# ESHO 1 Group 07 – Project Documentation


## Idea / Motivation

Have you ever dreamed about perfect on-point cooked eggs? And not just as lucky one in ten times event but every time? Our SmartEggTimer will satisfy every egg enthusiast. Our innovative smartEgg App calculates the perfect time to cook your eggs based on scientific research. But not just that, our clock also features a timer, a stopwatch, and an alarm clock — everything within a user-friendly and easy-to-use experience.

The idea of the project was to build a handy kitchen timer. The timer should be easy to use and should contain smart features. The heart should be a custom-built PCB with a powerful but energy-efficient STM32 microcontroller. The case of the timer should be completely 3D-printed.

## PCB

### Schematic, Selection of Parts

We started implementing our project idea by designing a schematic for our custom PCB. We got inspiration from a YouTube tutorial (https://www.youtube.com/watch?v=aVUqaB0IMh4&list=LL) and the provided extension board with the STM32F030CCT6. The schematic mainly consists of the following components:

#### Microcontroller: STM32F103RET6

We used the STM32 MCU Product Selector (https://www.st.com/content/st_com/en/stm32-mcu-product-selector.html) to choose this microcontroller. We chose a Cortex-M3 core with a maximum operating frequency of 72MHz because we wanted to have some performance headroom compared to the M0 on the extension board without a significant increase in price or power consumption. Additionally, we wanted slightly larger RAM and Flash storage space. Regarding the rest of the features, our microcontroller had to be similar or better to the STM32F0 on the extension board.

#### Voltage Regulator: Diodes Incorporated AP7361C-33E

To power our circuit, we use a Li-Pi battery (https://www.berrybase.de/lp-503562-lithium-polymer/lipo-akku-3-7v-1200mah-mit-2-pin-jst-stecker) and a simple TP4056 Charging board (https://www.berrybase.de/ladeplatine-fuer-3-7v-liion/lipo-akkus-mit-ausgang-usb-type-c-buchse-loetpads-1000ma) because we did not want to build our own BMS for safety reasons. We then needed a voltage regulator to deliver a constant voltage of 3.3V to our board at a reasonable current. We chose this one because of its

very low dropout voltage and because it can handle up to 1A of current.

### Battery Voltage Divider

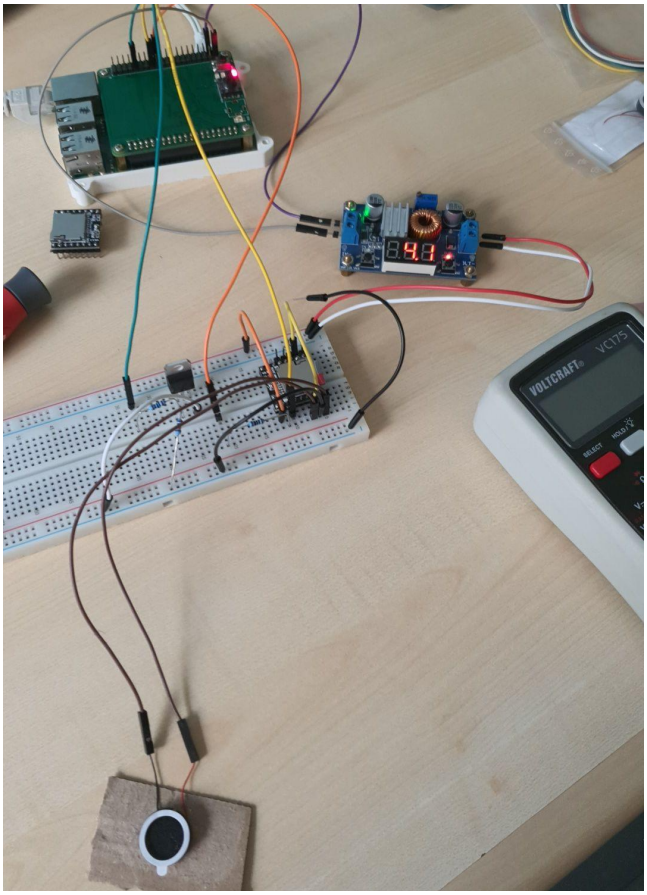
We wanted to utilize the STM's ADC to measure the battery voltage and calculate a rough estimate of the battery's charge level. As the battery voltage can go up to 4.2V and the STM is not capable of input voltages higher than 3.6V, we added a voltage divider to cut the battery voltage in half. However, this results in a constant current flow through the voltage divider, so we chose a relatively high resistance of 1 MΩ and added a 100nF capacitor, which acts as a 'buffer' and is charged up between the ADC's measurements. This results in a current flow of $I = \frac{U}{R} = \frac{4.2V}{2M\Omega} = 2.1\mu A$, which we consider managable for a 1200mAh battery. The hardware setup is inspired by this blog post (https://vivonomicon.com/2019/10/15/reading-battery-voltage-with-the-stm32s-adc/).

