```

Figure 3.7: *DTS equivalent to code in Figure 3.6.*

Even though simpler devices can usually work with only one node in the devicetree, some require more complex setups and thus need more nodes for a proper configuration. Controllers usually follow the latter case, having aspects such as voltage, clocks and so forth to be configured within the system. For example, the pl022 microcontroller requires a system clock and a voltage regulator within the expected bounds established in its datasheet. Figure 3.8 shows the code used to configure the pl022 devicetree node in QEMU's fdt framework, where `ms` is the machine state, the parent structure of the previous virtual machine state. It holds the current hardware state that is common to all platforms in QEMU, such as valid CPUs and RAM, and is defined in */include/hw/boards.h*

```c
// mclk node
qemu_fdt_add_subnode(ms->fdt, "/mclk");
qemu_fdt_setprop_string(ms->fdt, "/mclk", "compatible", "fixed-clock");
qemu_fdt_setprop_cell(ms->fdt, "/mclk", "#clock-cells", 0x0);
qemu_fdt_setprop_cell(ms->fdt, "/mclk", "clock-frequency", 24000);
qemu_fdt_setprop_string(ms->fdt, "/mclk", "clock-output-names", "bobsclk");
uint32_t clk_phandle = qemu_fdt_alloc_phandle(ms->fdt);
qemu_fdt_setprop_cell(ms->fdt, "/mclk", "phandle", clk_phandle);

// regulator node
const char *reg_node = "/regulator@0";
qemu_fdt_add_subnode(ms->fdt, reg_node);
qemu_fdt_setprop(ms->fdt, reg_node, "compatible", "regulator-fixed", sizeof("regulator-fixed"));
qemu_fdt_setprop_cell(ms->fdt, reg_node, "reg", 0);
qemu_fdt_setprop_cell(ms->fdt, reg_node, "regulator-min-microvolt", 3000000);
qemu_fdt_setprop_cell(ms->fdt, reg_node, "regulator-max-microvolt", 3000000);
qemu_fdt_setprop_string(ms->fdt, reg_node, "regulator-name", "vcc_fun");
uint32_t reg_phandle = qemu_fdt_alloc_phandle(ms->fdt);
qemu_fdt_setprop_cell(ms->fdt, reg_node, "phandle", reg_phandle);

// SPI node
char *spi_node = g_strdup_printf("/spi@%" PRIx64, vms->memmap[VIRT_SPI].base); // spi@address
qemu_fdt_add_subnode(ms->fdt, spi_node);
qemu_fdt_setprop_sized_cells(ms->fdt, spi_node, "reg",
                             2, vms->memmap[VIRT_SPI].base,
                             2, vms->memmap[VIRT_SPI].size);
qemu_fdt_setprop_string(ms->fdt, spi_node, "clock-names", "apb_pclk");
qemu_fdt_setprop_cell(ms->fdt, spi_node, "clocks", vms->clock_phandle);
qemu_fdt_setprop_cells(ms->fdt, spi_node, "interrupts",
                       GIC_FDT_IRQ_TYPE_SPI, vms->irqmap[VIRT_SPI],
                       GIC_FDT_IRQ_FLAGS_LEVEL_HI);
qemu_fdt_setprop_cells(ms->fdt, spi_node, "num-cs", 3);
const char compat[] = "arm,pl022\0arm,primecell";
qemu_fdt_setprop(ms->fdt, spi_node, "compatible", compat, sizeof(compat));
```

Figure 3.8: *QEMU code for creating the required devicetree nodes for the pl022 microcontroller.*

3.6 Standard Device

The device creation API in QEMU is unique per bus, hence each has its requirements to create new device instances, the reasoning behind it will be covered in Chapter 4. The only global requisite between all buses is a reference to the parent itself and the name of the peripheral being created. For SPI devices, these are the only arguments for device creation. Moreover, because of the similarities between the SPI and the SSI communication protocol, an SSI bus may be used to handle SPI devices. Given this, Figure 3.9 exhibits the code required for creating a new SPI device in the virtual machine, where `dev` is a sysbus device, a pl022 in this case – as with sysbuses, creating a device is equivalent to plugging it in the machine. Thus, most arm computers cannot properly recognize it. For it to be identified by the Operating System, a device tree node must be generated for the peripheral (see Figure 3.10).

