Design Report

BATA: Development of Embedded Firmware for Wireless LED Display and Time Registration System

Group 2:

Lucas Pruijssers — s3179176

Rune Ebbers — s3170268

Tristan Bottenberg — s3131599

Marijn de Haan — s3200175

Tim van Beek — s3194108

Case Owner: Bonne Zwaga

Supervisor: Gerwin Hoogsteen

April 2026

Abstract

The Batavierenrace is a large student relay race spanning from Nijmegen to Enschede. In the past, custom hardware and software has been implemented, including a time registration system that displays participant registrations. This project aims to refactor this system to separate the displaying logic from the registration logic. This is achieved by establishing connections dynamically and computing displayed contents using information received over the established connection, rather than receiving display instructions directly from the registration device. All required features are implemented, in addition to several optional features.

Contents

1 Introduction

The Batavierenrace is one of the largest relay races in the world, with 8500 participants and up to 300 runners active at the same time. The race spans 25 relay points, where lap times of runners are recorded. Over the years, custom hardware and software have been developed that meet the unique needs of the event, including a time registration system.

Currently, this system consists of a registration application which is linked to a display. To create registrations, the application requires certain information, such as the start time and the runner's number. This information is provided by a centralised database, which updates the application when necessary.

This application, originally intended for simple registrations, has since expanded to include a number of other features outside of its original scope. Consequently, the Batavierenrace's committee, eBART, requires a stronger separation of concerns in this application. One such separation is between the registration software and the display. These displays show information to runners, such as the current time, their ranking, and their stage time, among other information.

1.1 Project Goal

This project aims to refactor the old registration system to separate the displaying logic from the registration logic. In the new system, the display handles and processes data autonomously, rather than receiving instructions from the registration device. The secondary aim is to develop a testing environment for the new system, which aids future developers in maintaining or updating the produced codebase by enabling them to assert correct system functionality.

1.2 Concepts and Terminology

This section describes relevant concepts and terminology for this project.

1.2.1 Concepts

Every registration made at a relay point consists of a TeamID and a stage time. A TeamID is a number unique to each team, which is identified by a unique registration tag. The existing system specifies the following registration tags:

- S-tag: The standard tag used to identify a team. Only S-tags are displayed to avoid duplicate team numbers, as the S and R prefixes are not shown.
- R-tag: The Reserve tag acts as a replacement tag for a team which cannot use their S tag, for example, if they lost their vest or their transponder malfunctions. R tags are linked to a team's S tag, so registrations made with the R tag are linked to the correct team.

Each relay point has a sensor in close proximity, roughly 20 meters, before the relay point. Each time a tag passes it, a registration is made. The sensor specifications are outside this project's scope. Each registration must be shown on the display. In the case where an R-tag cannot be converted to an S-tag, meaning there is no TeamID to represent that registration, the TeamID is substituted by a placeholder value. Currently, three dash characters are used as a placeholder, since TeamID's are at most three digits long.

The RK requires a start and finish time to calculate the stage time. A start time indicates when a team started running a stage and a finish time indicates when they arrive at the respective RK. The RK creates a log book entry for each registration, which record registration information, such as the TeamID and finish time of the finished team. However, start times need to be retrieved from the relay point prior to this RK. It receives this information through the central database, which globally stores registrations made by RKs.

Additionally, registrations can manually be added, edited or given penalties. This is possible through a tablet, which connects to the RK. Lastly, teams that should prepare are displayed by typing their TeamID in a keypad. Both these components are outside of the scope of the new system, and no changes have been made to either of them.

An abstract overview of the system is shown in Figure 1.

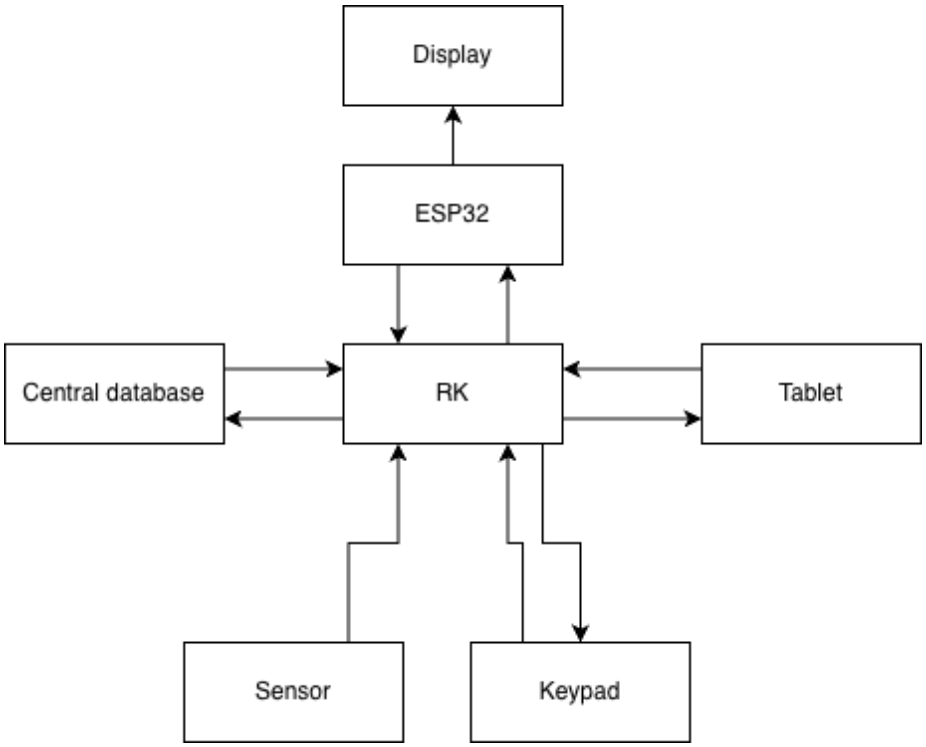


Figure 1: A system diagram showing a high-level overview of the system

1.2.2 Terminology

This section describes terminology used in this document.

- **BATA**: Abbreviation of the Batavierenrace.
- **Runner**: Someone participating in the BATA
- **RK**: A registration box, which registers that a runner has finished. Interchangeably used with a registration device or system.
- **Test Suite** The developed testing environment for the new system.
- **WP**: A relay point in the race.
- **Stage time**: Time taken to complete a stage from the previous relay point to the current relay point.

2 Project Planning

This section describes the project planning. This includes a section on design phases, which globally details each phase of the project, collaborative agreements within the team, communication with the client and communication with the team's supervisor. This will be reflected on later under the process section of the report.

2.1 Design Phases

The project is structured into five phases.

- The Planning Phase
- The Analysis Phase
- The Design Phase
- The Implementation Phase
- The Testing and Integration Phase

Each phase corresponds to a phase in the development life-cycle.

2.1.1 Planning Phase

The planning phase consisted of conferring with the client and making agreements regarding the project scope of. A planning was made during this phase, which outlined how long each phase should take and what should roughly be done in the respective phases. This phase started in week 1 and finished in week 2.

2.1.2 Analysis Phase

The aim of the analysis phase was to analyse the client's requirements and gather required data and information to implement the project. The client provided documents, including a system description. Requirements were developed based

on these documents, which were iteratively improved according to the client's feedback. The client did not provide all documents at once, rather they provided documents they found necessary for this stage of the project. Consequently, certain requirements were changed in later weeks, since the client deliberately withheld information relevant to the system's requirements. The phase was planned to start in week 2, and finish during week 3. The team did not deviate from this planning.

2.1.3 Design Phase

During the Design Phase, designs were made for the new display. This includes designs for how network connections are established between this system and the registration system and how they communicate. This was accomplished using data flow diagrams, class diagrams, and system mock-ups, where designs are verified and tested for correctness and feasibility. These mock-ups provided valuable insight into the design decisions, since they highlighted issues the team overlooked. This facilitated iterative design refinement prior to the finalization of the design. This phase was projected to start in week 3 and end in week 4. The team spent half a week longer on the design phase than planned; it ended halfway through week 5.

2.1.4 Implementation Phase

The purpose of the Implementation Phase is to realize the designs from the Design phase. Schemas and designs made during said phase serve as a baseline for implementation. The client expressed the need for a robust test suite, as this would improve the quality of life for future developers. Consequently, the implementation of a test suite was assigned high priority, and would be a primary focus for early stages of development. Besides this, the test suite would aid developing and testing other key components of the system.

In reality, the test suite was the last system component the team implemented. The team prioritized a Minimum Viable Product of the displaying

system over the test suite implementation. This phase was projected to start in week 4 and end around week 8. However, due to the extension of the design phase, bugs, issues, code dependencies etcetera, it was finalized in week 9. Consequently, development of the test suite started and finished within week 9.

2.1.5 Testing and Integration Phase

Finally, the Testing and Integration Phase consists of a thorough examination and testing of all the software. Although a separate phase is outlined for testing, Test Driven Development (TDD) would be adopted to ensure correct functionality while implementing the project. This Phase was projected to start in week 8 and end in week 10 at latest.

Although the aim was to use TDD, this was rarely done properly. Rather, a component would be developed, empirically tested and reviewed before tests were developed for the component. However, testing was still performed throughout the entirety of the project and the last week primarily focussed on testing code as planned.

2.2 Collaborative Agreements

2.2.1 Codebases

From the analysis phase and provided code, the team concluded that there would be two separate codebases: one for the RK and one for the display. Any code related to the RK was supposed to be written in Java, using camel case as a naming convention. Since the RK code already existed prior to this project, no agreement was made on the code structure there. All code in the display repository exclusively be written in C++. No agreements were made on coding conventions or code structure.

2.2.2 Git Usage

The team used GitHub as a development platform. The initial plan was to make a branch for each implemented feature. Once a feature is implemented,

a pull request should be created. A reviewer would be assigned to the Pull Request, who would provide feedback. All feedback must be implemented or resolved before merging with the main development branch. Additionally, the issue board would be used to indicate the status of issues. Lastly, each issue would be labelled according to the issue specifications.

2.3 Client Communication

The group planned to have weekly meetings with the client. During these meetings, the client could provide iterative feedback on what was developed the week prior. An agenda would be sent the day before, so the client had room to prepare if necessary. Besides meetings, the client and the team agreed to use mail as the primary communication method.