### Display

We used the 1.5inch OLED Module (https://www.waveshare.com/wiki/1.5inch_OLED_Module) from Waveshare. It features a 1.5-inch OLED screen with 16 different levels of grey. The display can be used with I2C and SPI. As we already knew the I2C protocol, we decided to use I2C. To make the display usable via I2C, we had to move two small 0 ohm resistors on the display's PCB and add another one (cf. here (https://www.waveshare.com/wiki/1.5inch_OLED_Module#Hardware_Configuration)). The display is connected to the STM via one of its I2C interfaces. Of course, we also added the necessary pull-up resistors.

### MP3 Player

To play alarm sounds, we decided to use a DFPlayer Mini (https://wiki.dfrobot.com/DFPlayer_Mini_SKU_DFR0299), a small MP3 player with a built-in amplifier and a UART interface. In the first place, we ordered a cheap clone of it, which turned out not to support all the features of the original one, so we later bought the proper one (https://www.berrybase.de/dfrobot-dfplayer-mini-mp3-player) instead. It is connected to a small speaker and one of the STM's UART interfaces. The DFPlayer is already equipped with an onboard voltage regulator, so we decided to power it with the unregulated battery voltage to avoid unnecessary load on the AP7361C-33E. The datasheet of the DFPlayer stated that it draws a constant standby current of about 20mA, which would drain our battery quite quickly. We, therefore, decided to add a MOSFET to turn the power on and off for the DFPlayer. However, it was quite challenging for us to find a MOSFET capable of letting through enough current at a gate-to-source voltage of 3.3V (or -3.3V), and is offered for sale in a German online store. We eventually found the STP60NF06L, which might be a bit overpowered for our needs but seems to do the job very well.
To ensure that this works properly, we implemented this part of the circuit on a breadboard:




*prototyping of the power circuit for the DFPlayer. The player is connected to a speaker and the M0 via UART (yellow and orange wires). A buck converter board is used to ensure that the DFPlayer can be powered with the battery voltage (red and white wires). The MOSFET is connected to a GPIO pin of the M0 (green wire).*

During prototyping, we observed that the UART communication between the DFPlayer and the M0 only works reliably in this setup if we add pull-up resistors to the RX and TX signals. Another difficulty we faced was that the breadboard we used sometimes did not make good contact between the power supply and the DFPlayer's VCC and GND pins, which led to the wrong assumption that the player could not be powered with lower voltages.

**Rotary Encoder and Tactile Switches**

We added some ordinary tactile switches and a rotary encoder so the user can control and set up the SmartEggTimer. We added pull-up resistors so that the buttons and the encoder connect to GND if pressed or turned. We also added RC low-pass filters to hardware-debounce the encoder and the switches. For the rotary encoder, we precautionally decided to use a weaker filter with a higher cut-off frequency because the datasheet only vaguely specified how long the pulses were. The outputs of the filters are connected to the STM in a way that each pin can be configured for an EXTI interrupt, respectively.

**Real-Time Clock**

The main functionality of our project is the timer, which should, therefore, be sufficiently accurate. We thought the microcontroller's internal oscillator might not be precise enough. Furthermore, we wanted our device to keep the current time even if it is turned off. The microcontroller itself already offers an RTC feature, but it requires an external oscillator and a backup power source. To reduce the cost of the PCB, we decided to use an external RTC instead. We got a module with a DS3231 (https://www.berrybase.de/ds3231-real-time-clock-modul-fuer-raspberry-pi), which already contains a backup battery. It is connected via I2C and uses the same bus as the display.

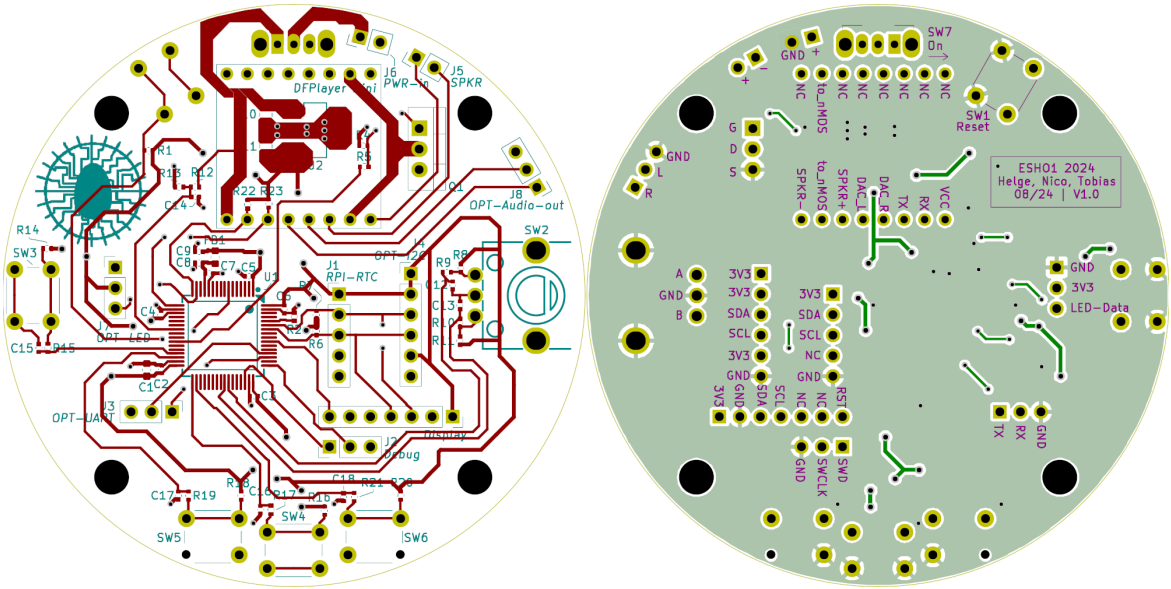
### Pin Headers

Lastly, we added pin headers to connect the battery power, the speaker, and a debugger. We also added pin headers for optional extensions (e.g., pressure sensor, LED, audio out jack).

