RASPBERRY PI 4 BUZZER DRIVER

Daniel Loaiza Noreña
Systems Engineering
Universidad de Antioquia
daniel.loaizan@udea.edu.co

Anderson Saldarriaga Castaño
Systems Engineering
Universidad de Antioquia
anderson.saldarriaga@udea.edu.co

Joan Manuel Muñoz Monroy
Systems Engineering
Universidad de Antioquia
joan.munoz1@udea.edu.co

Abstract—Device drivers are essential for communication between the operating system and hardware or software, acting as intermediaries that translate system instructions into hardware actions. Although many modern devices come with preinstalled drivers, understanding how these drivers are created is vital, especially for complex or undocumented hardware. This is crucial for developers designing custom devices, as specific drivers ensure optimal operation. This article examines the development of a driver for a buzzer on a Raspberry Pi, highlighting the importance of customizing drivers. Through this process, one learns to interact directly with the hardware, handle interrupts, and use specific control registers. The Raspberry Pi, popular among enthusiasts and developers, provides an ideal environment for this educational experience.

Index Terms—Raspberry Pi, buzzer driver, GPIO, Python, sound alerts.

I. INTRODUCTION

Device drivers are fundamental to enable hardware devices to function in a system. Nowadays, both developers and users are accustomed to drivers being automatically installed on their devices, as with a new mechanical keyboard or a high-precision mouse. But what about more complex devices? Not necessarily because they are complicated, but because they are either so old that no documentation is available or so novel that they tend to have bugs. For this reason, it is crucial to understand the process of creating drivers and how they function. As developers, we continuously create new devices, each with its own characteristics and mechanisms. If we want these devices to be compatible with our applications, it is beneficial to know how their drivers work and how we can adapt them to our specific needs. Having this knowledge does not mean that we can automatically solve any problem that arises during development, but understanding the process will make it more manageable and comprehensible. Furthermore, working with electronic devices like the Raspberry Pi opens the door to learning about many technological topics, such as circuit operation, and to gaining a deeper understanding of each hardware component by working with them in detail. It also teaches us how technology operates on a small scale. This latter aspect is especially important as it supports the famous quote from the movie about T.E. Lawrence: ”Big things have small beginnings,” which we can interpret as meaning that to create robust and functional software, it is essential to understand the basics.

II. THEORETICAL FRAMEWORK

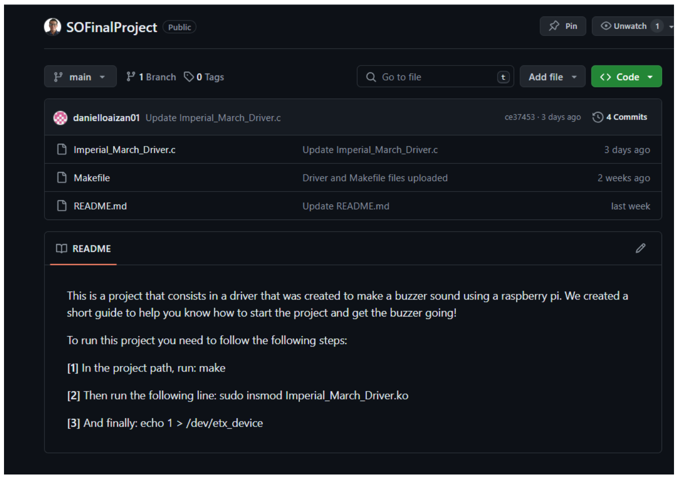
A. Raspberry Pi

The Raspberry Pi is a compact and affordable microcomputer running a Linux-based operating system, capable of performing multiple tasks typical of a desktop PC. For this project, the Raspberry Pi Model B model was used, with 512 MB of RAM, produced by Adafruit (see Figure 1).

Figure 1: Raspberry Pi 4

B. Device Drivers

”A driver is a software component that allows the operating system and a device to communicate.” In simple terms, it acts as a link between hardware and software. In this case, a developer must program the driver to ensure the correct operation of the hardware device, this was developed and track in a GitHub repository where the code was managed (see Figure 2).



danielloaizan01 Update Imperial_March_Driver.c ce37453 · 3 days ago 4 Commits

Imperial_March_Driver.c	Update Imperial_March_Driver.c	3 days ago
Makefile	Driver and Makefile files uploaded	2 weeks ago
README.md	Update README.md	last week

README

This is a project that consists in a driver that was created to make a buzzer sound using a raspberry pi. We created a short guide to help you know how to start the project and get the buzzer going!

