Diseño de un Controlador Horario Personalizado para Empleados
Designing a Customized Employee Time Tracker

Trabajo de Fin de Grado
Curso 2023–2024

Autor
Santiago Elias Rabbia

Director
Gonzalo Méndez Pozo

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Diseño de un Controlador Horario Personalizado para Empleados
Designing a Customized Employee Time Tracker

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Santiago Elias Rabbia

Director
Gonzalo Méndez Pozo

Convocatoria: Junio 2024

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

27 de Junio de 2024

Resumen

Diseño de un Controlador Horario Personalizado para Empleados

Cambios en las regulaciones modificaron el procedimiento por el cual los empleados fichan al entrar a sus trabajos en una compañía de gestión de instalaciones, volviendo obsoletos muchos dispositivos existentes. Se necesitaba una respuesta rápida, por lo que se desarrolló un dispositivo de fichaje rentable que utiliza microcontroladores, un PCB personalizado y una caja impresa en 3D.

Este dispositivo IoT debe ser responsivo, intuitivo de usar y ofrecer un servicio casi ininterrumpido, enviando información a través de WiFi o datos celulares. El intercambio de datos se realiza por medio de API alojada en la nube, desarrollada usando Java Spring, que pueden ser montada como un contenedor de Docker, o como una Google Cloud Function.

Palabras clave

fichador, microcontrolador, pcb, impresión 3D, cloud API, IoT, programación asíncrona, Docker, GCP, Java Spring

Abstract

Designing a Customized Employee Time Tracker

Regulatory changes have prompted a shift in employee clock-in procedures at a facility management company, rendering many existing devices obsolete. A swift response was necessary, leading to the development of a cost-effective clocking device utilizing microcontrollers, a custom PCB, and a 3D-printed case.

This IoT device has to be responsive, intuitive to use and offer close-to uninterruped service, sending information through WiFi or cellular data. Data exchange is facilitated through a cloud-hosted API developed using Java Spring, hostable as a Docker container or as a Google Cloud Function.

Keywords

clocking device, microcontroller, pcb, 3D print, cloud API, IoT, asynchronous programming, Docker, GCP, Java Spring

Contents

List of figures

Chapter 1

Introduction and objectives

Recent legislative changes introduced by both the Spanish Data Protection Agency and the national government have mandated alterations to employee clocking procedures. Notably, Spain now requires employees to clock in upon arrival at work, necessitating the implementation of reliable time-tracking systems [5]. Additionally, the Spanish Data Protection Agency has banned the use of clocking devices employing biometric identification methods such as fingerprint scanning or facial recognition [3].

A particular company manages many thousands of employees throughout Spain, and it was of utmost importance to expeditely replace the non-compliant devices for different ones, as many companies were already being imposed hefty fines, from tens to hundreds of thousands of Euros, for not meeting the new regulatory requirements.

The company already had providers for many types of clocking devices, and many of them were compliant, but were priced in the hundreds of Euros each. Now, faced with having to replace them, the prospect of replacing *thousands* of units at considerable expense loomed large. Moreover, outsourcing these devices often meant committing to complex time-tracking ecosystems, adding further complications and increasing the complexity of the data the company manages.

These issues were brought to light to the innovation team at the company, and were then tasked with bringing a solution to market.

1.1. Objectives

Given the preceding challenges and considerations, the development of a cost-effective device and the establishment of a cloud infrastructure were deemed necessary.

The objectives of this project are as follows:

- **Design and Prototyping**: Design a device utilizing microcontrollers and additional electronic modules, such as LCD displays and NFC readers, to fulfill the specified requirements.

- **PCB Design**: Develop a Printed Circuit Board (PCB) layout to facilitate easy interconnection of the various electronic modules used in the device, ensuring efficient and reliable performance.
- **API Development**: Create an Application Programming Interface (API) capable of deployment as a Docker container or Google Cloud Function, enabling seamless communication between the device and cloud-based services.
- **Microcontroller Research**: Investigate the constraints and capabilities of microcontrollers, particularly RP2040-based microcontrollers, to inform design decisions and optimize performance.
- **3D Printing and Modeling**: Explore the fundamentals of 3D printing and 3D modeling necessary for designing a suitable enclosure for the device. Additionally, provide an overview and comparison of various materials suitable for enclosure fabrication, considering factors such as durability, cost, and aesthetic appeal.
- **Cost Analysis**: Conduct a brief cost comparison between the new solution and the previous system, demonstrating the financial benefits of the new design.

Chapter 2

Breadboard Prototype

The first phase of developing the new clocking device involved thorough research into available components and microcontrollers in the market. Factors such as cost, functionality, and potential drawbacks were carefully evaluated to inform the selection process. Subsequently, selected components were tested by constructing a basic prototype on a breadboard to assess functionality and performance. Demonstrating the viability of the project at this stage was crucial for ensuring its continued development and success.

2.1. Hardware selection

Firstly, considering that the development was urgent, it would only be possible to use an already made controller. There were two routes to take:

- Using a **single-board computer**.
- Using a **microcontroller**.

The team's familiarity with *Raspberry Pi* products and their reputation for reliability led to the decision to utilize their offerings for the project. Given the lightweight processing requirements, the *Raspberry Pi Zero 2 W*, a cost-effective single-board computer with built-in WiFi capabilities, emerged as a suitable option. Alternatively, the *Raspberry Pi Pico W*, a microcontroller, presented another viable choice.

However, it's important to note that while the *Pi Zero* offers more features and functionality, it comes at a higher cost compared to the *Pi Pico*. In fact, it costs almost three times as much. Therefore, careful consideration is warranted to determine whether the additional expense justifies the benefits.

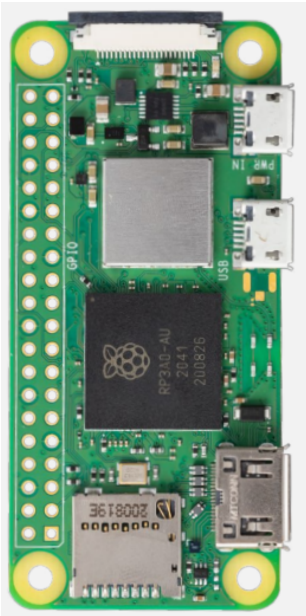

Figure 2.1: Raspberry Pi Zero 2 W. Image by Raspberry Pi licensed under CC BY-SA 4.0

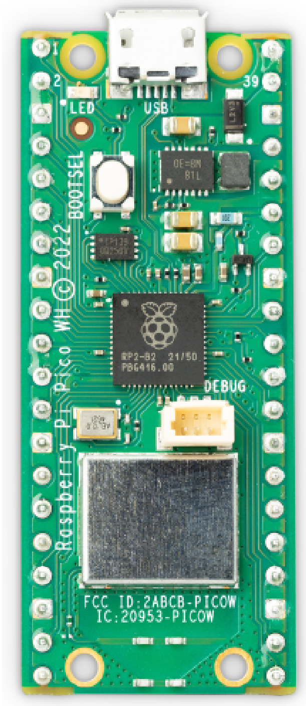

Figure 2.2: Raspberry Pi Pico WH. Image by Raspberry Pi licensed under CC BY-SA 4.0

2.1.1. Single-board Computers

A single-board computer (*SBC*) is a complete computer built on a single circuit board. It integrates all the necessary components required for a functional computer system, including a central processing unit (CPU), memory (RAM), storage (usually in the form of a MicroSD card), input/output ports, and sometimes additional features such as networking capabilities (e.g., Ethernet or WiFi), audio/video output, GPIO (General Purpose Input/Output) pins for connecting external devices, and even USB ports [18].

SBCs are designed to be compact and efficient, which is the case of the *Pi Zero*, measuring about 65mm by 30mm while drawing about one Watt of power. Additionally, SBCs can run an operating system, such as Ubuntu.

This ability to run an entire operating system significantly enhances their versatility compared to microcontrollers out-of-the-box. Many peripheral devices can simply be plugged into a USB port and function seamlessly without requiring additional configuration. For instance, a 3G SIM card adapter, which will be necessary for future stages of the project, can be effortlessly integrated into the system, letting the operating system take charge of the communication with the mobile network, abstracting all these problems from the programmer.

2.1.2. Microcontrollers

A microcontroller is a compact integrated circuit (IC) that contains a central processing unit (CPU), memory (both volatile RAM and non-volatile flash memory), input/output peripherals (such as digital and analog I/O pins), and various other hardware components necessary for interfacing with external devices. Unlike single-board computers, microcontrollers are typically designed for specific tasks and embedded applications, often with real-time requirements [18].

One key characteristic of microcontrollers is their ability to execute dedicated firmware or software code stored in their internal memory. This code typically controls the behavior of the microcontroller, processes inputs from sensors or other external devices, and generates outputs to control actuators or display information.

The *Raspberry Pi Pico W* is a development board that utilizes the *RP2040* microcontroller chip. This board offers a range of features beyond its microcontroller, including onboard flash memory for program storage, versatile GPIO pins for interfacing with external devices, built-in USB connectivity for programming and power supply, and WiFi connectivity.

In comparison to single-board computers, the *Pi Pico* does not have an operating system. Instead, firmware can be loaded onto it, and it is this firmware that provides functionality to the board. This firmware allows the programmer to use programming languages such as *C* or *MicroPython*, which then control the microcontroller's behavior and interactions with external devices.

The absence of an operating system reduces the overhead associated with system management and resource allocation, resulting in faster boot times and improved reliability for time-critical tasks.

Taking into account our previously established requirements, which prioritize minimal points of failure, low computational demands, and cost-effectiveness, the logical preference leans towards the utilization of a microcontroller. For instance, single-board computers often rely on SD cards for storage, which can be prone to failure after prolonged use due to factors such as wear and tear or data corruption. In contrast, microcontrollers typically have simpler storage mechanisms, such as onboard flash memory, which are less susceptible to such issues. Thus, lower operating costs.

As a conclusion, a **microcontroller will be used**, and in particular, the ***Raspberry Pi Pico W***.

2.1.3. Electronic Modules

Now that the microcontroller has been selected, additional components are necessary to meet the project's requirements. Specifically, the device must integrate an NFC reader and an LCD screen. Additionally, other peripherals, such as a buzzer or LED, may also be evaluated.

Available buses need to be taken into account to then choose modules accordingly.

Relying solely on the datasheet alone is not enough to determine the buses that are usable concurrently, since if two different buses use the same pin, then they cannot be used simultaneously. Referring to the pinout diagram of the *Raspberry Pi Pico W* in Figure 2.3, available buses can be identified:

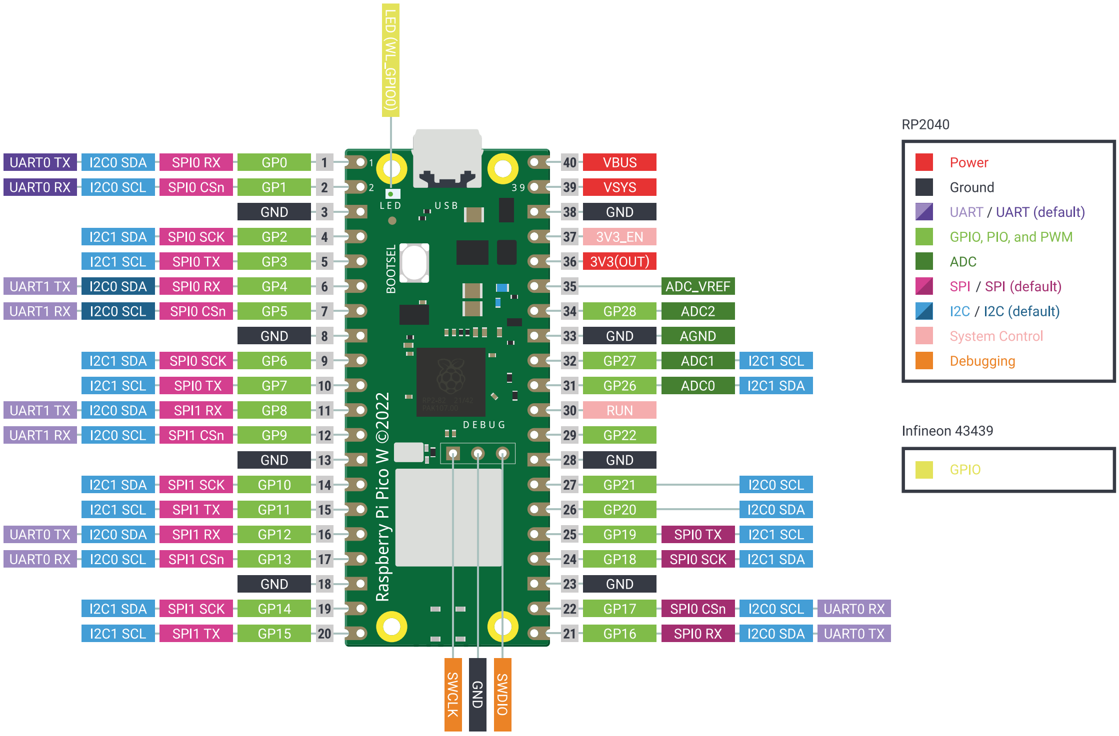

Figure 2.3: Raspberry Pi Pico W's Pinout. Image by Raspberry Pi licensed under CC BY-SA 4.0

For example, each UART bus conflicts with I2C buses. Therefore, when selecting components, it's crucial to ensure compatibility with this layout [1].

NFC Module

Various NFC boards are available on the market. Ideally, the desired NFC board would offer extended range, compact form-factor, low power consumption, and support for multiple communication protocols such as UART, I2C, SPI, among others.

The *PN532* chipset produced by *NXP* offers all three types of buses mentioned before, that is: High Speed UART, I2C and SPI. Numerous boards utilizing this chipset are available on the market, with *ElecHouse*'s offering standing out for its excellent form-factor (about 4cm×4cm) and impressive range [11]. This specific board has been extensively replicated and is accessible at a low cost, and various providers offer open-source code for controlling it.

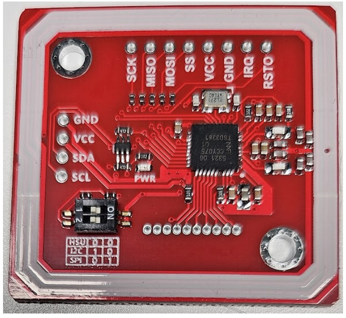

Figure 2.4: PN532 board

LCD Module

Initially, the idea of utilizing a black and white OLED display seemed appealing. However, as the size increased, so did the price, and even then, the displays were still too small. Consequently, the most practical decision was to adhere to LCD technology.

Upon researching LCD modules, the *LCD1602* module emerged as a suitable choice. It offers sufficient size, accommodating up to 16 characters per row across 2 rows, and provides satisfactory contrast. Additionally, it is available with I2C adapters for simplified control, requiring fewer pins on the microcontroller.

Figure 2.5: LCD1602 board

Buzzer

Beyond just relying on visual cues displayed on the screen to convey the device's status, it's essential to enhance the user experience by incorporating auditory feedback. When employees clock in to work, providing a distinct sound signal from a buzzer ensures they receive immediate confirmation of their action.