## Board Design

Placing the components onto and routing the PCB was quite straightforward. Relevant aspects of the design process include:

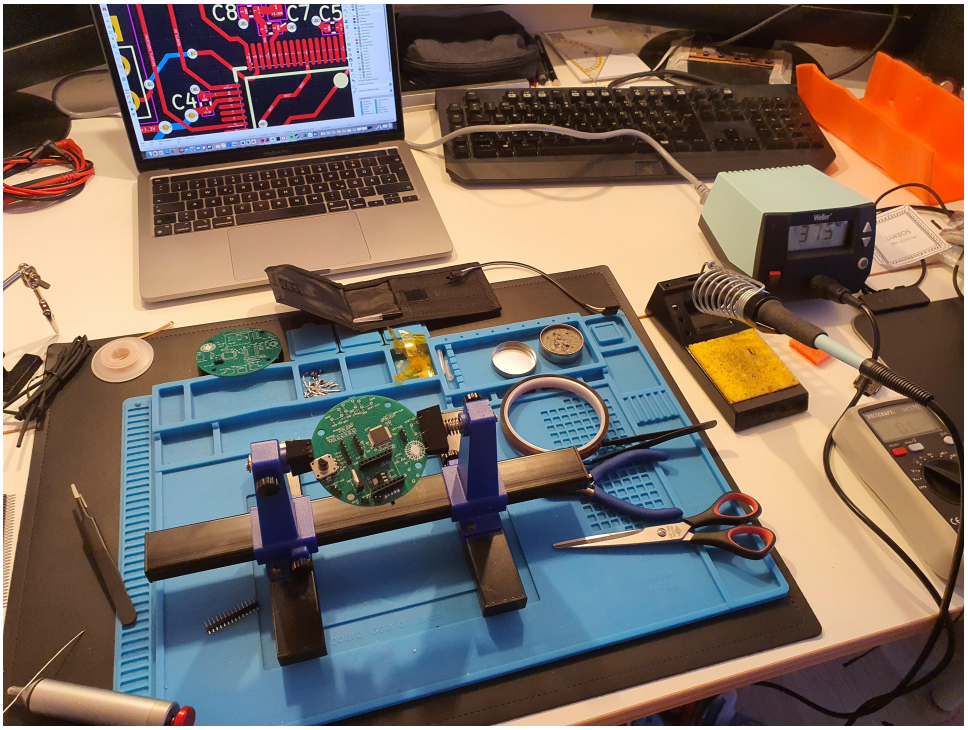
- We had to ensure that the placement of critical parts on the PCB, such as the switches, the rotary encoder, and the mounting holes, matched up with the case. This led to some constraints.
- As this was recommended in the tutorial we took as inspiration, we routed most of the traces on the front side and used most of the back side as a ground plane. We took care to use as few vias as possible and to only make a short trace on the back side if we had to jump a trace on the front side. Furthermore, we made sure to maintain a reasonable distance between parallel traces.
- To save some space, we moved the voltage regulator beneath the DFPlayer and parts of the microcontroller beneath the RTC.
- Because the DFPlayer acted quite temperamental with its power supply during prototyping, we added broad fill zones connecting the DFPlayer to the battery and to the MOSFET. This is probably unnecessary, and 0.5mm traces should be more than sufficient, but at least it looks quite funny. We also added some fill zones around the voltage regulator, which probably helps dissipate heat if the regulator ever gets warm.
- Two switches had to be placed relatively close to the edge of the PCB, so we edited the footprints and removed one pad, respectively. We also replaced the footprints of the microcontroller, the voltage regulator, and the rotary encoder with the footprints provided by JLCPCB to ensure they are correctly oriented upon ordering.




*Design of the front (left) and back (right) layers of the PCB.*

## Assembly and First Boot

Most of the PCB was assembled by JLCPCB. To save costs, we soldered the pin headers, the on/off switch, and the tactile switches manually:



*One of the PCBs during assembly*

After powering on the PCB for the first time, the microcontroller booted up properly. We connected it to the provided Raspberry Pi using the same GPIO pins for SWD and were able to flash the microcontroller using a slightly modified version of the config file from task one. One difficulty we faced at this point was that the SWD interface was not enabled by default for our microcontroller, so after flashing a simple test program, the microcontroller did not respond via SWD any longer. We got our board working again by dragging the BOOT0 pin of the microcontroller high, which we achieved by connecting the lower pin of R2 to 3.3V through a 1k resistor. This 'overwrote' the 10k pull-down resistor.

