Technical University of Denmark

34346 Networking technologies and application development for Internet of Things (IoT)

Project **Group:** 3

Student	Main	Secondary
	Report	
Asbjørn (s223987)	Battery	Challenges and improvements
Emil (s224025)	WiFi / UART / I2C / State evaluation / (BLE)	Challenges and improvements / Hardware (sensing / connectivity)
Victor (s202935)	Frontend / Backend	
Jonathan (s220579)	Hardware / PCB Motion sensor	Challenges and improvements
Henrik (s061795)	LoRa / Microchip RN2483 / LoRaWAN / TTN	GUI / Flutter
Manuel (s243549)	Introduction / Frontend / Backend	Hardware (sensing) / Challenges and improvements
	Code (and hardware)	
Asbjørn (s223987)	Battery monitoring	
Emil (s224025)	Main `.ino` file / Skeleton / WiFi / (BLE)	LED brightness / Light sensor
Victor (s202935)	FireBase / FlutterFlow / LED brightness	
Jonathan (s220579)	Motion sensor / PCB design / Enclosure design	
Henrik(s061795)	LoRa / Microchip RN2483 / TTN	GUI / Flutter
Manuel (s243549)	Light sensor / FlutterFlow	

Table 1: Task allocation overview

May 25, 2025

Link to GitHub with source code: `https://github.com/HenrikBach1/smart-bike-light`

Contents

1 Introduction **2**
1.1 Product . . . 2
1.2 Problem statement and objectives . . . 2

2 Communication technologies **2**
2.1 LoRa and TTN . . . 3
2.1.1 Theory of (Proprietary Semtech) LoRa IP Technology . . . 3
2.1.2 Microchip RN2483 LoRa Modem Module . . . 3
2.1.3 Theory of LoRaWAN . . . 3
2.1.4 The Things Network (TTN) . . . 4
2.1.5 Geo location in The Things Network . . . 4
2.2 WiFi . . . 4
2.3 UART . . . 5
2.4 I2C . . . 5

3 System setup **5**
3.1 Hardware . . . 5
3.1.1 Battery, sensing and connectivity . . . 6
3.1.2 PCB . . . 6
3.2 Firmware . . . 7
3.2.1 Battery monitoring . . . 7
3.2.2 LoRa Modem Module (RN2483) . . . 7
3.2.3 Sensing and LED . . . 8
3.2.4 WiFi / (BLE) . . . 8
3.2.5 State evaluation . . . 9
3.3 Frontend and backend . . . 9
3.3.1 FE Test App . . . 9
3.3.2 FlutterFlow . . . 10
3.3.3 Uplink and downlink communication . . . 10
3.3.4 App interface . . . 11

4 Challenges and improvements **11**
4.1 Battery Runtime Estimation . . . 11
4.2 Reevaluating LoRaWAN . . . 11
4.3 Dual-Core Architecture . . . 12
4.4 PCB and Hardware improvements . . . 12
4.5 PCB Optimization . . . 12
4.6 Power management and Sensor control . . . 12

5 Appendix **14**
5.1 PCB render . . . 14
5.2 Enclosure . . . 15
5.3 Physical product . . . 16
5.4 App . . . 17
5.5 PCB Schematics . . . 18

1 Introduction

Urban cycling has emerged as a sustainable and efficient mode of transportation, yet it is accompanied by persistent safety and security concerns. Inadequate visibility is a major contributor to collisions: Aalborg Universitet [1] estimates that up to 2,200 bicycle accidents per year in Denmark could be averted through reliable lighting systems. Concurrently, bicycle theft imposes a substantial economic burden, with annual losses in Denmark reaching approximately 250 million DKK [2]. These figures underscore the need for integrated technological solutions that both enhance rider visibility and deter theft.

1.1 Product

This project introduces a smart bike light system powered by an ESP32 microcontroller, designed to enhance cyclist safety and system intelligence through sensor integration, wireless communication, and adaptive mode management. The system operates in three main modes—Active, Park, and Storage—each tailored to specific use cases and controlled through button input and ambient light sensing - a fourth mode, Stolen, also exists and can be triggered from the backend. Upon startup, the ESP32 initializes modules including WiFi for location approximation, LoRaWAN for long-range communication, a light sensor, a battery monitor and not currently on the device, but to be implemented in the future: BLE for proximity detection. In Active Mode, the device monitors light levels and user input. In Park mode the device wakes up every set interval, estimates its location by scanning for the three closest WiFi access points and analyzing their RSSI values. It then transmits the system data and location updates via LoRaWAN to The Things Network (TTN), which are then forwarded through an MQTT broker to a cross-platform Flutter mobile application. Likewise, in Storage mode it wakes up and sends the battery percentage. The app allows users to remotely monitor real-time location, battery level, and system status. The light includes intuitive user interaction through button controls, and its modular hardware design supports future upgrades such as a custom PCB and 3D-printed housing.

1.2 Problem statement and objectives

The project tackles:

- **Safety:** Riders frequently neglect to switch on or recharge their lights, especially under variable ambient conditions, which elevates the risk of nighttime and low light accidents.
- **Theft:** High incidence of bicycle theft persists, as traditional lights offer no means of tracking or recovering stolen vehicles.

The Smart Bike Light aims to:

- **Autonomous Activation:** Employ movement detection and ambient light sensing to engage lighting automatically, eliminating reliance on manual operation.
- **Geolocation Tracking:** Utilize geolocation positioning (Wi-Fi triangulation) in combination with wireless communication (LoRaWAN), and the Google API to monitor and report the bicycle's location in real time, thereby aiding theft deterrence and recovery.
- **Remote Monitoring:** Provide a Flutter based mobile application and backend interface for users to view and interact with device status, battery level, and location data remotely.
- **Power Optimization:** Implement efficient power management strategies to maximize battery longevity in different modes.

2 Communication technologies

In the project, different communication technologies are utilized to allow the different modules of the bike light to exchange information interchangeably. This includes wireless and wired communication, as well as long and short ranged radio frequency (RF) based protocols. In the following, a brief explanation of these different technologies and their use cases in the project are presented.

2.1 LoRa and TTN

This section discusses the technologies utilized in the open LoRaWAN network stack, specifically the "The Things Network" (TTN) and front-end applications, which employ the use of the TTN API to receive uplink and send downlink data to IoT devices using LoRa and LoRaWAN technology, specifically the Microchip RN2483 LoRa Modem Module.

2.1.1 Theory of (Proprietary Semtech) LoRa IP Technology

LoRa (**Lo**ng-**Ra**nge - up to 15 km) is a physical (PHY) or "bits" layer implementation that uses RF technology to transmit information in the form of "bits", from sender to receiver. LoRa is proprietary, and its intellectual rights (core IP) are owned by Semtech, and it is selling their LoRa core IP to other chip vendors through different chip-sets.