To run this project you need to follow the following steps:

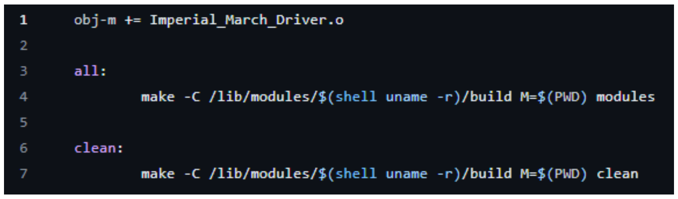
[1] In the project path, run: make

[2] Then run the following line: sudo insmod Imperial_March_Driver.ko

[3] And finally: echo 1 > /dev/etx_device

Figure 2: Repository

It is also important to address that the driver was created using a ”Makefile” that managed the creation and implementation of the driver in the Operative System (see Figure 3).



```
obj-m += Imperial_March_Driver.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Figure 3: Makefile

C. Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is a technique used to generate digital signals that emit analog signals (see Figure 4). This is achieved by varying the duration of time that a signal remains in the high (active) state within a complete cycle. The ability to modify the pulse width allows controlling devices such as lights, motors, or speakers, enabling visual effects, changes in speed, or sound tones efficiently.

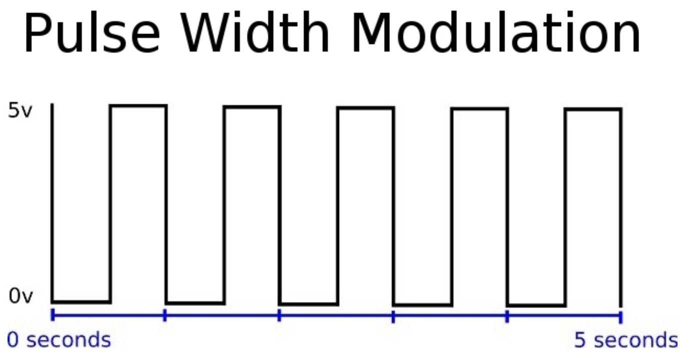


Figure 4: Buzzer wiring

D. General Purpose Input/Output (GPIO) Pins

General Purpose Input/Output (GPIO) pins are a series of pins on microcontrollers or computers like the Raspberry Pi, which can be configured as digital inputs or outputs (see Figure 5). These pins allow interaction with external devices, either by reading sensor signals or generating electrical signals to control devices such as LEDs.

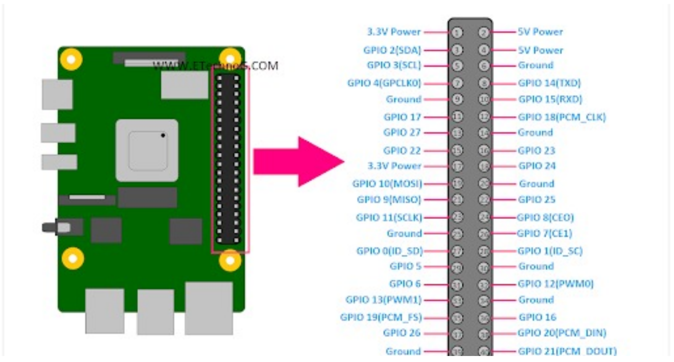


Figure 5: Raspberry Pi 4 Pins

E. Buzzer

A buzzer is an electronic component designed to produce audible sounds and it’s connected via wires with the hardware component (see Figure 6). There are two main types of buzzers: passive and active.

- Passive buzzer: Produces a constant sound when an electrical current is applied to it, with no internal control capability.