## Low-level software

### DF Player

Programming an interface for the DFPlayer was more involved than expected initially. Unfortunately, the manufacturer's documentation (https://wiki.dfrobot.com/DFPlayer_Mini_SKU_DFR0299) is partly crude and incorrect. We found a better datasheet here (https://github.com/DFRobot/DFRobotDFPlayerMini/blob/master/doc/FN-M16P%2BEmbedded%2BMP3%2BAudio%2BModule%2BDatasheet.pdf).

Not every DFPlayer behaves the same way or even supports the complete instruction set. Furthermore, the Arduino library the manufacturer provided seemed cluttered and did not really fit within our STM32 project. We thus wrote the interface entirely from scratch. During this, we had to deal with the following difficulties:

- DFPlayer requires an unavoidable delay between commands, otherwise the DFPlayer ignores some commands. To avoid using hard HAL_Delays all the time, we wait for an acknowledgment from the DFPlayer after each command is sent.
- Some commands only work if the DFPlayer is currently playing an audio file. Here, too, a certain amount of time elapses before DFPlayer actually starts playing a file once the "Play" command has been sent. Therefore, we have to check the actual status of the DFPlayer before sending some commands.
- DFPlayer always sends data after it has finished playing an audio file. However, we do not need this data and, therefore, do not receive it, as this would overcomplicate UART communication for our application. However, in this case, the DFPlayer may send individual

bits more than once after the next request, so we had to take special care to receive the messages from it properly.

## Display

Waveshare offers example code for the display on different platforms. We used this as a foundation and modified it to fit our needs and to get the display working. One main problem we had was the frame rate of the display. Our first implementation was really disappointing as it took about 10 seconds to paint the GUI on the screen. This was due to a misunderstanding of the GUI methods. With an easy fix, we quickly reached the maximum frame rate possible with the official library, which still resulted in a noticeable refresh rate. The library is inefficiently implemented, and we could nearly double the refresh rate as we started sending fewer commands with bigger parts of the image to the display at once instead of sending nearly pixel by pixel.
With the current implementation, it is still not enough to play games on it, but enough to display a timer.

Looking back, we maybe should have decided to use SPI for the display as it may have fewer issues with the data-transmission speed.

## RTC

The RTC was quite straightforward. We use an external RTC connected with I2C. The chip on the RTC handles all the date time-keeping functionality, and our clock does not 'forget' the time even if it is powered off, as the RTC contains a small battery. Our driver is inspired by the driver written by hasenradball (https://github.com/hasenradball/DS3231-RTC). As we use two I2C devices, we had some problems with concurrent access to the I2C devices. This is raised especially in the periodic timer interrupt to trigger the update of the timer on the display. We needed to get the time from the RTC in the interrupt, but often, the I2C bus was already blocked by other communication. We then decided to skip every interrupt while the i2c bus was blocked. This solved the issue, and we didn't notice any drawbacks.

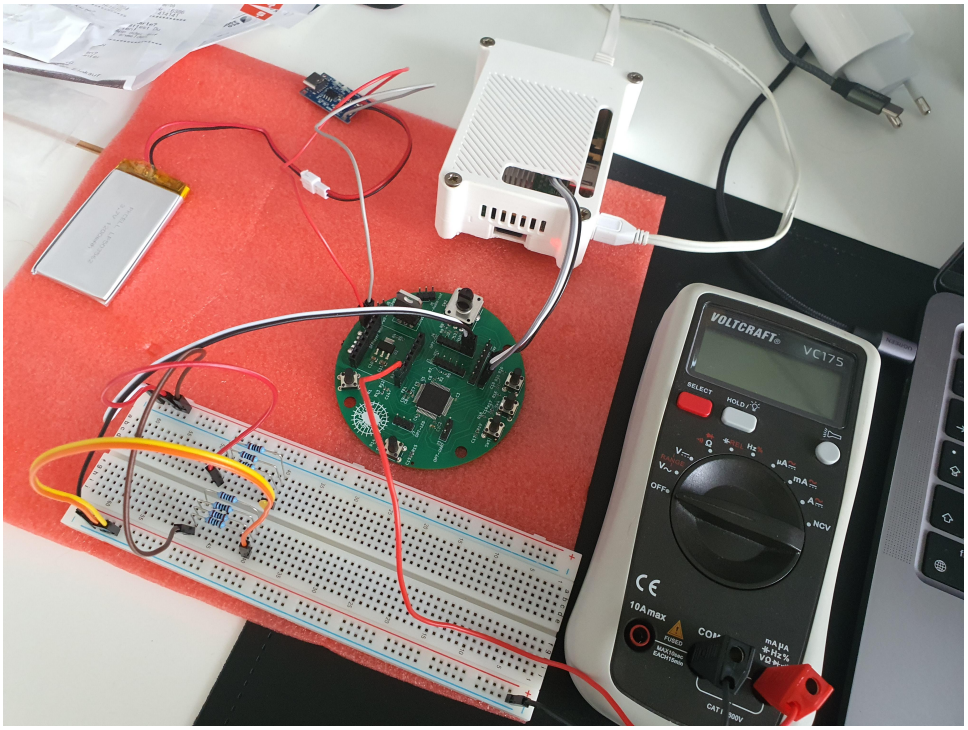
## Rotary Encoder

Each rotation of the rotary encoder triggers an EXTI interrupt. In this case, debouncing the rotary encoder correctly and preventing ghost inputs was more difficult. We use a combination of different solutions for this:

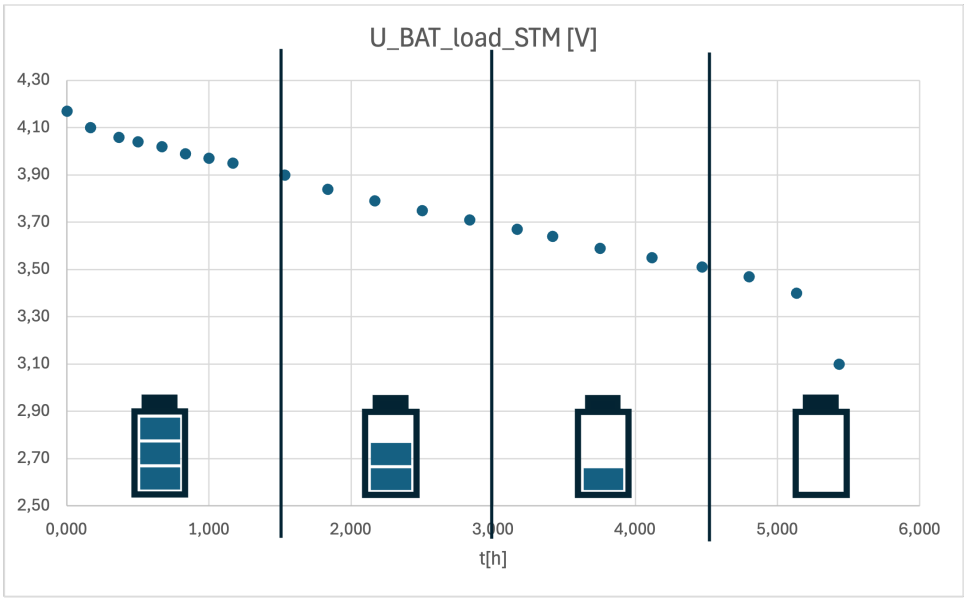
- We use the method presented here (http://www.technoblogy.com/show?1YHJ) to clean up bouncy inputs from the encoder.
- To prevent a single rotation from triggering multiple updates, we save the current value of HAL_GetTick for each rotation and ignore interrupts that were triggered too soon after the last accepted rotation
- The SysTick interrupt has a lower priority than the timer interrupt, which triggers an update of the display. This way, an update is only accepted after the display has been updated. This prevents the display from lagging behind.

## Measuring the Battery

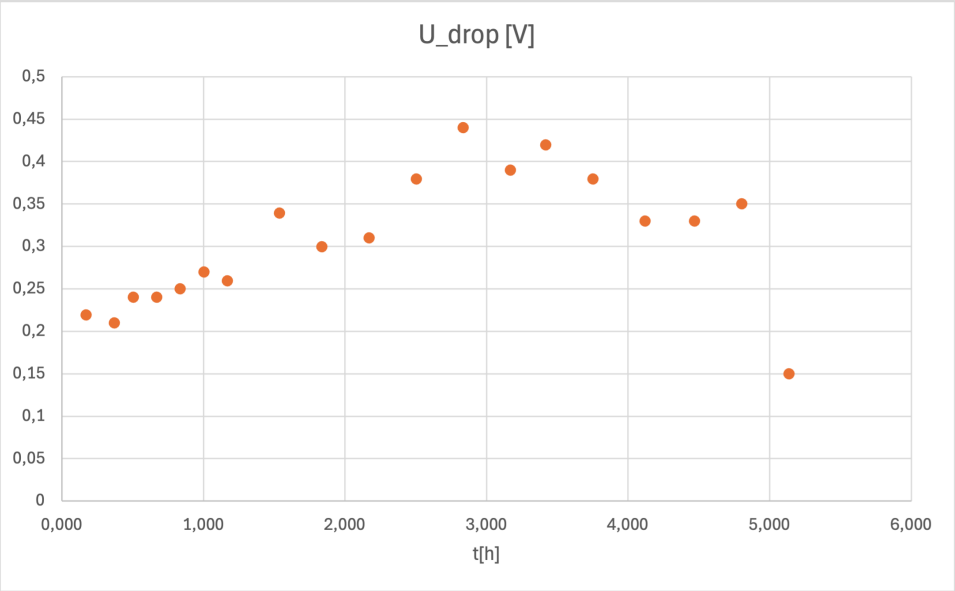
Reading the battery voltage with the ADC of the STM works without any problems. However, it is only possible to deduce the actual charge level from the current battery voltage to a limited extent. As we only use a simple BMS, the microcontroller has no further data about the battery. Nevertheless, in order to be able to make a qualitative and reasonably realistic estimate of the battery's charge level, we roughly determined the battery's voltage curve in a small experiment:




We slowly discharged the battery using resistors ($8\ 100\Omega$ resistors in parallel $\approx 12.5\Omega$ between the output of the voltage regulator and GND) while the microcontroller ran at 64 MHz and measured the voltage during this process.
Based on this curve, we divided the battery's state of charge into four parts ("full", "medium", "low", "empty" ):




