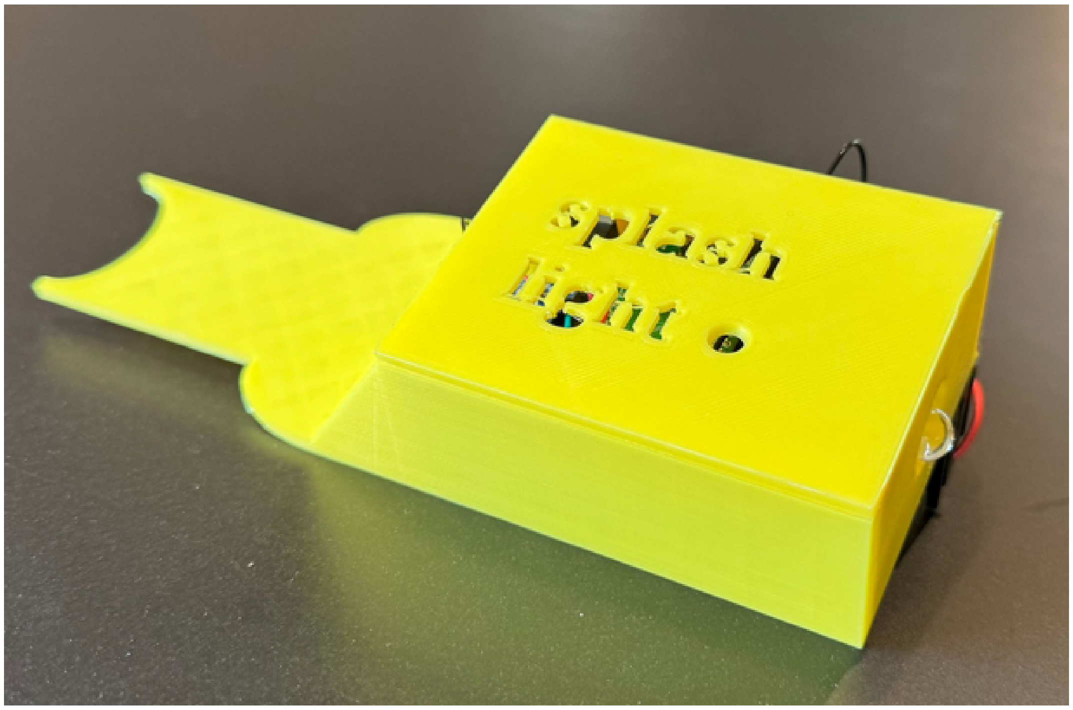
Technical University of Denmark



Splash Light 1.0 - Report

(34346) Networking Technologies and application development for IoT

Edoardo Santucci, s243100

Gea Staropoli, s243967

Julian Drachmann, s195152

Nicklas Mundt, s224218

Oskar Hvelplund, s194306

Pepe Rubbens, s247295

October 17, 2025

1 Introduction

Bikes have always been an integral part of Danish day-to-day life. They are convenient, easy, and largely free to use. According to the Municipality of Copenhagen, there are 1.37 bikes per citizen in the country[1] , and each of these citizens is mandated by law to have a bike light. Of all these cyclists, Sikkertrafik found that 4 out of 10 said they would continue to ride without a light[2]. This indicates that many people find bike lights inconvenient. This report introduces the *Splash Light*, which aims to solve this issue by integrating bike lights into the IoT ecosystem, making them smarter and easier to use by automating and enhancing their functionality.

1.1 User Needs

In order to evaluate the relevance of a smart bike light system, a large part of this project was spent on preliminary discussions about the key issues we aimed to solve with *Splash Light*. We identified four key areas:

- Lights are often forgotten at home or run out of battery unexpectedly.
- Cyclists value convenience, especially when commuting regularly.
- Visibility and safety at night or in dim lighting conditions are high priorities.
- Integration with mobile apps for tracking or theft prevention is desirable.
- Usability should be for everyone, especially rural areas.

The *Splash Light* system seeks to meet these needs by creating a seamless user experience: automatic lighting based on the current environment, power-saving features, and useful information on battery status and tracking via a connected app. Furthermore, by automating the on/off mechanism and minimizing user effort, the system encourages greater compliance with safety regulations and improves overall cycling safety.

This project aims to highlight these features and how they were implemented, along with insights into the design process of the *Splash Light*. Note that a link to the project's corresponding GitHub repository can be found in the appendix.