- Active buzzer: Includes an internal control circuit that allows generating a variety of sounds, with the ability

to adjust the frequency and duration of the sound as required.

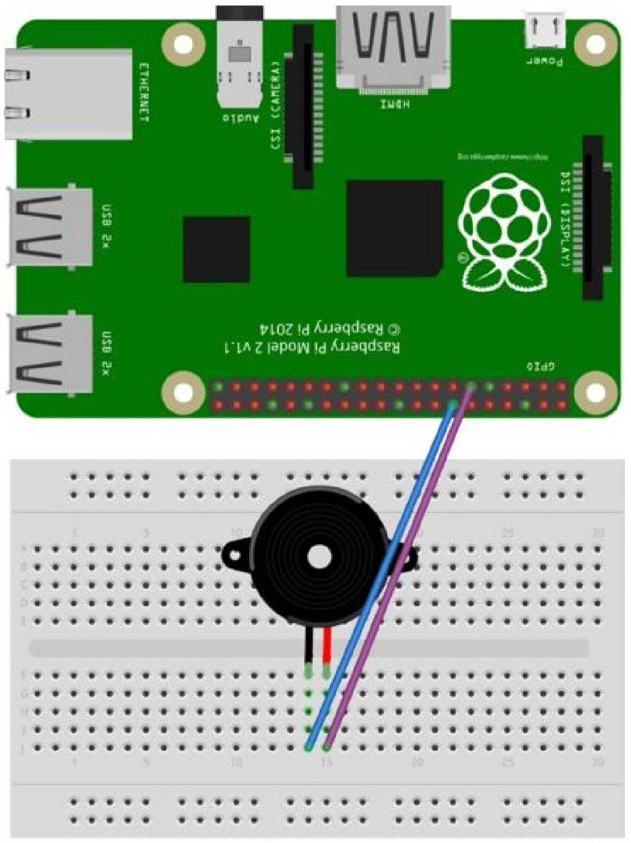


Figure 6: Buzzer connection

III. IMPLEMENTATION

The configuration and connection of the buzzer to the Raspberry Pi proved to be straightforward, following the instructions we had in the manuals we found (see Figure 7). Our main goal was to compile a device driver within the operating system kernel. Once the successful booting of the system on the Raspberry Pi was achieved, a Python program was used to evaluate the capabilities of the buzzer and understand its operation. However, if faster performance or greater control over the hardware is required, as is the case, we opted to migrate to a lower-level programming language, such as C. By rewriting the program in C, direct access to the system memory and hardware registers can be achieved, allowing for more precise control of the buzzer and more efficient performance . Furthermore, C code can be compiled into highly optimized binaries that run directly on the hardware, which can be especially beneficial in embedded systems like the Raspberry Pi. Ultimately, the process involved creating a custom device driver for the buzzer, integrated into the Raspberry Pi's operating system kernel (see Figure 8), then developing user software to interact with the buzzer, initially in Python and later migrating to C for improved performance and control. This approach provided a solid foundation for developing more advanced applications that leverage the capabilities of the buzzer connected to the Raspberry Pi.

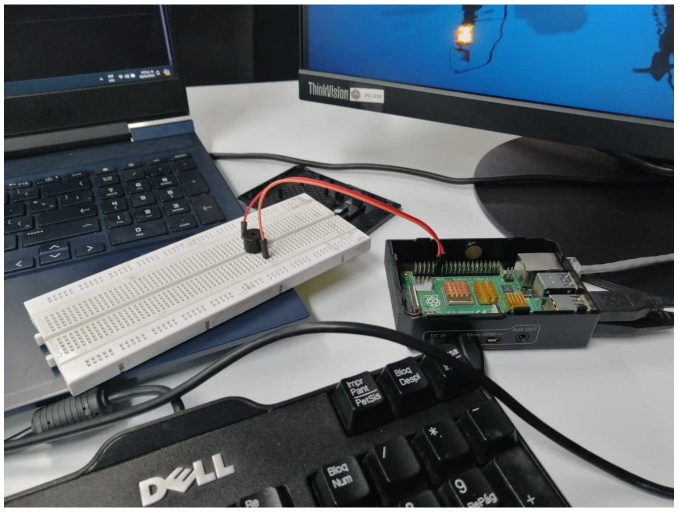
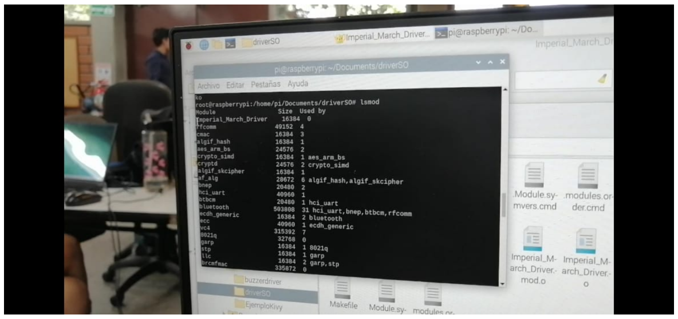


Figure 7: Buzzer wiring