```c
void *bus = qdev_get_child_bus(dev, "ssi");
DeviceState *spidev = ssi_create_peripheral(bus, "adxl313");
if (!spidev) printf("could not create adxl313\n");
```

Figure 3.9: *Creation of an ADXL313 device.*

```
const char compat2[] = "adi,adxl313";
char *nodename2 = g_strdup_printf("/spi@%" PRIx64 "/adc@%" PRIx64,
                                  vms->memmap[VIRT_SPI].base, 0x0l);

qemu_fdt_add_subnode(ms->fdt, nodename2);
qemu_fdt_setprop(ms->fdt, nodename2, "compatible", compat2, sizeof(compat2));

qemu_fdt_setprop_sized_cells(ms->fdt, nodename2, "reg", 1, 0x0);
qemu_fdt_setprop_sized_cells(ms->fdt, nodename2, "spi-max-frequency", 1, 1000000);

qemu_fdt_setprop_cell(ms->fdt, nodename2, "interrupt-parent", pl061_phandle);
qemu_fdt_setprop_cells(ms->fdt, nodename2, "interrupts", 4, 3);
```

Figure 3.10: *QEMU code for creating a devicetree node for the ADXL313 accelerometer.*

Now that the necessary background for adding new devices to QEMU's virtual machines is covered, we may focus on the actual implementation of one.

Chapter 4

Device Emulation Implementation

This chapter covers the code implementation of the emulated ADXL313 accelerometer in QEMU. The goal here is to be generalizable to the majority of devices. However, emulation code is highly coupled with the target device and, as the chapter progresses, its sections will gradually turn less general and more specific to the accelerometer. Later sections, especially 4.4, will go very in-depth about the intricacies of the ADXL313 and its implementation. Nevertheless, a significant part of its behaviour is standard, and it can be treated as a role model for emulating other devices with similar complexity.

About the device API, because of the great distinction between buses in the system, they often demand unique sets of requisites from their child devices. Some require little from them, with only initialization and transfer routines, while others demand complex sets of interactions, possibly unique to the particular bus. The difference complicates establishing a common API covering all use cases and tames the emergent code complexity. The API used by peripherals in QEMU's virtual machines is set per bus to mitigate that, where each bus defines its custom interface and set of operations [2][1]. The emulation code for a device varies drastically according to its parent bus. This section will focus on the SSI interface for SPI and SSI devices.

4.1 Class

The buses' custom API is set by a **class structure**. Classes borrow concepts from object-oriented programming, such as inheritance and virtual methods, to build a model of the available bus configuration and setup for its devices. A class instance is set per device type, covered with details in Section 4.2, where all the implementations particular for the device are set. Even though they are set in a different manner, the final result of this is similar to inheritance.

The API is defined for a given peripheral by setting the related fields with function pointers to the concrete implementations of the methods. Hence, the bus code calls the function present in the field. The fields may also have default values, which can then be overwritten, similar to how a method can be overridden in object-oriented programming. Function pointers compose the majority of the classes' fields. However, general bus configuration can also be present in them, as they are global to the device type, *e.g.* the polarity of the CS lane in SPI.

For SSI peripherals, the class is set by the `SSIPeripheralClass` structure, defined in */include/hw/ssi/ssi.h*. Holding the following fields for device configuration[1]:

- **parent_class:** A reference for the parent class (inheritance).
- **cs_polarity:** Defines whether chip select (CS) exists and if its active high or low.
- **realize:** Function called on device initialization.
- **transfer:** Default function for data transfers, used when the device has standard or no CS behaviour.
- **set_cs:** Function called at CS line change. Optional and only required when the device has side effects related to the CS line.
- **transfer_raw:** Function for non-standard CS behaviour, takes control of the CS behaviour at device level. When set, `transfer`, `set_cs` and `cs_polarity` fields are unused.

The CS line in SPI specifies if the peripheral[2] is selected for a data transfer. Depending on the device, the line can be high or low in standby, and, in devices with standard behaviour, it would be inverted by the controller to signify the start of a transfer. ADXL313 has standard CS behaviour where the line is set to low on transmission. Thus, only `realize` and `transfer` functions will be overwritten.

4.2 Device Type

A device implementation is separated into two concepts, a type and its instances. The **type structure**, in a sense, formalizes the device inside qdev, holding relevant attributes about the device itself, instance metadata, such as size and initialization method, and its class, all aspects global to all instances. A type is defined by the `TypeInfo` structure in */include/qom/object.h* and contains a large number of fields to enable extensive customizability for a variety of complex use cases. Different sets of these attributes have to be configured, all according to the custom requisites set by the target device [15]. SSI buses are fairly straightforward; to create a type for a standard SSI device, the following fields need to be set[3]:

- **name:** The name of the type.
- **parent:** The name of the parent type.

[1] From code at: https://gitlab.com/qemu-project/qemu/-/blob/master/include/hw/ssi/ssi.h#L34

[2] Also known as a slave, however the classic master-slave nomenclature present in SPI and other protocols is being discouraged by the community in favour of the more respectful controller-peripheral counterpart.

[3] From code at https://gitlab.com/qemu-project/qemu/-/blob/master/include/qom/object.h#L373

- **instance_size:** The size of an instance.
- **class_init:** Function called after all parent class initialization has occurred, allowing a class to set its default virtual method pointers. This is also used to override virtual methods from a parent class.

Before they can be used in any form inside QEMU, a user created type must first be declared and registered. The QEMU Object Model (QOM) provides a framework for registering types and instantiating objects. It also implements the object-oriented features found in the class and type structures, such as single-inheritance and multiple inheritances of stateless interfaces.

Types in QOM can be declared and registered in several ways, including defining the macros and calling the internal functions directly. Nevertheless, the most common way is using the `OBJECT_DECLARE_TYPE` macro and its variants for declaration and `type_init` and `type_register_static` macros for, respectively, creating a module with a callback function and for registering the type inside the callback [15]. The available functions, as well as their documentation, can all be found at */include/qom/object.h*. Figure 4.1 contains code snippets for creating, declaring and registering a new SSI device type, where the `ADXL313State` is a structure holding the device state.