2 Design

In this following section, the design considerations behind the smart tail light we call 'Splash Light' will be outlined. As written in the introduction, beyond simply alerting others in traffic, this smart bike system integrates hardware and software components to provide the user with features such as real-time bike tracking and intelligent power management. Furthermore, as an extension of the Splash Light system, a fully functional Android app has been developed, making the user able to get useful real-time information as well as controlling it from the app. The technical specifications of the app will be expanded upon in Section 3.

2.1 Features

The main features considered for the Splash Light are described as follows:

- *Battery-driven, chargeable*

[1]`https://byudvikling.kk.dk/mobilitet/verdens-bedste-cykelby#:~:text=p%C3%A5%20en%20gennemsnitlig%20hverdag%20i,hvor%20end%20du%20kigger%20hen.`

[2]`https://sikkertrafik.dk/presse/pressemeddelelser/for-mange-cykler-uden-lys-i-morket/`

The board will run on a lithium-ion battery, capable of being charged by a Micro USB. Furthermore, the battery will be measured, to allow the system to send data regarding the charge status to the mobile app. If the battery runs below a certain threshold, Splash Light will also be able to produce a sound warning.

- *Location tracking*

The system will be able to send GPS location data (exclusively) per request from the mobile app. To maximize security for the bike, this request can be sent and received in the app at any point in time.

- *Focus on battery lifespan*

To extend the battery life, the system will be implemented with a focus of saving power when possible. This can be done by turning off certain features that are not used in the moment, or by utilizing the modules' sleep modes.

- *Automatic light*

By being aware of the movement of the bike as well as the surrounding lighting, the system will be able to turn on/off its light from keeping track of multiple criteria.

2.2 Software Architecture

To create a smart light capable of automatically turning on/off and react to a given environment, the system itself was designed to be able to switch between 3 distinct states/modes. These modes, defined as **active**, **parking** and **storage** would enable the device to determine and enable its needed components, while keeping the others from drawing power. The idea behind these modes are as follows: when you are out biking, the system should automatically be in the *active* mode. When you park your bike at your destination, it should switch to *parking* mode. Finally, if it is in the garage, or some other place where it will be stored for a longer period of time, the bike should be able to switch to *Storage* mode. In Figure 4 in the appendix, a flowchart visualizing the system described in this section is shown. It includes the three modes, as well as how they work and interact. Still, the three modes will be described below:

Active mode

This mode will be the most battery-consuming mode (battery life will be adressed in Section 4.3), as it will have the system continuously deciding if the light needs to be turned on/off, e.g. if you during a sunny day suddenly drive through a dark tunnel, the light should instantly react to this change in environment lighting. Also, the bike will check if the bike is moving, since it should be able to switch to parking mode if standing still for a certain period of time.

Parking mode

In parking mode, the light turns off, but the system should still check for movement, so the bike can go back to active mode when movement is resumed. This will allow for a smooth transition for when the user has to use it again.

Storage mode

The storage mode is where the light will use the least amount of power, since it is stored away e.g. in the garage at home. Here, the system will not react to movement, and the light will always be turned off. However, it should still send its geolocation uplink within a certain interval, to allow it to be tracked, in the unfortunate event of grand theft bicycle.

In every mode, wireless communication was assumed necessary for the device to be responsive to the mobile app at all times. The implications regarding this assumption is discussed further in Section 5.

2.3 Hardware choices

Based upon the features, as well as the software specifications mentioned, certain hardware components are required for the smart light to work. For the automatic light and mode switch, the system will need an LED, photoresistor and accelerometer. To send the Location values, either WiFi or a GPS (or both) could be used. Also, an audio buzzer is needed to produce sound warnings. Lastly, a 3D printed enclosure will be made to protect the components. More on the implementation of these modules and features in Section 4.

3 Communication

Splash Light communicates with the The Things Network (TTN) server via Long Range Wide Area Network (LoRaWAN). The data is then forwarded through Node-RED to the MQTT broker, from which the application retrieves it directly. This process works symmetrically for both uplink and downlink communication. A schematic of this communication is shown in Figure 1.