There are two types of buzzer [6]:

- **Active buzzers**: they incorporate an internal oscillator circuit that generates the sound signal when voltage is applied. They are self-contained and do not require external circuitry to produce sound. They are commonly used in applications where simplicity and ease of use are prioritized, as they can be directly connected to a power source to emit sound.

- **Passive buzzers**: they require an external oscillating circuit to produce sound. An alternating current is applied to create vibrations that produce sound waves. They offer more flexibility in sound frequency and intensity control but require additional circuitry for operation.

Due to the project's time constraints and the preference for simplicity, **active buzzers will be employed**. Additionally, since a single sound frequency suffices for the intended use, active buzzers are well-suited for the task.

Cellular Connectivity Module

In certain locations, access to WiFi networks may be limited or unavailable altogether. Even in areas where WiFi is accessible, signal strength may vary, leading to instances of poor connectivity. Consequently, it becomes imperative to incorporate cellular connectivity options to ensure internet access across diverse environments and conditions. By integrating cellular connectivity capabilities into the system, users can rely on alternative means of internet access, mitigating the limitations posed by WiFi availability and signal quality fluctuations.

Considering the time limitations, it would be benefitial to opt for a ready-made solution. However, it's worth noting that the Raspberry Pi Pico, being a microcontroller, cannot simply utilize a USB dongle with a SIM card adapter for cellular connectivity. A more intricate solution is necessary. Fortunately, there exists a module specifically designed for compatibility with the Pi Pico, developed by *Waveshare*. This module integrates the *SIM7020E* module, manufactured by *SIMCOM*, a prominent provider of wireless modules and electronics.

The *SIM7020E* module, known for its affordability and low power consumption, is specifically designed for NB-IoT[1] applications, making it an optimal choice for M2M[2] communication scenarios. Integrated into the *Waveshare* board, this module establishes communication with the Pi Pico through UART, using *AT*[3] commands for control and data exchange [23].

However, a significant tradeoff exists as this module only supports NarrowBand IoT. NB-IoT operates within the LTE (Long-Term Evolution) spectrum and is tailored for narrow bandwidths, making it ideal for low-power, wide-area IoT applications. While NB-IoT typically utilizes the 4G LTE spectrum, it operates at reduced data rates compared to conventional LTE, facilitating efficient communication with IoT devices while minimizing power consumption and costs.

[1] NB-IoT, short for Narrowband Internet of Things, is a low-power cellular technology designed for efficient communication between IoT devices and networks.

[2] M2M stands for Machine-to-Machine, and encompasses a wide range of applications where devices or machines communicate and exchange data without human intervention.

[3] AT commands are a standardized set of instructions used to communicate with and control modems and other serial devices.

Nonetheless, this limitation poses challenges as the range constraints of LTE networks persist, potentially leading to signal issues in underground facilities. In an ideal scenario, a module supporting 2G and NB-IoT would be preferable for the project. However, such modules are not readily available on the market for seamless integration with the Pi Pico. While *SIMCOM* does offer several variants of these modules with diverse capabilities, developing a custom solution would necessitate significant investment in terms of both time and resources.

However, the downside of this approach is that the module requires soldering of fourteen pins (the ones whose text is highlighted in white in Fig. 2.7), introducing an additional layer of complexity to the device assembly process. Furthermore, it is important to note the *stacking headers* featured in Figures 2.6 and 2.7, which consist of female headers with extended male pins. These pins are designed to penetrate the PCB and connect to another female header underneath. While this setup may appear standard, improper soldering, with excess tin reaching too high on the male part of the header, can lead to faulty connections with the PCB or breadboard. Therefore, if the team intends to assemble the device on-site, careful soldering is a must.

Figure 2.6: Waveshare's SIM7020E module, top view

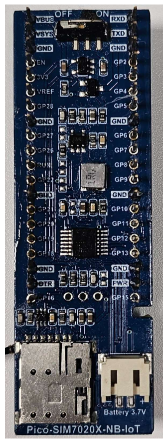

Figure 2.7: Waveshare's SIM7020E module, bottom view

Buses

As previously mentioned, various buses are available for interconnecting components. While the LCD is already set up for I2C, the NFC reader offers more flexibility in bus selection. However, to maintain simplicity, the NFC reader will also utilize the I2C bus.

I2C, short for Inter-Integrated Circuit, is a serial communication protocol commonly utilized for connecting microcontrollers and peripheral devices. It operates using two wires: *SDA* (Serial Data) and *SCL* (Serial Clock). Devices connected to the bus are addressed individually by unique addresses, and communication follows a master-slave configuration. The master device initiates communication and controls the bus, while one or more slave devices respond to commands. Data is transferred sequentially, with the master device generating clock pulses to synchronize communication [19].

However, I2C does have limitations. It operates at relatively low speeds compared to other protocols, which can impact performance in applications requiring high data transfer rates. The length of the I2C bus is also limited due to signal integrity issues, typically to a few meters, which may restrict the physical layout of devices in larger systems. In any case, none of these limitations affect this project's use case.

The *Pi Pico* features two I2C buses, providing an additional bus beyond what is required. It's important to mention that the two modules can share the same I2C bus since they have different I2C addresses. For instance, the PN532 always uses the *0x48* address, which cannot be modified. Consequently, two PN532 readers cannot share the same bus and must be placed separately. However, this is not an issue for the LCD as it uses a different address.

Note: While the PN532 produced by *NXP* does allow for the modification of the I2C address [20], it is conventionally maintained at *0x48* [21].

Matrix Keypad

A matrix keyboard offers an alternative means for employees to clock in or out by entering a numeric code, which can be particularly useful if an employee forgets or misplaces their card. The working principle of matrix keyboards is straightforward: they consist of intersecting rows and columns. Each key on the keyboard is positioned at the intersection of a row and a column. When a key is pressed, it creates a connection between a specific row and column, resulting in a unique electrical signal that can be interpreted by the microcontroller.

Figure 2.8: Matrix keypad

Matrix keyboards utilize a scanning technique to detect key presses. The microcontroller sequentially activates each row, while monitoring the columns for any signals. If a signal is detected, the microcontroller identifies the corresponding key based on the activated row and column, allowing it to register the key press.

One important consideration when using matrix keyboards is the potential for key ghosting or masking. Ghosting occurs when multiple keys are pressed simultaneously, causing the keyboard to register unintended key presses. Masking, on the other hand, happens when certain key combinations prevent other keys from registering. To mitigate these issues, careful design and debounce techniques must be employed to ensure reliable key detection.

During testing, it was observed that with sufficiently fast scanning and debouncing, the occurrence of ghosting or masking on the keyboard would likely not present a problem in practical usage scenarios. This assessment stems from the understanding that the likelihood of multiple key presses happening simultaneously on a wall-mounted device, such as the one being developed, is minimal.

Another challenge arises from the substantial use of GPIO pins on the microcontroller. The 4×4 keyboard depicted in Figure 2.8, consisting of 4 rows and 4 columns, requires 8 GPIO pins for operation. Considering that the Pi Pico offers a total of 29 GPIO pins, this allocation accounts for nearly 30 percent of available pins.

2.2. Prototype Assembly

Once the components have been selected and basic schematics drawn, the assembly process can commence. All components can be interconnected on a breadboard using jumper wires. It is important to highlight that the Raspberry Pi Pico WH comes with pre-soldered headers, facilitating easy integration for testing purposes. Only minimal soldering is required for certain components, such as the four pins on the NFC module and the fourteen pins for the Waveshare module.

Many of the components necessary for this prototyping phase can be obtained by purchasing a basic electronics kit, which typically includes a breadboard and jumper wires, and other basic components such as a matrix keyboard and buzzer.

In addition to the minor soldering required, the majority of components were effortlessly connected to the breadboard using jumper wires. The small footprint of the Pi Pico further simplified the connections to its pins.

In the subsequent sections, detailed explanations of the wiring configurations for all modules will be provided, offering insights into their respective functionalities and interconnections within the system.

2.2.1. Matrix keypad wiring

Row Pins Configuration

- The 4 row pins are configured as INPUT with PULL-UP resistors enabled. This configuration allows the microcontroller to detect changes in voltage on these pins.

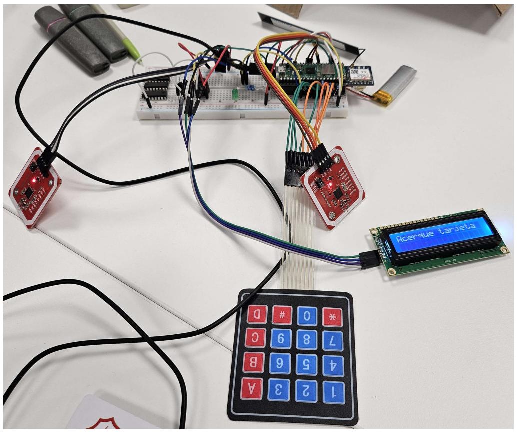

Figure 2.9: First prototype mounted on a breadboard

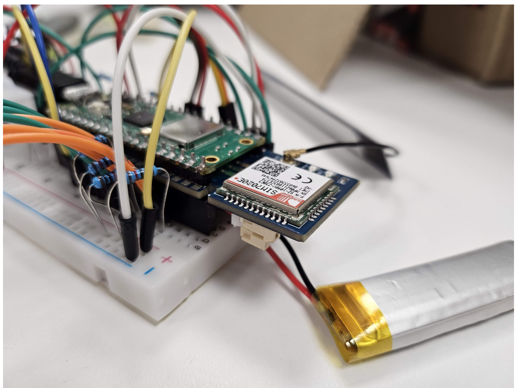

Figure 2.10: Zoom in on the Pi Pico and SIM7020E module

- Each row pin is connected to the positive supply voltage (3V) through a 10kΩ resistor. This ensures that when no button is pressed, the row pins maintain a high voltage level due to the PULL-UP resistors.
- When a button in a particular row is pressed, it creates a connection between the row and column pins, effectively pulling the voltage level of the corresponding row pin down to ground.

Column Pins Configuration

- The 4 column pins are configured as OUTPUT. During scanning, each column pin is individually set to false (logic low), one at a time, while the others remain at a high logic level (logic high or true).
- By setting a column pin to false, it creates a path to ground for the corresponding row pins. If a button in the corresponding column and row is pressed, the voltage on the row pin is pulled low, indicating a button press.

Button Press Detection

- When a button is pressed, it forms a connection between a row and column pin. During scanning, when the corresponding column pin is set to false, it effectively pulls the voltage on the row pin down to ground.
- The microcontroller continuously scans each column pin, one by one, and if it detects changes in voltage on one row pin, then it already pinpointed the row-column pair, corresponding to a key press.

In summary, this configuration allows the microcontroller to detect button presses on the keypad by scanning each column and monitoring the voltage levels on the row pins. When a button is pressed, it creates a connection between a row and column pin, causing a change in voltage on the row pin, which is detected by the microcontroller.

2.2.2. Devices on the I2C bus

Thanks to the straightforward nature of I2C communication, connecting both the LCD display and the NFC reader requires just two wires each: one for the Serial Data Line (SDA) and the other for the Serial Clock Line (SCL), in addition to the necessary VCC and GND connections. With the Pi Pico offering two I2C buses, either or both buses can be used. This versatility is possible because the two devices possess distinct I2C addresses, as specified in their respective datasheets.

I2C Pull-up

Ensuring proper signal integrity is crucial for the reliable operation of I2C communication. One essential consideration in this regard is the implementation of pull-up resistors on both the SDA and SCL of the bus.

Pull-up resistors serve to maintain the default high logic level on the SDA and SCL lines when they are not actively being driven by the master or slave devices. Without pull-up resistors, the I2C lines may float, leading to undefined voltage levels and potential communication errors.

The pull-up resistors effectively "pull" the voltage level of the I2C lines to the high logic level when they are not actively being driven low by a device. This ensures clear signal transitions and prevents signal distortion or noise interference that could disrupt communication.

Typically, pull-up resistors are connected between the SDA and SCL lines and the positive supply voltage (VCC) of the system. The value of the pull-up resistors determines the strength of the pull-up effect and should be chosen carefully to balance signal integrity with power consumption.

During the initial stages of the project, pull-up resistors were manually added to the SDA and SCL of the I2C bus to ensure proper signal integrity. However, it was later discovered upon examination of the device schematics that both the LCD display and NFC reader modules already incorporated pull-up resistors on their respective boards, with voltage level translation from 5V to 3.3V.

This unintentional duplication of pull-up resistors led to a reduction in the overall resistance of the pull-up network and an increase in the strength of the pull-up effect on the I2C lines. The existing 4.7kΩ resistor in both the NFC reader and LCD modules inadvertently paralleled with an additional 10kΩ resistor, effectively reduced the overall resistance to approximately 3.2kΩ [10]. While this theoretically could result in faster signal transitions, it also heightened susceptibility to noise and interference, potentially leading to signal integrity issues. Additionally, the increased current draw from the additional pull-up resistors slightly elevated the power consumption of the system.

Despite these consequences, it is important to note that the unintentional duplication of pull-up resistors is unlikely to cause permanent damage to the devices involved.

Problems with the NFC reader

During a later phase of the project, an issue surfaced wherein the NFC reader would cease to respond after prolonged periods of operation. The only indication of this malfunction was rapid flickering of its red LED. Attempts to command the reader yielded no response, needing a power cycle to restore functionality. This could be achieved either by disconnecting and reconnecting the power source or by pulling down the "RSTPD_N" pin on the board. To mitigate this recurring problem, a solution involving the insertion of a transistor between the microcontroller and the VCC pin on the NFC board was devised. This approach was favored over direct soldering onto a minuscule 1-millimeter pin on the NFC board, which presented challenges and fitting problems. This will be covered in more detail in the next chapter.

2.2.3. The SIM7020E board

With the Waveshare board purposefully engineered to align with the pinout of the Pi Pico, no wiring is necessary beyond connecting the two components. This module interfaces with the microcontroller via UART for communication and utilizes additional GPIO pins for auxiliary functions such as power management.

A critical consideration is to avoid utilizing any pins already allocated by the SIM7020E board. This precaution can be ensured by referencing the datasheet and wiki documentation provided by the manufacturer [27].

Although the device includes a 3.7V Li-Po battery in the box, it was decided against its utilization due to concerns about introducing an additional point of failure to the device. Furthermore, its presence impedes the Raspberry Pi from undergoing a hard reset when disconnected from the power source, since this battery keeps feeding the 5V rail on the Pi Pico.

2.2.4. Buzzer and other components

The active buzzer, requiring minimal setup, simply needs a transistor to toggle its state using a control signal from the microcontroller. It's essential to note that it must still be connected to a 5V source, as the 3.3V from the GPIO pins is insufficient, hence the requirement for the transistor.

Adding an LED to the project is similarly straightforward, as it can be controlled directly by a GPIO pin due to its lower voltage and current requirements. When paired with a 50Ω resistor, the LED emits sufficient brightness, even in well-lit conditions. However, ultimately, it was deemed unnecessary and excluded from the final design.