```c
#define TYPE_ADXL313 "adxl313"
OBJECT_DECLARE_SIMPLE_TYPE(ADXL313State, ADXL313)

static void adxl313_class_init(ObjectClass *klass, void *data)
{
        DeviceClass *dc = DEVICE_CLASS(klass);
        SSIPeripheralClass *k = SSI_PERIPHERAL_CLASS(klass);

        k->realize = adxl313_realize;
        k->transfer = adxl313_transfer;
        k->cs_polarity = SSI_CS_LOW;

        set_bit(DEVICE_CATEGORY_INPUT, dc->categories);
}

static const TypeInfo adxl313_info = {
        .name          = TYPE_ADXL313,
        .parent        = TYPE_SSI_PERIPHERAL,
        .instance_size = sizeof(ADXL313State),
        .class_init    = adxl313_class_init,
};

static void adxl313_register_types(void)
{
        type_register_static(&adxl313_info);
}

type_init(adxl313_register_types)
```

Figure 4.1: *Code for creating, declaring and registering the ADXL313 type with QOM.*

4.3 Device State

In QOM, instances of a device type are called objects, and they store the values required for emulating the peripheral itself. They represent the device's state. Hence, their types are often referenced as state structures and usually achieve this by mimicking the hardware found in the real-world counterpart, such as registers and memory. Objects also include their peripheral parent structure among their fields. For instance, a SSI device would need to include a `SSIPeripheral` inside of it (see Figure 4.2).

```c
struct ADXL313State {
        // private
        SSIPeripheral ssidev;

        // SPI
        uint8_t address;
        uint8_t spi_mode;

        // device registers
        uint8_t bw_rate;
        int16_t accels[3];
        int8_t offsets[3];
};
```

Figure 4.2: *State structure of the ADXL313 accelerometer.*

The use of embedded structures in the state structure is a common workaround to implement inheritance in the C programming language using pointer arithmetic. It includes the parent structure inside the child types as one of its fields. With this, the pointer to the parent structure is used. For the cases that have to deal with the individual implementations, the pointer to the wrapper structure can be calculated from the field offset inside the structure. Since the pointers to the parent are all the same type, this abstracts the implementation details of the peripherals in a concise way and memory-efficient format without overruling C's type system [4].

Pointer arithmetic is often considered dangerous as it can easily lead to undefined behaviour. However, QOM generates macros for calculating the child structure reference from the parent pointer at device declarations, thus making it safe for users. The macro in question is named after the type name given as an argument in the declaration and is callable, taking only the pointer to the parent device as a parameter. Converting the type often is the first line of code in the interfaces' implementations, as the following example:

```c
ADXL313State *s = ADXL313(dev);
```

4.4 SSI Interface Implementation

The device emulation code presented so far is only related to the peripheral setup. It does not dictate the actual behaviour of the device. The intricacies of how it will work are defined exclusively by the concrete implementations of its bus's interface and can vary according to the user's intention. The behaviour can be guided towards real-world emulation, being as close as possible to the real-world counterpart, or be used as a stub, focusing on predictability to test some part of the pipeline. For this work, the latter makes more sense. Thus, the example ADXL313 emulation uses easily verifiable return values instead of mimicking the actual accelerometer behaviour.

The following paragraphs go in-depth about the ADXL313 implementation of the standard SSI interface and its particularities, which means only the `realize` and `transfer` functions will be covered. As expected, this is heavily coupled with the actual device, the most among any other section covered so far. Nevertheless, the foundation concepts of the presented code should be fairly similar, and thus it can be used as a model while targeting other devices with similar complexity.

When creating boards, one of the hardware designers' goals is to minimize wastage. For this reason, bytes and registers tend to hold multiple information. For example, a boolean value can be represented with only one bit, so allocating an entire byte leaves 7 of those bits unused. The D_i notation will be used to reference specific bits of a value, where i indicates the index of bit, from lower to higher bits, starting from zero.

4.4.1 Realize

The `realize` function is called at device startup, right after object instantiation. Thus it is mainly used to run any initialization procedure of the device and set up its state. The code present in it is often simple and limited to variable initialization, and the argument structure reflects this, taking only a reference to the device and an error pointer for passing failure conditions up the stack. For the ADXL313 implementation, since all fields in the state are set to zero at instantiation, realize only sets arbitrary acceleration values on each axis of the object state (see Figure 4.3).

```
static void adxl313_realize(SSIPeripheral *dev, Error **errp)
{
        ADXL313State *s = ADXL313(dev);

        s->accels[0] = -200;
        s->accels[1] = 0;
        s->accels[2] = 200;
}
```

Figure 4.3: *Realize implementation of the ADXL313 accelerometer.*

4.4.2 Transfer

While the `realize` function tends to be simple, a lot of the complexity in sensor devices' emulation code comes from the `transfer` implementation. As the name suggests, it is called at every data transfer of the peripheral, taking as parameters the raw transferred value and a reference to the device. However, while the SPI protocol specifies how to convey data, it does not define how devices ought to grip incoming data. Thus, data handling complexity is left entirely on the device burden. As a result, SPI compliant devices must define a data-transfer layout to communicate with the controller device. Therefore, the device implementation is responsible for parsing the input accordingly, updating the device state when appropriate, and returning an output when expected [21]. The data transfer layout is often unique to each device model, so the implemented parsing procedure for device emulation is tightly coupled with the ADXL313 design.

```c
static uint32_t adxl313_transfer(SSIPeripheral *dev, uint32_t value)
{
        ADXL313State *s = ADXL313(dev);
        uint32_t out = 0;

        switch (s->spi_mode) {
        case ADXL313_SPI_MULREAD:
                if (value == 0) {
                        out = adxl313_read(dev);
                        s->address += 1;
                        break;
                }
                s->spi_mode = ADXL313_SPI_STANDBY;
                /* fallthrough */
        case ADXL313_SPI_STANDBY:
                s->address = value & 0x3F;
                switch (value >> 6) {
                case 0:
                        s->spi_mode = ADXL313_SPI_WRITE;
                        break;
                case 1:
                        // Undefined behaviour for multiple writes
                        break;
                case 2:
                        s->spi_mode = ADXL313_SPI_READ;
                        break;
                case 3:
                        s->spi_mode = ADXL313_SPI_MULREAD;
                        break;
                }
                break;
        case ADXL313_SPI_READ:
                out = adxl313_read(dev);
                s->spi_mode = ADXL313_SPI_STANDBY;
                break;
        case ADXL313_SPI_WRITE:
                adxl313_write(dev, value);
                s->spi_mode = ADXL313_SPI_STANDBY;
                break;
        }

        return out;
}
```

Figure 4.4: *Transfer implementation of the ADXL313 accelerometer.*

ADXL313's transfers are composed of one byte (8 bits), and its datasheet defines as transmission the group of transfers made while the CS line is active. During a full transmission, multiple calls are made to `adxl313_transfer` as it is invoked at every transferred byte. Usually, communications with sensor devices are started by a controller device that writes a control value to the communication line. This specifies an operation to the peripheral and indicates how it must treat the following transfers, *e.g.* defining how many bytes are expected to be read from the device. ADXL313 functions under the same logic. All communications with it are started with a byte written by the controller. The control byte then designates if the next operation will be a read or write from the controller's perspective. The controller must set bit D_7 to request a read operation or unset it to request a write. Setting bit D_6 tells whether multiple bytes will be transferred, while bits D_5 to D_0 encode the register address for the operation [11].