2.4 Supervisor Communication

The group planned to have weekly meetings with the supervisor. During these meetings, the group would discuss their progress and findings made the previous week. This would be used to discuss any obstacles or issues the group encountered. An agenda would be sent the day before, so the supervisor had room to prepare if necessary. Besides meetings, the supervisor and the team agreed to use mail as the primary communication method.

2.5 Team Communication

The team planned to use both WhatsApp and Discord for communication. The team wanted to meet as frequently as possible. The aim was to meet daily at half past nine in the morning. At the start of these daily meetings, a small briefing would be held where team members inform each other what they would be doing or what they did the day prior.

3 Requirement Specification

Requirements describing the system were developed, based on the client's needs. The MoSCoW method is used to prioritize these requirements, assigning higher priority to requirements more integral to the product. Additionally, a distinction is made between functional and non-functional requirements. The team completed all **must** requirements. Additionally, most **should** and **could** requirements are implemented, with some exceptions. The below list shows the developed requirements. A later section will elaborate on why certain requirements were not (fully) implemented.

3.1 Requirements

This section covers the requirements established during the analysis phase. These requirements serve as a contract between the team and client, detailing what this project promises to deliver. All **must** requirements are considered mandatory, while **should** and **could** requirements are considered optional.

3.1.1 Display requirements

This section states requirements pertaining to the display.

Functional:

Must:

- The display **must** be able to show relevant data gathered from the RK.
- The display **must** show the current time, such that the runners know what time it is.
- The display **must** show the stage time when a runner passes, such that their team knows how long the stage took.

- The display **must** show the WP number, such that the runners are aware which WP they are located at.
- The display **must** show the total number of teams that have passed this WP, such that third parties, such as medical support, are aware of the current advancement of the race.
- The display **must** show the team number and running time upon a runner passing a WP, and keep this visible for a to-be-determined minimal duration, such that this is visible to spectators and runners alike.
- The display **must** show the team number of the teams that need to ready for their change at the WP, such that the runners can be prepared for their switch.
- The display **must** normally show a maximum of 3 team registrations. If the system is busy, more of the display can be freed up for showing more registrations by hiding the scrollable list of registered teams.
- The display **must** show the team number of a team that needs to be ready at the WP for a limited time, depending on whether the runner has been registered.
- The display software **must** be updatable to allow new versions to be deployed.

Should:

- The display **should** show the ranking of a runner when they pass, such that their team knows how they are doing.
- The system **should** not display all already displayed finishing participants after a reboot or connection reset.
- The display **should** handle time fluctuations, such as sudden jumps in the RK time after a reboot.

- The display **should** show the team number correctly when they have a reserve transponder.
- The display **should** show its software version during startup.
- The display **should** indicate that it has an active connection to the RK.
- The display **should** indicate that it has an active network connection.
- The display software **should** be buildable in a Windows environment.
- The display **should** include a test mode that can simulate a variety of patterns to verify system robustness.

Could:

- This display software **could** have logging of important events.
- The display **could** show all registered teams in a scrollable list, including leading text.

Non-Functional:

Must:

- The display text/items **must** be visible in different weather and lighting conditions.

Should:

- The display **should** maintain a maximum deviation of 500 milliseconds from the RK time, preferably smaller if possible.

3.1.2 Networking requirements

This section states requirements for establishing and maintaining a network connection between the display and RK.

Functional:

Must:

- The system **must** connect through a router with independent SSIDs to other components of the system (e.g. RK, Tablet, Command Center).
- Protocol **must** be compatible with the RK and ESP32.
- The display **must** automatically establish a connection to the network and to the RK after a reboot of either the display or the RK, or after a temporary network interruption.

3.1.3 Test Suite requirements

This section states requirements for the testing environment, including how tests are run on the system, what components are testable and how the testing environment is structured.

Functional:

Must:

- The test suite **must** contain a test mode that can be used to simulate a variety of patterns, such that the system can be tested for robustness.
- System components **must** be able to be tested individually.

Should:

- The test suite **should** be independent of hardware.
- The test suite **should** be able to simulate system components.
- The test suite **should** include a stub implementation of the RK that simulates RK behaviour, implemented in Java and compatible with Java 8.
- The test suite **could** include control over RK and display to automatically test scenarios.

- The test suite **could** include representation of display data internals. This is to assist in developing, testing and debugging.

3.2 Altered Requirements

This section outlines the requirements whose interpretations changed over the course of the project, resulting from knowledge and context gain through development.

- *The display should handle time fluctuations, such as sudden jumps in the RK time after a reboot.*
- *The display should maintain a maximum deviation of 500 milliseconds from the RK time, preferably smaller if possible.*

The interpretation of both requirements are changed for the same reason. In the old system the RK would communicate its time to the ESP32 given their direct connection. However, in the new system the ESP32 retrieves time from an NTP server, since it already has access to the internet. Consequently, although this ensures the displayed time is accurate, it may lead to desynchronization with the RK's time.

3.3 Unimplemented Requirements

This section describes the requirements that were not implemented or only partially implemented. Each item includes an explanation for why these requirements are ultimately considered unfinished.

Unimplemented Requirements:

- *The system should not display already finished participants after a reboot or connection reset.*

  It is not possible to persist the data within the ESP32 since it does not have persistent storage. However, the system is designed to support this

requirement regardless, as it is possible to distinguish whether registrations should be displayed through the use of a bit flag in the packet sent by the RK. However, this feature is not yet implemented in the RK. Until the RK code is updated to support this feature, all finished participants are displayed again after a reboot.

- *The display software should support logging of important events.*

  Implementing logging was not feasible, since the ESP32 does not have persistent storage where logs could be stored. It is theoretically possible to store logs, by sending them to the RK, but this interferes with the client-server architecture of the system and is not an elegant solution. Besides this, all important events are executed and logged by the RK. Consequently, adding logs to the ESP32 would be redundant.

- *The test suite should be independent of hardware.*

  Although the test suite can be executed on any local machine, visualizing results requires a connected ESP32 and display. This is necessary because the relevant frameworks target the ESP32 hardware and depend on its peripherals. Technically, you should be able to run the test suite on an emulator but due to time restrictions it was out of the scope of our project.

- *The test suite should be able to simulate system components.*

  Simulating the ESP32 proved too complex. Simulating the RK is possible, through a socket simulation, allowing test interactions with an ESP32.

- *The test suite should include a stub implementation of the RK in Java (Java 8 compatible).*

  This requirement was not implemented because simulating the ESP32 introduced unnecessary complexity. Instead, an RK socket is simulated and relevant parts of the existing RK codebase are reused. A separate stub implementation was deemed redundant as a result. Furthermore, since only the stub implementation of the RK required Java 8 and the

test suite is independent of the RK. Java 24 has been chosen such that a more modern version of the GUI framework can be used.

4 Technical Design

The following section presents details and choices regarding hardware specification, system architecture and codebase structure.

4.1 Hardware Specification

The ESP32-WROOM-32 microcontroller is used to interact with the display and registration device. Whenever this document refers to interactions between the registration device and the display, these are processed and communicated by the ESP32, as shown simplified in Figure 2.

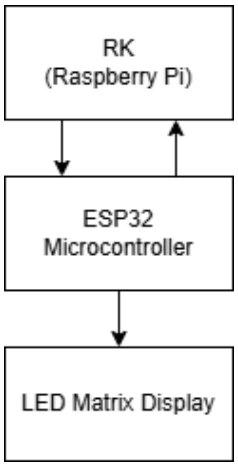


Figure 2: Interactions between the RK and the Display, with an ESP32 as a medium.

The display consists of five 64x32 RGB LED Matrix - 320x160mm panels. The ESP32 is mounted on a custom PCB and controls five daisy-chained RGB LED matrices using the HUB75 protocol. Power is provided by the battery box in the WP's vehicle. It is a 5V system, as to prevent straining the ESP32 with a high current. However, this does mean that the output of the 12V battery needs to be converted. Both the display and the power supply are provided by the client. The team did not assist with the design or creation of this hardware.

4.1.1 PCB Structure

The PCB architecture is shown in Figure 3. Only the ESP32 and HUB75-D Connector figures are relevant to the displaying system. The pins from these figures are used in the developed display software to send instructions to the display.

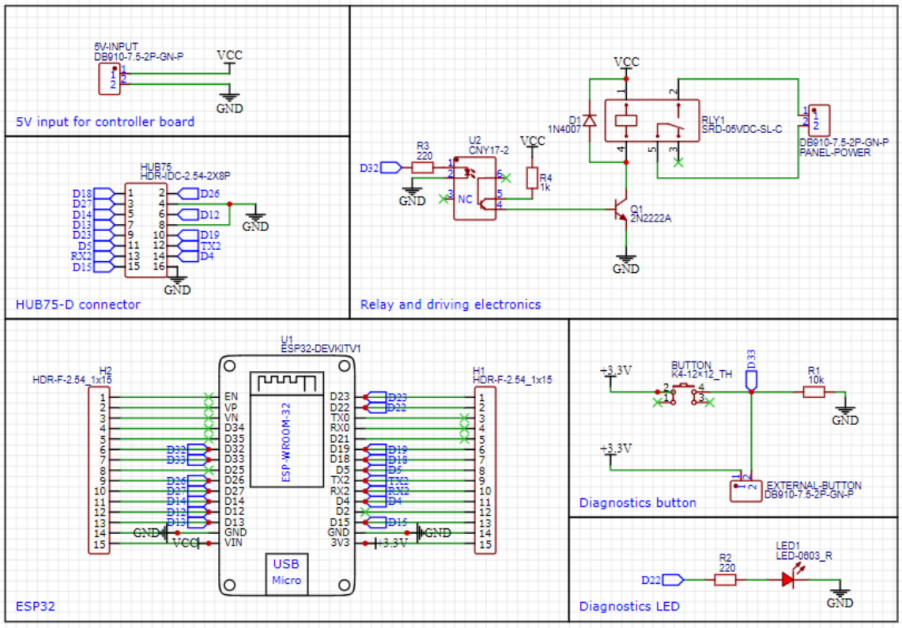


Figure 3: PCB layout

4.2 System Architecture

The system assumes a client-server architecture, where the RK acts as a server providing data and the ESP32 acts as a client. It was briefly considered to use a client-to-client architecture, where both the ESP32 and the RK could request data. The protocol was initially designed around this notion, however, after implementing this, the team realized the ESP32 does not need to request data in the first place. Consequently, since the ESP32 only consumes data, the protocol was changed to a client-server architecture.