2.2.5. Considerations

The initial prototype incorporated two NFC modules, as can be seen on figure 2.9, intending to use one module for clocking in and the other for clocking out. This approach was adopted initially to streamline compatibility with a specific provider. However, this strategy was subsequently abandoned in favor of a simpler solution, whereby the worker's entry or exit status is determined through software processing.

Initially, the matrix keyboard was contemplated as an alternative method for clocking in or out by entering a numeric code. However, this approach was deemed impractical due to several concerns. Using the worker's national identification number as the input code would involve handling sensitive information, which was not ideal. Additionally, distributing individual codes to each worker would present logistical challenges. As a result, this approach was ultimately abandoned.

Chapter 3

PCB Design

Following the initial development on a breadboard, the natural progression in the design evolution was to transition to a perfboard assembly. This intermediate step not only served to condense the device's footprint but also offered a preliminary glimpse into its form and functionality. Subsequently, the culmination of this iterative process involved the design and fabrication of a PCB tailored to the project's specifications, marking a significant milestone in its realization.

3.1. Assembling on a Perfboard

Perfboards, short for perforated boards, are commonly used prototyping platforms in electronics. They consist of a board with a grid of holes spaced at regular intervals. These holes are surrounded by copper pads or traces that can be connected using solder to create electronic circuits. Perfboards allow electronic components, such as resistors, capacitors, integrated circuits, and other discrete components, to be soldered onto the board, facilitating the creation of temporary or semi-permanent circuits for testing and development purposes [8]. They serve as a practical and versatile tool for electronics hobbyists, engineers, and designers to quickly prototype and iterate on circuit designs before moving to more permanent solutions like printed circuit boards (PCBs). Examples can be seen in Figures 3.1 and 3.2.

In the context of this project, transitioning to a perfboard prototype served multiple crucial purposes. Firstly, it offered a tangible visualization of the future device's physical footprint, providing executives with a concrete representation of the project's direction and potential. This visual aid not only conveyed the scale and form of the device but also showed its feasibility and progress, instilling confidence in its development trajectory.

To commence this phase, a diagram detailing the placement of each component and its connection to the corresponding pins on the microcontroller was drafted. This diagram served as a blueprint, guiding the subsequent assembly process.

With the schematic as a roadmap, the assembly of the perfboard prototype commenced. Components were transitioned from the breadboard to the perfboard, ensuring each element found its designated place. Careful consideration was given to the layout, optimizing spatial organization for efficient circuitry and minimal interference.

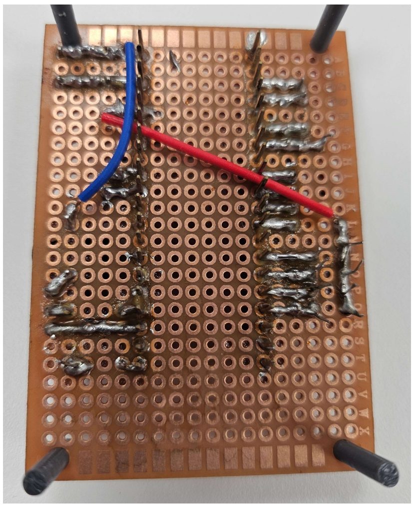

Figure 3.1: Back side of the assembled perfboard

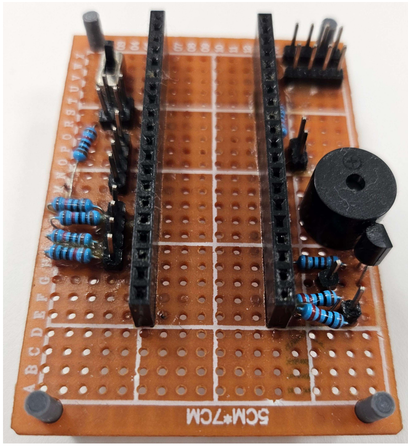

Figure 3.2: Front side of the assembled perfboard

3.2. Designing the PCB

Having used the perfboard as a preliminary platform to consolidate components and understand the spatial dynamics of the circuit, the focus now shifts towards the design of the *printed circuit board* (PCB).

The primary function of the PCB mirrors that of the perfboard: to integrate and interconnect all components seamlessly. However, unlike the perfboard, the PCB offers advantages in terms of aesthetics, functionality, and ease of assembly.

Moreover, the transition to a multi-layer PCB introduces a realm of design possibilities previously unattainable with the perfboard. By using multiple layers, traces can now traverse both above and below the surface, allowing for more flexibility in component placement and routing. This feature allows for the implementation of continuous male headers for each component, which were not easily implemented with the perfboard.

3.2.1. Requirements

As mentioned earlier, this PCB is designed primarily as an interconnection platform for the various components of the device, making its complexity relatively low.

The SIM7020E board would be positioned beneath the Pi Pico using stacking headers, as illustrated in Figure 2.6, thereby reducing the number of interconnections required.

Additionally, it requires integration of the buzzer and its transistor, while certain parts were retained in case of future requirements, given their minimal cost. These include the matrix keyboard's header, the additional NFC reader's header, and the LED's header. Further details on the cost will be provided in the next section.

Another critical requirement driving the development of a custom PCB is the need to streamline the device's assembly processes. In line with this objective, it's essential for the PCB to arrive pre-assembled. While it's common for boards to be sold without soldered components, pre-assembled PCBs are a must for this project to be economically and logistically viable. Thus, an ideal provider would offer both manufacturing and assembly services.

3.2.2. Design and Production

In the quest for a user-friendly PCB design program, *EasyEDA*[1] emerged as a promising option. Offering a schematic design tool, it facilitated the transition from hand-drawn schematics to digital diagrams. EasyEDA allows users to seamlessly translate these diagrams into PCB layouts, enabling the placement of components, definition of board dimensions, and automatic routing of traces to interconnect all elements. Thanks to its simplicity, a final design was swiftly achieved within a matter of days.

It was discovered that EasyEDA has a partnership with *JLCPCB*[2], a company offering comprehensive PCB manufacturing and assembly services. Additionally, EasyEDA seamlessly integrates with *LCSC*[3], a supplier providing a comprehensive parts catalog with detailed specifications for each component used in the project. Notably, the final pricing was highly competitive, and with all these providers working together, support extended throughout the entire production process, from schematic design to the final product.

A few considerations regarding part selection:

- Since active buzzers can only be turned on or off, the sound's frequency is already pre-determined, and cannot be changed. This has to be taken into consideration when choosing one.

- The transistor responsible for controlling power to the buzzer must meet the power specifications and respond to changes in voltage from a GPIO pin (3.3V). Fortunately, given that the voltage at the drain is 5V and the current is in the milliamp range, virtually any transistor would be suitable for this purpose.

[1] EasyEDA's website: `https://easyeda.com/`
[2] JLCPCB's website: `https://jlcpcb.com/`
[3] LCSC's website: `https://www.lcsc.com/`

- Since regular *DuPont* jumper wires will be used, each one of the headers' pins have to be 2.54mm apart (which is the equivalent to 0.1 inches), thus compatible headers need to be selected.
- Mounting holes would be needed on each corner to attach it to a box, which would be designed in the future.

After navigating through a learning curve with the parts catalogue and sorting through dozens of providers for each component, all necessary parts were successfully chosen, and the PCB was prepared for production.

Figure 3.3: PCB's top view

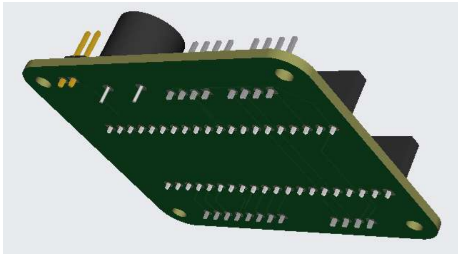

Figure 3.4: PCB's bottom view

During the ordering process, a 3D model of the PCB with the selected parts is displayed, similar to Figures 3.3 and 3.4. It's important to note that the PCBs are manufactured in China and then shipped internationally. Factors such as import tariffs and shipping costs need to be taken into account. Despite these considerations, the price remained competitive compared to local alternatives, even after factoring in customs' duties and shipping fees. Remarkably, the PCBs arrived within a week of ordering.

The quality of construction exceeded expectations, with all components fitting perfectly on the first try. The selected buzzer emitted the expected pitch, and everything functioned as intended. Executives were pleasantly surprised by the remarkably low price per unit.

Solving the Problem with the NFC Reader

As discussed before, the NFC reader needs to be periodically reset by cutting its power supply and restoring it after a few miliseconds.

Unfortunately, this was discovered after 25 units of the PCB were produced. In order to reduce waste, it was decided that all the PCBs would be fixed by adding a transistor to each of them. Additionally, to control this transistor, a resistor needs to be placed between the gate and the Pi Pico's GPIO pin that controls it. This can be seen in Figures 3.5, 3.6 and 3.7.

In this way, the transistor can interrupt power from the ground pin on the Pi Pico, to the ground pin on the PN532 board.

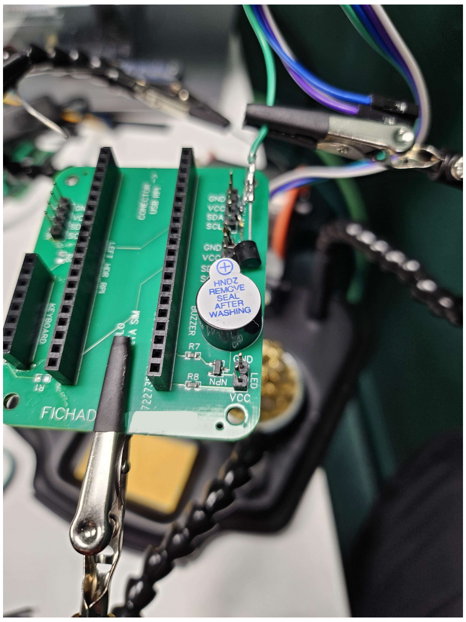

Figure 3.5: PCB fixed on helping hands

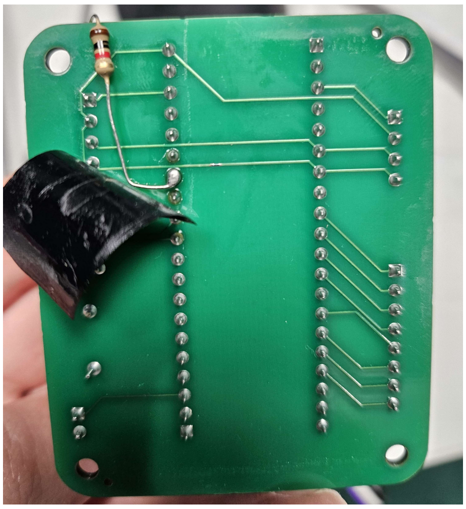

Figure 3.6: PCB's back side after being modified

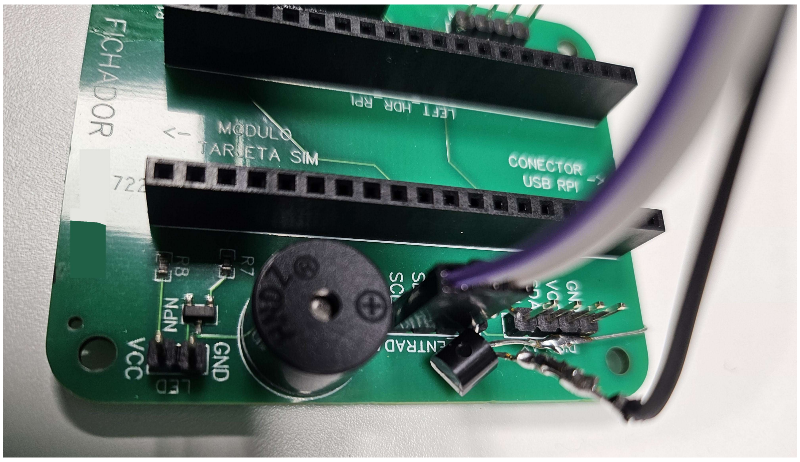

Figure 3.7: PCB's top side after being modified

Chapter 4

Code Implementation and Development Challenges

The software development phase represents a significant portion of the project timeline, demanding meticulous attention to detail and thorough problem-solving skills. In the forthcoming sections, an overview of the software requirements will be provided, shedding light on how these requirements were effectively addressed. Additionally, detailed insights into the encountered challenges and the corresponding solutions devised will be explored.

One crucial decision made early on was the selection of CircuitPython as the programming language for the device. This decision stemmed from several key factors, including its open-source nature, extensive library support, and the availability of pre-existing libraries with MIT licenses, particularly those essential for handling NFC readers. Furthermore, CircuitPython's user-friendly syntax and simplified microcontroller programming paradigm were deemed advantageous compared to alternatives like MicroPython, contributing to smoother development workflows.

However, it's important to note that despite the benefits of CircuitPython, the project encountered limitations imposed by the microcontroller's memory capacity, capped at 264kB. This constraint was exacerbated by the use of Python, a high-level language known for its memory-intensive nature. Throughout the development process, various memory optimization strategies were implemented to mitigate these limitations. These optimizations will be thoroughly explained, providing valuable insights into managing resource constraints in microcontroller-based projects.

In addition to the aforementioned aspects, it's crucial to highlight the iterative nature of software development, where frequent testing, debugging, and refinement cycles were integral to achieving desired functionality and performance. This iterative approach enabled the project team to address emerging issues promptly and iteratively enhance the software's robustness and reliability.

Furthermore, a detailed examination of the software architecture and design decisions made will offer valuable insights into the project's development methodology and the rationale behind specific implementation choices.

4.1. CircuitPython: Advantages and Disadvantages

CircuitPython is an open-source programming language designed specifically for microcontroller-based development. Developed primarily by Adafruit Industries, CircuitPython is built on top of the Python programming language, offering a simplified yet powerful platform for programming microcontrollers. Unlike traditional embedded programming languages, CircuitPython abstracts many low-level complexities, making it more accessible to beginners and hobbyists while still providing advanced features for experienced developers.

Figure 4.1: CircuitPython's logo. Image by Adafruit licensed under MIT

One of the key advantages of CircuitPython is its user-friendly syntax and high-level abstractions, which resemble those of Python, a language renowned for its readability and simplicity. This makes CircuitPython particularly attractive to beginners and educators, enabling them to quickly grasp programming concepts and develop projects without being bogged down by intricate syntax or complex setup procedures.

Furthermore, CircuitPython simplifies the process of interacting with hardware peripherals and sensors commonly used in embedded systems. It provides a consistent and intuitive API (Application Programming Interface) for accessing GPIO pins, I2C and SPI interfaces, analog inputs, and other hardware features, streamlining the development process and reducing the learning curve associated with embedded programming.

Another notable advantage of CircuitPython is its extensive library support, particularly for popular microcontroller boards and peripherals. Adafruit maintains a vast repository of CircuitPython libraries, covering a wide range of sensors, displays, actuators, communication modules, and other components commonly used in electronics projects[1]. These libraries abstract the complexities of interfacing with specific hardware, allowing developers to focus on application logic rather than low-level hardware details.