```c
static uint32_t adxl313_accel(ADXL313State *s, int i)
{
        // on full resolution, the offsets' scale is 4x the acceleration's scale
        int32_t value = s->accels[i] + (4 * s->offsets[i]);
        return *((uint32_t *) &value); // int from raw bit value
}

static uint32_t adxl313_read(SSIPeripheral *dev)
{
        ADXL313State *s = ADXL313(dev);

        switch (s->address) {
        case ADXL313_REG_DEVID0:
                return 0xad;
        case ADXL313_REG_DEVID1:
                return 0x1d;
        case ADXL313_REG_PARTID:
                return 0xcb;
        case ADXL313_REG_POWER_CTL:
                return 0x4b;
        case ADXL313_REG_DATA_FORMAT:
                return 0x0b;
        case ADXL313_REG_BW_RATE:
                return s->bw_rate;
        case ADXL313_REG_DATA_AXIS_X0:
                return adxl313_accel(s, 0) & 0xff;
        case ADXL313_REG_DATA_AXIS_X1:
                return (adxl313_accel(s, 0) >> 8) & 0xff;
        case ADXL313_REG_DATA_AXIS_Y0:
                return adxl313_accel(s, 1) & 0xff;
        case ADXL313_REG_DATA_AXIS_Y1:
                return (adxl313_accel(s, 1) >> 8) & 0xff;
        case ADXL313_REG_DATA_AXIS_Z0:
                return adxl313_accel(s, 2) & 0xff;
        case ADXL313_REG_DATA_AXIS_Z1:
                return (adxl313_accel(s, 2) >> 8) & 0xff;
        case ADXL313_REG_OFS_AXIS_X:
                return s->offsets[0];
        case ADXL313_REG_OFS_AXIS_Y:
                return s->offsets[1];
        case ADXL313_REG_OFS_AXIS_Z:
                return s->offsets[2];
        }

        return 0;
}

static void adxl313_write(SSIPeripheral *dev, uint8_t value)
{
        ADXL313State *s = ADXL313(dev);