4.3 Firmware

The ESP32 is responsible for display functionality, including receiving data from the RK, processing said data and turning this data into instructions. The codebase is divided into two parts; those being the part that processes the RK information and the part that communicates with the display.

4.3.1 Core Distribution

The display needs to be updated and be fed instructions by the ESP-32 constantly due to the scrolling elements and the clock. Consequently, separating the generation of frames and the handling of data onto separate cores will ensure frames will be processed continuously even if there is a lot of data to handle.

The ESP32 supports the use of RTOS software to aid with internal task and process assignment, and comes bundled with FreeRTOS. FreeRTOS works with tasks, which are pieces of code that can be assigned to a thread or core. To achieve the separation between cores, FreeRTOS provides a function called xTaskCreatePinToCore, which takes a task as an argument and which core should run said task, among other input parameters such as task priority. Additionally, the libraries that are needed for the implementation all require and use FreeRTOS, which allows for smooth integration and configuration of the libraries' functionalities.

Whenever a message from the RK is received, it is first processed by a class that determines what kind of information is included in the packet. It does so using the custom protocol built for RK and ESP32 communication, which will be elaborated upon later in this section. When the packet type is determined, the information is sent to a class that puts the obtained information into memory. This part of the system runs on its own thread and tries to write to memory whenever data is received from the RK. These classes and the interactions between them can be seen in Figure 4. The Network, RKCommunicator, and DataProcessor class are particularly of interest, since these handle the core system logic. In Figure 5, the functionality of this part of the system is encap-

sulated in loop 1. In other words, loop 1 is the sequence of actions that are continuously executed on this ESP32 core.

Running in parallel is another core that is solely dedicated to reading the gathered information from memory, formatting it into a frame, and sending this frame to an instruction sender class which sends appropriate instructions to the display. In Figure 4, you can see that the FrameGenerator class obtains its data from the same memory that DataProcessor writes to. Once it is done reading memory, it uses Formatter and a Frame object to create an internal representation of the display and what LEDs need to be which colour. Such a Frame object is a 2-dimensional array of tuples containing integers, representing RGB values. That Frame object is then passed on to multiple formatters that each take in their own data and edit a specific part of the Frame object. For simplicity, only one formatter class is shown Figure 4. FrameGenerator calls certain formatters based on what data it reads from memory. After all the necessary formatters have been called, the Frame object is considered ready and is subsequently passed on to the InstructionSender. This InstructionSender iterates over the Frame object and sends instructions to the display, after which the received RK data is visible. In Figure 5, this core's actions are visually represented in loop 2.

In the same figure, it can be seen that both of these cores try to access the same memory simultaneously. To account for this, memory read and write operations are managed using a mutex object. In this way, the system prevents the most prominent concurrency issues, ensuring that the data displayed on the screen is reliable.

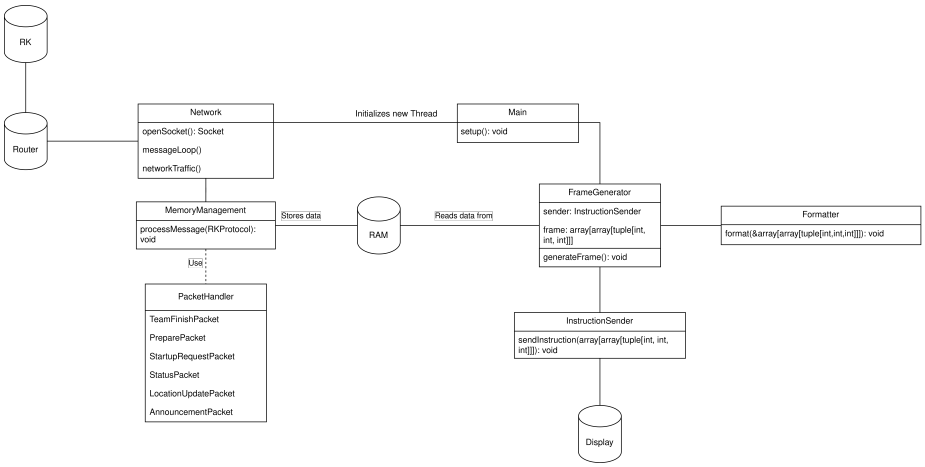


Figure 4: Class diagram of the ESP32

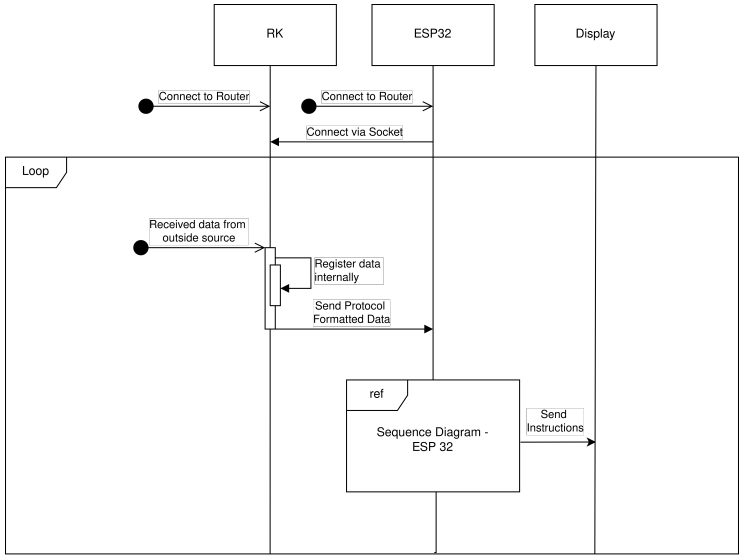


Figure 5: Sequence diagram of the ESP32

4.4 Connection Architecture

A network connection is required for the RK and ESP32 to transmit data to one another. The RK and ESP32 are always in close proximity with one another, so a local network suffices. A TCP socket is used for communication between the RK and ESP32, since this is a simple communication medium with the minimum requirements of being able to transmit all data while ensuring its validity.

The client argued that WebSockets could be used instead of sockets. The inclusion of a TCP socket would break the uniformity of the current system, which exclusively uses WebSockets. This might cause issues for future developers who are unfamiliar with WebSockets. However, WebSockets provide additional higher level functionality, which is unnecessary for the newer system. Consequently, they would cause unnecessary overhead.

After careful consideration and discussion, the decision to use a TCP socket instead of a WebSocket was approved by the client as long as sufficient documentation is provided, which would provide clarity for maintainers of the system.

The sequence diagram in Figure 6 shows how a network connection is established between the RK and the Display.

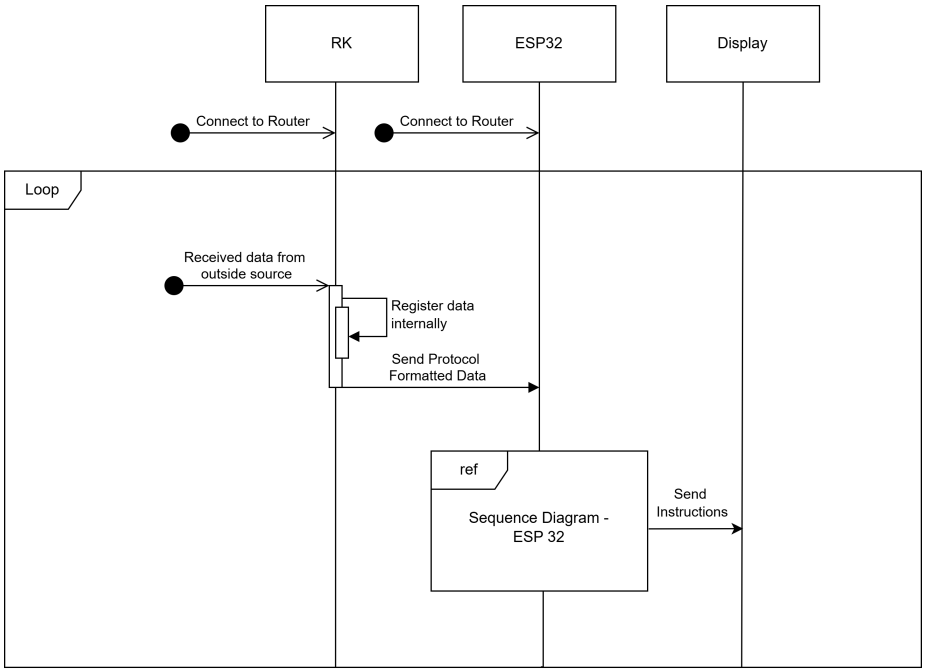


Figure 6: A sequence diagram showing how network connections are formed between the RK, ESP32 and the Display.

Initially, both the RK and ESP32 have to connect to a router. It is integral that they connect to the same router, otherwise they are unable to discover each other. Once both parties have established a network connection, they will try to connect with each other. After their connection is established, the main displaying loop starts. It is not stated in the diagram, but the component responsible for sending instructions runs on a separate thread. Whenever the RK receives data that should be displayed, such as a team registration, it transmits this data over the network using a communication protocol, which will be elaborated on after this section. Subsequently, this data is received by the ESP32 and stored in memory. The core responsible for frame generation generates frames at a

certain frequency. These frames are based on the information in thread global variables, which are shared by both cores. An illustration of the full data flow is shown in Figure 7.

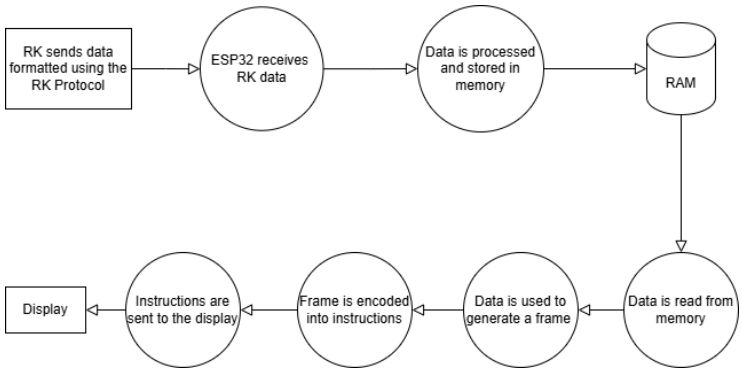


Figure 7: A data flow diagram showing how data sent by the RK flows in the ESP32.