LoRa is non-cellular RF modulation technology using a proprietary chirped spread spectrum (CSS) modulation and orthogonal spreading factors (OSF/SF) in unlicensed radio frequency spectrum bands (band widths) of 433, 868 (EU), 900 (US) MHz and 2.4 GHz with a fixed bandwidth carrier channel of either 125 or 500 kHz (for uplink channels), and 500 kHz (for downlink channels). OSF and CSS allow the IoT device to preserve battery life while making adaptive optimizations of power levels and data rates when connected. Note that OSF signals with different spreading factors transmitted on the same channel do not interfere with each other and simply appear as noise to each other.

The spreading factor (SF) determines how many "chirps" are used per symbol, and thus how long each symbol lasts in time. Higher SF also (indirectly) improves the sensitivity at the receiver end, by making it easier to decode weak signals from the sender. LoRa groups bits of the message into symbols depending on the spreading factor (SF x, where x is equal to the number of bits of a message that are grouped together into one "chirp" symbol) and uses MSB-first ordering for symbol construction. Each symbol is transmitted as a distinct frequency-modulated chirp.

LoRa uses ultra low power data rate communication over various frequencies (channels) for upload and download of small hexadecimal formatted messages. These range in size from 11 (recommended) to 242 bytes, using variable bit rates and CSS symbols from about 250 bps (SF 12) to 50 kbps (SF 7) depending on the distance between the sender and receiver. Lower frequencies provide better propagation and penetration into buildings due to reduced attenuation, likewise, higher spreading factors are used to reach longer distances, thus resulting in longer symbol durations and lower data rates. This increases transmission time, but improves the receiver's sensitivity to weak signals.

If the firmware of the IoT device powers down the LoRa communication module when not actively transmitting or receiving, but maintains minimal power to preserve its register state and avoid re-initialization, the module consumes only a few milliwatts. This allows battery-powered IoT devices to potentially operate for up to 10 years, provided that the overall system is energy-optimized and designed appropriately.

2.1.2 Microchip RN2483 LoRa Modem Module

The Microchip RN2483 LoRa Modem Module (chip) implements the LoRa RF technology and functionality of the "LoRaWAN 1.1 Specification".

To communicate with the LoRa Modem module, a connection to its onboard UART. The firmware of the module interprets the Modem AT commands to configure settings in its onboard SRAM, enabling it to join a network, transmit, or receive data. The various AT commands are wrapped into the (Arduino-based) rn2xx3 library, which simplifies programming by providing built-in error handling, removing the need for users to manage command specific quirks themselves.

2.1.3 Theory of LoRaWAN

In the OSI seven-layered network model, LoRa covers the physical medium layer (using RF transceiver technology) and the Data Link layer with physical addressing together with LoRaWAN.

LoRaWAN, as defined in "LoRaWAN 1.1 Specification", is an open standardized and secure protocol that enables half duplex, bidirectional communication using a (multi-)star network topology built on LoRa (RF transceiver) technology. In this architecture, each "star" in a LoRaWAN network represents a gateway that relays messages between IoT devices and a

central network server. The network servers then forward messages to/from an application server, where user applications can process the data. Messages on an LoRaWAN network may be 11 (recommended) up to 242 bytes long.

An application is created in the application server and each IoT device is registered for the application with a predefined DevEUI of its LoRa module. Then, front-end applications may subscribe to (to read uplink)/or publish (to send downlink) messages to specific IoT devices.

Devices are identified and authenticated using a set of cryptographic keys, including the DevEUI (device identifier), which is typically preassigned by the LoRa module manufacturer. User-facing applications can then subscribe to uplink messages or publish downlink messages to specific devices via the application server. LoRaWAN supports message payloads ranging from 11 bytes (recommended minimum) to a maximum of 242 bytes, depending on the region and data rate.

When an IoT device wants to join a LoRaWAN, it uses the join command, where both the join-/APP_EUI and the created APP_KEY keys are registered beforehand in the LoRa module. The network server instead communicates with a join server, which sends back two computed session keys (NwSKey and AppSKey) through the network server to the LoRa module to create a secure communication between the network server (NwSKey) and the application servers (AppSKey). Like in the Tel-co world, each LoRa gateway must be configured specifically for an LoRaWAN network provider.

2.1.4 The Things Network (TTN)

We use the open and free platform "The Things Network", also known as TTN, to connect to our IoT devices. In the TTN Console, we create several credentials to secure the communication:

- **Application key (AppKey:** Used by the IoT device to authenticate and join the network.
- **Application id (AppId):** Identified the specific application within TTN.
- **API Key:** Grants access to the TTN services and can be adjusted to different permission levels depending on its use case.

These API keys allow external front-end (FE) applications that don't necessarily support the TTN API, to read data from or send data downlink to connected devices. Additionally, each IoT device must be registered in TTN using its globally unique DevEUI and an AppEUI (typically 0x00 00 00 00 00 00 00 00 for TTN's public network). This results in, that only devices and gateways registered on the same LoRaWAN network are able to communicate. Lastly, both LoRaWAN (regional parameters) and TTN enforce a "Fair Use Policy", which limits a public community network to a device duty cycle below 1%, ensuring network availability for all their users.

2.1.5 Geo location in The Things Network

When an IoT device sends a message via TTN, each gateway that receives the signal attaches additional metadata containing the geo location of the gateway (if available) and the signal strength (RSSI) of the received message. This additional metadata together with the original payload and other contextual information is then packaged into a JSON structure and sent to the network server.

If multiple gateways on the same network receive the same message, the metadata from all of them is aggregated into a single grouping. If instead only one gateway receives a message, the RSSI may instead be interpreted as a rough range estimate, forming a circular area around the gateway's position indication where the device is likely located. If at least two or more gateways receive the message, their location and RSSI values can instead be used to perform triangulation, resulting in a better estimate of the device's actual positioning.

In our project, both single-gateway and multi-gateway location estimates are done in the **Trilateration** function found in the ttn_api.dart file of the smart_bike_light project under the FE-Flutter directory.

2.2 WiFi

An alternative method for locating IoT devices is WiFi-scanning. This technique detects nearby WiFi access points by identifying their mac addresses and signal strengths (RSSI), to get an approximate location (coordinates + radius) of a

device. To obtain the actual location, the Google `Geolocation API`, which among other things, contains a database of different WiFi access points (MAC addresses + signal strengths, RSSI) and their respective location.