        int8_t raw_int = *((int8_t *) &value); // int from raw bit value

        switch (s->address) {
        case ADXL313_REG_BW_RATE:
                s->bw_rate = value;
                break;
        case ADXL313_REG_OFS_AXIS_X:
                s->offsets[0] = raw_int;
                break;
        case ADXL313_REG_OFS_AXIS_Y:
                s->offsets[1] = raw_int;
                break;
        case ADXL313_REG_OFS_AXIS_Z:
                s->offsets[2] = raw_int;
                break;
        }
}
```

Figure 4.5: *Implementation of reads and writes in the emulated ADXL313 accelerometer.*

A device instance stores the current SPI state in the `spi_mode` field, which can be either `ADLX313_SPI_STANDBY`, the initial state waiting for a new control byte transfer, `ADLX313_SPI_READ`, `ADLX313_SPI_WRITE` or `ADLX313_SPI_MULREAD`, for multi-byte reads. Multi-byte writes cause undefined behaviour in the current implementation. Hence, they do not need a state of their own.

Single reads and writes, when D_6 is low, operations are composed of only two bytes. The first transfer carries the already detailed control byte. The second byte contains the operation itself, *e.g.* the value to be written in the device address or the value read from it. Single-byte reading and writing are straightforward to implement. First, the device instance parses the control byte, updates the SPI state, then sets the current address pointer. Next, the transfer function calls either `adxl313_read` or `adxl313_write` to perform the requested operation. Lastly, `spi_mode` is reset to standby so forthcoming transfers may be handled (see Figure 4.4).

When D_6 is high, the operation has no predefined number of transfers to complete because the end of a multi-read operation is only signaled by the end of the transmission, with the inversion of the chip select line. However, the current QEMU SSI implementation activates the CS line at device initialization and keeps it active throughout the device functioning. This renders it impossible to determine the end of a multi-byte write in the emulation code. The effect of this CS policy is that the ADXL313 device instance operates as if all transfers are done within a single transmission. Yet, as the ADXL313 driver does not make writes with more than one byte, this does not hinder the emulation by any means.

When the controller device issues a read command, the `value` argument to the `adxl313_transfer` function is set to zero. With that, the device instance can identify the end of a multi-byte read request by checking out the value that comes with the call to `adxl313_transfer`. While in multi-read mode, read requests are handled as a single read with the difference that the register address pointer is incremented at the end of each transfer. This multi-read feature is convenient for reading several registers in a row. When a non-zero argument is passed to `adxl313_transfer`, the state is set back to `ADLX313_SPI_STANDBY`, and the input is parsed as a normal control byte, defining a new operation (see Figure 4.4).

In the current emulation code, the actual implementation of reads and writes are separated from `adxl313_transfer` into distinct functions in order to enhance code readability (see Figure 4.5). Both functions resolve the operation with simple lookups using switch cases, where the value in the address pointer is tested against the register addresses present in the `REG_` macros. Nevertheless, two particularities found in them are relevant to be discussed.

Firstly, input and output of `transfer` use unsigned integers; however, acceleration and offsets are defined by signed integers. This causes a problem, as the input and output must match, in the bit level, the C programming language implicitly converts the state integers and these types. A cast is applied to the pointer type instead to avoid any modification done to the bits, which bypasses C's type conversion system. Secondly, in ADXL313's full resolution mode, offsets use a scale four times larger than the acceleration's. Hence, their value must be multiplied by four before they are added.

4.5 Emulation as a Development Tool

In summary, all core aspects of an emulated device implementation are now covered. As this chapter goes through many in-depth concepts about QEMU's architecture, the devices inner workings and the used protocol, all of which require a strong background in low-level programming, the chapter itself gets quite dense, the most among the others. It is easy to lose the sense of what this work is aiming to achieve in the midst of it. Given the context, it is best to look at the bigger picture of the state we currently are in and where we are heading.

Using the custom virtual machine set in Chapter 3, it is possible to test the ADXL313 driver present in Chapter 2 against the emulated device detailed in this chapter. Doing such confirms that both implementations are functioning properly, as the Linux driver can run its setup and its device access functionality within the VM communication against the emulated ADXL313 with no errors. Using the sysfs interface of the ADXL313 driver also returns the expected values set by the emulated device. In a sense, this confirms that emulation can actually be used to test devices, as the driver pipeline is being entirely tested. Nevertheless, it does not imply that the effort that needs to be put in developing all this infrastructure is a good investment.

Chapter 5

Final Remarks