We also measured the voltage drop caused by the load of the resistors:



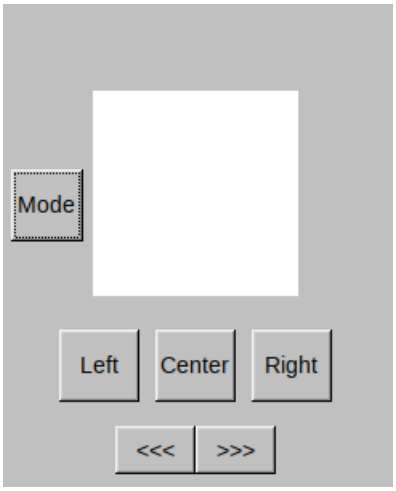

These values vary relatively widely but are mainly concentrated within an interval of about 0.25V-0.4V.
During measurement, we only powered the microcontroller (on at 64 MHz). In our final project, the display and RTC add a small additional and fairly constant load, for which we assume a constant voltage drop for the sake of simplicity.
As long as the DFPlayer is in use, the battery is subjected to a higher load, which fluctuates greatly depending on the current volume. Nevertheless, we also assume a generously estimated constant voltage drop here, and every time we measure the battery voltage, we check whether the DFPlayer is currently switched on.

To avoid fluctuations around the thresholds between two states of charge, only transitions from a higher state of charge to a lower state are allowed. This also ignores false and too-high results if the ADC reference voltage falls below 3.3V when the battery is very low.
After the battery has been charged, our device must be restarted or resetted to update the charge status.

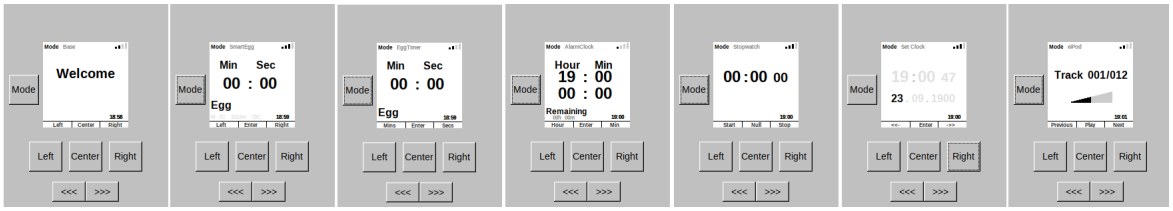
More detailed information and the specific values can be found in our repository (https://gitlab.esa.informatik.tu-darmstadt.de/esho1_24/Group07/-/tree/master/Project/Documentation/Battery?ref_type=heads).
In summary, the calculation of the charge status is based on many simplifying assumptions, but our tests have shown that this is more than sufficient for our application and gives the user a good idea of how long the battery will last.

## High-level Software overview

The high-level software provides the direct user interface and the functionality that justifies the project. It has to take into account the hardware constraints of the system regarding computational load as well as possible interactions. In the chapter regarding the PCB and the case, the reader will be informed about the hardware. For now, we will refer to an abstract system where the software is implemented on.
This system has a 128x128 Pixel monochrome display, a single mode button, three egg buttons, and a rotation input device. Aside from visual information, it can output acoustic information through a speaker.




We initially developed the high-level software on an emulated system using a GUI library on a personal computer. Using this emulated system, we programmed, designed, and debugged apps. This programming approach enabled quick iteration in all design and debugging. By keeping the emulated system as close to the real hardware constraints as possible and using proper abstraction in the C++ code, we could easily transfer the resulting applications to the actual hardware.




## Functionality user stories

### General

As a user of the Smart Egg Timer device, I want to know its basic state. This includes its current application, its battery state, the functionality of the buttons, and the current time. The current time of day must be manually adjustable. The functionality used on the Smart Egg Timer should be changeable.

### Smart Egg App

As a user of the Smart Egg App, I want to set a timer for cooking an egg by setting egg information instead of the time itself. The app should take necessary information like egg size, initial temperature, air pressure, and desired end temperature and compute the optimal cooking time. The optimal cooking time should then be counted down, and an alarm should attract attention once the time is up.

### Egg Timer App

As a user of the Egg Timer, I want to set a specific time for cooking an egg. The cooking time should then be counted down, and an alarm should attract attention once the time is up.

### Alarm Clock App

As a user of the Alarm Clock, I want to set a specific time of day. Once this time is reached, an alarm should be able to wake up a sleeping person. The app is supposed to show the current time of day, the alarm time, and the remaining time till the alarm is on the display.