4.4.1 Protocol

Communication between the RK and ESP32 relies on a communication protocol, which specifies the packets that can be sent over the network. This ensures data integrity, since transmitted data must be in the correct format for the RK to send it and for the ESP32 to receive it, acting as a data contract. A bit-protocol is used, since bit-protocols have a high efficiency and have little overhead. However, these are relatively excessive for the system's needs. Alternative data formats, like JSON, could be used as well without noticeably affecting performance. Nevertheless, the bit-protocol does give a performance benefit, which is significant in cases where a large number of packets are transferred, like during a restart of the system.

The below tables briefly describe when certain packets are sent and the corresponding packet size and bit-code. Packet size is always fixed, except for announcement packets, which have a variable length. These packets are explained in more detail under Appendix A.

Situation	**RK data sent**	**Packet size (Bytes)**	**Message Bit-Code**
Team finishes stage	Team Finish packet	10	0000
Tablet person revokes time	Team Finish Revoke Packet	2	0001
Team should prepare	Prepare packet	2	0010
RK Opens or Closes / Keep-Alive	RK Status packet	1	0011
Startup of the connection	Startup packet	2	0100
Announcement	Announcement Packet	2 + Announcement character Length	0101

Table 1: Table detailing packets sent by the RK.

Situation	**ESP data sent**	**Packet size (Bytes)**	**Message Bit-Code**
Heartbeat received	Heartbeat response Packet	1	1000

Table 2: Table detailing packets sent by the ESP32.

5 Functional Design

This section describes the functional design of the display.

5.1 Overview Functional Design

The new display design shown in Figure 11 is inspired by the matrix display design from Figure 12, as well as the old display design shown in Figure 8. The final design was inspired by both designs, lending how information is structured from Figure 11 and taking the colour formatting from Figure 12. The display is structured as follows:

Header

- WP number (Relay point number). If the WP is not opened, it displays WP-X.
- RK Connection, only shown when there is no active RK connection. Displayed by WP-?.
- Internet Connection, only shown when there is no active internet connection. Displayed by a crossed-out Wi-Fi symbol.
- Current Time
- Number of teams finished

Runner Data

- Team number, Stage Time and Ranking
- Teams that should prepare to start the stage

Footer

- Listing team numbers that have finished already

Figure 8: Old Display design

Several display designs were made, from which the client chose the design shown in Figure 9. The team suggested an alternative coloured version of the design, shown in Figure 11. The colours could be used to distinguish teams from one another. However, this would require extensive testing to guarantee this information is visible. As such, the client decided to use the red monochrome design.

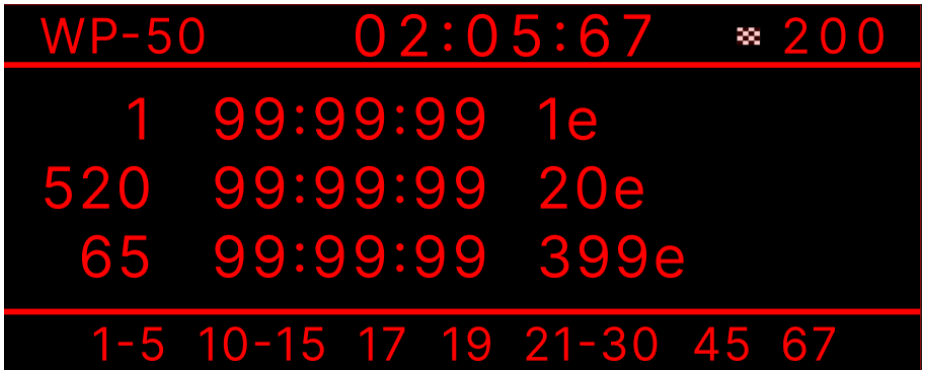


Figure 9: Red Display Design

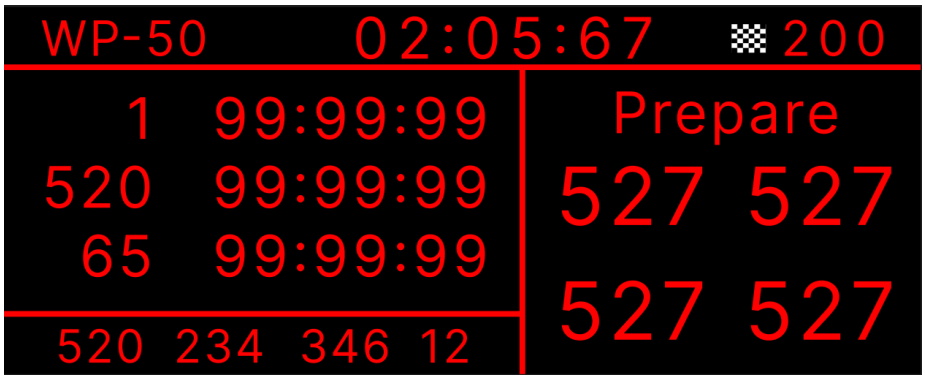


Figure 10: Red Display Design Prepare

Figure 11: Coloured-Display-Design

Figure 12: Display Inspiration

VBZ passenger information display clone.

Note. From Schüller (2023).

5.2 Header Design

The header displays the WP number, current time, and the number of teams finished by default. In the event that the RK or network disconnects, the WP number is replaced by a status indicator icon corresponding to the status type. Since the WP number can only be retrieved when both RK and internet connections are available, this approach makes sure that connection problems are communicated clearly without affecting the display of other information. Time can still be estimated during a network outage and remains accessible without an RK connection, while the number of finished teams should always be present for medical staff to estimate the current distribution of runners between stages.

If an RK disconnect occurs, the WP number is replaced with “WP-?” to indicate loss of RK connectivity. If there is an internet disconnect, a no-WiFi icon is displayed instead. This can be seen in Figure 13. If the WP is closed, the display shows “WP-X”.

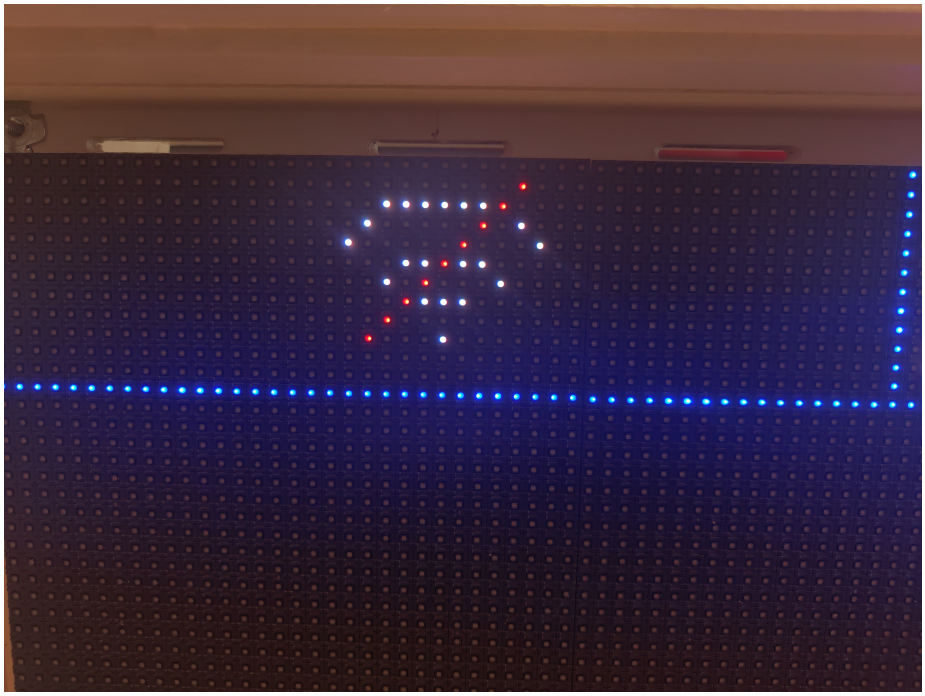


Figure 13: Display with no connection

5.3 Design Central Data Display

The central area of the screen displays data relevant to the stage results of recently finished teams. On the left-hand side, it shows the respective team number, time, and virtual ranking of the stage (see Figure 9). An entry appears whenever a team crosses the stage finish line and remains visible for at least 10 seconds. Up to three entries can be displayed simultaneously; after this period, the oldest entry is replaced by a newer one if additional registrations are available.

If a team should prepare for its stage, it will be displayed prominently on

the right-hand side until the team has started the stage, see Figure 10. The case owner highlighted that this information has priority over the ranking/time registration and finished team numbers. Hence, the decision was made to remove the virtual ranking temporarily whenever the prepare field holds data. This field itself is able to hold up to six team numbers. If more than six teams are supposed to prepare, the field will paginate through all teams to notify each team to prepare. An entry in the prepare field will be displayed on one of the pages for a maximum of 3 minutes, but is removed 20 seconds after the team has finished the stage.

Upon an Announcement, only the header is displayed, along with the announcement taking up the rest of the display. The header is not overwritten, as the case owner stated that the number of teams finished in a stage is vital information for medical services. Other information that is overwritten is not deemed crucial and therefore not displayed. For an example announcement, see Figure 14.

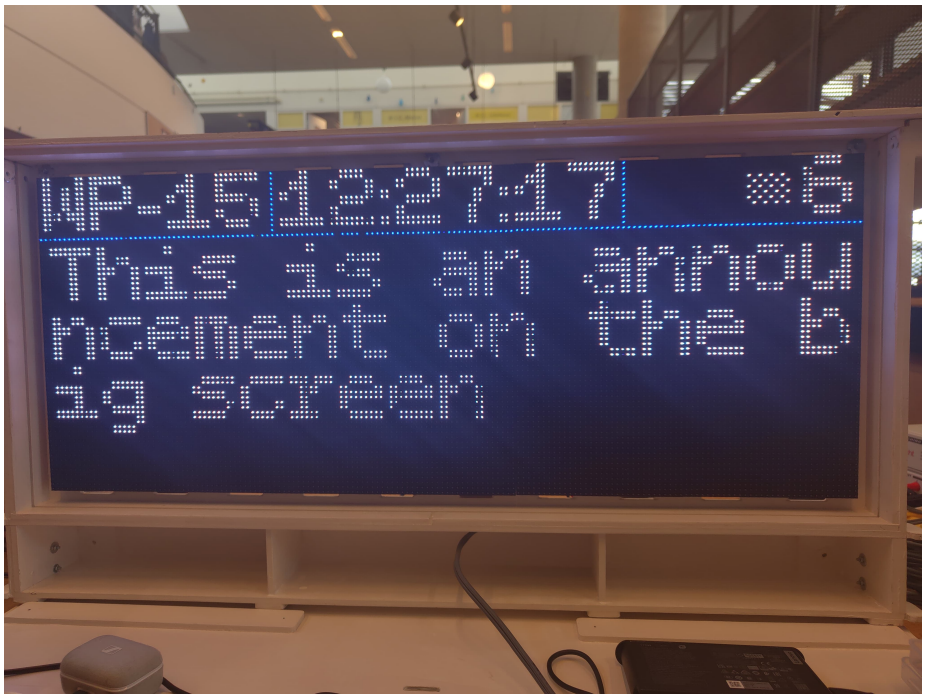


Figure 14: Display Announcement

5.4 Footer Design

The bottom of the screen scrolls through a list of teams that have finished the stage infinitely, see Figure 9. The team numbers are grouped together (e.g. if teams 1 through 5 have finished, it will display 1-5). By grouping consecutive team numbers, it remains viable to scroll through the list of finished teams in a relatively short time span.