The main results of this work include the driver for ADXL313 devices, an ADXL313 emulated device using QEMU and, finally, a brief analysis of how the latter can aid the development and testing of the first.

The driver itself was developed following the open-source development best practices in the Linux kernel. Testing the device properly against multiple hardware configurations, gathering feedback from community reviews, implementing the proposed improvements and properly documenting non-trivial code decisions are among the practices adopted during development. As a result, the driver has been accepted by the IIO subsystem and will be globally available in the Linux kernel from version 5.16 onwards[1][2]. This first version allows users to use ADXL313 devices to read acceleration measurements, set offsets for the acceleration data and change the sampling frequency of the device.

On the other hand, the emulation implementation was developed to be tested against the driver. Its development process was more exploratory than not. Therefore, the final result is not production-ready code, and, consequently, there is no plan to send it out for review in any QEMU community. However, even with the workarounds presented in Chapter 4, it serves its purpose. It can be targeted against the ADXL313 driver implementation without any errors within an emulation environment as it handles all accesses made by the driver.

As stated in Section 4.5, in a sense, the fact that the emulation against driver setup works confirms that it can be used as a bare minimal testing tool. However, it is highly inconvenient for such. As machine emulation in QEMU performs full system emulation, every kernel part has to be run, as with any regular computer. This approach includes processes that waste time and resources but are not relevant to the parts of the code being tested, such as system boot. The interaction is not automatic; hence the actual developer has to go over the sysfs directory and manually test the read and write accesses of the device. Even the actual implementation of the emulated device is a bit unwieldy, as it requires too much knowledge about QEMU intrinsics. This scenario invalidates its use as

[1] https://github.com/torvalds/linux/commit/636d44633039348c955947cee561f372846b478b

[2] https://github.com/torvalds/linux/commit/af1c6b50a2947cee3ffffb32aa8debb100c786bb

a continuous integration tool. Nevertheless, it can still be relevant to assist developers in specific scenarios.

Using QEMU to assist driver development is not a novel idea. Jonathan Cameron, the maintainer of the IIO subsystem, has already declared using a custom QEMU emulation as a target for one of his patch series[3][4]. The work done there consisted mainly of standardizing an old driver's code to the newer conventions of the subsystem. Since Cameron did not have the hardware in his hands for proper testing, he developed an emulation target based on the current driver and refactored it against the emulated device. In a way, his approach is similar to an integration test.

The emulated device implemented in this work had a similar development process as Cameron's. Both targeted an already functioning device driver. Then the working emulation was used for its purpose, in Cameron's case, the refactoring and in mine this brief analysis.

Given the current state, which doubtedly will improve, this is the correct way to use emulation for testing drivers. It is a short-term solution for specific problems, such as refactoring an existing driver while the kernel lacks a proper CI interface. More complex use cases may see QEMU's shortcomings compared to proper testing frameworks.

Building a driver from scratch using emulation is also not a good idea. It would be easy to misinterpret the datasheet and make minor deviations from the real-world device. With a collection of such deviations, the final result might be a device driver that only supports a device's wrong implementation. A driver which supports no device is no more than a useless implementation.

Apart from using it as a reference for building a proper CI for device drivers in Linux, there is not much to go from here with device emulation. It has a purpose in other areas; however, forcing QEMU to be a CI interface does not seem to be a path to go.

Nevertheless, ADLX313 devices have plenty of features that their driver currently lacks support, which can be suggestions for future work. Such features include:

- add proper FIFO support
- add support for interrupt requests for data ready
- add support for handling trespassing the activity and inactivity thresholds
- add support for configurable automatic sleep mode

[3] https://lore.kernel.org/linux-iio/20210614113507.897732-1-jic23@kernel.org/

[4] https://github.com/jic23/qemu/blob/ad7280a-hacks/hw/misc/ad7280a.c

Chapter 6

Personal Appreciation

On a personal note, working on this project has been fairly enjoyable. Before starting it, I had never had experience with either open-source software or device driver development. I had done only minor experiments emulating old gaming consoles. However, in the span of one year, I have now become the sole maintainer of a Linux driver, shipping soon to millions of users. I have completed an entire essay about an experiment with device emulation. Learning about these topics and developing the software present here has been a challenging but gratifying experience.