### Stopwatch App

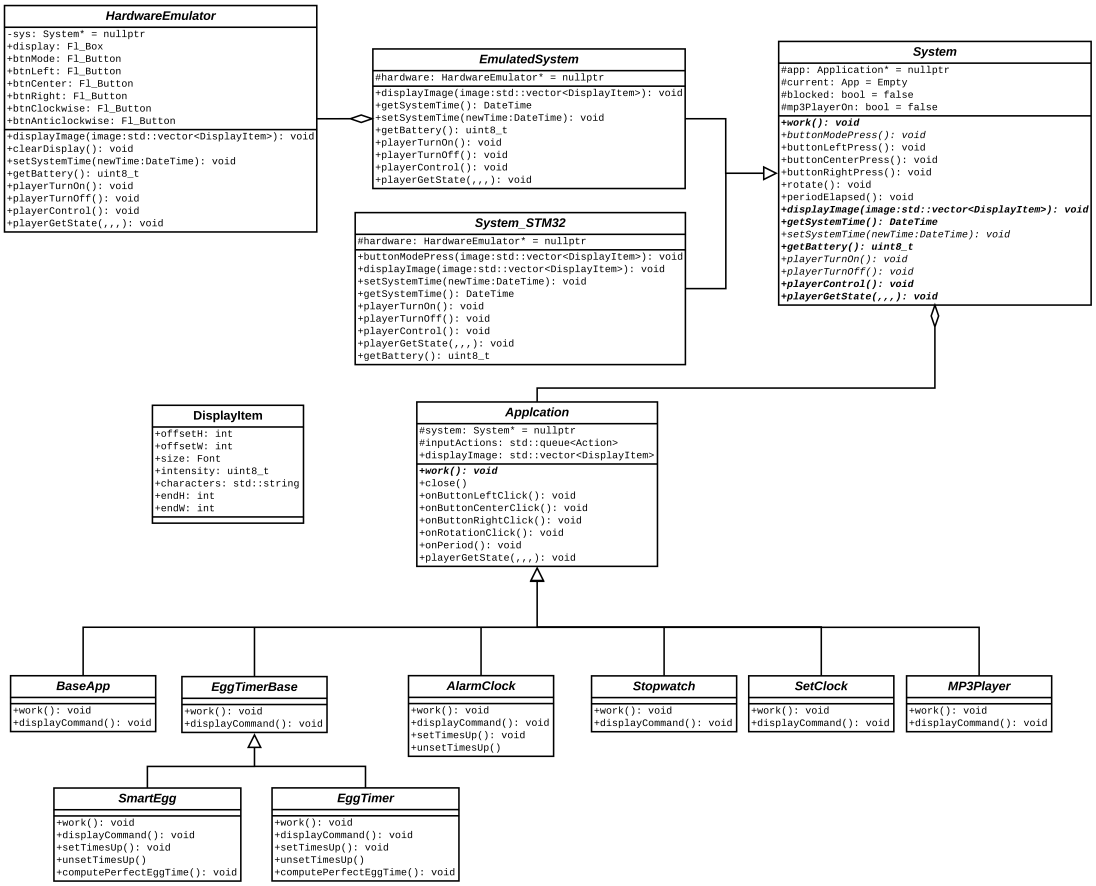
As a user of the Stopwatch, I want to know the time between two events, which I determine by

pressing a button. I want to see the time precisely in seconds. In addition, I want to be able to take the time of intermediate events without losing the continued time count.

#### eiPod App

As a user of the eiPod I want to listen to acoustic tracks that are stored on the SD card. I want to start and stop the acoustic track. There should be a visualization of the number of tracks loaded on the device and the current track ID on the display. I want to be able to change the track IDs. In addition, I want to be able to change the volume of the speaker.

## Software architecture




We choose an object-oriented approach to allow for an emulated development environment and keep the software quality high.