During system start-up, the display will show a message that the system is booting alongside with the current software version. This can be seen in Figure 15. The term eRUNE stands for electronic Relay User Notification Engine, a representation of our product in line with the naming conventions of the BATA committee.

Figure 15: Startup Design

Lastly, these designs decisions concern the display requirements related to displaying information. See Chapter 3 for the requirements.

6 Test Plan

To ensure the correct functioning of the Display under any given circumstance, a test plan has been contrived. The test plan consists of multiple layers, from small unit tests to full system tests. This means that if the system passes the test plan, many failures can be avoided.

6.1 Unit tests

Testing is performed primarily on a unit-by-unit basis. Most of the classes can be seen as one unit. Testing these separately will ensure the inner working of the code, which can then be abstracted as functioning correctly for the broader tests, which span over multiple units. Furthermore, if future development to the code is done, the unit tests can be used per unit to ensure they still function according to their original design. The following units are tested:

- Network: tests to ensure the capability of setting up a socket connection to a server, receiving messages, and detecting a dead connection.
- Packet Handling: tests to ensure packets can be correctly decoded, but will also handle malformed packets correctly.
- Memory management: tests to ensure data can be correctly stored in memory, no dangling pointers are left over in memory, and all structures stay correctly organized.
- Frame generation: tests to ensure the frame generation calls the correct formatting functions based on the data in memory and will thus generate correct frames at the correct times.

6.2 Integration tests

After unit tests ensure the correct functioning of each unit, the connections between units are tested as well. The following cases are integration tested:

- Sending and Receiving Data: The connection between receiving data and parsing the data are done by testing the network connection over the socket, with the decoding of packets by the packet handler. The test sends binary data over a socket to the system, which then calls on the memory manager to correctly decode the packet and store the necessary data in memory. Afterwards, the memory values are read to test if the correct data has been stored, based on the packet.

- Memory Functions: Between the memory management and frame generation units, data is shared. This test ensures that all data is correctly written and read without any possible concurrency issues present.

6.3 System tests

The proposed unit and integration tests ensure the correct functioning of the majority of the system. However, some functionality must be tested empirically, like that information is correctly displayed. System tests test the entire system. These tests can be done using the RK simulation of the test suite. The display can be connected to the test suite, after which selected commands and situations can be used to simulate the functioning of the display. Using this, all functional requirements can be tested empirically.

6.4 User tests

The unit, integration, and system tests listed above test the functional requirements of the system. However, non-functional requirements cannot be tested with these tests. Therefore, these requirements are empirically tested using the following tests:

- Visibility: A major concern of the client is the visibility of the new display. Therefore, the visibility should be tested in different conditions. For these tests, all combinations of the following variables should be taken into account:

  - Time of day: As the Batavierenrace is a race from the evening through the night until the next morning, the visibility of the display should be tested in all conditions, including, among others, sunny, dark, rainy, and misty.
  - Weather condition: Different whether condition affects the visibility of the display. Visibility should therefore be tested with sunny, cloudy, foggy and rainy weather,
  - Relative speed: While the staff will be stationary next to the display, runners will run past it, and medical staff will drive by in a car. In all of these situations, the visibility of the display and its information should be tested.
  - Distance: The system should deliver a broad range from which the data can be easily read. For this, different distances from the screen should be tested.

- Stability: Each WP has a maximum deployment time of roughly 4 hours, according to the case owner. Because of this, the system must function correctly for at least 6 hours. The system should be stress tested during these 6 hours, to highlight any latent memory leaks or other longevity issues. The test should add, remove, and revoke registrations/preparations. The maximum number of registrations and preparations should be above the range of the expected amount during the race. In addition to this, announcements should be displayed and revoked often.

6.5 Test Results

This section describes the results of the aforementioned tests.

6.5.1 Unit Tests

The Unit tests test each individual function of each respective test. Some exclusions are functions from FrameGenerator whose main job is to output a raw

pixel array. The complexity of verifying these results is not worth the returns, especially as such results could also be verified through system tests. The unit test results are shown in Table 3. All tests pass.

Test	Status
Network	Passed
Packet	Passed
Memory Management	Passed
Frame Generation[1]	Passed

Table 3: The performed unit tests with their results

6.5.2 Integration Tests

Integration tests focus on whether different components of the system integrate with one another. The integration test results are shown in Table 4.

Test	Status
Memory Functions	Passed
Sending & Receiving Data	Passed

Table 4: The performed integration tests with their result

6.5.3 System Tests

To perform system tests, the test suite is utilised in order to verify the correct functionality of the system. The team performed several actions in the test suite, such as registering teams, adding teams to the preparation list, and verifying whether what works correctly according to the stipulated requirements. All implemented requirements are tested using these system tests and are confirmed to work as intended.

[1] Frame generation test is partly an integration test; most functions require the use of memory management. Alternatively, the FrameGenerator class could be altered to take alternative parameters mimicking what would be stored in memory.

6.5.4 User Tests

For the specified user test categories, the team performed different types of testing:

- Visibility: To confirm the visibility of text on the display, the team tested the display at different times of day, with different brightness levels and colours. This led to the conclusion that, during the night, the display was most readable at a brightness level between 20% and 30%, with the colour scheme of white text and blue lines being polled as most preferable between the team and the case owner. Unfortunately, due to surprisingly good weather, it was impossible to test the device in different weather conditions.

- Stability: In order to verify the stability of the system, the team left the display running on multiple occasions. This led to the discovery of multiple memory leaks, which have since been fixed. The team currently believes that there are no memory leaks present. The system has run for over six hours without crashing.

  The system has been stress tested to roughly estimate its capacities. The maximum and minimum number of registrations and preparations that can be made at the once seems to be roughly 800 registrations. This maximum can be increased further, by allocating more RAM to the socket buffer as this buffer is currently the bottleneck for the maximum registrations at once. Regardless, this is far above the estimated number of registrations which will be made during the actual race, given that there are usually no more than 350 teams participating (Batavierenrace — Home, n.d.).

7 Final Product

The displaying system is implemented, as shown in Figure 11. The majority of the stipulated requirements, seen in Section 3, are complete. This section details the product's source code, repositories and provides a manual for system usage.

7.1 Source Code

At a higher level, the repository is split among three separate directories: Display, RK and Testing.

7.1.1 Display Directory

The display directory contains all ESP32/C++ code. This includes the communication with the RK, unpacking RK packets, storing these in memory, the formatting of information, generating the virtual display frame and finally sending the instructions to the display. The file structure can be seen in Figure 19 under Appendix B.

Below are short descriptions of the relevant subdirectories or files.

- **lib**: Contains source code for implemented features. Each feature has its own include folder, containing header files, and its own src file, containing source files.
- **Formatter**: Contains formatting functions for formatting information correctly on a frame.
- **Frame**: Contains a type definition for a frame, which is a virtual representation of the display.
- **FrameGenerator**: Contains code for the generation of a frame, including the main frame generation loop.
- **InstructionSender**: Contains code for sending a frame to the display as instructions.

- **MemoryManagement**: Contains code relating to the storing of data in memory.
- **Network**: Contains code for setting up and maintaining a network connection to an RK.
- **PacketHandler**: Contains code for the unpacking of packets and formatting them as c++ structs.
- **Settings**: Contains a header file with environment variables or macros.
- **src**: Only contains main.cpp, which is the file executed on the ESP32.
- **README.md**: A README describing the directory in more detail.
- **platformio.ini**: Environment file for PlatformIO.

7.1.2 RK Directory

This directory contains all RK code and is written in Java. Since the team only developed and altered the code required for interacting with the ESP32, only those subdirectories are clarified here. The file structure is shown in Figure 20 under Appendix B

Below are short descriptions of the relevant subdirectories or files.

- **display_communicator**: This directory contains all code for communicating with the display.
- **DisplayCommunicator**: This file is responsible for interacting with the ESP32.
- **PacketBuilder**: This file builds and formats packets to send to the ESP32, as specified by the RK protocol document.
- **SocketHandler**: This file is a wrapper class of a Java socket.
- **RK**: This class is the monolith software application of the RK. The team changed it to interact with the display and only claim responsibility for those components of this class.

7.1.3 Testing Directory

The testing directory contains the test suite created to easily simulate scenarios. The test suite is written in Java, requiring at least Java 24, and the JavaFX library for creating and managing the GUI. The test suite is set up with a Maven project, configured in the `pom.xml` file in the root directory.

Below are short descriptions of the files used in the testing suite as seen in Figure 21 under Appendix B.

- **Main.java**: This is the entry point of the test suite. It sets up the JavaFX module and starts the GUI.
- **MainController.java**: This Java file handles all function calls of the GUI. Every user input has functions here that manage actions that need to happen accordingly.
- **PacketBuilder.java**: This class is the exact same as the one in the RK described above, to ensure that packets are built exactly the same.
- **SerialHandler.java**: This class handles the serial connection to the display to set up the connection and send messages.
- **main.fxml**: This fxml file contains the GUI format such that the JavaFX library can correctly display the GUI.
- **main.css**: This css file adds structure and formatting of the elements in the `main.fxml` file.

7.2 Manual

This section is a manual of the system, which includes instructions for deployment and instructions on test suite usage.