Despite its numerous advantages, CircuitPython does have some limitations and drawbacks. One notable limitation is its higher memory footprint compared to lower-level languages like C or assembly, which can be a concern for projects with strict memory constraints. Additionally, while CircuitPython abstracts many hardware complexities, it may sacrifice some performance compared to bare-metal or lower-level programming approaches.

[1] Adafruit's Library Bundle: `https://github.com/adafruit/Adafruit_CircuitPython_Bundle`

Overall, CircuitPython offers a compelling combination of simplicity, accessibility, and versatility for microcontroller-based development, making it an excellent choice for a wide range of projects, from educational endeavors to commercial products. Its rich ecosystem of libraries, ease of use, and strong community support contribute to its popularity among hobbyists, educators, and professional developers alike [2].

For this project, CircuitPython version 9.0.0 will be used.

4.2. Code Requirements

The device must prioritize responsiveness and intuitive usability for all employees. This ensures that the device can be easily operated by users of varying technical proficiency levels, promoting efficient and accurate clocking processes.

Furthermore, the device must maintain the highest possible uptime to ensure continuous operation without interruptions. This is crucial for accurate time tracking and prevents any disruptions that could lead to discrepancies in employee attendance records.

Data integrity is of paramount importance, as any loss of data could result in unfair deductions from employees' wages. Therefore, the device must be designed to prevent data loss under any circumstances, even in the event of power outages or connectivity issues. It should have the capability to store clockings locally and reliably transmit them to the server once connectivity is restored.

In situations where internet connectivity is temporarily lost, the device must continue to store clockings and transmit them as soon as connectivity is regained. This ensures that no clocking data is lost and enables seamless operation regardless of network conditions.

Efficient data transmission is essential for real-time monitoring and analysis of employee attendance. Therefore, the device should send clocking data as soon as it is available, without any delays, to ensure timely reporting and accurate statistics.

Flexibility is also key, as the device must be capable of operating using either WiFi or cellular data connectivity, depending on the specific configuration and deployment requirements.

Moreover, the device should incorporate a mode-switching feature, allowing it to display relevant information such as the device's ID and enabling easy implementation of additional modes in the future. This ensures scalability and adaptability to changing needs and requirements.

Additionally, the device must display the current time on the screen to provide users with real-time information and facilitate accurate clocking processes.

To enhance user experience and ensure successful clocking operations, the device should provide visual and audible cues to indicate successful clockings. It should also alert users to errors, such as attempting to clock in multiple times with the same keycard.

Finally, all clocking data should be securely transmitted to an API using POST requests, ensuring that it is reliably delivered to the server for processing and storage. This maintains data integrity and enables seamless integration with existing systems and workflows.

4.3. Overview

In this section, a brief overview of the main workings of the code will be provided, with more detailed explanations to follow in subsequent sections.

The code operates two main tasks running asynchronously, which share CPU time efficiently.

- **Main Loop**: This task continuously listens for NFC cards. When a card is detected, it adds the clocking data to the device's storage. Additionally, this task periodically instructs the second task to send a "keepalive" message to the server.
- **Data-Sending Task**: This task reads the device's storage for any stored clockings. When it finds clocking data, it sends a POST request to the server. This task manages all internet communications.

This division of labor ensures that the device remains responsive and does not hang. If a worker had to wait for a POST request to complete before clocking in, the delay could range from 10 seconds to a few minutes (due to the NB-IoT's network slow responses), which is unacceptable. By making the main loop independent from the data-sending task, the system maintains its responsiveness and efficiency.

4.4. Code Implementation

4.4.1. Asynchronous Programming

Asynchronous programming is a programming paradigm that allows multiple tasks to be executed concurrently, without the need for explicit parallelism or multithreading. It enables programs to perform non-blocking operations, where tasks can be started and completed independently of each other, leading to improved performance and responsiveness.

In CircuitPython, asynchronous programming is achieved using the `asyncio` module, which provides a framework for writing asynchronous code using coroutines. Coroutines are functions that can be paused and resumed during execution, allowing for efficient task scheduling and coordination.

The key concept in asynchronous programming in Python is the `async` and `await` keywords. Functions defined with the `async` keyword are called asynchronous functions, and they can be paused with the `await` keyword when they encounter an operation that may block, such as I/O operations or network requests.

When an asynchronous function is awaited, it returns control to the event loop, allowing other tasks to continue executing in the meantime. Once the awaited operation completes, the function resumes execution from the point where it was paused.

The event loop is a central component of asynchronous programming in Python and CircuitPython. It manages the execution of asynchronous tasks and ensures that they are scheduled and executed efficiently. The event loop continuously checks for tasks that are ready to run, executes them until completion or until they are paused, and then moves on to the next task [2].

The code extensively utilizes asynchronous programming to enhance the perception of concurrency and prevent the device from becoming unresponsive. This is crucial, especially considering that the device displays the time, including seconds, on the screen. However, while this approach offers significant benefits, it also introduces certain drawbacks, which will be addressed later.

Later on, this aspect will be further explored. Essentially, two tasks will run within the event loop: one managing the main loop, waiting for workers to approach the device with their keycards, while the other will handle the transmission of clockings to the internet.

4.4.2. Main Loop and Operating Modes

The main loop executes functions on the current operating mode. These modes, essentially Python classes, implement an interface containing four functions: `enter_mode`, `exit_mode`, `card_received`, and `no_card_found`. Depending on the event, one or more of these functions may be invoked.

Initially, the device starts in a default mode. When the "configuration keycard" is detected by the NFC reader, the mode transitions to a different one, triggering the `exit_mode` function on the mode that is being exited, and `enter_mode` in the mode that entered execution.

As part of its operation, the NFC reader is configured with a timeout to detect the presence of any card near its antenna. Upon successful detection, it reads the UID of the card, calling the `card_received` function. If after the timeout, no card is read, then the `no_card_found` function is called.

Additionally, the main loop is responsible for periodically signaling the other asynchronous task to send a "keepalive" message to the server to maintain connectivity.

4.4.3. Interchangeable WiFi and Cellular Data Operation

In certain deployment locations, access to WiFi may not always be available. Furthermore, even in areas with existing WiFi coverage, circumstances can change, requiring the device to be relocated to areas without such connectivity. In such scenarios, cellular data operation becomes essential.

To facilitate internet usage, an interface was created with fundamental functions for internet communication. Essentially, the primary requirement was the capability to send POST requests, or a series of them. Consequently, two classes were designed to adhere to this interface: one serving as a controller for WiFi connectivity, and the other for the SIM7020E module.

Either of these controllers could be employed seamlessly, with the device operating without any noticeable difference.

Regarding the WiFi functionality, Adafruit offers comprehensive code for executing requests via WiFi under an MIT license. This code was subsequently modified to operate asynchronously, aiding in preventing device hang-ups. Details regarding the SIM7020E code will be provided in the subsequent subsection.

4.4.4. SIM7020E Module

The SIM7020E module, which facilitates cellular data connectivity, operates using *AT commands*.

AT commands, short for "Attention commands", are a standardized set of instructions used to communicate with modems, including those integrated into cellular modules like the SIM7020E. These commands are typically sent from a host device, such as a microcontroller or computer, to the modem via a serial communication interface (such as *UART*, which is the case for this Waveshare board).

AT commands follow a specific syntax, starting with the characters "AT" (hence the name), followed by a command mnemonic and optional parameters. They are used to configure various settings, initiate actions (such as making a phone call, sending a text message or sending a POST request), query status information, and handle other modem-related tasks.

For example, to check the signal strength of a cellular connection, the command "AT+CSQ" may be sent to the modem, which would respond with a value indicating the signal quality. Similarly, to establish a data connection, the command "AT*MCGDEFCONT" may be used, followed by parameters specifying the APN (Access Point Name) and other connection details. It is worth noting that some of these commands may vary depending on the module used, and some are manufacturer specific [22].

AT commands provide a standardized interface for interacting with modems, allowing developers to integrate modem functionality into their applications without needing to understand the intricacies of the underlying hardware.

SIMCOM offers a specific manual for the SIM7020E, delineating the AT commands, their parameters, usage instructions, and responses. This manual proved indispensable for using the module, especially considering the scarcity of code examples available online for this specific module, apart from the brief code snippets provided by Waveshare as illustrative examples.

Commands are transmitted to the transmit UART buffer, where they await processing by the SIM7020E module. Following a command transmission, the module responds, and its responses are stored in the designated memory region allocated for the receive UART buffer.

Subsequently, the receive UART buffer must be periodically checked to ascertain whether the entire response has been received. This periodic monitoring is essential since not all responses may be promptly received without any delay, potentially leading to device hang-ups. This necessity for asynchronous programming arose primarily from this requirement. Consequently, with asynchronous programming implemented, a command is dispatched, and the response is continuously read until the entire message is received, allowing for subsequent commands to be processed.

It's crucial to note that only one "thread" is engaged in communication with the SIM7020E module, obviating the need for locks or synchronization mechanisms to prevent the dispatch of additional commands while awaiting a response. Such simultaneous command dispatch would risk causing errors within the module.

Certain commands, particularly those related to establishing connections with the internet and external servers, inherently entail longer execution times, necessitating longer wait times during the response retrieval process.

Regarding POST requests, a sequential series of steps must be meticulously followed to ensure successful transmission. These steps will be explained to provide a clearer understanding of the module's interaction mechanisms.

CHTTPCREATE

First, an HTTP or HTTPS client instance must be created within the module. Here, the host is specified:

```
AT+CHTTPCREATE="https://example.com"
```

And after waiting for the corresponding response, the following will be read:

```
+CHTTPCREATE: 0\r\n\r\nOK\r\n
```

Notice the appearance of "`\r`" and "`\n`", due to the use of Python's byte strings.

Most commands end with the appearance of "`OK\r\n`", which allows for easily checking if a command has finished executing, to then send the following one.

CHTTPCON

The next step is to establish the HTTP connection with the server, by executing the following:

```
AT+CHTTPCON=0
```

Notice the “0” used in the command. That number has to be the same one as the one returned as a response in the `CHTTPCREATE` command. What that number means, is the slot that the host is occupying in the module’s memory. In the SIM7020E, there can be only 5 hosts in total (numbers 0 to 4). Thus, by connecting to the index “0”, the device will establish the connection with the host just created.

The following response will be received in case of success:

```
\r\nOK\r\n
```

CHTTPSEND

Now, the actual POST request can be sent. The structure of the command is the following: “`AT+CHTTPSEND=<httpclient_id>, <method>, <path>, <header>, <content_type>, <content_string>`”.

The values for this command will be the following:

- `httpclient_id`: “0”, as explained before.
- `method`: the number “1” corresponds to the POST method, as explained by the commands manual.
- `path`: this is the endpoint. The host’s URL must not be specified.
- `header`: the header contents, encoded in hexadecimal.
- `content_type`: the content type, as in any regular POST request. For example: “application/json”.
- `content_string`: the data contained in the POST request, encoded in hexadecimal.

In the following example, the following headers will be sent, encoded in hexadecimal:

```
Accept: */*
Connection: Keep-Alive
User-Agent: RPI_PICO
```

And the following data:

```
{"data": "some information"}
```

This is the resulting command:

```
AT+CHTTPSEND=0,1,"/some_endpoint",
4163636570743a202a2f2aa436f6e6e656374696f6e3a204b65657
02d416c697665a557365722d4167656e743a205250495f5049434fa,
"application/json",
7b2264617461223a22736f6d6520696e666f726d6174696f6e227d
```

CHTTPDISCON

The last command to be executed is the disconnection from the HTTP server:

```
AT+CHTTPDISCON=0
```

The number sent in the command is a “0” again, and the response will be another OK. With this, the POST request will have been sent, and the connection terminated.

The HTTP field could be destroyed with the `CHTTPDESTROY` command, but this is not needed, since it will be reused, and would only increase inefficiencies in sending a new request. In fact, if there are many POST requests queued to be sent, then they could even be sent continuously before sending the `CHTTPDISCON`.

4.5. Challenges

This section examines the challenges faced during the development process. From technical obstacles to unforeseen setbacks, each hurdle provided an opportunity for problem-solving and refinement.

4.5.1. Lack of Multithreading

While CircuitPython incorporates asynchronous programming via the `asyncio` library, it's essential to note that this differs from traditional multithreading, a feature that is in fact supported by MicroPython. Asynchronous programming with `asyncio` allows for concurrent execution of tasks, but it operates within a single thread, managing task switching based on input/output operations or explicit yields. In contrast, multithreading involves the simultaneous execution of multiple threads, each running independently and potentially executing different code segments concurrently.

One significant challenge arises from the nature of CircuitPython's asynchronous programming model. While `asyncio` allows for the concurrent execution of tasks within a single thread, certain code segments lacking explicit yields may monopolize the event loop, hindering task switching and potentially causing sluggish performance. This becomes particularly evident in scenarios where tasks, such as establishing a WiFi connection, encounter delays. In such cases, where CircuitPython lacks native support for asynchronous WiFi operations, the device may become unresponsive for extended periods, up to 15 seconds, before timing out.

Regrettably, there exists no straightforward solution to address this challenge within the current framework. A potential avenue for improvement involves migrating the codebase to MicroPython, which offers multithreading capabilities. However, this approach necessitates significant reworking of existing libraries, presenting a formidable task exacerbated by time and budget constraints.

4.5.2. Memory Constraints and SSL Certificates

Python's inherent memory footprint, compared to C, is often larger, which poses a challenge given the limited 264kB of RAM available. Upon loading CircuitPython during startup, approximately 120kB of memory remains free. However, once all libraries are loaded and devices initialized, this dwindles to around 60kB.

While this may seem sufficient, complications arise when attempting to send a POST request to an HTTPS server over WiFi. SSL certificate validation requires a minimum of 60kB of available memory, dangerously close to the device's operational limit. The lack of descriptive error messages compounds this issue, necessitating extensive investigation to decipher the underlying problem. Anecdotal evidence suggests that errors may occur when available memory dips below 42kB [15], though during testing, errors consistently manifested when nearing the 60kB threshold. This discrepancy may be attributed to various factors, including CircuitPython version and memory fragmentation.

To mitigate memory constraints, a strategy was devised to import libraries only when needed. For instance, if the device operates in WiFi mode, code related to the SIM7020E module remains dormant. Additionally, the garbage collector is called upon constantly to minimize memory fragmentation and ensure sufficient space for SSL certificates.

During testing, the following errors were encountered:

- When available memory is insufficient by a big margin:

```
x509-crt-bundle:PK verify failed with error FFFFBD70
x509-crt-bundle:Failed to verify certificate
Exception:  <class 'RuntimeError'> Error connecting socket:
 (-12288, 'MBEDTLS_ERR_X509_FATAL_ERROR')
```

- When available memory is nearly adequate but still falls short a regular `MemoryError` is raised.

In testing, it was challenging to ascertain the precise threshold between encountering the two errors. However, observations indicated that the first error surfaced at approximately 50kB or below, while a `MemoryError` occurred within the range of 50kB to 60kB.