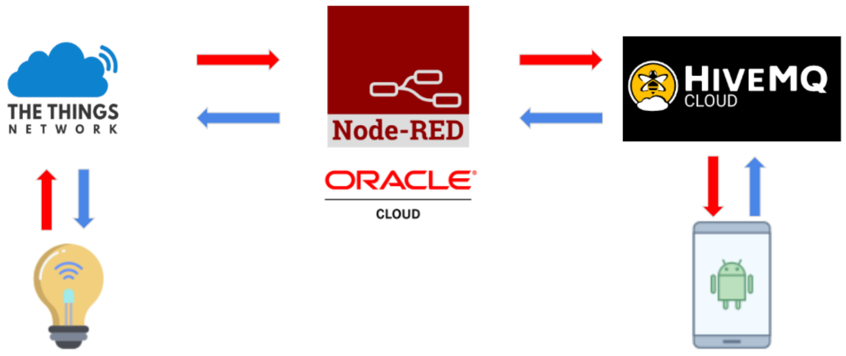


Figure 1: Schematic of the communication

3.1 LoRaWAN and TTN

For the communication between the device and the server, we decided to use LoRaWAN because it is a wireless technology, with low power consumption, that operates over long distances. It is also cost-effective, as it does not require any contracts with providers to enable communication. On the server side, we chose to use The Things Network (TTN) because it provides an interface for monitoring all exchanged data, which is useful for both debugging and assessing coverage.
The data sent from the device to the server includes the LED status, geographical location, mode status, and battery level (all of this information is always sent in separate data frames). To optimize efficiency, we decided to send only one byte per information, except in the case of geo-location data. The different message types and their corresponding meanings are detailed below in Table 1.
It is worth mentioning that both the GPS coordinates and the battery level are transmitted directly as numerical values. We carefully ensured that the hexadecimal codes used for these values do not conflict with the numbers corresponding to the battery status. Additionally, since

GPS coordinates are longer messages, they are formatted in a way that does not interfere with any of the reserved codes used for other data types.

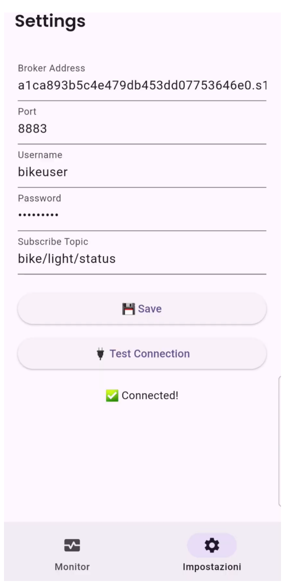
Data in HEX	Explanation
02	Active Mode ON
03	Parking Mode ON
04	Storage Mode ON
05	GPS coordinates requested
0B	LED ON
0C	LED OFF

Table 1: Codes and their explanation

Moreover, at the beginning of each connection when the device turns on, the device transmits the current mode (active, parking, or storage), the LED status, and the battery level to establish the initial state. To maintain a consistent connection and enable downlink communication, we implement a keep-alive mechanism consisting of a 4-step cycle: the LED status is sent three times consecutively, followed by the battery level once. These keep-alive messages are transmitted every 15 seconds to avoid overloading the TTN gateway while ensuring reliable data reception. This timing is determined by the structure of the LoRaWAN protocol, which is an open standard and allocates each device a 1% duty cycle. Finally, the GPS coordinates are sent only upon request from the app.
Although we initially chose to use LoRaWAN for the Splash Light, during development we realized that it might not be the best fit for our needs. First, we currently experience poor coverage due to the limited number of gateways across Denmark. Furthermore, we discovered that LoRaWAN is primarily designed for uplink communication and is not well suited for downlink messaging. This is because downlink messages are only transmitted in response to an uplink message.
Another issue we encountered with LoRaWAN is that not every message sent from the device is reliably received by the server, and consequently, not by the rest of the data chain. To address this, we decided to send duplicate messages at the start of each connection to improve overall reliability.

3.2 MQTT Broker

To transmit messages from the TTN server to the application reliably, we use an MQTT broker. In our case, we chose the broker provided by HiveMQ, due to its ease of integration and straightforward implementation for bridging communication between the TTN server and the application. The MQTT broker allows both parties to publish and subscribe to topics, enabling smooth and symmetrical bidirectional communication.


Settings

Broker Address
a1ca893b5c4e479db453dd07753646e0.s1

Port
8883

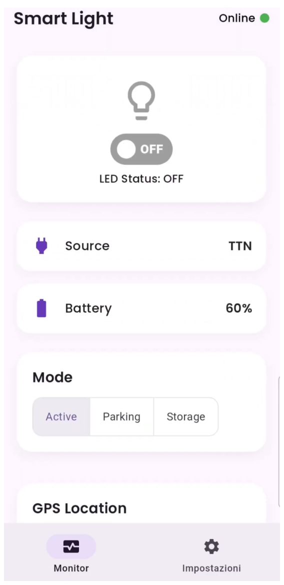
Username
bikeuser

Password
•••••••••

Subscribe Topic
bike/light/status

✅ Connected!