```
ko
root@raspberrypi:/home/pi/Documents/driverSO# lsmod
Module                  Size  Used by
Imperial_March_Driver    16384  0
rfcomm                 49152  4
cmac                   16384  3
algif_hash             16384  1
aes_arm_bs             24576  2
crypto_simd            16384  1 aes_arm_bs
cryptd                 24576  2 crypto_simd
algif_skcipher         16384  1
af_alg                 28672  6 algif_hash,algif_skcipher
bnep                   20480  2
hci_uart               40960  1
btbcm                  20480  1 hci_uart
bluetooth             503808  31 hci_uart,bnep,btbcm,rfcomm
ecdh_generic           16384  2 bluetooth
ecc                    40960  1 ecdh_generic
vc4                   315392  7
8021q                  32768  0
garp                   16384  1 8021q
stp                    16384  1 garp
llc                    16384  2 garp,stp
brcmfmac              335872  0
```

Figure 8: Driver implementation

IV. RESULTS

Upon completing the solution and mounting the driver on the operating system (see Figure 9), the buzzer produced the assigned sounds as specific instructions were executed. However, during the music playback process, certain difficulties related to the modulation of the buzzer frequency were identified. Some tones were not recognized correctly because the buzzer generated them too quickly, causing distortion in the playback of certain notes, especially those requiring a specific duration or precise frequency.

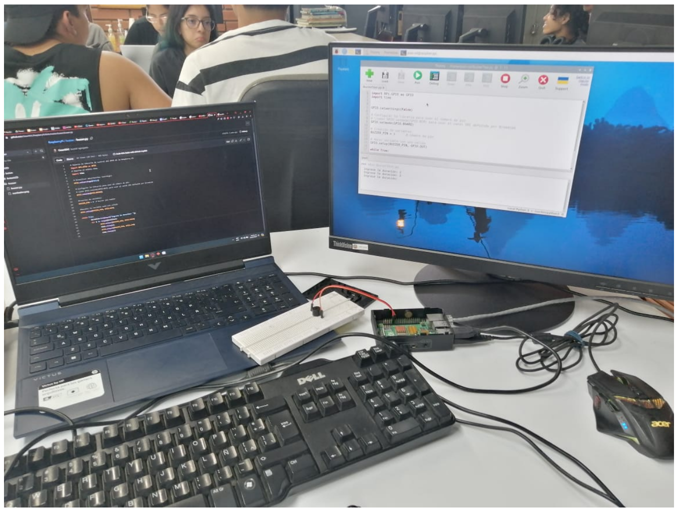


Figure 9: Complete hardware and software connection

V. CONCLUSION

- The successful implementation of the buzzer driver on the Raspberry Pi demonstrated the practical application of the acquired knowledge. This project not only served as an exercise in driver development but also highlighted the relevance of understanding hardware at a fundamental level. Additionally, the difficulties encountered during music playback underscore the importance of considering hardware limitations and the need to adjust the configuration appropriately to ensure optimal performance.
- This project equipped the team with practical skills essential for embedded systems development. The hands-on experience of connecting, configuring, and programming a hardware component from scratch is invaluable for those looking to work in fields related to embedded systems or IoT devices.
- The transition from Python to C for the buzzer driver highlighted the necessity of efficient coding practices in embedded systems. Writing the driver in C allowed for better performance and more precise control over the hardware, demonstrating the impact of language choice on system efficiency and responsiveness.
- Despite the challenges encountered, music playback was successfully achieved using the buzzer connected to the Raspberry Pi. This demonstrates the system's ability to meet the basic requirements of sound reproduction and validates the effectiveness of the implemented solution. Ultimately, this project highlighted both the achievements made and potential areas for improvement, offering valuable lessons on device driver development and hardware integration in embedded systems like the Raspberry Pi.
- Throughout the development process, various challenges related to timing, signal integrity, and hardware-software synchronization were encountered. These issues underscored the significance of robust error handling and debugging techniques in ensuring the reliability and stability of device drivers.
- The development of a custom driver for the buzzer on the Raspberry Pi deepened the understanding of direct hardware interaction. This project emphasized the importance of learning how to manipulate hardware registers and handle interrupts, providing insights crucial for advanced hardware interfacing and low-level programming.
- The project demonstrated how a single component, like a buzzer, can be integrated into a larger system to provide meaningful feedback or alerts. This integration is crucial for designing comprehensive systems where multiple components work together seamlessly to achieve the desired functionality.

VI. REFERENCES

[1] 330ohms, "Cómo conectar un buzzer activo a raspberry pi," 330ohms, https://blog.330ohms.com/2020/06/18/como-conectar-un-buzzer-activo-a-raspberry-pi/ (accessed Jun. 3, 2024).

[2] Uaeh, "Buzzer," Buzzer | Arduino, http://ceca.uaeh.edu.mx/informatica/oas$_final/red4_arduino/buzzer.$ $text = Tipos$

[3] Raspberry Pi, https://www.raspberrypi.com/ (accessed Jun. 4, 2024).