Addressing this challenge proves daunting, as memory optimization can only achieve so much, and the alternative—removing code to reduce functionality—is undesirable.

4.5.3. NFC Reader Unresponsiveness

As detailed in Chapter 2 (see Chapter 2), during the later stages of testing, a significant issue emerged with the NFC reader. It would abruptly cease responding to requests until power was disconnected and reconnected. The only visible indication of this malfunction was the erratic flickering of the red LED on the board.

In software, the NFC reader would not trigger any error when polled solely for the presence of a keycard nearby. Therefore, continuous polling of the PN532 for its firmware version was used to ascertain its operational status. If it failed to respond, a reset was initiated with the assistance of a transistor.

4.5.4. Storage

The problem

Due to the necessity of retaining all clockings until their transmission to the server and even after system resets, the data had to be stored in the internal memory of the Raspberry Pi Pico. Despite its ample 2-megabyte capacity, sufficient to accommodate tens of thousands of clockings, the implementation was far from straightforward.

Initially, the process seemed straightforward. However, CircuitPython imposes limitations on writing to internal storage unless configured in the `boot.py` file, utilizing the `storage.disable_usb_drive()` function. Unfortunately, activating this function disables the Pi Pico's appearance as a drive on connected computers, thereby negating one of CircuitPython's key advantages for rapid debugging against MicroPython. Consequently, file transfers were relegated solely to the COM Port.

Once this hurdle was overcome, clockings could be stored in the internal storage. However, a critical scenario emerged: if the device experienced prolonged connectivity loss and accumulated 1 megabyte of clockings in internal storage, each clocking sent to the server necessitated removal from internal storage. Given that stored clockings should be sent in order of arrival, that is, a queue, the deletion process began with the first entry. Unfortunately, the provided filesystem lacked a "delete" method to selectively remove file content, forcing the entire file's rewriting.

This predicament not only engendered prolonged read and write times, potentially degrading flash memory, but also posed a fundamental memory constraint. With the Pi Pico lacking 1 MB of RAM to retain all data in memory simultaneously for file rewriting, the process proved doubly challenging.

The solution

To address the storage issue, an algorithm was developed to create multiple files, each with a fixed maximum number of clockings. For simplicity, let's assume each file holds up to 50 clockings, amounting to approximately 5kB per file.

As clockings accumulate, new files are created, each containing 50 clockings. These files are sequentially named starting from `file0` (e.g., `file0`, `file1`, `file2`, ..., `fileN`). When connectivity is restored, the clockings are removed from `file0` onward, maintaining a queue structure where the oldest entries are processed first. After removing many of the oldest clockings, the file structure may be, for instance: `file5`,..., `fileN`, which has to be accounted for in the algorithm.

This approach ensures that when a clocking is added or removed, the maximum amount of data handled in memory at any given time is limited to 5kB. This data is then quickly rewritten to storage, significantly reducing the risk of memory overload and enhancing system performance.

4.5.5. Connecting to the Google Cloud Platform

During initial testing with an API hosted on a local computer, connectivity issues were not encountered. However, after deploying the API on the Google Cloud Platform (GCP), the SIM7020E module began experiencing intermittent problems establishing the HTTP connection. Specifically, the `CHTTPCON` command would occasionally error out, causing significant delays in sending POST requests.

Attempts to establish the connection varied widely, sometimes requiring anywhere from 1 to 40 attempts to successfully execute the `CHTTPCON` command. This inconsistent behavior severely slowed down the data transmission process, necessitating an urgent resolution.

After extensive testing over several months, it was determined that this issue was unique to the GCP, particularly when using Google Cloud Functions. This discovery prompted a thorough investigation into all potential causes, following every lead possible.

Initially, a source suggested that the problem might be related to the lack of support for Server Name Indication (SNI) by the SIM7020E module [25]. However, this theory was initially dismissed since the issue was intermittent, which is not typical of a fundamental incompatibility like SNI support.

Complicating the investigation further, in some locations, the device could not connect to the GCP at all, despite having internet access and being able to synchronize time with Google's time servers. Meanwhile, other locations could establish connections after only a few attempts, indicating a potential environmental or network-related factor.

Signal quality and strength were also considered as possible culprits, but even with excellent signal conditions, the connectivity issues persisted. Additionally, swapping SIM cards between different operators appeared to affect the frequency of the problem. For instance, using Movistar SIM cards resulted in more frequent failures, whereas a multi-operator provider, Diferenza, showed fewer issues.

Seeking a definitive answer, the team contacted SIMCOM, the manufacturer of the SIM7020E module. SIMCOM's response was that Google had discontinued their IoT Cloud services, and they recommended switching to Azure or AWS. They also mentioned that the SIM7020E module was discontinued, with the SIM7022 as its successor, which after investigating this new variant, it turns out it supports SNI.

Despite these insights, a clear resolution remains elusive. The most plausible speculation is that the issue might indeed be related to SNI. The variability in GCP's internal server behavior could mean that some instances of their servers require SNI, while others do not. This inconsistency might explain the intermittent nature of the problem and its location-dependent variability.

The next steps involve considering alternative hardware solutions. The SIM7022 module, which supports SNI, could potentially resolve the issue, but it would require board-level modifications beyond the team's current capabilities. Alternatively, Waveshare produces another module with the SIM7080G, which does support SNI, this would be more accessible but a costlier solution.

Chapter 5

API Development

After developing a basic working device, the next major milestone was enabling it to send data over the internet. Achieving this milestone was crucial for demonstrating the project's progress to executives, proving that the project is on track for success. Such progress not only instills confidence in the project's viability but also helps secure additional financing for further development. This financing is essential for acquiring necessary resources, such as a 3D printer, which will be discussed in the next chapter.

Developing an *Application Programming Interface* (API) was a fundamental step in this process. The API acts as an intermediary, facilitating communication between the device and a database where clockings are stored. It ensures that data from the device is accurately and efficiently transmitted to the database. This capability is vital for the real-time tracking and management of employee clockings, which forms the core functionality of the device.

This approach adds an extra layer of security. The alternative—directly inserting clockings into the database—would pose a significant security risk. If someone were to open the device and access its code, they could potentially see the database credentials. Given that the database contains personal data, maintaining its security is paramount. By using an API, the devices are restricted to only sending data, without the ability to read any data. This ensures that even if the device is compromised, the database credentials remain protected and the integrity of the personal data is maintained.

5.1. Requirements and Design Decisions

This time, the requirements are straightforward. The API needs to receive a JSON payload via a POST request, verify an API key, and process the data format to make it compatible with the database. For instance, it adapts the time format sent by the RTC clock of the Pi Pico into the format recognized by the database. This is more efficiently handled in an environment with extensive library support, rather than on a microcontroller running CircuitPython. After processing the data, the API must send it to the database and respond with an *HTTP 200 status* if the operation was successful.

Given the team's prior experience with Java and the *Spring Framework*, it was decided to use Spring to quickly deploy the API. Spring is a comprehensive framework for enterprise Java development, providing tools and features to build robust, scalable, and maintainable applications. It offers extensive support for web development, including RESTful services, through its Spring MVC module [24]. By leveraging Spring, the team could efficiently create a secure and reliable API.

The code for the API was intentionally kept very simple to ensure ease of development and maintenance. The first step involved creating the "Clocking" entity, which served as a model for the data structure representing each clocking event. This entity allowed Spring to seamlessly connect to the database, providing a straightforward way to store and retrieve clocking data.

Spring Security was then configured to require an API key for authenticating requests. This setup added an essential layer of security, ensuring that only authorized devices could send data to the API. By using API keys, the system could verify the source of the requests, preventing unauthorized access and potential data breaches.

In more detail, the process involved defining the "Clocking" entity with appropriate fields such as employee ID, timestamp, and any other relevant information. This entity was annotated with *JPA* (Java Persistence API) annotations to map it to the corresponding database table, allowing Spring Data JPA to handle the *CRUD* (Create, Read, Update, Delete) operations automatically.

For the security configuration, *Spring Security* was set up to intercept incoming HTTP requests and validate the presence and correctness of the API key. This was achieved by defining a security filter that checked the API key in the request headers and compared it against a predefined valid key. If the key was missing or incorrect, the request was rejected with an appropriate HTTP error code.

5.2. Containerizing the Java application with Docker

Containerizing is a method of packaging an application and its dependencies into a standardized unit, called a container, which can run consistently across different computing environments. This approach isolates the application from its environment, ensuring that it performs the same regardless of where it is deployed. For this API, *Docker* will be used to achieve containerization.

Docker is a platform that allows developers to automate the deployment of applications inside lightweight, portable containers. A Docker container includes everything needed to run the application: the code, runtime, libraries, and system tools.

Advantages of Containerizing with Docker

- **Consistency Across Environments**: Containers ensure that the application runs consistently across various environments—development, testing, and production. This eliminates the common "it works on my machine" problem, as the container encapsulates all the dependencies and configurations.
- **Simplified Deployment**: Docker containers can be easily deployed and managed. With Docker, new instances of the application can be spun up, scaled across multiple servers, and be updated with minimal downtime.
- **Resource Efficiency**: Containers are lightweight and use system resources more efficiently than traditional virtual machines. Multiple containers can run on a single host without the overhead of a full-blown hypervisor, leading to better utilization of server resources.
- **Isolation and Security**: Docker containers provide process and filesystem isolation, which enhances security. Applications running in separate containers are isolated from each other and from the host system, reducing the risk of conflicts and vulnerabilities.
- **Portability**: Docker containers can run on any system that supports Docker, whether it's a developer's laptop, an on-premise server, or a cloud-based virtual machine. This portability simplifies moving applications between different environments.

These advantages significantly outweigh the minimal time required to configure and deploy the container [4] [7].

In fact, the only setup needed in particular for this project, was just creating its *Dockerfile*, which is used to create the *image*, from which containers can be created.

```
FROM eclipse-temurin:17-jdk-jammy
ARG JAR_FILE
COPY build/libs/*.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

After the Dockerfile is created, the `docker build` command can be called to create the image [9].

```
docker build -t fichador-api .
```

Then, a container can be created from that image, starting an instance of it:

```
docker run -d -p 8080:8080
    -v fichadorLogs:/logs -env-file ./.env
    -rm -name api fichador-api
```

With the `docker build` command, an image was created with the name "fichador-api", then, with the `docker run` command, a container was started on the port 8080, storing the logs in the `/logs` file, and using a `.env` file to initialize variables, such as the database username and password. The container's name is "api".

5.3. Adaptation to Google Cloud Functions

Hosting an entire Docker container in the cloud can be quite expensive, especially if it needs to run continuously. Given the simplicity of the API and its low resource requirements, this approach is not cost-effective. Consequently, alternative hosting methods were explored. Since the company already utilizes Google Cloud services, Google Cloud Functions was recommended as a suitable solution.

Google Cloud Functions is a serverless execution environment that enables the execution of code in response to specific events without the need to manage or provision servers. It automatically scales based on the load, ensuring that only the compute resources used during the execution of the functions are paid for. This service is ideal for lightweight, event-driven applications, offering a cost-effective and efficient solution for deploying small-scale APIs and microservices. By using Google Cloud Functions, developers can concentrate on writing code while Google manages the infrastructure, scaling, and availability [16].

Fortunately, Spring provides support for Google Cloud Functions. However, adapting the code was not straightforward, and due to the lack of comprehensive documentation and code examples, it took nearly a week to accomplish.

The first challenge encountered was the scarcity of examples using Gradle as a dependency manager for building a project on Cloud Functions. Consequently, the project had to be migrated to Maven.

Additionally, conventional Spring Controllers could no longer be used and had to be transformed into a function format specified by Spring's documentation. This format expects a specific input and output data type. In this case, the function received a Clocking object and could output various response types, such as a simple string response like "OK".

Another problem was that it seems Spring was not able to read the JSON in the POST request and encode it into a Spring Entity. For this reason, a simpler class was added, which would be automatically parsed from the JSON in the POST request, and then be manually converted to an actual Spring Entity inside the function's code. In this way, it could be persisted to the database.

A working prototype was successfully developed, but it is currently pending approval for use. For security reasons, executives have decided to use an existing API and make minor adaptations to fit the current use case, rather than introducing entirely new code.

Chapter 6

Designing and Fabricating the Enclosure: 3D Modeling and Printing

Designing and fabricating enclosures for electronic devices involves choosing the most suitable fabrication method. Various methods exist, ranging from traditional machining and molding techniques to more modern additive manufacturing processes. Among these, 3D printing has become one of the most accessible and cost-effective methods, especially for small-scale productions and prototypes.

Unlike traditional manufacturing processes that may require expensive tooling and setups, 3D printing has minimal upfront costs to get started. This makes it particularly attractive for rapid prototyping and custom design iterations. The ability to create complex geometries and designs with relative ease also contributes to its popularity in the field of electronics.

In this chapter, the process of designing and fabricating the enclosure for the employee time tracking device using 3D modeling and printing techniques is explored. The chapter discusses the design considerations, the software tools used for modeling, and the intricacies of the 3D printing process. Additionally, it examines the challenges encountered and the solutions implemented during this phase.

6.1. 3D Modeling

3D modeling plays a crucial role in the design and fabrication of the enclosure for the employee time tracking device. It enables precise visualization and customization of the physical form of the device, accommodating its electronic components and ensuring functional and aesthetic requirements are met. This section explores the process of 3D modeling, detailing the software tools utilized, the design considerations, and the iterative development of the enclosure design.

6.1.1. An Iterative Process

The first step in designing the enclosure was to decide on a general shape. The primary objective was to create an enclosure that was as compact as possible while remaining visually appealing and functional. A rectangular shape was chosen, with the layout designed to maximize usability and accessibility. The top third of the enclosure would be occupied by the screen, providing a clear display for users, while the lower two-thirds would house the NFC reader, the Raspberry Pi, and the PCB. This can be seen in Figure 6.1.

Figure 6.1: Enclosure view from the top

To bring this design to life, Fusion 360[1], developed by Autodesk, was selected as the modeling software. Fusion 360 is an industry-leading solution renowned for its versatility and widespread use in 3D modeling for various applications, including product design and engineering.

Ensuring the secure attachment of the components to the enclosure was critical. The mounting holes present in the NFC reader's board, the display, and the PCB were utilized for this purpose. Precise measurements were taken using a caliper to model the studs accurately in Fusion 360. These measurements ensured that the components would fit snugly and securely within the enclosure. Considering that the device had to be as compact as possible, the NFC reader and the PCB were stacked one on top of the other, making for an extremely compact design.

In the first design iteration, threaded studs were incorporated into the model. Corresponding nuts were also designed to secure these studs. Although the threads functioned as intended, they proved to be problematic. The studs, being less than 3 millimeters in diameter, were too fragile. They tended to break easily if the enclosure was dropped, and the threads often sheared off, making it difficult to release the components without breaking the studs. This initial challenge highlighted the need for a more robust solution in subsequent design iterations.

[1] Fusion 360's Website: `https://www.autodesk.es/products/fusion-360/overview`

It is also worth noting that even with the studs at the full 3-millimeter diameter, they remained fragile and prone to breaking when dropped. This issue manifested during testing, necessitating a more robust solution.

Consequently, walls were added to surround the NFC board and the PCB. These walls featured extra support on their lower half but lacked support near the top. This design allowed the walls to act like springs during a drop, absorbing impacts and preventing the boards from breaking the studs. This solution proved highly effective and was incorporated into the final design. This can be seen in Figure 6.2.

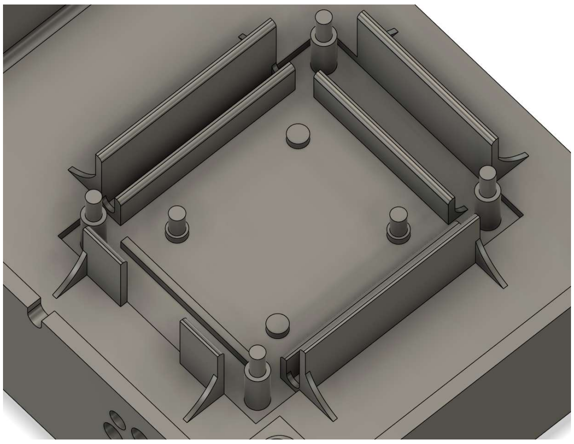

Figure 6.2: Enclosure's "walls"

However, a new issue arose with the removal of the threaded studs: there was no way to prevent the components from moving up and down or shooting out of their positions. The solution came in the form of a backplate that would be screwed on the back of the enclosure. The height of the device was designed so that the backplate would press the Raspberry Pi Pico against the PCB and the base of the studs, preventing any movement. For the LCD display, four hollow cylinders were designed to press against the back of the LCD, ensuring it remained securely in place. This can be seen in Figure 6.3. Notably, the LCD display did not require additional walls for support, as it was already held in place by protruding outside the enclosure. In this way, the final design successfully addressed the issues of component stability and impact resistance, ensuring a robust and reliable enclosure for the device.

Another challenging aspect of the design was the mounting mechanism for attaching the backplate to the enclosure. Ultimately, it was decided that both parts would be secured together using screws, which would be modeled and 3D printed for simplicity. This design required incorporating a threaded section on the enclosure side to tighten the screws.

Additionally, there is a recess surrounding the screw holes on the backplate. This recess accommodates the screw heads, ensuring they fit flush within the backplate and do not protrude, which allows for flat mounting to a wall.

Another important feature of the backplate is the perimeter that protrudes along its edge, as can be seen in Figure 6.3. This perimeter holds the backplate laterally, so the screws only need to keep the backplate and the enclosure together, without

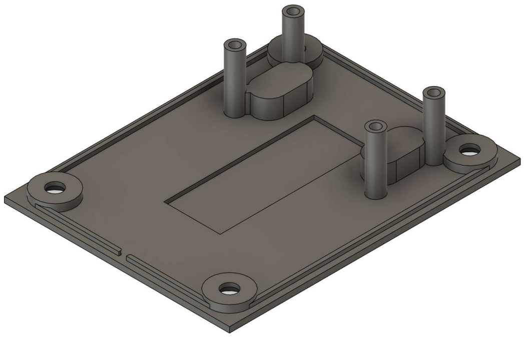

Figure 6.3: Front side of the enclosure's backplate

Figure 6.4: Rear side of the enclosure's backplate

bearing any lateral forces. This design detail enhances the structural integrity and durability of the assembled enclosure. Moreover, mounting holes were added, for the device to be wall-mounted. This detail can be seen in Figure 6.4.

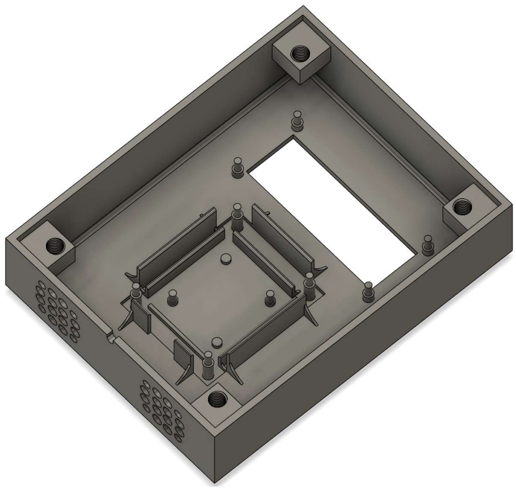

Figure 6.5: Enclosure view from the back

Details have already been provided regarding the mounting mechanism for the backplate to the enclosure. Now, the modifications on the enclosure-side will be explored.

A separate component was designed specifically to contain the threads that the screws would screw into. These threaded components were then placed in all four corners of the enclosure, positioned at the correct distance from the top to ensure the backplate fits flush against them. The individual threaded component is illustrated in Figure 6.6, while all four components integrated into the enclosure are shown in Figure 6.5.

Additionally, the screws were 3D printed and modeled using the threading tool in Fusion 360. This tool facilitated the creation of an M8 threaded screw, with a

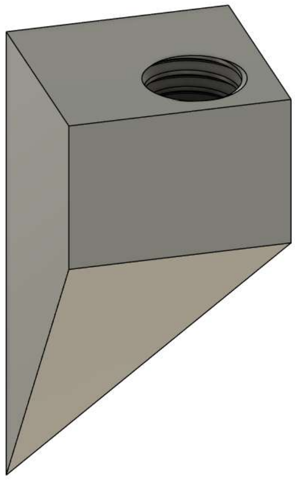

Figure 6.6: Enclosure's mounting component

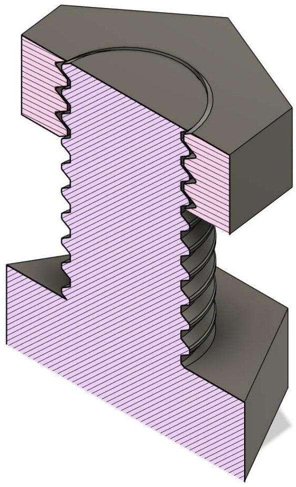

Figure 6.7: Screw's cross section

diameter of 8 millimeters. To ensure functionality, the screw was tested with a nut to verify that it could be smoothly screwed in and reused. The cross-section of the screw and nut assembly is illustrated in Figure 6.7.

The final design details include ventilation holes at the bottom of the enclosure and a slot for cable routing. The ventilation holes are intended to allow air circulation, particularly if the enclosure is exposed to direct sunlight, to prevent heat buildup. The slot provides a pathway for the power cable. Both features are illustrated in Figure 6.5.

6.2. 3D Printing

After completing the design of the enclosure, the next crucial step is to bring the digital model into the physical world through 3D printing. It is important to note that 3D modeling and printing were not strictly sequential processes but rather iterative and concurrent. This allowed for rapid prototyping and real-time adjustments to the design based on the physical prototypes. In this section, the focus will be on the 3D printing process itself, including the selection of materials, the configuration of the printer, and the iterative testing required to achieve a functional and durable final product. The flexibility and precision of 3D printing make it an ideal method for fabricating custom enclosures for electronic devices, enabling quick refinements and improvements throughout the development process.

There are two primary types of 3D printing technologies: *Fused Deposition Modeling* (FDM) and *Stereolithography* (SLA). FDM printers work by melting and extruding thermoplastic filament through a heated nozzle, which deposits the material layer by layer to build up the object. This method is known for its affordability, ease of use, and the ability to produce durable parts, making it ideal for creating functional prototypes and larger objects. However, FDM prints may have visible layer lines and may require post-processing for a smoother finish.

On the other hand, SLA printers use a laser or projector to cure liquid resin layer by layer. The resin hardens upon exposure to light, resulting in highly detailed and smooth parts. While SLA printing is excellent for intricate designs and small parts due to its high resolution, it can be more expensive because of the cost of resin and the need for post-processing. Additionally, the build volume of SLA printers is typically smaller compared to FDM printers [12].

For this project, FDM printing is the better choice. It is more cost-effective, with both printers and filament materials being more affordable. The thermoplastic materials used in FDM printing are durable and suitable for functional parts that need to withstand handling and usage. Furthermore, FDM printers are easier to set up and operate, which is beneficial for in-house production. They also offer larger build volumes, allowing for the printing of larger enclosures in a single piece, reducing the need for assembly. Overall, FDM printing provides a practical balance of cost, durability, and ease of use, making it well-suited for producing the enclosures needed for this project.

6.2.1. Choosing a 3D Printer

Given that the production of the enclosures would be done in-house, it was necessary to purchase a 3D printer. The market for 3D printers spans a wide range of price points, from approximately 200€ to several thousand euros.

For the sake of simplicity, 3D printers can be categorized into three general types based on their price points: budget (200€ to 600€), mid-range (600€ to 1500€), and high-end (1500€ and above). While this is a broad approximation and may not apply to all printers (and in fact, it does not even properly cover industrial 3D printers, which may be well above 10.000€), it serves as a useful framework for understanding the options available on the market for this project.

The following information is based on merging many sources and adapting them to this project [14] [26] [17].

3D Printers in the Range of 200 to 600 Euros

- **Capabilities**: Printers in this price range are typically entry-level models suitable for beginners and hobbyists. They offer basic functionalities for printing small to medium-sized objects with decent accuracy.
- **Features**: These printers usually have a single extruder and a limited build volume. They may lack advanced features such as automatic bed leveling, filament sensors, and Wi-Fi connectivity.
- **Limitations**: While affordable, printers in this range may have lower build quality, resulting in less reliable prints. They may require more manual calibration and maintenance compared to higher-end models.

3D Printers in the Range of 600 to 1500 Euros

- **Capabilities**: Printers in this price range offer improved build quality, reliability, and features compared to entry-level models.
- **Features**: These printers may include features like automatic bed leveling, filament sensors, touchscreen displays, and larger build volumes. They offer better print quality and consistency.
- **Limitations**: While more capable than budget models, mid-range printers may still lack some advanced features found in higher-end machines. They may require occasional calibration and maintenance but generally offer a good balance between affordability and performance.

3D Printers Priced at 1500 Euros and Above

- **Capabilities**: High-end printers are designed for professionals, industrial users, and those with demanding printing needs. They offer the highest build quality, reliability, and precision.
- **Features**: These printers often come equipped with advanced features such as multiple extruders for multi-material printing, enclosed build chambers for better temperature control, and compatibility with a wide range of filament types.
- **Limitations**: While offering unparalleled performance, printers in this price range come at a premium cost. They may require significant investment upfront and may be overkill for casual users or hobbyists with limited budgets.

The Final Decision

The final decision was to opt for a 3D printer in the mid-range price category, around 1000 euros. This range was chosen to balance cost with the necessary features and reliability for in-house production. Two prominent companies stood out in this price range: *Prusa*[2] and *BambuLab*[3].