(a) MQTT Settings Page


Smart Light Online

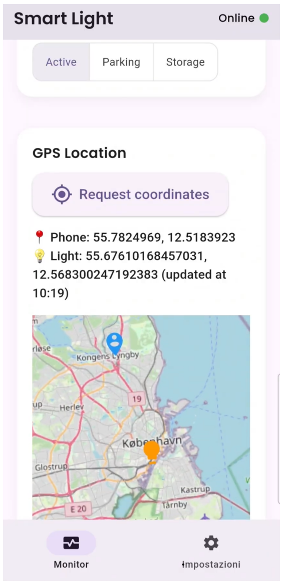
OFF
LED Status: OFF

Source	TTN
Battery	60%

Mode

GPS Location

(b) Monitor and Control Page


Smart Light Online

GPS Location

📍 Phone: 55.7824969, 12.5183923
💡 Light: 55.67610168457031, 12.568300247192383 (updated at 10:19)

(c) GPS Map View

Figure 2: Mobile application: settings, control interface, and GPS map view

3.3 Backend

NodeRED

For the backend implementation, we integrated Node-RED as a central component to handle both uplink and downlink communication and data processing between the TTN MQTT broker and our private HiveMQ broker. Node-RED is a lightweight, open source tool designed for flow-based programming with built-in MQTT support, resulting particularly suited for our IoT implementation. Once deployed, Node-RED performs two critical tasks:

1. **Message Forwarding**: It subscribes to MQTT topics exposed by TTN, which follow the structure:

   ```
   v3/<app-id>@ttn/devices/<device-id>/up
   ```

   Upon receiving an uplink message from the smart light, it parses the payload and republishes a new message to the corresponding topic on the private HiveMQ broker. For downlink communication, Node-RED subscribes to a designated HiveMQ topic for incoming user commands, reformats the data according to TTN's required structure, and publishes it to the appropriate topic on TTN's MQTT broker:

   ```
   v3/<app-id>@ttn/devices/<device-id>/down/push
   ```

2. **Message Translation**: As discussed previously, due to LoRaWAN's strict bandwidth and duty cycle constraints, the smart light transmits compact hexadecimal payloads to minimize message size. When these payloads are received by TTN, they are automatically Base64-encoded before being forwarded to the backend. However, the mobile application expects structured, human-readable data such as JSON or decimal values. To bridge this gap, Node-RED converts the Base64-encoded uplink payload from TTN into decimal or JSON for the app and vice versa for the downlink.

Cloud VM

Initially, Node-RED was tested locally on a personal computer. However, this approach had a clear limitation: the backend would only be active when the host computer was turned on, limiting service availability and reliability. To address this, Node-RED was deployed on a cloud-based VM hosted on Oracle Cloud Infrastructure. For remote management and real-time monitoring of the backend, a public-facing endpoint was established through the following configuration:

- A custom domain name (`https://bikehost1.ddns.net`) was registered and dynamically mapped to the VM's public IP address using Dynamic DNS service.
- A secure connection that relies on HTTPS access enabled by an autogenerated SSL certificate using Certbot and Let's Encrypt.

3.4 Frontend

The frontend of the system was developed as an android application using Flutter and consists of two main views.

The first is the *Settings Page* (Figure 5(a)), which allows configuration and testing of the MQTT connection to the HiveMQ broker. This view was primarily designed for development and debugging purposes but remains accessible to ensure connectivity is maintained.

The second and main interface is the *Monitor and Control Page* (Figure 5(b)). It displays the status of the smart light, including LED state, battery level, and type of communication (TTN). Users can manually switch the operating mode or view automatic changes made by the system. Additionally, the app enables GPS location requests: upon user action, a downlink message is sent through the backend to the smart light, which replies with its current coordinates. These are then displayed on an embedded map, showing both the position of the bicycle and the user (Figure 5(c)).

4 Hardware

In this section, the implementation of the hardware will be covered. Because of the lack of applicable pins on the used micro-controller, some modules and sensors were not able to be used. Note however, that they were still soldered onto the final prototype (just without being connected to pins), to illustrate how they would be used if enough pins were available. How the components were soldered onto the development board for the prototype can be seen in Figure 5 in the Appendix. The code for every component was also implemented with the same assumption that the pins were available.

4.1 Components

The **microcontroller** used in the project is the **Heltec ESP32-C3**. It features an embedded LoRa transceiver (SX1262), making it possible to connect our bike light to an application without an additional module.