WiFi is a wireless communication technique that is most famous for enabling WLANs (Wireless Local Area Networks) in homes, schools and practically anywhere. Like any other wireless technology, it operates based on certain "rules" that dictate the protocol. WiFi uses the IEEE 802.11 standard, which defines a baseline of both the physical and data link layer protocols [3]. Since its introduction in 1997, the standard has evolved significantly. WiFi uses the unlicensed 2.4 GHz and 5 GHz frequency band to communicate, offering an array of different channel widths depending on the specific protocol version. It is commonly used for high-bandwidth data transmission over short distances between multiple users and an internet access point functioning as a gateway.

In our application, however, we do not use WiFi for communication. Instead, we merely exploit the fact that most buildings contain a WiFi router (with a unique MAC address), that it constantly broadcasting its presence. This enables us to utilize the power of WiFi scanning for geolocation, as it is a precise and reliable locating mechanism in urban areas (because of the abundance of different WiFi routers). The main downside to using WiFi locating, is an increased power consumption and a decreased performance in rural areas.

2.3 UART

UART **Universal Asynchronous Receiver-Transmitter** is a serial communication protocol commonly used for direct device-to-device communication. In this project, UART is used for two things:

- Communication between the ESP32 and our computer/IDE (code upload and debug console).
- Communication between ESP32 and RN2483 (LoRa) module.

To enable UART communication between two devices, each device is required to have an UART port, typically implemented via hardware or software, which manages data transmission over two lines: TX (Transmit) and RX (Receive). UART is especially advantageous, as it supports both half-duplex and full-duplex communication and is relatively simplistic in setup. However, UART is not without its limitations. It only supports point-to-point communication, meaning a max of two devices per bus, and is arguably slower than other protocols [4]. Since this project utilizes two UART communcation channels, from ESP32-IDE and ESP32-RN2483, we require two separate UART ports. Luckily, the ESP32 microcontroller has three UART [5]. For the final PCB, we also need two UART ports, but only the connection between ESP32-RN2483 is in use when the bike light is running.

2.4 I2C

Another type of serial communication is the Inter Integrated Circuit (I2C) protocol. This is a faster and more scalable technology than UART - however, it also requires a bit more hardware in the implementation than UART. I2C uses two open-drain wires called SCL (clock line to synchronize the communication) and SDA (data lines that carries the information). The clever thing about I2C is that it uses a (7 bit) addressing system, which means it can accommodate up to 128 different devices on the same bus. In this project however, we only use I2C for communication with a single device, namely the accelerometer. Since I2C provides data validation and flow control, it will in general provide a better data integrity compared to UART [6, 7].

3 System setup

In the next section, the focus will be to describe the actual implementation of our project. This includes the hardware setup used, the firmware on the ESP32 and the topologies of the system.

3.1 Hardware

In the following subsection, we describe the various hardware components used to develope the bike light system. This includes things such as power management, processing, sensors, connectivity and a description of how the components

were integrated on a custom PCB.

3.1.1 Battery, sensing and connectivity

The system is powered by a Samsung INR18650-35E Li-Ion cell (nominal 3.6 V, full charge 4.20V, cutoff 2.65V). Sicne the ESP32 requires a 3.3V supply (VDD), a linear voltage regulator (AMS1117-3.3) is used to step down the battery voltage down to 3.3 V. For sensing, three different types of sensor were implemented: buttons, light sensor and accelerometer.

- **Buttons:** The buttons were implemented using the ESP32's internal pulldown resistors. Each button closes the circuit between the 3.3V line and the ESP32 input through an external 10kΩ resistor, which limits current during usage.
- **Light sensor:** The light sensor, or LDR, is connected to an analog input pin via a voltage divider, formed by the LDR and a 10kΩ resistor connected to ground. This specific setup allows the light sensor to read light intensity as a voltage level.
- **Accelerometer:** The accelerometer is connected to the ESP32's I2C bus. Pin `SDO` is shorted to `GND` and pin `CS` is shorted to `3V3`.

The light output is controlled by a BJT transistor, which acts as a switch, driven by a digital output from the ESP32. A series resistor is also placed between the power line and the light, specifically to allow safe and flexible switch between different light sources during testing. For connectivity we utilize:

- **WiFi scanning:** Is handled directly using the ESP32's built in WiFi module. This provides and easy straightforward implementation.
- **LoRa communication:** Is managed via the RN2483 module, which is connected to the TTN network. This modules utilizes the standard UART configuration; `3V3` and `GND` pins, as well as `RST`, `TX` and `RX` pins.

3.1.2 PCB

The PCB was designed with modularity in mind, allowing external components to be plugged directly into headers, rather than being fully integrated into the PCB. Specifically, the motion sensor (ADXL345) and the LoRa module (RN2483) are connected via two dedicated headers on the PCB. A similar principle is applied to device programming. In order to reduce both design complexity and cost, an external converter is used to flash the firmware instead of embedding a USB-to-UART converter directly on the PCB. In order to program the device, the external converter must therefore be plugged into a header, after which the converter handles the programming.

The main components present on the PCB include the ESP-32-WROOM-32E microcontroller, buttons, a light dependent sensor (LDR), a BJT based light driver, a micro USB port for charging, a battery management system (BMS), and an AMS1117-3.3V voltage regulator.

The micro USB port is solely connected to the input of the BMS, and cannot be used for programming. The 5V bus from the USB is fed into an TP4056 Lithium-ion charging IC, which is responsible for charging the battery. The battery voltage is monitored using a voltage divider composed of a 50kΩ and a 180kΩ resistor. This reduces the battery voltage to a safe level for the ESP32's analog-to-digital converter (ADC), allowing direct connection to the ADC input. To prevent the battery from decharging too much, as DW01A Lithium-ion protection IC is used, alongside a FS8205A chip, which houses two MOSFET transistors. This combination disconnects the battery's negative pole fro mthe PCB, if the voltage drops too low.

The AMS1117-3.3V voltage regulator is connected to the battery, and is responsible for regulating the battery's voltage to a stables 3.3V. A diode is connected between the output of the regulator and the 3.3V input on the programming header, to prevent reverse current that could damage the system.