7.2.1 Deployment

The deployment of the system requires a one-time setup to be performed, after which the system should operate successfully out of the box when connected

to a power supply. The one-time setup of the system requires the platformio software to be installed.

As the system needs to interact with both a router and an RK it needs to know how to reach these. For this, specific values must be hardcoded in the compiled code of the ESP32 running the system in the `network_config.ini` file of the Display project source code. In this configuration file, the following values must be supplied:

- **SERVER_HOST_NAME:** The host name of the RK appended with ".local". For example, if the host name is the default "raspberrypi", the server host name becomes "raspberrypi.local".
- **WIFI_SSID:** The SSID of the router to which the system is supposed to connect.[2]
- **WIFI_PASS:** The password of the router to which the system is supposed to connect.[2]

Once all values have been configured, the source code can be compiled and uploaded to the ESP32, using PlatformIO, by running the following command in the root directory of the Display project source code (PlatformIO, n.d.):

```
platformio run --target upload
```

7.2.2 Test Suite

The usage of the test suite requires an installed Java runtime of version 25 or higher and small changes in the display setup. For the test suite, a separate environment for the display has been defined, which exposes a serial connection to receive data, instead of the socket. To build the environment for the test suite, enter WIFI_SSID and WIFI_PASS in the same structure as the above normal deployment. Then build and upload the code in the Display directory (PlatformIO, n.d.):

[2]If the SSID and password are set to the same value in all the routers used in the race, the system will be able to connect to any of these routers.

```
platformio run --target upload -e esp32simulation
```

After successfully configuring the display, the test suite can be easily used by connecting the display to the computer using a cable and launching the test suite using:

```
java -jar test_suite.jar
```

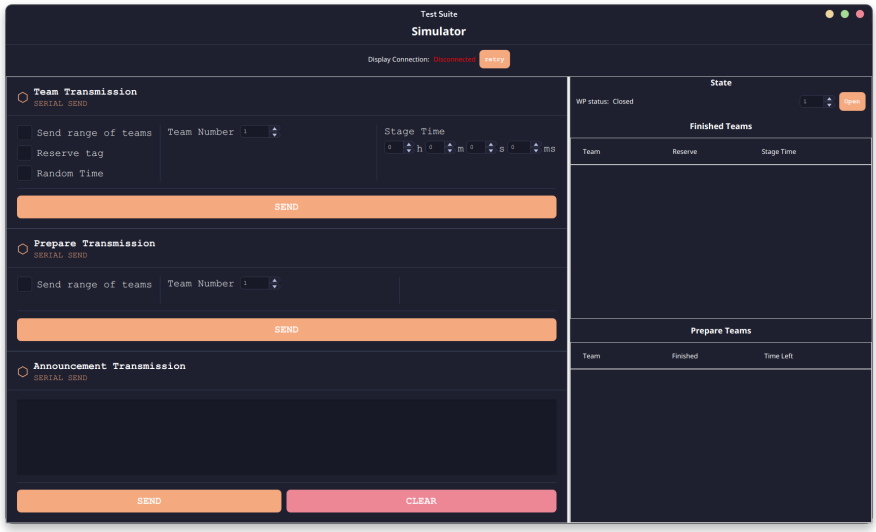


Figure 16: Full overview of the test suite after startup

After launching the test suite, a screen as seen in Figure 16 pops up. At the top of the screen, the connection status will be shown. If it does not connect automatically, the retry button can be used to retry connecting to the display. When a connection has been established, the interface can be used to send commands to the display. Using the Team Transmission tab, team registrations can be sent, either one at a time or a whole range, with a set interval. The teams can be sent as a reserve tag and with a random time if needed. The time for the team can also be manually entered in the stage time tab. After configuring everything, the registrations can be sent to the display with the orange send button. The Prepare Transmission tab works in a similar manner to the Team

Transmission tab, where either a single team or a range of teams can be sent with a set interval. Finally, the Announcement Transmission tab has a simple text input for putting announcements on the screen, which can afterwards be cleared from the screen using the clear button.

The right part of the screen shows the current state of the display. If the WP is in a closed state, it can be opened with a WP location number and the open button, after which it can be closed again. The state also shows the finished teams and the teams that are to be prepared in their corresponding tables. These registrations or preparations can be revoked using the trash bin button attached to every entry.

8 Process

This chapter aims to reflect on the process of our project and to compare it to the initial planning made.

8.1 Reflection on Design Phases

Each design lifecycle was planned to last a certain number of weeks. The planning and analysis phases went according to plan, but unfortunately, the design and implementation phases took longer than anticipated. The aim was to finish the designs by week 4, which ended up being done halfway through week 5. This influenced the implementation trajectory, since the implementation phase was planned to take 4 weeks. Consequently, the implementation phase ended in week 9, leaving little wiggle room for errors. As a result, we had to work more during this week to make up for this fact. We aim to improve this in future projects by accounting for delays in our planning, such that situations like these are prevented.

8.2 Reflection on Requirement Analysis

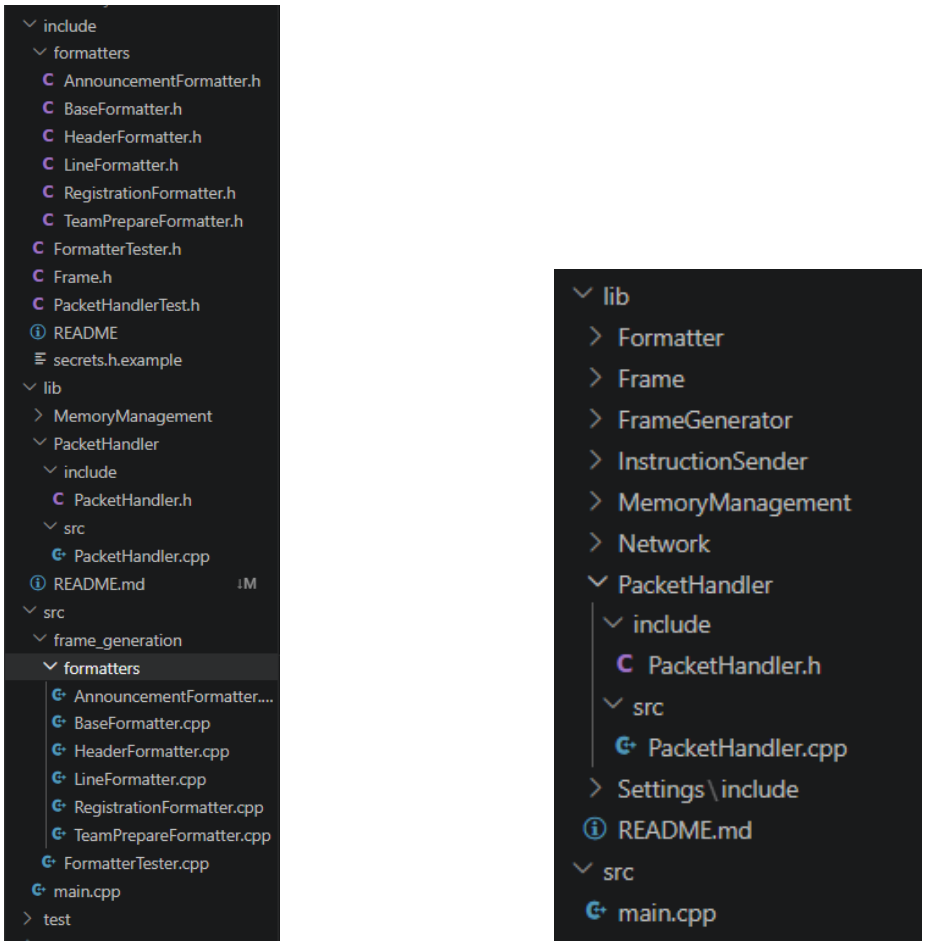
Throughout the implementation trajectory, minor changes in interpretations were made to some system requirements. Take, for example, the requirement "The display **must** show the team number of a team that needs to be ready at the WP for a limited time, depending on whether the runner has been registered". Initially, this meant that the display must show a team which should prepare for a maximum time, until it is registered. It turned out that that is not what the client had in mind. His vision was that a team that has been prepared should be displayed for a maximum of 180 seconds, unless the team is registered before that time, in which case the team can remain displayed for a maximum of 20 seconds.

Changing requirements introduces problems, namely delays in developments and the need for communication with the client. Whenever a requirement is

changed, the team has to alter previous designs to correspond to the new requirement. These changes can influence the rest of the project by proxy, since certain components depend on others. For example, changing how data is stored in memory affects how it is read from memory. Additionally, the team has to propose this change to the client to ensure the new requirement still aligns with their vision of the system. Although the communication with the client was fast, it cost time to make such changes. Consequently, it is imperative that in future work the team improves the process of requirement analysis by reviewing designed requirements more thoroughly, such that less changes are required during implementation and valuable time is saved.

8.3 Reflection on Code Planning

Our initial structure of the codebase, based on the analysis phase, proved fruitful. There were little structural issues regarding the respective RK and Display codebases, at least pertaining to feature implementations. However, the fact we did not account for any C++ naming convention or structure caused issues during pull requests. Since we were unfamiliar with C++ and there was no clear structure, we had to do a large refactor halfway through the implementation phase to fix inconsistencies in code structure.



(a) Before refactoring

(b) After refactoring

Figure 17: Comparison of code structure before and after refactoring.

Ultimately, we agreed that features should be placed in the 'lib' directory and that each feature should have an include folder with a header, and a src folder for the C++ code, as seen in Figure 17b.

This ambiguity regarding code structure caused confusion during development, since it was unclear at times where certain code lived in the codebase. In the future, the team will communicate a structure in advance, in order to avoid issues like these.

8.3.1 Git Usage

Overall, our use of Git throughout the project was effective and well-structured. We consistently relied on issues to define tasks, used the issue board to track progress, and organized our workflow around feature branches. Pull requests were an integral part of our process, and we made a strong effort to review changes thoroughly before merging them into the main branch. This helped maintain code quality and ensured that all team members stayed aligned.