Prusa, founded by Josef Prusa, is renowned for its high-quality, reliable 3D printers. Prusa printers are particularly known for their user-friendly design, open-source philosophy, and robust community support. Prusa is based in the Czech Republic.

BambuLab, on the other hand, is a newer entrant in the 3D printing market but has quickly gained a reputation for innovation and high-performance machines. BambuLab focuses on integrating advanced features such as multiple extruders, high-speed printing, and sophisticated software to enhance the printing experience. BambuLab is based in China.

[2] Prusa's website: `https://www.prusa3d.com/`

[3] BambuLab's website: `https://bambulab.com/en-eu`

In the end, despite BambuLab offering a faster printer that might require less tinkering, the final decision was to go with a Prusa printer. This choice was driven by several key factors. Prusa is renowned for its comprehensive support and availability of parts, ensuring that maintenance and repairs can be handled efficiently. Their commitment to offering extensive resources and a robust support network provides a level of reliability and assurance that is critical for sustained, in-house production.

While BambuLab presents an attractive option with its innovative features and high-speed printing capabilities, it has not been around long enough to establish the same level of trust and reliability as Prusa. The long-standing reputation of Prusa for quality and support, combined with their open-source approach and active user community, made it the more secure and dependable choice for this project.

The newest printer that Prusa sells in the selected price range is the Prusa MK4, which was purchased with the enclosure. The printer can be seen in Figure 6.8. The reason for purchasing the enclosure is that it allows printing materials that may be sensible to sudden changes in temperature, which are usually caused by rushes of air close to the printer.

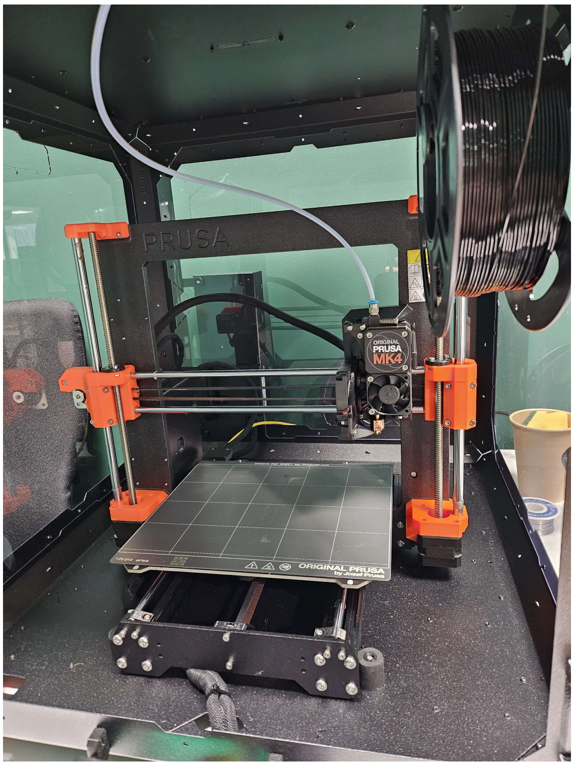

Figure 6.8: Prusa MK4 in its enclosure

6.2.2. Materials

Choosing the right material for 3D printing is critical to achieving the desired properties in the final product. The selection depends on various factors, including strength, flexibility, heat resistance, and ease of printing. Here is a brief overview of some of the most commonly used materials in 3D printing [13]:

PLA (Polylactic Acid): PLA is one of the most popular materials for 3D printing due to its ease of use. It is made from renewable resources like corn starch and sugarcane. PLA is great for beginners because it prints at lower temperatures and does not require a heated bed. However, it is not very heat-resistant and can become brittle over time.

ABS (Acrylonitrile Butadiene Styrene): ABS is a strong and durable plastic often used in industrial applications. It has better heat resistance compared to PLA but can be more challenging to print with due to its tendency to warp. ABS emits fumes during printing, so it requires a well-ventilated area or an enclosed printer with ventilation.

PETG (Polyethylene Terephthalate Glycol): PETG is a versatile material that combines the ease of printing found in PLA with the strength and durability of ABS. It is also resistant to moisture, chemicals and UV light, making it suitable for functional parts. PETG does not warp as much as ABS and emits fewer fumes, making it a safer choice for indoor printing.

Nylon: Nylon is known for its excellent strength, flexibility, and durability. It is commonly used for producing parts that need to withstand mechanical stress. Nylon is more difficult to print with due to its tendency to absorb moisture from the air, which can affect print quality. Proper storage and drying are essential when using nylon.

TPU (Thermoplastic Polyurethane): TPU is a flexible, rubber-like material that is ideal for printing items that need to be elastic and impact-resistant. It is commonly used for phone cases, gaskets, and wearables. TPU can be challenging to print due to its flexibility, requiring careful adjustments to print settings and slower print speeds.

Composites: Composite filaments are materials infused with fibers or powders, such as carbon fiber, wood, or metal. These composites can provide unique properties like increased strength, reduced weight, or aesthetic finishes. However, they can be abrasive and may require hardened steel nozzles to prevent wear during printing.

After evaluating the various materials, PETG emerged as the clear choice for the project. This decision was driven by several key factors. Given that some devices may be exposed to direct sunlight and high temperatures, PETG's excellent resistance to UV radiation and heat made it the most suitable material. Additionally, PETG offers a good balance of strength, flexibility, and ease of printing, which are crucial for creating durable and reliable enclosures. Its moisture and chemical resistance further enhance its suitability for this application, ensuring that the devices remain robust and functional in various environmental conditions.

6.2.3. Printing

With the design finalized and the material chosen, the next step is to bring the digital model to life through 3D printing. This section will cover the various aspects involved in the printing process, including printer setup, calibration and slicing the model. The aim is to ensure that the printed enclosures meet the required specifications and quality standards.

6.2.3.1. Printer Setup and Calibration

One of the standout aspects of the Prusa MK4 is its minimal setup and calibration requirements, thanks to its integrated sensors and automated processes. Upon receiving the Prusa MK4, the initial assembly is straightforward and well-documented, with comprehensive guides provided by Prusa Research.

The Prusa MK4 is equipped with an advanced bed leveling system that uses a mesh bed leveling sensor. This sensor measures the distance between the print bed and the nozzle at multiple points, creating a precise mesh that ensures the first layer is perfectly even. This automated process eliminates the need for manual bed leveling, which is a common challenge with many 3D printers.

Additionally, the printer features a filament sensor that detects the presence of filament and can pause the print if the filament runs out or jams. This feature is particularly useful for long prints, ensuring that the print does not fail due to filament issues.

The Prusa MK4 includes a self-test and calibration wizard that guides users through the initial setup. The wizard checks all critical components, such as the print bed, extruder, and sensors, ensuring everything is functioning correctly before starting a print.

Due to these advanced features, the Prusa MK4 requires very little manual setup and calibration. Users can rely on the automated systems to handle most of the initial adjustments, allowing them to focus on the design and printing process rather than troubleshooting and fine-tuning the printer.

Furthermore, the Prusa MK4 comes with pre-configured print profiles in the *PrusaSlicer* software. These profiles are optimized for various filament types, including PETG, and take into account the printer's capabilities, ensuring high-quality prints with minimal manual adjustments.

However, while the stock printing profile of the Prusa MK4 is highly optimized for general use, some adjustments were necessary to achieve optimal performance for the project's specific requirements. These modifications ensured better print quality and ease of use, particularly for producing PETG enclosures.

Firstly, the nozzle height was adjusted slightly by raising it 0.05mm from the bed. The default setting caused the nozzle to print too close to the bed, resulting in the first layer rippling due to excessive compression. This minor adjustment improved the first layer's smoothness and adhesion, providing a more consistent foundation for subsequent layers.

Additionally, the bed temperature settings were lowered. The default temperatures provided excellent adhesion, but this sometimes resulted in the PETG enclosure adhering too strongly to the print bed, making it difficult to remove without damaging the print or the bed. By reducing the bed temperature, sufficient adhesion was maintained for successful printing, while also allowing for easier removal of the completed prints.

6.2.3.2. Slicing the 3D Model

Before sending a 3D model to a 3D printer for fabrication, it must go through a crucial step known as slicing. Slicing is the process of converting a 3D model (typically in STL format) into a set of instructions (G-code) that a 3D printer can understand. This G-code provides specific instructions on how the printer should move its print head and build platform to create the desired object layer by layer.

A slicer software plays a central role in this process. It takes into account various parameters set by the user, such as layer height, infill density, print speed, and support structures, and generates the G-code accordingly. These parameters directly influence the quality, strength, and appearance of the final print.

Different slicer software packages are available, each with its own set of features and capabilities. Some popular slicers include Ultimaker Cura, PrusaSlicer, Simplify3D, and Slic3r. These programs offer user-friendly interfaces and a range of customization options to optimize prints for specific materials, print quality, and printer hardware.

For this project, PrusaSlicer was employed, as can be seen in Figures 6.9 and 6.10.

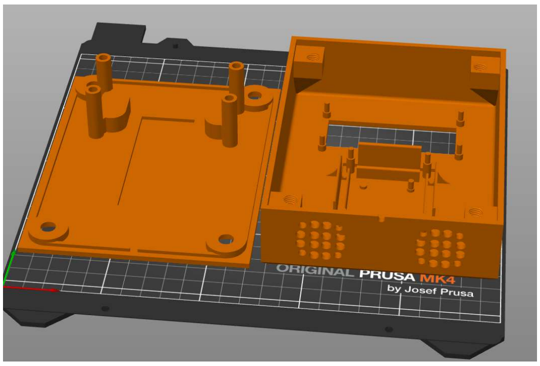

Figure 6.9: Objects placed on the bed on PrusaSlicer

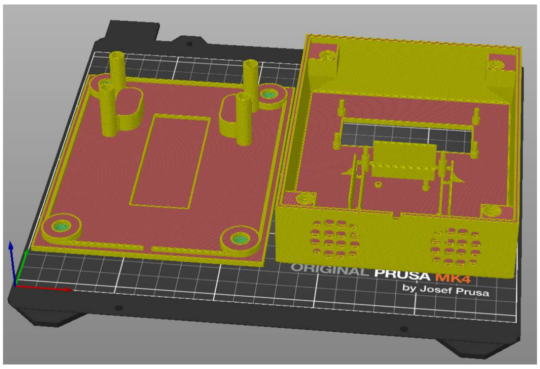

Figure 6.10: Sliced objects on PrusaSlicer

6.2.3.3. Printing and Post-Processing

With the 3D model sliced and the printer calibrated, the final step is straightforward: pressing start and waiting for the printer to finish. Once the printing process is complete, the next task involves removing the supports from the backplate. These supports are necessary due to the recesses where the screws fit, which are elevated from the print bed.

After completing this post-processing step, the printed enclosure is ready for use. The successful integration of the device into its newly printed enclosure can be seen in Figure 6.11, demonstrating the practical realization of the design and fabrication plan.

Figure 6.11: Device in its enclosure

Chapter 7

Cost Analysis and Competitive Comparison

The primary motivation for undertaking this project was to significantly reduce costs associated with the previous solution, which relied on mobile phones. This chapter will provide a detailed analysis of the costs incurred during the development and implementation of the new device, alongside a comparison with the costs of the previous mobile phone-based solution. By examining both the financial aspects and the competitive landscape, this analysis aims to demonstrate the economic advantages of the newly developed system and its potential for broader application.

7.1. The Previous Solution

The previous solution for processing employee clock-ins involved using mobile phones mounted on walls, equipped with an application from a software provider designed for this purpose. While the application itself functioned effectively, the hardware presented significant issues. These phones, which were constantly plugged into power sources and often exposed to direct sunlight, suffered from severe battery degradation. Within six months, the batteries would swell and become unusable, necessitating frequent replacements.

Even if these phones were on the lower-end of performance, their cost was not negligible, and with the need for replacements every six months, the expenses quickly escalated. Additionally, the mounting cases used for the phones were notably brittle and unable to withstand impacts from falls. Despite their cost not being nearly as high as the phones, they were too costly for the purpose they served, adding to the overall expense and contributing to the recurring costs of maintaining the system.

This combination of frequent hardware failures and the associated costs highlighted the need for a more durable and cost-effective solution, prompting the development of the new device detailed in this project.

Component	Approximate Cost
Raspberry Pi Pico WH	8€
Waveshare SIM7020E Board	20€
NFC Reader	7€
LCD Display	8€
1kg of PETG	20€

Table 7.1: Pricing table

7.2. The Proposed Solution

The proposed solution aims not only to significantly reduce costs but also to greatly enhance the longevity of the system. The new device is designed with durability and modularity in mind, ensuring that it can withstand prolonged use and adverse conditions far better than the previous phone-based system.

It is estimated that the electronics in the device should last at least three years, a substantial improvement over the six-month lifespan of the phones previously used. This extended longevity translates into fewer replacements and lower long-term costs. Additionally, in the event of a component failure, the modular design allows for the replacement of the specific faulty part rather than discarding the entire device. This repairability further contributes to cost savings and reduces electronic waste.

By addressing both the economic and operational shortcomings of the previous system, the proposed solution offers a more sustainable and efficient approach to managing employee clock-ins.

7.2.1. Pricing

To maintain confidentiality regarding proprietary costs, this analysis will focus exclusively on the prices of publicly available parts. For example, while custom PCB costs are excluded due to their sensitive nature, prices for readily available components such as the Raspberry Pi Pico, NFC reader, and 3D printing materials will be included. This approach ensures that the analysis remains transparent and informative without compromising any sensitive financial information.

The table with approximate pricing for each component at the time of publication can be seen on Table 7.1. Consider that the enclosure's weight is significantly less than 1 kilogram, and that costs related to assembling the device and developing software are not being included.

Chapter 8

Conclusions and Future Work

The project has been a great success and is currently being rolled out to production. This achievement marks a significant milestone in the pursuit of creating a more cost-effective and durable solution for employee clocking systems.

In conclusion, the development process—from the initial concept through to the final deployment—has demonstrated the effectiveness of meticulous planning, innovative design, and iterative testing. Key accomplishments of the project include:

- **Cost Efficiency**: The newly developed system significantly reduces costs compared to the previous solution using mobile phones. By utilizing a Raspberry Pi Pico and custom-designed enclosures, the project has lowered both initial and long-term expenses.
- **Enhanced Longevity**: The robust design of the new device promises a much longer lifespan than the mobile phones previously used. The components are expected to last at least three years, and in case of any failure, individual parts can be replaced rather than discarding the entire device.
- **Customizability and Flexibility**: The modular nature of the system allows for easy upgrades and customization. This adaptability ensures the system can evolve with changing technological requirements and company needs. This also applies to the code, which can be modified if there are changes in requirements.
- **Improved Reliability**: The new enclosure and internal design provide superior protection against environmental factors, such as heat and impact, which plagued the mobile phone-based solution.
- **Advanced Functionality**: The integration with Google Cloud Platform and the use of Spring for API development have ensured a seamless and secure data transmission process, enhancing the overall functionality and reliability of the system.

The project's success underscores the importance of selecting appropriate materials, technologies, and design methodologies. The choice of PETG for the enclosure, the adoption of 3D printing for rapid prototyping, and the iterative design process have all contributed to the creation of a highly effective solution.

As the rollout continues, ongoing monitoring and feedback will be essential to ensure the system meets all operational requirements and continues to perform as expected. Future improvements may include further optimization of the design, enhancements in software functionality, and potential expansion of the system's capabilities.

This thesis not only documents the journey from concept to deployment but also serves as a guide for similar projects aiming to leverage technology for practical, cost-effective solutions. The insights gained and the methodologies developed during this project can be applied to a wide range of applications beyond employee clocking systems, demonstrating the broader impact and potential of this work.