I am also grateful for this project. It allowed me to get to know and interact with very interesting people who helped me throughout the year. Many of them are developers from different parts of the world (England, Romania, India, and The United States of America), leading to a great cultural exchange between their peers. It was a privilege to have such a great experience interacting with people considering the current worldwide pandemic.

References

[1] Markus Armbruster. *QEMU's device model qdev: Where do we go from here?* 2011. URL: https://www.youtube.com/watch?v=Cpt5Zqs_Iq0%5C&t=128s (visited on 12/05/2021) (cit. on p. 29).

[2] Markus Armbruster. *QEMU's new device model qdev.* 2010. URL: https://www.linux-kvm.org/images/f/fe/2010-forum-armbru-qdev.pdf (visited on 11/11/2021) (cit. on pp. 20, 21, 29).

[3] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track.* Vol. 41. Califor-nia, USA. 2005, p. 46 (cit. on p. 19).

[4] Arpit Bhayani. *Powering inheritance in C using structure composition.* 2020. URL: https://www.codementor.io/@arpitbhayani/powering-inheritance-in-c-using-structure-composition-176sygr724 (visited on 12/11/2021) (cit. on p. 32).

[5] Hossein Bidgoli and Andrew Prestage. "Operating Systems". In: *Encyclopedia of Information Systems.* Ed. by Hossein Bidgoli. New York: Elsevier, 2003, pp. 377–390. ISBN: 978-0-12-227240-0. DOI: https://doi.org/10.1016/B0-12-227240-4/00126-X. URL: https://www.sciencedirect.com/science/article/pii/B012227240400126X (cit. on pp. 22, 23).

[6] Jonathan Cameron. *10 Years of the Industrial I/O Kernel Subsystem.* United Kingdom, Oct. 2018. URL: https://www.youtube.com/watch?v=644oH1FXdtE (visited on 09/18/2021) (cit. on p. 8).

[7] Andy Chou et al. "An empirical study of operating systems errors". In: *Proceedings of the eighteenth ACM symposium on Operating systems principles.* 2001, pp. 73–88 (cit. on p. 2).

[8] Jonathan Corbet. *An alternative device-tree source language.* 2017. URL: https://lwn.net/Articles/730217/ (visited on 10/30/2021) (cit. on p. 18).

[9] Jonathan Corbet. *Kernel development.* 2001. URL: https://lwn.net/2001/0704/kernel.php3 (visited on 10/24/2021) (cit. on p. 18).

[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005. ISBN: 0596005903 (cit. on pp. 9, 13).

[11] Analog Devices. *ADXL313 Data Sheet.* 2019. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL313.pdf (visited on 12/11/2021) (cit. on p. 34).

[12] QEMU documentation. *'virt' generic virtual platform (virt).* 2021. URL: https://github.com/qemu/qemu/blob/master/docs/system/arm/virt.rst (visited on 11/25/2021) (cit. on pp. 19, 20, 22, 25).

[13] QEMU documentation. *qdev device use*. 2021. URL: https://github.com/qemu/qemu/blob/master/docs/qdev-device-use.txt (visited on 11/10/2021) (cit. on p. 21).

[14] QEMU documentation. *The memory API*. 2021. URL: https://qemu.readthedocs.io/en/stable/devel/memory.html (visited on 12/09/2021) (cit. on p. 22).

[15] QEMU documentation. *The QEMU Object Model (QOM)*. 2021. URL: https://qemu.readthedocs.io/en/stable/devel/qom.html (visited on 12/07/2021) (cit. on pp. 30, 31).

[16] The Linux Kernel documentation. *Core elements*. 2021. URL: https://www.kernel.org/doc/html/latest/driver-api/iio/core.html (visited on 11/21/2021) (cit. on p. 8).

[17] Pavel Dovgalyuk et al. "QEMU-based framework for non-intrusive virtual machine instrumentation and introspection". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 944–948 (cit. on p. 19).

[18] Alexander S. Gillis. *Device Driver*. 2020. URL: https://searchenterprisedesktop.techtarget.com/definition/device-driver (visited on 10/09/2021) (cit. on p. 1).

[19] Andrea Höller et al. "QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks". In: *2015 Euromicro Conference on Digital System Design*. IEEE. 2015, pp. 530–533 (cit. on p. 19).

[20] Greg Kroah-Hartman Jonathan Corbet. *2017 Linux Kernel Report*. 2017. URL: https://www.linuxfoundation.org/wp-content/uploads/linux-kernel-report-2017.pdf (visited on 07/09/2021) (cit. on p. 1).

[21] Frederic Leens. "An introduction to I^{2}C and SPI protocols". In: *IEEE Instrumentation Measurement Magazine* 12.1 (2009), pp. 8–13. DOI: 10.1109/MIM.2009.4762946 (cit. on p. 33).

[22] Grant Likely. *Linux and the Devicetree*. URL: https://github.com/torvalds/linux/blob/master/Documentation/devicetree/usage-model.rst (visited on 12/05/2021) (cit. on p. 18).

[23] Linaro Ltd Linaro Ltd. *Devicetree Specification*. Feb. 2020. URL: https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.3 (visited on 12/05/2021) (cit. on p. 18).

[24] Yan Luo et al. "Qsim: Framework for cycle-accurate simulation on out-of-order processors based on QEMU". In: *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*. IEEE. 2012, pp. 1010–1015 (cit. on p. 19).

[25] John Madieu. *Linux Device Drivers Development: Develop customized drivers for embedded Linux*. Packt Publishing Ltd, 2017, pp. 201–216 (cit. on p. 6).

[26] John Madieu. *Mastering Linux Device Driver Development*. 1st. UK: Packt Publishing Ltd., 2020. ISBN: 978-1-78934-204-8 (cit. on pp. 6, 7, 22).

[27] S. O'Dea. *Mobile operating systems' market share worldwide from January 2012 to June 2021*. 2021. URL: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (visited on 07/09/2021) (cit. on p. 1).

[28] Marta Rybczyńska. *Device-tree schemas*. 2018. URL: https://lwn.net/Articles/771621/ (visited on 10/30/2021) (cit. on p. 18).

[29] Marcelo Schmitt. "Linux Device Driver Development: a report from the trenches". São Paulo: Universisade de São Paulo, Dec. 2019 (cit. on p. 5).

[30] TOP500 TEAM. *Operating System Family / Linux*. 2021. URL: https://www.top500.org/statistics/details/osfam/1/ (visited on 07/09/2021) (cit. on p. 1).