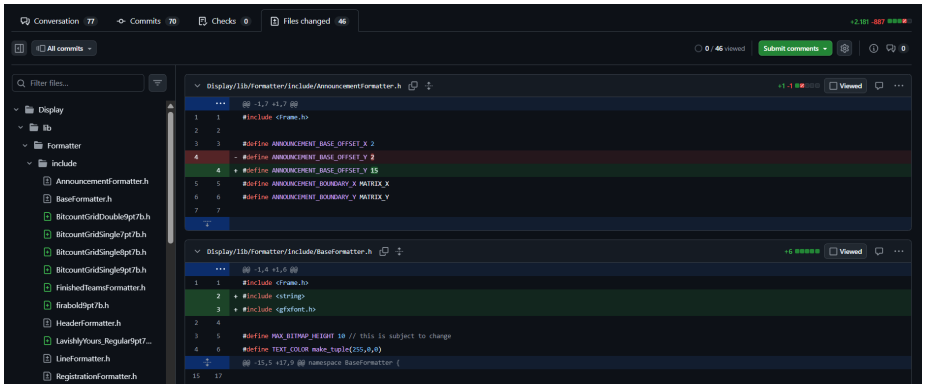


Figure 18: The large pull request, resulting from improper git usage.

There was one notable exception to this otherwise smooth workflow. Due to the interconnected nature of several features, we occasionally depended on each other's in-progress work. Instead of creating temporary stub implementations, we chose to pull changes directly from one another's branches. This ultimately resulted in a single large pull request, as shown in Figure 18, containing 2188 added and 887 removed lines. Reviewing this pull request took considerable time due to its size.

After merging this pull request, we adjusted our workflow to avoid similar situations by only pulling from the main branch. This change helped us return to a more streamlined and manageable development process.

Importantly, this situation was an outlier rather than a pattern. For the majority of the project, Git was used in a disciplined and collaborative manner.

In retrospect, we recognize that the issue could have been further mitigated by breaking down larger tasks into smaller, more manageable sub-issues and submitting pull requests more frequently. This is an improvement we will carry forward into future projects.

Overall, our Git practices—particularly our use of issues, structured branching, and thorough code reviews—contributed positively to the project's organization and code quality.

8.4 Reflection on Client Communication

The team generally had a pleasant experience with our client. The client responded quickly, which enabled us to solve certain issues without having to wait until our next meeting. The meetings themselves were helpful too. It was planned to meet once every week, which was not deviated from. The feedback provided in these meetings ensured that the product aligned with the case owner's vision.

However, there were some issues with the client, in particular with how the client provided projected related information. On several occasions, the client withheld information which would have proved useful if shared earlier. For example, the source code of the RK was only provided well after the design phase. The source code, after being analysed, showed that the view of the system made in the orientation phase and used in the design phase was incorrect and would not work with the designs made. Furthermore, a logbook specification, which the group did not know existed, was sent at the start of the implementation phase. This would have been useful during the analysis and design phases, respectively, since the designs changed resulting from this document. Besides this, the provided RK code did not adhere to its own logbook specifications. Similar situations happened throughout the project, which disrupted the development process.

Furthermore, there were miscommunications in the expectations of certain requirements and the scope of the project, which ultimately resulted in certain

requirements remaining incomplete. Initially, the team was of the understanding that only minor changes to the RK code were necessary, as understood from the meetings and requirements which the client approved of. Later, the client contradicted himself, claiming that refactoring the RK was an integral part of the project from the start. A mutual agreement was made not to include this in the current project, based on the limited time left within the project.

Although miscommunications are somewhat inevitable, the team hopes that by communicating clearly with future clients, these situations can be prevented. Additionally, matters such as project scopes will be established and written more clearly and directly.

8.5 Reflection on Supervisor Communication

Neither the team nor the supervisor deviated from the planned communication. The team would like to thank the supervisor, Gerwin Hoogsteen, for his supervision. He responded fast, gave useful feedback and generally supported us during the project. For example, during the aforementioned conflict of interest with the client, Gerwin helped us navigate the situation. This was greatly appreciated, since this helped the team navigate the situation professionally. Consequently, the team knows better how to handle such cases, whenever they may occur in professional careers.

8.6 Reflection on Team Communication

Overall, the communication within the team went smoothly. The planned daily meetings ensured team members kept each other accountable and that everyone was up to date regarding each other's progress. The team stopped doing small briefings at the start of each meeting, since it was found somewhat redundant given that the team was already up to date on everything. Although the communication went well, there is nevertheless room for improvement. Team members were not always clear about their scheduling, leading to them suddenly leaving during a project meeting. In addition to this, team members were

regularly late, which decreased morale to work in the morning. In the future, the team plans to mitigate this by communicating scheduling concerns better and by vocalising the need to start later if necessary.

8.7 Contributions

It is quite difficult to show how the exact workload was distributed, since the team members often helped each other out or provided quick fixes in parts of the codebase they initially were not assigned to. Nevertheless, the tables below provide some rough estimates on how the workload was balanced during the duration of the project. It is worth mentioning that showing workload division in a table does not accurately cover actual workload distribution, as some parts of the system took significantly more work than others, e.g. the implementation of the instruction sender took noticeably less time than creating the formatting functions. The largest amount of development time was allocated to memory management, frame generation and the formatting functions. These parts were either simply very large or were susceptible to bugs and thus took a lot of debugging effort. Please refer to Table 5 for an estimation of the development workload division. The rows in the table show for each part of the system how much of that part was written by whom. Other tables in this section should be read in a similar way. The percentages next to the part names aim to estimate how much of total development time went to each part.

Table 5: Distribution of coding workload across the team. Each part sums to 100%.

Part of system	Lucas	Marijn	Rune	Tim	Tristan
Network socket (10%)	30%	-	-	-	70%
Packet handling (15%)	-	50%	-	50%	-
Memory management (20%)	-	80%	-	-	20%
Frame generator (15%)	80%	-	20%	-	-
Formatting functions (25%)	15%	-	80%	-	5%
Instruction sending (5%)	-	-	-	-	100%
Testing suite (10%)	-	-	-	20%	80%

The workload distribution for creating the designs and diagrams of the sys-

tem was relatively equal, as most of these designs were collaborative efforts. In general, designs were first created to establish a kind of baseline, after which the respective implementations evolved naturally. This evolution is most obvious in the protocol and functional designs. Both of these designs underwent numerous changes throughout the course of the project, mainly due to insights gained during development and client feedback respectively. Please refer to table 6.

Table 6: Distribution of design workload across the team

Part of system	Lucas	Marijn	Rune	Tim	Tristan
Protocol design	40%	-	-	-	60%
System class diagram	20%	20%	20%	20%	20%
ESP32 sequence diagram	-	-	50%	50%	-
System sequence diagram	-	-	50%	50%	-
ESP32 data flow diagram	100%	-	-	-	-
Unit and integration tests	-	25%	25%	25%	25%
Functional design	-	100%	-	-	-

Lastly, there is the miscellaneous work, which could also be described as workload regarding meetings and client contact. With these matters, there were not any strict workload assignments apart from who would lead meetings and who would be the main point of contact. To see how the workload was divided among these matters, refer to table 7.

Table 7: Miscellaneous work

Part of system	Lucas	Marijn	Rune	Tim	Tristan
Leading meetings	95%	-	-	5%	-
General client communication	10%	-	-	60%	30%
Note taking	-	20%	50%	20%	10%

9 Conclusion

In conclusion, a large majority of the requirements have been finalised, and a robust, complete system has been delivered to the case owner. As seen in Section 3, we have accomplished a highly complete product, with only minor changes and exemptions made to the original requirements.

The system meets the primary objectives set at the beginning of the project and demonstrates robust functionality across its core features. Additionally, the system has been thoroughly tested, and an extensive testing suite has been provided in order to verify the functionality of the system at any point in time.

Throughout the development process, careful consideration was given to design choices, implementation quality, and usability, resulting in a system that is both effective and practical.

Although a small number of requirements were adjusted or omitted, these decisions were made deliberately to improve overall system performance, feasibility, and maintainability. Future work could focus on addressing these remaining aspects, which are also stipulated in Section 10.

Overall, the project can be considered a success, delivering a well-functioning and valuable product that satisfies the needs of the case owner.

10 Future Recommendations

Based on what has been implemented during our process, there are still improvements that could be made to the system before being used in the actual Batavieren race to improve functionality and usability.

10.1 Unimplemented Requirements

As mentioned in Section 3.3, the new system does not account for every specified requirement. Since these features were desired, it is recommended that these unimplemented requirements still get implemented as far as possible.

10.2 Restart Robustness

In the current RK system, finish times are stored in a log book using persistent storage. However, the start times registered at the previous WP, which are supplied by an outside server, are only stored in memory and will not be present during a restart of the system. Because of this, when the RK fails and restarts, there is no start time present for the calculation of a stage time. Consequently, it is impossible to make registrations during this time. To resolve this issue, the following solution proposals are made.

Band-aid Solution

A simple fix to ensure proper functioning of the system can be made by adding a cache to the display controller of the RK. The display controller could interact with a local file that would store all start times that have been communicated in a table. This could be in a simple format, such as CSV. The display controller could use this cache file to properly send the stage times to the display on a restart.

Refactor solutions

Although the aforementioned solution ensures proper functioning of the sys-

tem, it does not improve the general code structure and quality of the RK system. Initially, the RK's scope started small, but it has since had numerous additions, which give extra responsibility to the system, according to the case owner. Because of this, the RK has turned into a large, intertwined monolith. If an RK refactor were to take place, the storage of received start times should be taken into account. When moving away from this monolith structure, a single persistently stored file would be used for the log book, start times, and other data that might be necessary late in the process. All of this would then have a single point of access for reading and writing, keeping it separated and abstracted. This proposal would ensure proper functioning and good code quality, but it is infeasible to implement before the Batavierenrace in May 2026. For further development of future competitions, it would be recommended to improve upon the project.

10.3 Improved Registration Identification

The old system and the created system both contain a major flaw in the logic for removing teams from the internal display data which was discovered during development. When a team registration is removed, the registration is only identified with the team number. In the case a team has multiple registrations, for example, an automatic and manual registration, if one registration were to be removed, just the team number will not suffice for identifying which registration is to be removed. To handle this flaw, an extra property needs to be added to uniquely identify registrations. If one were to assume that multiple registrations will never have the same time, since they would otherwise come down to the same registration, adding the time to the revoke packet would suffice for uniquely identifying registrations. If this is not the case, a unique identifier could be added to each registration with which the display can identify the registration to be removed. Furthermore, this unique identifier would have to be stored in persistent storage to ensure that after a crash or reboot of the RK registrations, they can still be correctly removed.