8.1. Future Work

The development of this project is still ongoing, with new improvements being added on a daily basis. There is a comprehensive roadmap for future development, including implementing remote updates for the device's code, which will significantly enhance its maintainability and flexibility.

Very shortly, the devices will be updated to periodically send "keepalive" messages to the server. These messages will not only confirm that the device is operational but will also report any ongoing hardware issues. This proactive approach will enable quick identification and resolution of potential problems, ensuring the devices operate smoothly and reliably.

However, several challenges remain. The limited memory capacity of the Raspberry Pi Pico and the intermittent connectivity issues with Google Cloud have been significant obstacles. Addressing these problems is not straightforward, but overcoming them could unlock immense future opportunities.

The optimal solution to these issues involves designing a custom PCB that integrates both the microcontroller and the cellular connectivity module. This ambitious endeavor would require months of meticulous planning, prototyping, and testing, but it would represent a substantial upgrade to the current system.

Additionally, it is likely that the microcontroller will need to be upgraded to a model with greater memory capacity, such as the ESP32-S3. The ESP32-S3 not only offers more memory but also provides similar functionality, making it a suitable replacement for the Raspberry Pi Pico.

While the process of designing this integrated PCB has not yet begun, and discussions about its feasibility are ongoing, it is clear that such a development would greatly enhance the device's capabilities. This future work promises to further improve the reliability, efficiency, and overall performance of the system, building on the significant advantages already realized in the current project.

Bibliography

[1] Raspberry Pi Documentation. https://www.raspberrypi.com/documentation.

[2] Adafruit. CircuitPython Documentation. https://docs.circuitpython.org/en/9.0.x/docs/index.html.

[3] AEPD. Guía sobre tratamientos de control de presencia mediante sistemas biométricos. https://www.aepd.es/guias/guia-control-presencia-biometrico.pdf.

[4] Balakirev, D. Four Ways Docker Boosts Enterprise Software Development. https://www.docker.com/blog/four-ways-docker-boosts-enterprise-software-development/.

[5] BOE. Real Decreto-ley 8/2019, de 8 de marzo, de medidas urgentes de protección social y de lucha contra la precariedad laboral en la jornada de trabajo. https://www.boe.es/buscar/act.php?id=BOE-A-2019-3481.

[6] CircuitBasics. Active Buzzers VS Passive Buzzers. https://www.circuitbasics.com/what-is-a-buzzer/.

[7] Cotton, B. 5 Benefits of a Container-First Approach to Software Development. https://www.docker.com/blog/5-benefits-of-a-container-first-approach-to-software-development/.

[8] DigiKey. How to use Perfboards. https://www.digikey.com/en/maker/blogs/2022/start-building-cleaner-perfboard-projects-using-these-simple-tips.

[9] Docker. Docker Documentation. https://docs.docker.com/reference/.

[10] ElecHouse. PN532 Board Schematics. . https://www.elechouse.com/elechouse/images/product/PN532_module_V3/PN532_shematic_drowing.pdf.

[11] ElecHouse. PN532 User Guide. . https://www.elechouse.com/elechouse/images/product/PN532_module_V3/PN532_%20Manual_V3.pdf.

[12] FormLabs. FDM vs. SLA: Compare Filament and Resin 3D Printers. . https://formlabs.com/blog/fdm-vs-sla-compare-types-of-3d-printers/.

[13] FormLabs. Guide to 3D Printing Materials: Types, Applications, and Properties. . https://formlabs.com/blog/3d-printing-materials/.

[14] Fusion3. HOW MUCH DOES A 3D PRINTER COST. https://www.fusion3design.com/how-much-does-a-3d-printer-cost/.

[15] GitHub. Deceiving exception handling messages when available memory is low. https://github.com/adafruit/Adafruit_CircuitPython_Requests/issues/138.

[16] Google. Google Cloud Functions. https://cloud.google.com/functions.

[17] MakeShaper. How Much Does a 3D Printer Cost? https://www.makeshaper.com/what-is-the-price-of-a-3d-printer.

[18] Mouser. Picking The Right Tool For The Job: MCU, SBC or FPGA? https://www.mouser.es/blog/picking-the-right-tool-for-the-job-mcu-sbc-or-fpga.

[19] NXP. I2C Bus Specification. . https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[20] NXP. PN532 Specification. . https://www.nxp.com/docs/en/nxp/data-sheets/PN532_C1.pdf.

[21] Romkey, J. PN532 I2C Information. https://i2cdevices.org/devices/pn532.

[22] SIMCOM. SIM7020E AT Commands Manual. . https://files.waveshare.com/upload/a/a4/SIM7020_Series_AT_Command_Manual_V1.03.pdf.

[23] SIMCOM. SIM7020E Datasheet. . https://files.waveshare.com/upload/e/e7/SIM7020E_SPEC_EN_190424.pdf.

[24] Spring. Spring Framework Documentation. https://docs.spring.io/spring-framework/reference/index.html.

[25] StackOverflow. SIM7020 - Error during execution of AT+CHTTPCON. https://stackoverflow.com/questions/70083705/sim7020-error-during-execution-of-atchttpcon-based-on-example.

[26] UltiMaker. How much does a 3D printer cost? https://ultimaker.com/es/learn/how-much-does-a-3d-printer-cost/.

[27] Waveshare. SIM7020E Board Wiki. https://www.waveshare.com/wiki/Pico-SIM7020E-NB-IoT.

Appendix A

Recycling High Density Polyethylene for 3D Printing

As the project was nearing the production phase, the Sustainability team proposed a collaboration to enhance environmental responsibility. Their suggestion was to explore the feasibility of 3D printing the device's enclosure using *High Density Polyethylene* (HDPE), sourced from the recycling of chemical containers used by the company for cleaning purposes.

In parallel, they established contact with the "Centro de Innovación en Economía Circular" (CIEC), a facility owned by the Community of Madrid, dedicated to promoting circular economy practices. The CIEC is equipped with a fabrication laboratory (*fablab*) that houses the necessary tools for recycling HDPE and converting it into 3D-printable filament.

With the support of the CIEC, the team gained access to the essential equipment for this project. The CIEC staff were extremely supportive, providing the machinery and guidance needed on-site, free of charge.

This appendix outlines the process of transforming plastic containers into 3D-printable filament. The following steps provide a summary of this process, which will be elaborated upon in subsequent sections:

- **Initial Preparation**: The first step involves cutting the HDPE containers into smaller pieces that can be fed into a plastic shredder.
- **Shredding**: These pieces are then shredded into fragments smaller than pea-sized particles, with additional post-processing as necessary.
- **Filament Production**: The shredded plastic is fed into a filament-making machine, which is configured to produce the final 3D-printable filament.

By repurposing recycled HDPE, this initiative not only supports the project's sustainability goals but also demonstrates a practical application of circular economy principles in modern manufacturing.

A.1. Initial Preparation

The initial preparation process, although the least time-consuming step in recycling HDPE, is crucial to ensure the quality of the final product. Without proper care, the filament produced can be contaminated to the point of being unusable.

First, it is essential to ensure that the chemicals previously stored in the containers are non-toxic. The team did not have the facilities to properly clean and neutralize hazardous chemicals, so containers that held dangerous substances were not used.

Additionally, many containers had visibly dirty sections. Including these contaminated parts in the recycling process was found to significantly increase impurities in the final filament. These impurities can compromise the print quality and mechanical properties, rendering the filament unsuitable for use.

Therefore, meticulous inspection and cleaning of the containers are necessary. All parts that cannot be thoroughly cleaned or show signs of significant contamination should be discarded. Although this might seem wasteful, ensuring the purity of the HDPE is critical for producing high-quality 3D printing filament. Ensuring that only clean, non-toxic containers are used is a key step in achieving a successful recycling process.

A.2. Shredding and Post-Processing

Shredding the plastic is straightforward with the *GP20 Plastic Shredder*[1] by *3devo*. However, due to HDPE's elastic nature and resistance to breaking apart, the shredder occasionally needed to reverse its blades to effectively cut through the material. This happens automatically nonetheless.

Regarding post-processing, most filaments require drying after shredding but before filamenting. This is because moisture in the plastic can lead to steam vapor during extrusion, which deforms the plastic and ruins the filament roll. Despite this, multiple attempts to remove water from the HDPE plastic using 3devo's AIRID dryer[2] yielded negligible results. The weight difference of a 500g bag of shredded filament after drying was only 0.5g, resulting in 499.5g, which could be attributed to residual plastic left in the dryer and falls within the margin of error.

Given this experimentation, it was concluded that drying the shredded plastic was unnecessary for this process. In Figure A.1, the final result of the shredding process can be seen, and in Figure A.2 both machines can be seen.

[1] GP20 Plastic Shredder: `https://www.3devo.com/gp20-plastic-shredder`

[2] AIRID dryer: `https://www.3devo.com/dryer`

Figure A.1: Shredded HDPE. Image provided by Gonzalo Alonso García

A.3. Filament Production

The filament production process was conducted using 3devo's *Filament Maker ONE Composer*[3]. This machine is highly versatile, allowing fine-tuning through the configuration of numerous parameters. The most relevant parameters and their impact on filament quality are detailed below.

Extrusion and Filament Diameter

The filamenting machine is equipped with a sensor that continuously measures the diameter of the extruded filament. To maintain a consistent diameter, the machine uses a puller mechanism. This puller clamps onto the filament and adjusts its speed: increasing the speed when the filament is too thick and decreasing it when the filament is too thin. This adjustment process stretches or compresses the filament accordingly to achieve the desired diameter. Additionally, the machine can produce a graphical representation of the filament's diameter over time, aiding in the monitoring and adjustment process. This graph can be seen in Figure A.3. It can be seen that the diamater is fairly unstable, missing by a considerable margin the targets set by the two horizontal lines, delimiting the 1.650 and 1.850 millimeters range.

[3]Filament Maker ONE Composer: `https://www.3devo.com/filament-maker-one-composer-precision`

Figure A.2: 3devo's shredder and dryer. Image provided by Gonzalo Alonso García

Heaters

The filamenting machine features four heating zones along an extruder screw that gradually pushes the shredded filament through the machine, ultimately extruding it at the end. These heating zones are sequentially numbered from 4 to 1, with Zone 4 being the entry point for the shredded plastic and Zone 1 being closest to the extruder.

3devo recommends an increasing temperature profile for HDPE, where the temperature is lowest at Zone 4 and highest at Zone 1. However, during testing, this setup resulted in an extremely uneven filament diameter and a visible texture, indicating incomplete melting and the presence of small plastic chunks.

Through extensive experimentation, the team discovered that a decreasing temperature profile produced superior results. Starting at approximately 220°C in Zone 4 and reducing the temperature to about 170°C in Zone 1 yielded a more consistent and smoother filament. It is speculated that this variation in optimal temperature profiles may be due to the recycled nature of the HDPE, which alters its melting properties compared to virgin material.

Cooling Fans

Immediately at the extruder's output, two cooling fans are positioned to cool the molten plastic as it exits. The speed of these fans needs to be meticulously adjusted to achieve optimal cooling. If the plastic is not cooled sufficiently, it remains too soft, causing the puller to crush it. Conversely, if the plastic is cooled too much, it becomes too hard, making it difficult for the puller to stretch it effectively.

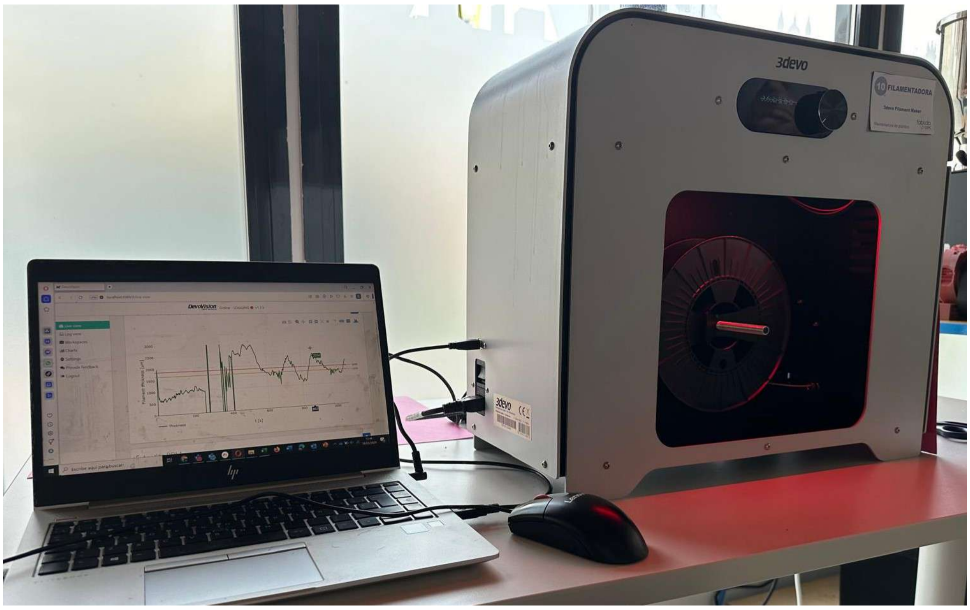

Figure A.3: Filament diameter graph with 100% recycled HDPE. Image provided by Gonzalo Alonso García

Furthermore, HDPE presents a unique challenge because it consists of two microstructures that shrink at different rates when cooled. Rapid cooling can exacerbate this differential shrinkage, leading to filament ovalization, a phenomenon where the filament cross-section becomes oval rather than circular. This issue is particularly critical because ovalized filament can lead to inconsistent extrusion in 3D printers, affecting the quality of the printed parts.

3devo emphasizes the importance of controlled cooling to mitigate these issues[4]. The fans must be set to a speed that ensures the plastic is solid enough for handling by the puller but avoids rapid cooling that could cause structural inconsistencies. Finding the right balance in fan speed is crucial for producing high-quality, round filament that meets the standards required for reliable 3D printing. Through careful tuning and testing, the team was able to achieve a cooling rate that maintained the filament's integrity and dimensional accuracy.

A.4. Challenges and Solutions

Throughout the filament production process, several challenges emerged. The initial batches of extruded filament displayed visible impurities, notably black spots, which resulted in clogging a 0.4 mm nozzle on the Prusa MK4 3D printer. These impurities are believed to stem from the residual dirt on the original plastic containers that were shredded. This underscores the necessity of using as-clean-as-possible plastic for shredding to ensure the quality of the final filament.

Another major hurdle was the instability in the filament's diameter. Variations

[4] 3devo's note on HDPE: `https://support.3devo.com/why-is-hdpe-difficult-to-extrude`

in diameter can drastically impact print quality, leading to issues such as under-extrusion when the filament is too thin, or even preventing the filament from fitting into the nozzle's 2 mm tube when it is too thick. To address this issue, the team experimented with blending virgin HDPE with the shredded recycled plastic.

A mixture of 70% recycled HDPE and 30% virgin HDPE yielded significantly more stable filament diameter, as illustrated in Figure A.4. Although a blend with 20% virgin HDPE also improved stability, it did not achieve the same level of consistency. This blend strategy proved to be a viable solution for mitigating diameter instability and improving the overall quality of the filament.

These solutions highlight the importance of both material purity and the strategic use of virgin plastic in producing high-quality, reliable 3D printing filament from recycled HDPE. By carefully addressing these challenges, the team was able to enhance the performance and usability of the recycled filament, paving the way for more sustainable and efficient 3D printing practices.

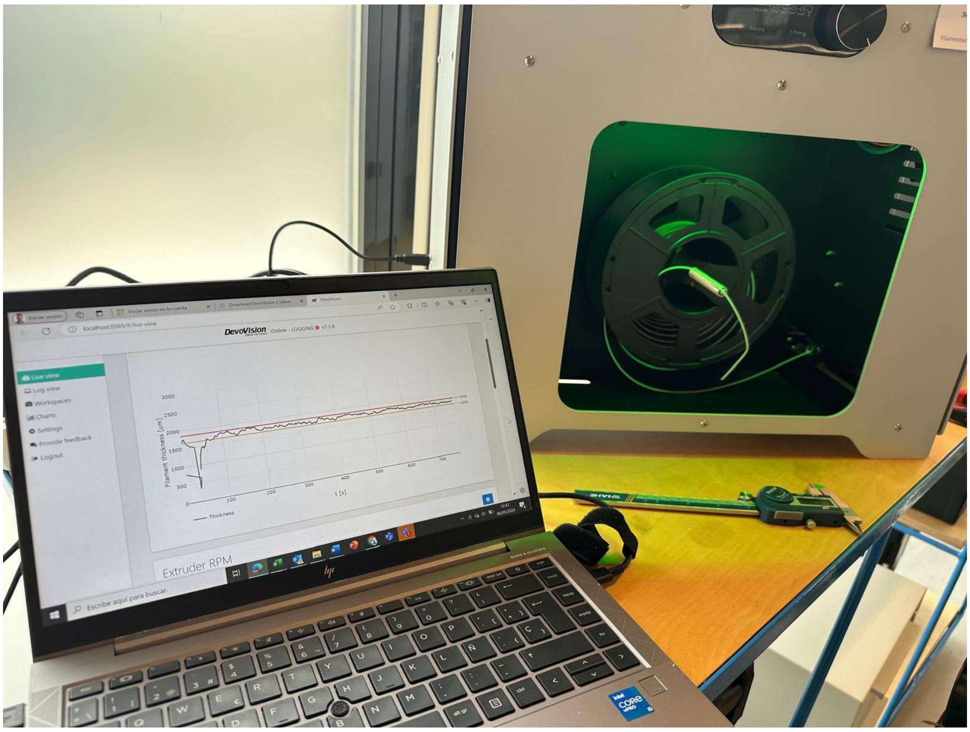

Figure A.4: Filament diameter graph after adding 30% virgin HDPE. Image provided by Gonzalo Alonso García

A.5. Conclusions

In conclusion, the exploration of recycling HDPE for 3D printing filament has been both challenging and enlightening. The initial stages of filament production highlighted critical issues such as impurities and diameter instability, both of which were addressed through meticulous preparation and the addition of virgin HDPE. These steps ensured a cleaner, more consistent filament suitable for 3D printing.

Despite the early setback of a clogged nozzle, the groundwork has been laid for further experimentation with 3D printing using this recycled filament. The next phase involves testing the printability of HDPE, addressing potential challenges such as warping and adhesion problems that are commonly associated with this material. With the setup now optimized, future trials will focus on overcoming these obstacles to fully realize the potential of recycled HDPE in 3D printing applications.

The collaboration with the Sustainability team and the support from the "Centro de Innovación en Economía Circular" have been invaluable, demonstrating the feasibility and benefits of integrating circular economy principles into technological projects.