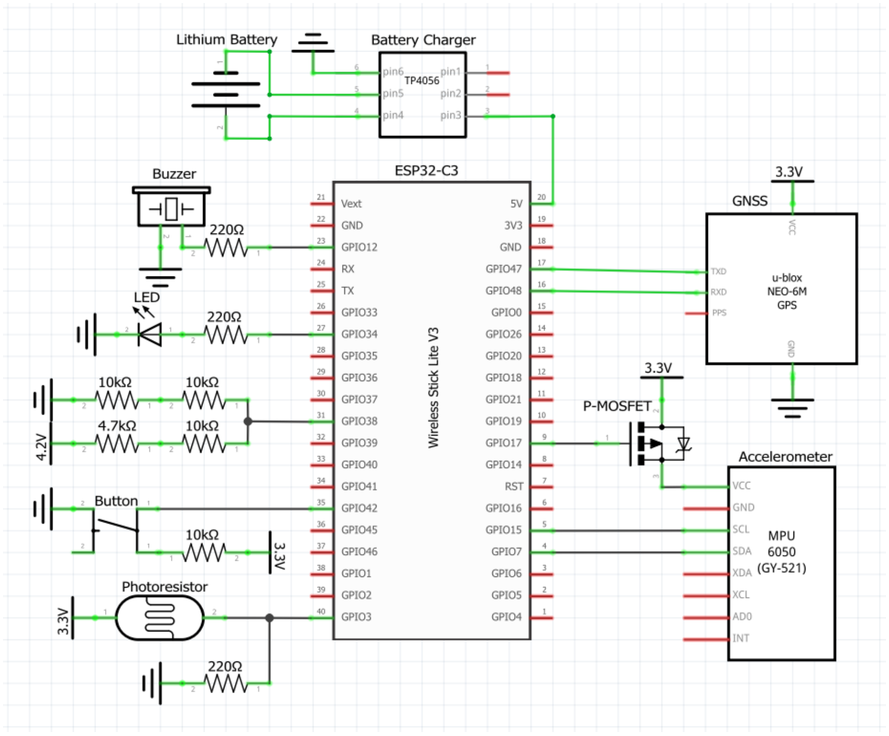


Figure 3: Schematics of the prototype. The micro-controller used in the schematic is not the same module as the one used in the prototype, but it is only used for visualization purposes of the design.

The system uses a **GY-521 module**, which features the **MPU6050** accelerometer and gyroscope. It communicates using I2C, which could help save pins, if multiple components uses it. It's primary function is to detect motion. When the bike starts moving, the Splash Light automatically turns on, removing the need for user interaction and improving safety through increased visibility. This is done through the Euclidean distance in three-dimensional space, and a threshold function.

The accelerometer is also used to help with switching between the 3 states as described in Section 2, active, parking, and storage. As the bottom-right of Figure 3 illustrates, the accelerometer uses three pins, as its power state is controlled using a P-MOSFET. This is used to turn off the sensor in the appropriate states to conserve power. The different states it is used in can be seen in Table 2.

The system includes a **GY-NEO6MV2 GNSS (Global Navigation Satellite System) module** to enable live location tracking through LoRa communication. GNSS uses GPS signals from multiple satellites to calculate accurate positioning anywhere on Earth. In the Splash Light system, this allows the user to track the bike's real-time location via a mobile app. This feature adds an extra layer of functionality by improving theft recovery and enabling ride logging, without relying on traditional cellular or WiFi networks.

A **photoresistor** is used in tandem with a built-in analog-to-digital converter to measure light levels. When it detects low brightness, such as at night or in dark conditions, the Splash Light automatically activates. This ensures visibility without requiring the user to manually switch the light on or off.

The **3D-printed enclosure** is designed so that it can be mounted onto a bike seat. This is done by shaping it in such a way that it slides in the back of the seat and can be clamped stuck by another 3D-printed component. To reduce weight and printing time, the floor is perforated with square holes.

Using four resistors (counting up to 20kΩ and 14.7kΩ) that split the voltage proportionally, a **voltage divider** is created to scale down a 4.2V battery voltage for safe input into an ADC. It would give an output of 1.76V in between the resistors when the battery is fully charged. The lower voltage is readable for the microcontroller and is converted into a percentage.

For recharging the battery a **TP4056-based lithium-ion battery charging module** is used. It has overcharge and overdischarge protection. Taking 5V input via a micro-USB port it charges the battery, that is connected to the B+ and B- terminals. The OUT+ and OUT- terminals regulate the output voltage to the microcontroller.

A **white LED** working as the final output as the backlight itself. It needs a current of 30 mA. Its status is dependent on the accelerometer and the photoresistor, as it will turn on only when movement is detected while it is dark outside.

For the battery warning a simple **buzzer** is used. It works if the microcontroller sends a power signal through the module.

The bikelight features a **button** with which the whole device can be turned on and off.

The battery used is the **ICR18650-26J, a lithium-ion rechargeable cell** from Samsung SDI. It has a typical capacity of 2600mAh and a nominal voltage of 3.63V. It delivers a consistent output and supports a maximum continuous discharge current of 10A. A voltage regulator is integrated in the microcontroller.