#### System

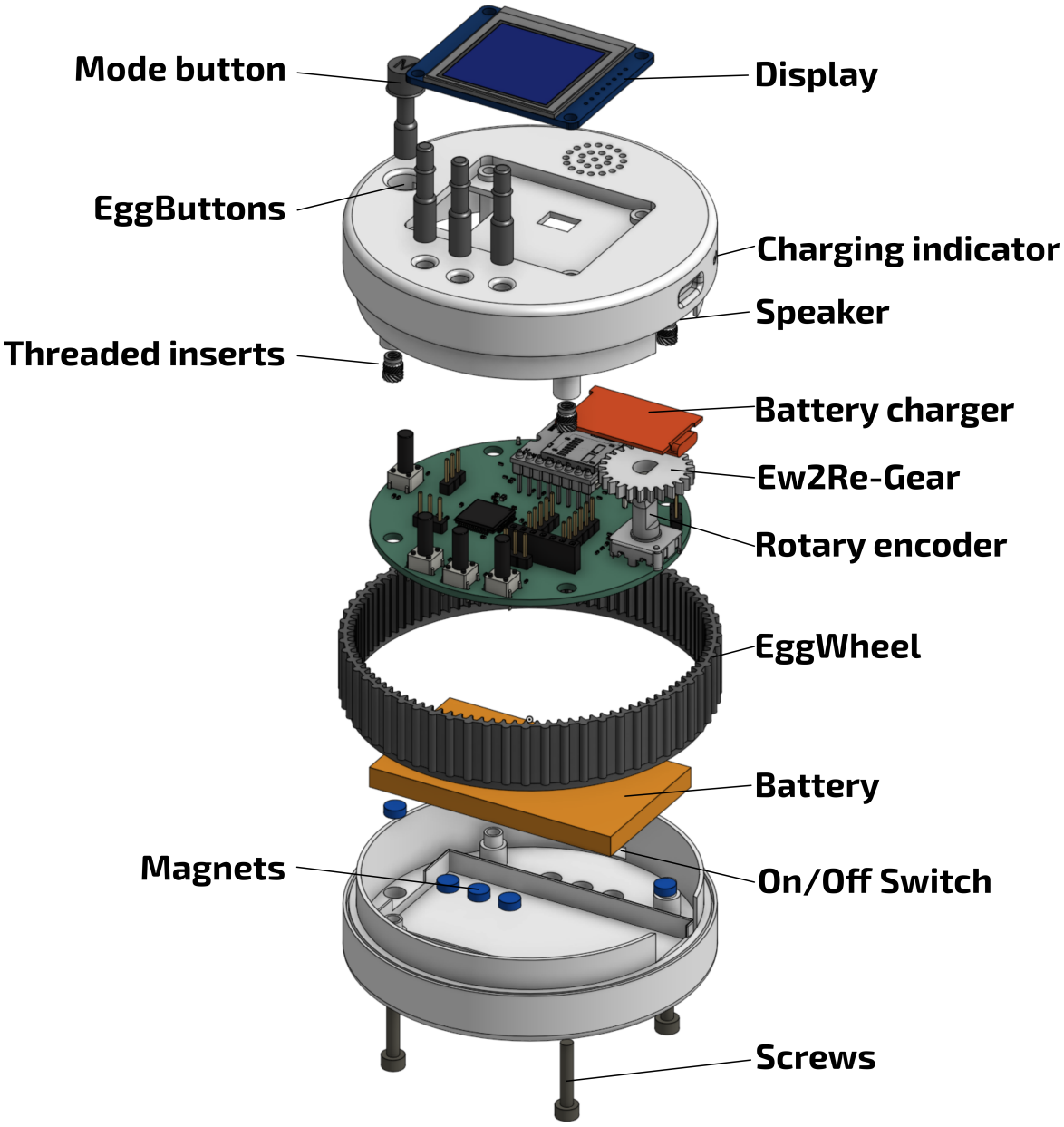
Central to the architecture is the abstract System class. It fulfills an OS-like role and provides functionalities for the applications. No application directly interacts with hardware but only with the system class. In addition, the System class can be implemented on the actual hardware or on an emulator without any effects on the applications. This is realized in the classes System_STM32 and EmulatedSystem. The latter implements its methods using a HardwareEmulator class based on the FLTK library, while the former implements them on the hardware itself.
The methods of system that trigger the application are work(), buttonModePress(), buttonLeftPress(), buttonCenterPress, buttonRightPress(), rotate(), and periodElapsed(). work() calls on the application to do work similar to a process scheduler in an OS. buttonModePress() handles the application changing process. buttonLeftPress(), buttonCenterPress, and buttonRightPress() notify the application of button interaction by calling the application’s respective button press methods. rotate() does the same with the rotary device.

periodElapsed() forwards a cyclic and time-controlled signal to the application. Additionally system offers methods for the applications that they can trigger. displayImage(...) provides a way to write an image to the display. Other methods allow the applications to ask for the current time or battery status. Finally, system offers methods for interacting with the MP3 player.

#### Application

All applications in the Smart Egg Timer are derived from an abstract base class. It defines basic functionality for all other apps. Triggers from the System class (pressed buttons e.g.) are handled by pushing the information to a queue. In every working step, this queue is worked through. This structure mitigates the risk of losing an interaction due to interfering interrupts and allows all apps to be implemented in a finite automata fashion. The work method is abstract in the class. Specific applications must override it, taking an action from the queue in every run.

## Case




The case of the SmartEggTimer evolved over time. After many trial and error and small improvements, we can proudly say that we developed a case that is handy, easy to use, and

stylish. The case is entirely 3D-printed out of PLA, with a print time of around 6 hours.

In the beginning, the main focus was on the mechanics to allow the user to rotate a wheel to set the clock of the timer. We tried some approaches and now use gears as they are used in most gearing mechanisms. As all components are 3D-printed out of PLA, they had a smooth- but still too rough surface-finish in terms of minimizing frictional resistance. We had to sand the surface of the EggWheel with fine sandpaper and used a lubricant for a smooth rotation experience. We tried some different lubricants and Oils and chose the best one. We found that the ‘Lager Fett’ from the local hardware store offered the best experience.

With this solved, we fitted all the other components into the case, which was a challenge because we wanted to keep the case as handy as possible. With close collaboration between the PCB design and the Case design, we found an excellent way to fit everything in the case. The PCB sits over the battery and is held by four holes which are also used by 4 M3x16 Screws to screw the two parts of the case together. For a good user experience, we decided to raise the four buttons with an extension. The extension can be put over the button on the PCB, and the buttons can be pressed from the outside nicely and smoothly. The buttons have a nose that prevents them from falling out. We then noticed that the mode button could be rotated, which resulted in the label being turned the wrong way around. To also avoid this, there is an additional small nose so the button can not be rotated but only pressed downwards.

One additional experimental feature is the charging indicator. The charging board for the battery features two LEDs to signal the state of the charge. We wanted to make them visible from the outside of the case. We used a short part of 3D-printing filament without color as fiber optics. This way, the LED light is directed to the outside.