A block diagram of the components on the PCB can be seen in figure 1. A 3D render of the PCB can also be seen in the appendix in figure 8. The PCB is enclosed in a custom 3D-printed housing, which was modeled in Autodesk Fusion

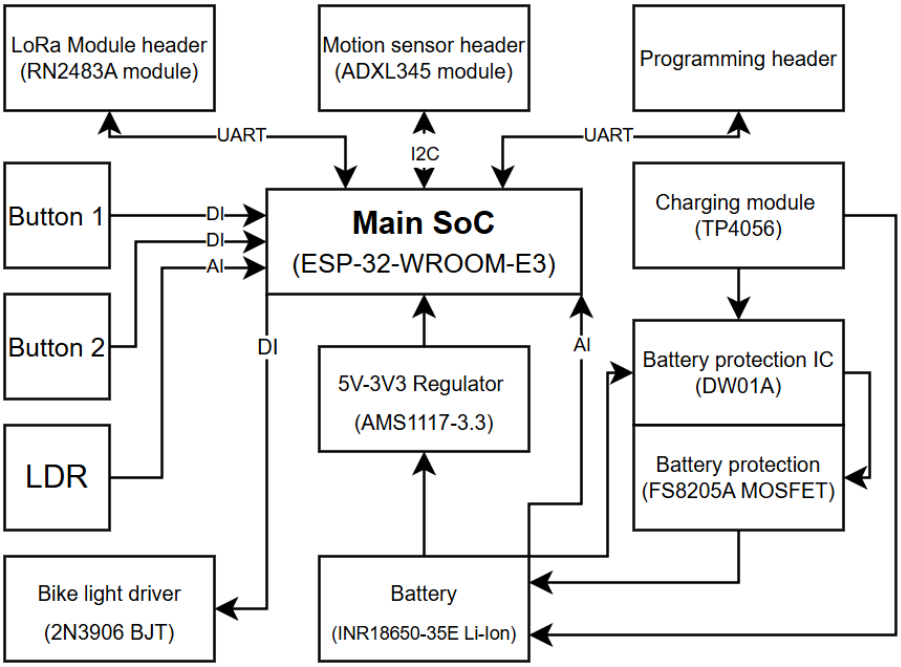


Figure 1: Block diagram of the components on the PCB

and 3D-printed on a Bambu Lab P1S. A screenshot of the model can be seen in the appendix in figure 4. The completed assembly of the housing, PCB, the battery etc. can be seen in the appendix in figure 5.

The PCB is using a four layer construction, where the top, inner layer 1 and bottom layer is primarily serve as ground planes. The inner layer 2 is dedicated to a 3.3V plane, where the different IC's are connected through vias. The different ground planes are connected with stitching placed on the entire PCB. This ensures the ground planes have the same potential. The use of big ground planes also minimizes noise. The PCB was made and assembled by JLCPCB and drawn in EasyEDA Pro. The electrical schematic can be seen in the appendix figure 7-10.

3.2 Firmware

This section describes how the various hardware components were integrated and controlled via the ESP32 microcontroller. To enable a an efficient and parallel workflow, the team was divided into smaller groups, each responsible for implementing a smaller firmware module, designed to work independently. Once accomplished, the individual modules were combined into a single common project and adjusted to fit and functions together.

3.2.1 Battery monitoring

The `battery_monitoring` module provides three main functions: `charging()`, `battery_percentage()` and `time_left()`. The `readCellVoltage()` function is a helping function used internally to get the battery cell voltage from the ADC-value input. It is a private function not accessible in other parts of the program. Based on the discharge profile from [8], the cell voltage spans from 4.20 V (100%) down to 2.65 V (0%), with two approximately linear regions: 3.40–4.20 V corresponds to 20–100% and 2.65–3.40 V corresponds to 0–20%

3.2.2 LoRa Modem Module (RN2483)

In this section, we introduce the functionality written within the CustomLoRa module implementing functionality for both the LoRa Modem module and the TTN network. Table 3 contains the functions to provide functionality for the LoRa Modem module.

Function Name	Description
`charging`	Returns `true` if the battery is currently being charged. Reads a GPIO pin connected via a voltage divider to the charger-status output on the PCB.
`readCellVoltage`	Reads the raw ADC value and converts it to the actual cell voltage using a resistor divider. Used internally by other functions.
`readBatteryPercentage`	Estimates battery percentage based on the measured cell voltage. Uses a two-part linear mapping between 2.65–4.20 V.
`time_left`	Estimates remaining runtime using battery percentage and current draw based on mode (ECO, MEDIUM, STRONG). Divides remaining capacity by expected load.

Table 2: Summary of battery monitoring functions

Function Name	Description
`initialize_LoRaWAN`	Initializes the LoRa Modem module.
`deepSleep`	Puts the LoRa module into minimal power consumption.
`wakeUp`	Returns the LoRa module to normal power mode.
`tranceive`	Sends and receives messages to/from the TTN network application server.
`join_TTN`	Joins the TTN network using OTAA (Over The Air Activation).
`is_joined_TTN`	Returns `True` if connected to the TTN.
`leave_TTN`	Resets OTAA parameters (including NwSKey and AppSKey) in the LoRa Modem module.

Table 3: Summary of LoRa Modem and TTN-related functions

3.2.3 Sensing and LED

The firmware for detecting button inputs on the two buttons is handled using the Arduino library `OneButton`. This library handles timing, debouncing and detection of the buttons clicks seamlessly.
The LDR firmware is similarly quite simple, utilizing an analog read on an ADC pin, which is checked against a certain threshold for when it is dark/bright.
The bike light system uses an ADXL345 accelerometer module for motion sensing. This module includes some basic components needed for the IC, is plugged directly into the main PCB and communicates with the ESP32 via the I2C bus. The firmware implementation makes use of the output interrupt pins from the module. This means that the ESP-32 receives a hardware interrupt whenever an "inactivity", "activity", "free fall", "double tap" or a single tap occurs on the ADXL345. This is used to control the different device modes.

To perform motion traction from the accelerometer, a function is used to retrieve the X, Y, and Z acceleration values in in real-time. These raw values are adjusted with an offset value, either subtracted or added, based on the calibration. This ensures that when the device is standing still, it will indicate no motion detected. This is needed, as many things can influence the base readings. The readings are then processed, and utilized to increase the light brightness proportionally when the device is braking. This is done by both looking at the sign of the data from the ADXL345 and how much acceleration is detected. The LED brightness itself is controlled by adjusting the duty cycle of the PWM (Pulse Width Modulation) signal on one of the ESP32's GPIO pins. This is easily implementable, as the ESP32 already has compatable PWM pins.

3.2.4 WiFi / (BLE)

The WiFi functionality has been implemented using the Arduino WiFi library, which makes it super simple to set up and scan for nearby WiFi access points used in the geolocation. Additionally, BLE proximity detection was also developed, (which essentially just looks for a specific Service UUID close to it), but was not implemented due to a lack of storage space on the ESP32's memory. With additional time and memory, both WiFi and BLE would be include in the bike light.

3.2.5 State evaluation

To keep track of the device at any point in the program, the module called `Skeleton` was used. This module keeps track of the device state with a global `struct` called `deviceState`, and "glues" the above-mentioned firmware modules together, to achieve the desired overall functionality. This includes checking and altering the state of the device, reading relevant sensor values, and establishing wireless connections whenever needed. The functional states the device can occupy are: `Active`, `Park`, `Storage` and `Stolen`.