4.2 Board architecture

For the architecture of the physical prototype itself, it was created and soldered to be as compact as possible, to reduce the space it would take up on the bike.

As seen in the schematics in Figure 3 and as already mentioned in this section, lots of pins was needed for the components. This did create some challenges, as will be discussed in section 5. It was initially assumed that certain pins could be used for the prototype, but it was later discovered that many of them were unavailable due to being reserved for LoRaWAN. As a result, the board needed to be resoldered to exclude the reserved pins. Even though the datasheet for the Heltec module revealed another, unused pin slot, it also revealed that fewer than the expected number of pins were available, meaning there were not enough for all the modules. This limitation led to some components being left unconnected. To illustrate how the Splash Light would function with sufficient pins, all components were still soldered onto the prototype board—they were simply not connected to the ESP32.

Component	Current	Comment	Active	Park	Storage
Microcontroller	40-60 mA	According to basic CPU load	x	x	x
Accelerometer	0.5 mA	not using gyro	x	x	
GNSS	20-30 mA	approx. 2-3s when asked for.	x	x	x
LED	30mA	on/off	x		
LoRa uplink	120-130 mA	100-200 ms per uplink (every 15s)	x	x	x
LoRa downlink	10-12 mA	1-2s after uplink	x	x	x
Miscellaneous	3mA	Always on like V-divider, or neglectable like buzzer	x	x	x

Table 2: An overview of the components and their contribution to the power consumption. In the right part of the table, it is specified *when* when the specific component is used

4.3 Power Consumption

In Active mode all components are running. The following things were assumed making a calculation according to the table 2 above:

- GNSS is not used while in active mode, since the person is riding the bike and knows where it is. This is assumed for normal use, where the bike is not stolen or lend out
- The microcontroller has an average CPU load of 60 mA.
- Miscellaneous components are constantly taking a current of 3 mA

So, the current used in Active mode when it is bright enough outside is 63,8mA. When the light is turned on this will be 93,8mA. It takes 40,7 hours and 27,7 hours to drain the 2600mAh battery respectively.

In Park mode the following assumptions are used to make the calculation.

- GNSS is used once every hour for 10 minutes, coming up to an average usage of 5mA.
- The microcontroller has an average CPU load of 50 mA. Due to less modules needing steering.
- Miscellaneous components are constantly taking a current of 3 mA

In Park mode the LED is always off making the usage of the bikelight count up to 58,8mA. This will drain the battery in 44,2 hours.

In Storage mode the following assumptions are used to make the calculation.

- GNSS is used once every four hours for 10 minutes, coming up to an average usage of 1,25mA
- The microcontroller has an average CPU load of 40mA. Due to less modules needing steering.
- Miscellaneous components are constantly taking a current of 3 mA

In Storage mode the LED is always off making the usage of the bikelight count up to 43,5mA. This will drain the battery in 59,8 hours.

Sketching a situation where a biker uses the bike light to go to work early in the morning and go back home late in the evening (2 hours roundtrip). While the light is in Active mode the LED is on for 60 percent of the time. At day the owner keeps the light in Park mode for the 9 hours while he is at work and at night he stores it in the shed for 13 hours total in Storage mode.

Coming out on average 50,7mA usage. In conclusion, in normal everyday use, the battery has to be charged every 51,3 hours.

5 Challenges and Discussion

As mentioned in both the Design{2} and Hardware{4} Section, there were some complications during development. Before soldering, we made sure that each component was functioning correctly by running independent tests on the ESP32 board. However, by doing those tests independently, we did not realize that four of our available pins corresponded to the SX1261's lora pins, which thus meant we could not use said pins and therefore lacked a sufficient number of GPIO's. A theoretical solution could be to include an IO-extender, or intermediate I2C modules for each component, so that the photoresistor, buzzer and the GNSS component can be put onto the I2C bus. This would also require for the photoresistor to have an ADC component as well.

As for the complication raised by LoRaWan being kept on constantly and thus ensuring that the Splash Light remains responsive to the mobile app at all times, we do not have a clear solution, for to meet our requirement, the LoRa component must persist through deepsleep (and we need deepsleep to avoid/significantly improve the otherwise 2-day battery life that we concluded in the Hardware Section{4}.). Now to grasp the issue, our preemptively implemented design for deepsleep has to be understood:

Each time the ESP32 wakes up from deepsleep, it would check its RTC memory, which would contain its mode, the time since movement was detected, and a bitmask of activated modules. First, it would use the data in these registers to enter into either active, parking or storage mode. Now, active mode is arguably the most intriguing of the bunch, for it is possible to make it fairly power efficient using deepsleep: when entered, it would first detect if the LED should be on, and act accordingly. Afterwards it would utilize the bitmask to activate the accelerometer - either the hardware component and digital representation, or just its digital representation (no need to recalibrate if it was already calibrated during the ESP32's last wakeup). The Splash Light will then assess if it is moving, reset/update the "time since last movement," decide whenever to enter parking mode next time it wakes up, set the watchdog timer, and prepare for deepsleep.