10.4 Coloured Information

As discussed in functional design of the display, adding colours to the display would require extensive testing, but could be helpful for a visually appealing design. Furthermore, colours could be used for displaying subtle data without taking up extra screen space. An example of this would be registration types; changing the colour of the registration based on whether it is an automatic or manual registration would be useful for staff to immediately notice this, while it would not influence participants. Implementing this requires the display to know the type of registration. The protocol between the RK and the display would need to be changed to include this information.

10.5 Dynamic brightness and colours

The user testing done to determine which colour scheme and brightness level would be best suited for the system did not result in one concrete answer. Different situations call for different visual styles, and so it would be worth considering ways to be able to change colour schemes and brightness levels dynamically based on the time of day and weather conditions. For example, the active colour scheme could be changed based on the current time of day to make the displayed information more readable during the night, or the brightness level could be dynamically determined through external light sensors (which are currently not present and would have to be added separately). Another approach would be to make colour and brightness values depend on buttons and dials, which could be manually pressed or tweaked by the staff operating a relay point.

10.6 Improvements to announcements

The new system handles announcements up to a certain length. This means that if an announcement contains too many characters, the announcement gets cut off, and crucial information may not be displayed properly. A possible fix for this would be to implement a form of pagination for announcements, i.e.

splitting the total announcement string across multiple pages that are displayed one by one. Another approach would be to make overflowing announcements scroll from the bottom up, similarly to how finished teams scroll from right to left at the bottom of the screen, drawing characters row-by-row instead of column-by-column.

Additionally, the announcement formatter currently adds new lines when no more characters can be drawn horizontally. Although this helps improve the readability of announcements, this approach is far from perfect, as words are often split across new lines. An option could be to check during formatting whether the following word can still fit in the remaining horizontal space, and force a new line if this is not the case. This solution seems simple and would work for most situations; however, it may fall flat whenever a word takes up more than the entire width of the display. A syllable-based cut-off would prevent this issue from occurring, but considering the rarity of the previously described edge case, this approach may not be worth the effort.

10.7 Time retrieval

As mentioned in the altered requirements 3.2, the system currently requires an internet connection to an NTP server to retrieve the time. The RK uses a similar system, relying on an active internet connection to retrieve the time, and is thus unable to register times correctly without an internet connection. Furthermore, some relay points are located in remote areas, making the internet connection unreliable. To improve on this, a local NTP server can be used at each WP. The local NTP server would then be used by both the RK and the display to ensure they have the same synchronised time and do not rely on an internet connection any more.

11 AI Usage

During the preparation of this work, the authors used LLM Tools such as ChatGPT and GitHub Copilot to debug errors, such as memory leaks. The usage of LLM generated code was limited to one function in the BaseFormatter class that extracts a character bitmap from AdafruitGFX and the frontend of the test suite. The tests themselves were written by the authors. After using these tools and services, any generated content was thoroughly reviewed by the authors as needed, with the authors taking full responsibility for the outcome.

12 Bibliography

References

[1] Batavierenrace. (n.d.). *Batavierenrace — Home*. Retrieved April 17, 2026, from `https://www.batavierenrace.nl/`

[2] PlatformIO. (n.d.). *Your gateway to embedded software development excellence*. Retrieved April 17, 2026, from `https://docs.platformio.org/en/latest/`

[3] Schüller, S. (2023, November 12). *Building a VBZ display clone*. GitHub Pages. `https://sschueller.github.io/posts/vbz-fahrgastinformation/`

Appendix

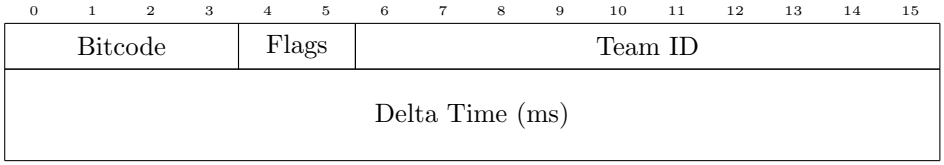
A Packets

All used packets in the protocol with their structure and fields. The grayed bits are unused.

Team Finish packet

Bit-code: 0000

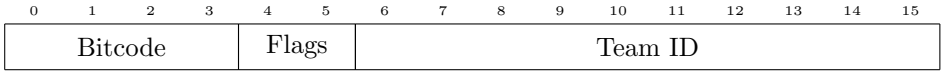
Description: Whenever a team finishes a relay, i.e. arrives at a relay point, the RK sends a packet detailing which team finished and the start/finish times.



Revoke packet

Bit-code: 0001

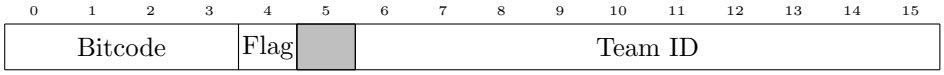
Description: Whenever the staff through the tablet revokes a registered time the RK sends a packet invalidating the time of this team.



Prepare packet

Bit-code: 0010

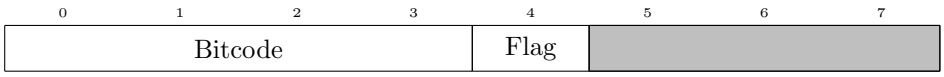
Description: Whenever a team has to get ready to start running, the RK sends a packet with that team's ID.



RK status packet

Bit-code: 0011

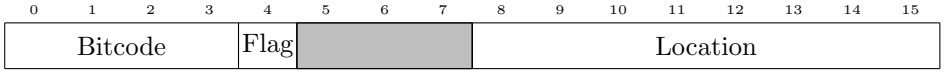
Description: Whenever the RK is closed or being opened, this is communicated with a packet.



Startup packet

Bit-code: 0100

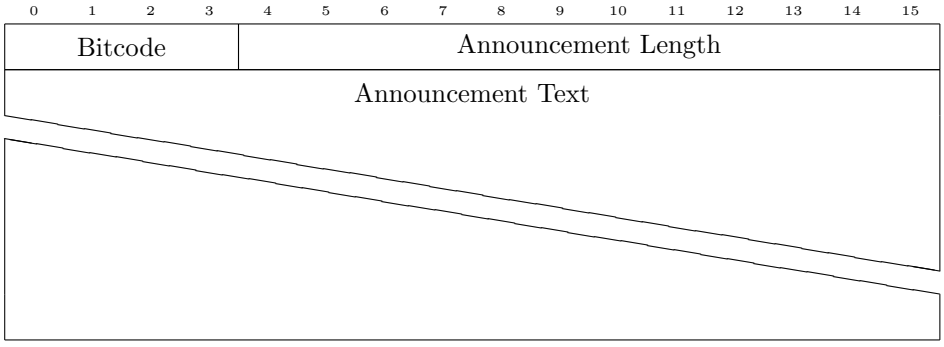
Description: Whenever the RK starts up, it has to send requests to the ESP32 informing them that it exists, where it is located, and if it is already opened.



Announcement packet

Bit-code: 0101

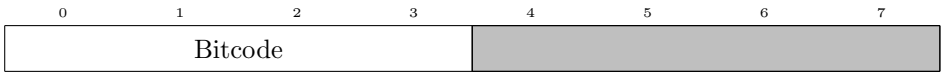
Description: Whenever an announcement is made, the RK sends a packet with the announcement. Each character is one byte, assuming ASCII format. If the Announcement Text is empty, it means that an announcement is revoked. An announcement can have a maximum length of $2\hat{1}0 = 1024$ characters.



Heartbeat response packet

Bit-code: 1000

Description: As a response to a heartbeat message a single byte will be sent.



B Directories

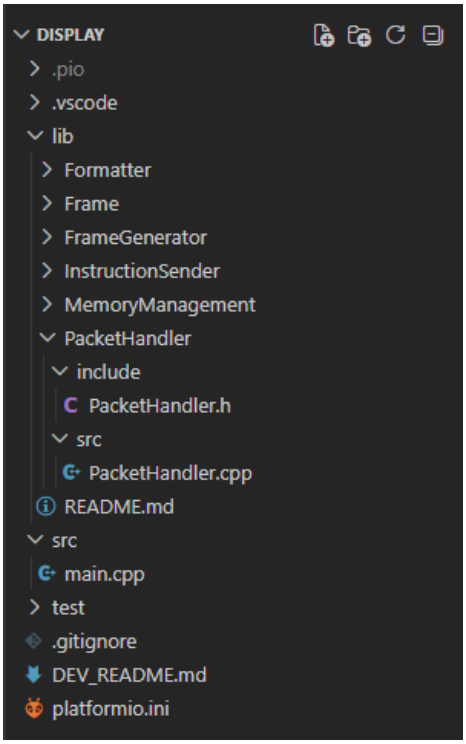


Figure 19: File structure of the Display directory.

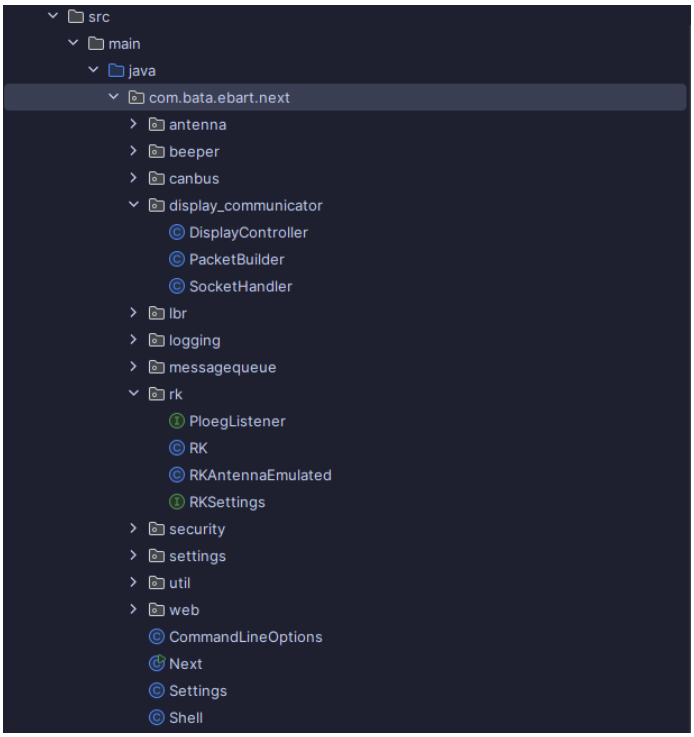


Figure 20: File structure of the RK directory.

Figure 21: File structure of the Testing directory.