Function	Description
`tick_stuff()`	This function ticks the button handlers, reads sensor values and adjusts light intensity.
`tick_stuff_interval()`	This function reads less important sensor values two times a second.
`esp_sleep_get_wakeup_cause()`	Function provided by Espressif to determine which interrupt that caused the wakeup.
`goToDeepSleep(int sleepingTime, bool mpu_interrupt, bool stolen)`	Our main routine for setting the ESP32 in deep sleep. Depending on the function arguments, we set the wake-up interrupts accordingly.

Table 4: Selected important functions regarding state operation of the ESP32.

A key component in optimizing energy efficiency, is the use of the ESP32's sleep modes. The ESP32 has two different sleep modes: Deep sleep and Light sleep. As the names suggests, the former halts more processes in the processor than the latter, hereby also decreasing the power draw significantly [9]. In this project we only utilize deep sleep mode, as it offers the lowest power consumption. When the processor is "put to sleep", a collection of interrupt wake up sources are set, depending on which state the device is currently in. This ensures that we have full control of when, how and why the device wakes up - which is always checked upon waking up from deep sleep. The most important functions are listed in table 4.

3.3 Frontend and backend

With both the hardware and the firmware described, the final piece of the puzzle involves examining the backend and frontend structure. The backend is responsible for handling the data transmitted and received via LoRaWAN, and enabling this data to accessible by the frontend (user interface).

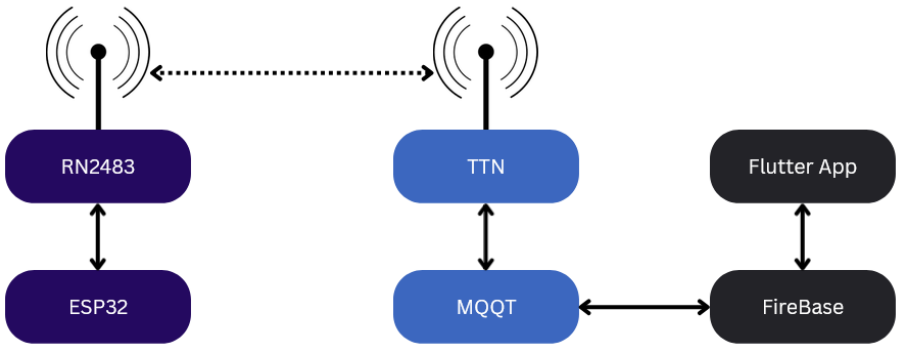


Figure 2: Backend/frontend topology

3.3.1 FE Test App

In the beginning, when the full scope of the project was unclear, our focus was to quickly identify 1) how the communication between the backend application of an IoT device and the frontend application, for the user owning the IoT device, using

the TTN API should be designed and 2) which cross-platform frontend app framework to use to support iOS and Android smart phones.

Through research and evaluation, the Flutter framework was chosen, due to its multitude cross-platform capabilities, which allowed us to utilize a single code base in the dart programming language for all our supported platforms. It seemed to be a great tool for initial app development. However, as development progressed, it became clear that the Flutter UI design and darts programming model was quite complex, even for a simple UI. Due to this and strong time constraints, we opted to switch to FlutterFlow, and utilize Firebase to communicate through the TTN API with the IoT device.

3.3.2 FlutterFlow

FlutterFlow is a no-code Internet application that, through what could typically be described as block-programming or vibe-coding, allows the user to create apps and websites without extensive knowledge of the dart coding language. As part of the project was to create a seamless cross-platform application, Flutterflow greatly helped overcome the hurdles of time constraints and lack of previous knowledge on the dart coding language that coding directly in flutter pose.

The application itself had to display relevant data, such as location, bike mode and battery life in real-time, whilst also allowing that data to be altered through the uplink and the downlink streams. This means that the frontend needs to store the relevant variables, and account for and react to the changes made by the user directly on the bikelight or through the app. To do this, a database is required, preferably one with custom functions to control the data-flow. Here Firebase was chosen, as it the application directly integrated by FLutterFlow and provides the ability to create custom functions for handling the incoming data.

3.3.3 Uplink and downlink communication

The smart bike light system utilizes LoRaWAN for its primary long-range communication, interacting with The Things Network (TTN) as its network provider. This communication occurs in two main directions with three different "middle men" sending and receiving data:

Uplink Communication (Device to Application)

- **Data Transmission:** The ESP32 microcontroller, acting as the main SoC, collects data from various sensors and modules, including the light sensor, battery monitor, and motion sensor (ADXL345). It also performs a Wi-Fi scan to identify nearby access points, analyzing their mac addresses, RSSI values and signal-to-noise ratio, to later approximate its location. All this data are then packed into a single string containing the module mode, battery life, and locational data.
- **LoRa Module Interaction:** The collected data are then transmitted through the RN2483 LoRa Modem Module. Communication between ESP32 and the RN2483 module occurs over UART. The LoRa module uses specific AT commands to manage connections and data transmission.
- **TTN Integration:** The LoRa module sends small, hexadecimal formatted messages (up to 242 bytes) to the TTN network. When a message is received by a gateway within the TTN network, it is "decorated" with the gateway's geolocation (if available), the signal strength (RSSI) and signal-to-noise ratio. If multiple gateways receive the message, trilateration can be performed to estimate a more precise location of the device.
- **MQTT Broker and Firebase:** From TTN, the data is forwarded through an MQTT broker to a Firebase database. This database is directly linked to a cross-platform Flutter mobile application. The FlutterFlow application uses Firebase as a database to store and manage incoming data, allowing custom functions to control data flow.

Downlink Communication (Application to Device)

- **Remote Control:** Users can interact with the mobile application to send commands to the bike light. This includes altering the device's mode or requesting locational data.

- **Firebase, TTN and LoRa Module:** Data requests and other commands are published from the Flutter application to Firebase, through custom commands formatted in a payload in the same hexadecimal type as the data received. Firebase then relays the payload to TTN, via. the TTN's api and an MQTT broker. TTN then sends these messages via LoRaWAN to the RN2483 LoRa Modem Module on the bike light.
- **Device Response:** The LoRa module on the bike light receives the formatted downlink messages, which are then processed by the ESP32 firmware to implement the requested changes, such as altering the device's operational mode (Active, Park, Storage, Stolen) or requesting location. The application itself stores relevant variables and reacts to changes made by the user directly on the bike light or through the app.

3.3.4 App interface

The app is as mentioned built in Flutterflow and meant to be an intuitive user experience with primary focus on close to real-time data updates. The interface itself consists of two screens, a main device screen and a "Find My Bike" geolocation screen, that can be navigated to and back from.