Before the microcontroller goes back to deepsleep, it would check whenever the LoRa pin caused it to wake up, and if so, it would wait for an incoming message, act upon that next message's command, and go back to sleep. Essentially, the application would start any command or request by first sending a blank or meaningless message, to cause the SX1261 Lora Device to activate its data pin, which would be set as an interrupt pin, waking up the ESP32. It matters not if the packet is lost, for it is the message after the first one which carries the command. That way, the Splash Light would remain responsive, even if it is in deepsleep. Unfortunately, we were unable to prevent the built-in SX1261 from also powering down during deepsleep - though we did hear from other groups that immediately after it wakes up, then it ought to still retain its connection - which would solve our problem, as storage mode and parking mode could have a much shorter sleep interval to ensure that it still remains responsive. This, and given that both the bitmask functionality and deepsleep, interrupt and wakeup timer is already implemented, meant that the main future works would be to convert our active, parking and storage mode upper function into a deepsleep-variant.

A final consideration is that WiFi scanning is less power-intensive than GNSS, so it could potentially be an alternative for location tracking. However, as our clients are active in both rural and urban areas, and WiFi scanning either requires a connection to "Google Location Services" or to have a local database - not ideal for microcontrollers which have limited memory -, we decided to go with GNSS instead. One could argue for future work to make an implementation utilizing both GNSS and WiFi scanning, as WiFi scanning could be more dependable in urban areas, while GNSS would be better in rural areas.

Were we to implement WiFi Scanning, then VANET I2P interaction could have been a worthwhile addition for urban clients, but this might have complicated deepsleep further. another potential expansion, would be to communicate with the "Google Location Services" through LoRa - but that would require a significant increase to the development time.

6 Conclusion

In conclusion, Splash Light offers an IoT-based solution aimed at improving the safety and usability of bike lights in urban environments like Copenhagen, where cycling is an essential part of daily life. The system was designed in response to identified user needs, defined in Section 1.

Our implementation addresses these challenges through a smart microcontroller-based design using the Heltec ESP32-C3, which integrates a photoresistor, accelerometer, and GPS module. Automatic lighting ensures the light is always on when needed, without user intervention. The system is able to monitor battery level and communicates status updates, helping avoid sudden power loss. The mobile app allows users to track the device, check battery status, change modes, and locate the bike, enhancing both usability and security.

Ultimately, while the proposed implementation successfully addresses key user needs, limitations in available GPIO pins on the chosen microcontroller prevented the integration of some tested components. Future developments could overcome these constraints through more capable hardware or by implementing the solutions proposed in Section 5.

7 Work Distribution

Section	Responsible
Introduction	Oskar
Design	Nicklas
LoRaWAN and TTN	Gea
MQTT Broker	Gea
Backend	Edoardo
Frontend	Edoardo
Sensor components	Oskar
Power components	Pepe
Board architecture	Nicklas
Power Consumption	Pepe
Challenges and Discussion	Julian
Conclusion	Edoardo

Project work	Edoardo	Gea	Julian	Nicklas	Oskar	Pepe
Sensor high-level			15%		85%	
Sensor low-level			75%		25%	
Pin mapping			60%	40%		
Setup of body of the main code				50%		50%
Main code troubleshooting	20%		30%	50%		
LoRaWAN implementation	20%	80%				
MQTT setup	50%	50%				
VM and NodeRED setup	50%	50%				
Application development	100%					
Battery				50%		50%
Soldering				60%		40%
CAD, 3D Printing						100%