- **Main device screen:** The main device screen shows the make and model of the bike. A circular battery indicator is also visible, which displays a visual and numerical indicator of the remaining battery life. A "Change mode" button is also usable. This button displays the current bike mode as it was last received and allows the user to remotely change it. Lastly, a large yellow geolocation button at the bottom of the screen provides quick access to the bike's location tracking feature.
- **"Find My Bike" Geolocation Screen:** This screen displays a map with the last known location of the bike. The locational data are sourced using the Google Maps API, which takes the mac addresses, RSSI values, and signal-to-noise ratio data it receives from the Flutterflow backend, and sends back a lat, lng, and accuracy measurement. These values are then used to display a location on the map. A last known location and date display is also implemented. This shows the last approximate address of the bike and the timestamp for when that location was received. The bike's location is also ping-able via. a large yellow button labeled "Ping location". This button sends a command downlink to the ESP32 to trigger a Wi-Fi scan, after which it performs an uplink transmission via. LoRaWAN as usual. Additionally, zoom features and standard buttons are present on the maps display for navigation.

Pictures of the app interface can be found in figure 6 in the appendix, with example data retrieved through the uplink from the ESP32.

4 Challenges and improvements

The project accomplished many of the set out tasks, but we believe some aspect could use improvements, and have highlighted here some aspects we believe could really improve the design.

4.1 Battery Runtime Estimation

Estimating the remaining battery runtime of our bike light battery presented a unique challenge. Initially, we considered an empirical testing approach. The idea was to perform separate discharge tests for each light mode (low, medium and bright). In each test, the battery would be fully charged and discharged down to its cutoff voltage, while logging the process. This method would result in an accurate estimate completely relying on our specific setup and battery. However, the time we would spend on each discharge test taking several hours to complete made us go another way. We instead used a public discharge curve that matches the characteristics of our cell to estimate the remaining power. This method offered a reasonable accurate way to measure and then display battery life for the user, but provided us with no quantitative results for total battery time.

4.2 Reevaluating LoRaWAN

From a practical standpoint, we would likely avoid using LoRaWAN again in a similar future project. While LoRaWAN offers low power consumption and is cost-effective for transmitting small amounts of data over long distances, its reliability

is heavily dependent on the availability of nearby gateways. The very limited LoRa coverage is inherently not well suited for a moving device. This makes the system unreliable in scenarios where consistent geolocation tracking or real-time status updates are needed. In hindsight, a GPS tracker with cellular connectivity would have provide a far more robust and scalable solution for off-campus use, albeit at a higher cost and power demand. In retrospect, the limited range and infrastructure dependency of LoRaWAN significantly restrict its practical deployment for mobile applications, such as ours.

To address this, we propose supplementing LoRaWAN with a GPS module integrated with cellular connectivity (e.g., LTE-M or NB-IoT). A GPS module, such as the u-blox NEO-6M, would provide precise and infrastructure-independent geolocation, critical for theft deterrence and recovery. While cellular connectivity increases power consumption and cost compared to LoRaWAN, it ensures reliable operation across diverse environments. A hybrid approach could also be explored, where LoRaWAN is used in gateway-rich areas to conserve power, and cellular connectivity is activated in areas with poor LoRaWAN coverage, leveraging the ESP32's dual-core architecture to manage these protocols efficiently. Perhaps a version where the GPS system is only utilized and turned on when the precision of the triangulation falls below a threshold, or the user requests it, could be implemented in the future.

4.3 Dual-Core Architecture

Another significant opportunity for enhancing the smart bike light's performance and responsiveness lies in leveraging the ESP32's dual-core architecture, by dedicating one of its cores specifically to LoRa communication tasks. Currently, various firmware modules, including communication protocols, sensor readings, and state evaluation, are sharing processing time on the same core. By assigning LoRa communication (e.g., handling AT commands with the RN2483 module, managing LoRaWAN join procedures, and facilitating data transmission and reception) to a separate core, the system could achieve notable improvements. This separation would lead to improved real-time performance for critical communication operations, as they would not be interrupted by other computational tasks. This would thereby enhancing the overall reliability of data exchange and simplifying the development and debugging processes by clearly segmenting the firmware architecture.

4.4 PCB and Hardware improvements

Future iterations of the PCB would also offer substantial opportunities for improvement. With the potential for directly integrating key components such as the motion sensor or a smaller LoRa module onto the PCB itself, the overall physical size and complexity of the product could be reduced. Furthermore, enhancing the status indication beyond a single LED could involve incorporating more comprehensive visual cues, such as distinct LEDs for charging status, connectivity status (LoRaWAN, Wi-Fi, BLE), current operational mode, and multi-color or blinking patterns for low battery warnings. This could provide much clearer and more intuitive user experience.

4.5 PCB Optimization

In terms of improvements on the PCB, a general size reduction would be the easiest implementation to make. By relocating components closer together and integrating the modules as mentioned, the size of the overall size of the PCB could be greatly reduced. However, this might also require mounting components on both sides of the PCB, which will make the board more expensive and difficult to produce.

4.6 Power management and Sensor control

A further possibility to lower the power usage even more, would be to introduce a *true* shutdown of external modules when not in use, perhaps by the use of transistors on the power pins. The last aspects we want to point out is introducing a low pass filtering on the light sensor, to make the device react to more general changes in environment, instead of quick fluctuations.

References

[1] A. Universitet, "Bike safety report," 2020.

[2] F. Stiftstidens, "Bike theft statistics," 2012.

[3] I. S. A. (SA), "The evolution of wi-fi technology and standards," 2023. [Online]. Available: https://standards.ieee.org/beyond-standards/the-evolution-of-wi-fi-technology-and-standards/?utm_source=chatgpt.com.

[4] E. Pena and M. G. Legaspi, "Uart: A hardware communication protocol understanding universal asynchronous receiver/transmitter," 2020. [Online]. Available: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html.

[5] Espressif, "Universal asynchronous receiver/transmitter (uart)," 2025. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/uart.html.

[6] Espressif, "Inter-integrated circuit (i2c)," 2025. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/peripherals/i2c.html.

[7] J. Hopkins, "Understanding the differences between uart and i2c," 2020. [Online]. Available: https://www.totalphase.com/blog/2020/12/differences-between-uart-i2c/.

[8] Samsung, "Specification of product for lithium-ion rechargeable cell model name : Inr18650-35e," 2015. [Online]. Available: https://www.orbtronic.com/content/samsung-35e-datasheet-inr18650-35e.pdf.

[9] Espressif, "Sleep modes," 2025. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/sleep_modes.html.

5 Appendix

5.1 PCB render

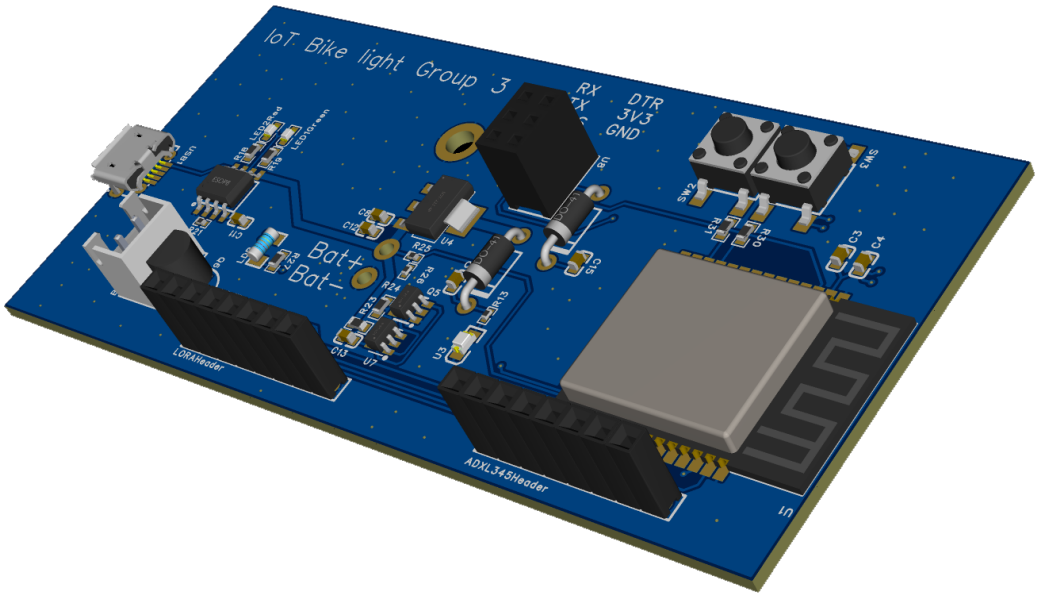


Figure 3: 3D render of the PCB with mounted components. Generated in EasyEDA Pro.

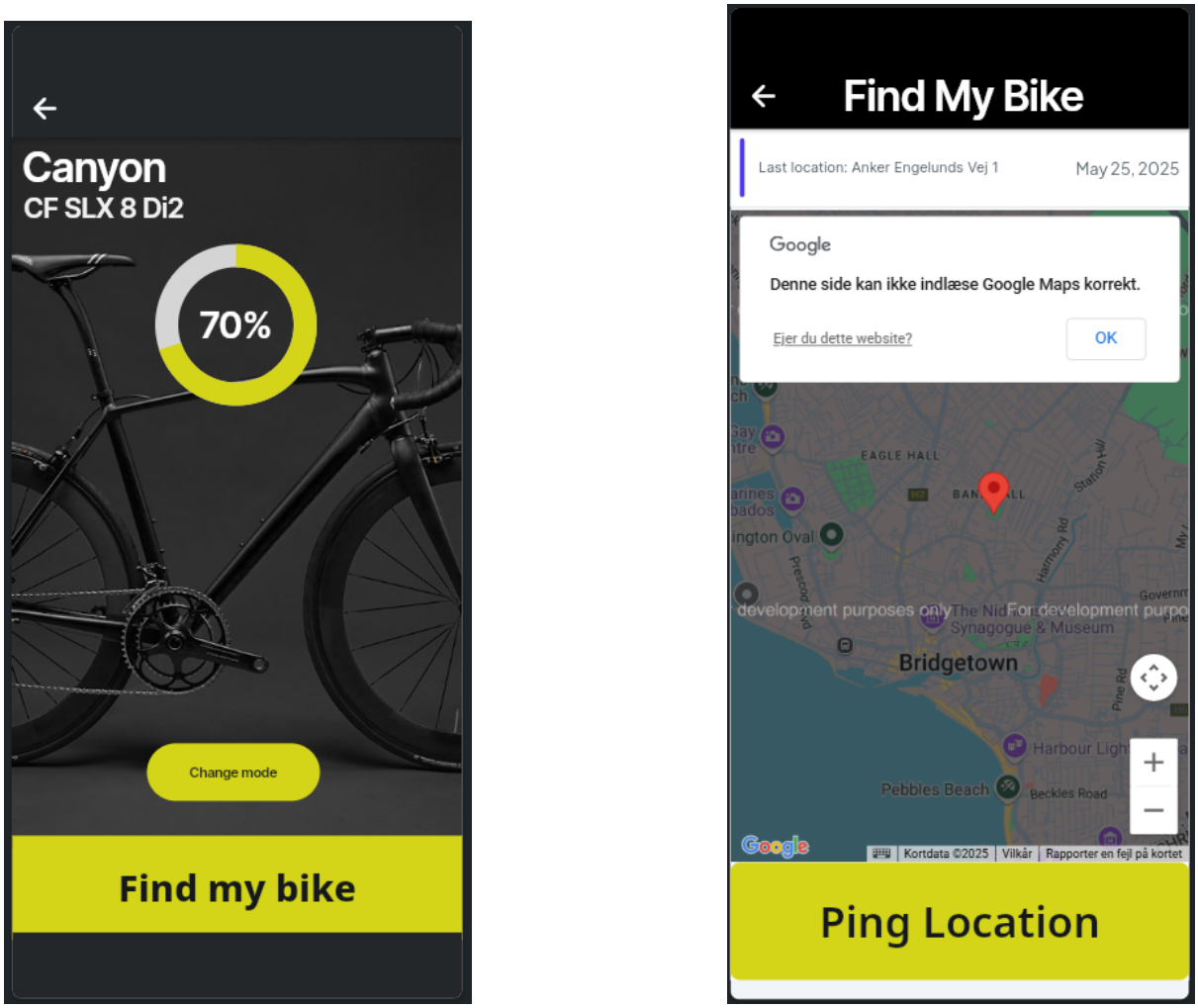
5.2 Enclosure

Figure 4: The 3D model of the housing for the device. Drawn in Fusion.

5.3 Physical product

Figure 5: The physical implementation of the PCB and 3D printed housing. The housing was printed using a Bambu Labs P1S.

5.4 App



(a) Main interface of the app

(b) Geolocation interaction view

Figure 6: Screenshots from the test application frontend. Note, due to an unknown fault in the Flutterflow application's test mode, the app isn't visible in test mode, but the is possible to confirm that the correct variables are loaded, including addresses and location. This means they are successfully being sent and fetched from the Google API. The screenshots are therefore of updated values, but the it isn't possible to actually load the Google maps interface in developer mode.