8 Appendix

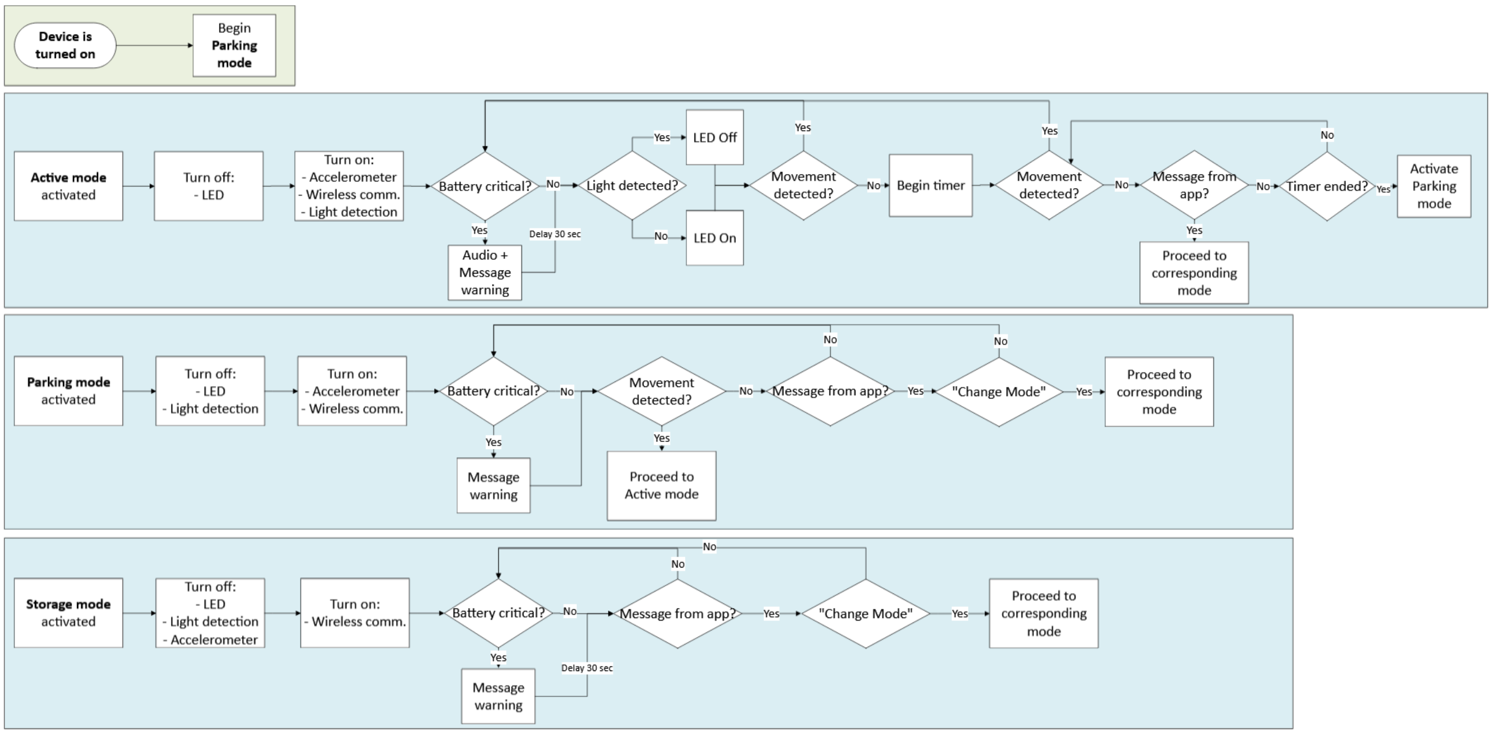
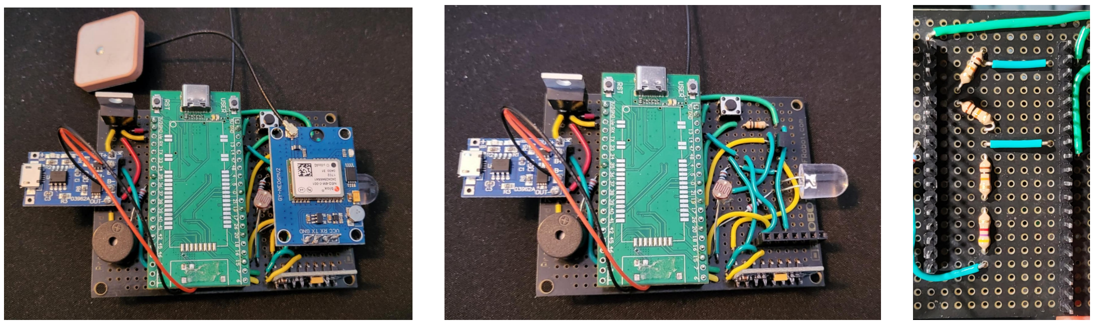


Figure 4: The dynamics of the different modes is visualized through the flowchart.



(a) Full prototype architecture (b) Prototype without GPS (c) Voltage divider

Figure 5: Three pictures of the prototype soldering, also showing the more hidden components underneath the GPS and ESP32.

Link to GitHub Repository

You can find all the code for this project in the GitHub repository called ”SplashLight”. You can find it here