5.5 PCB Schematics

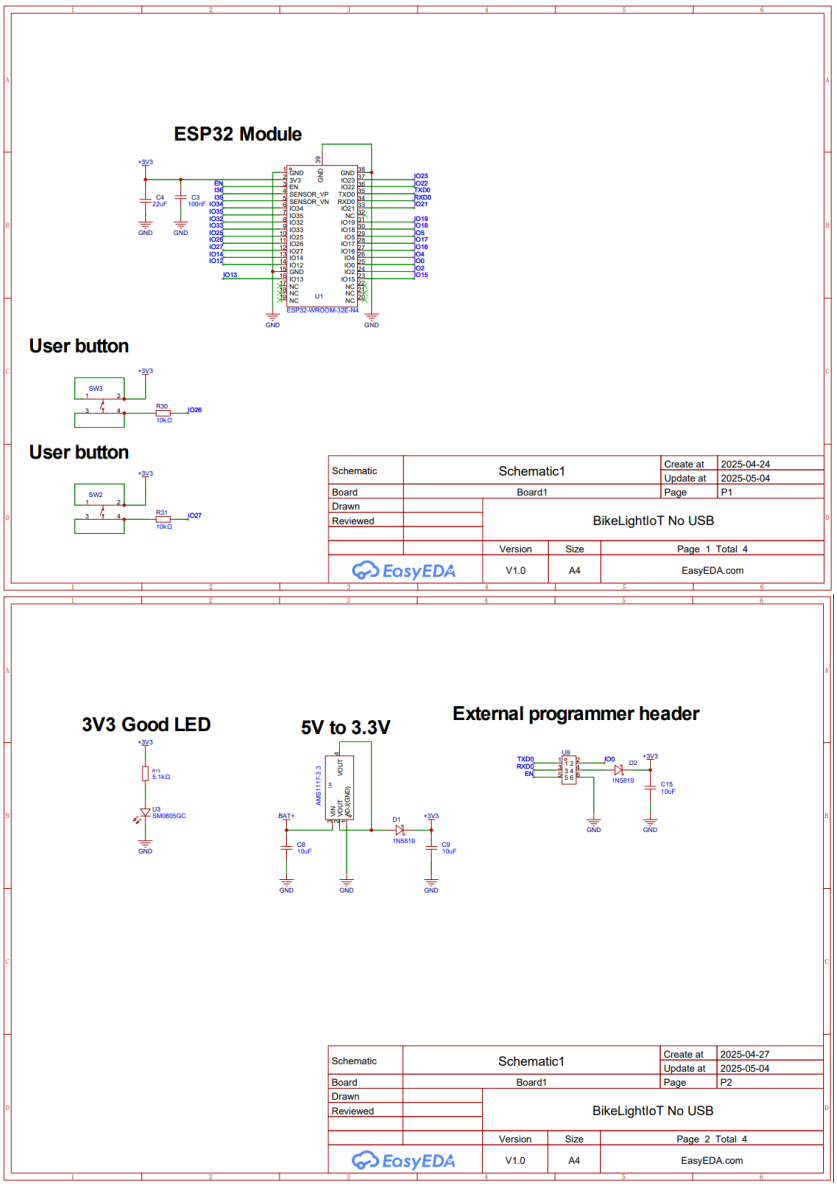


Figure 7: Page 1-2 of the electrical schematics for the PCB.

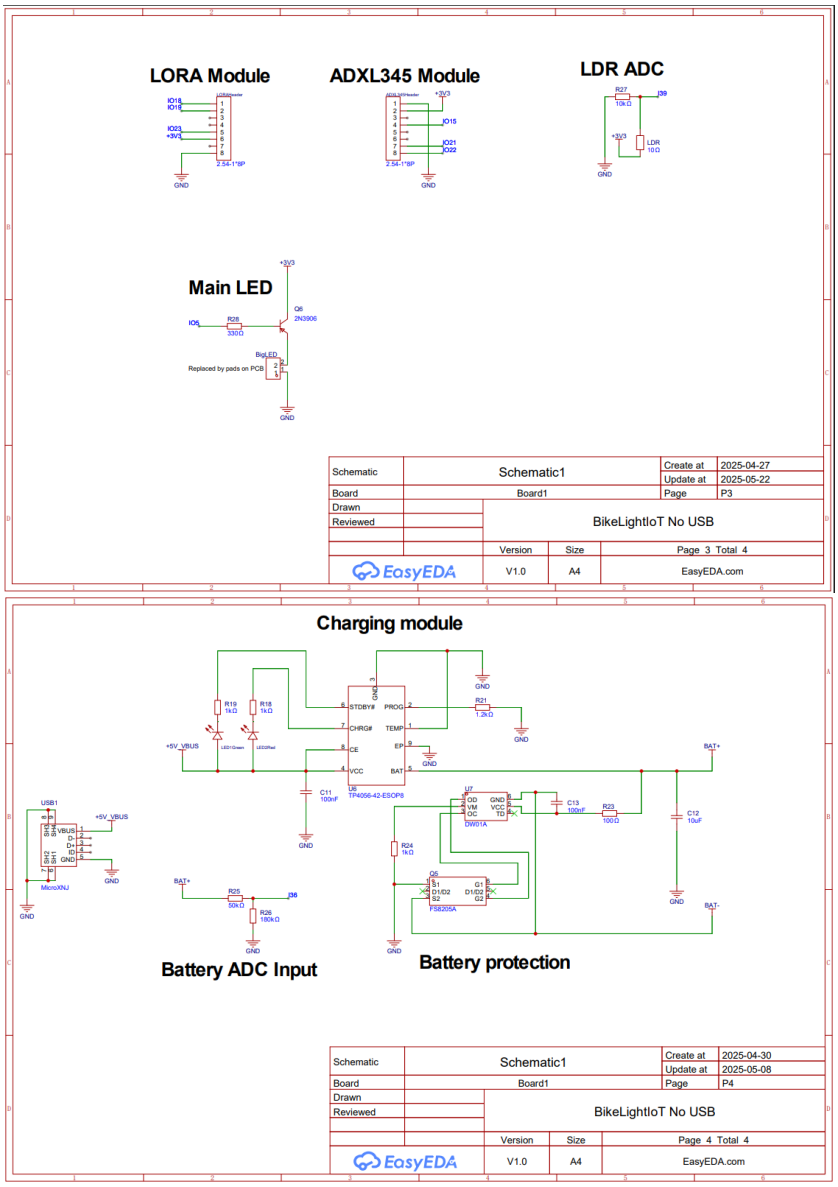


Figure 8: Page 3-4 of the electrical schematics for the PCB.