VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV STROJÍRENSKÉ TECHNOLOGIE

INSTITUTE OF MANUFACTURING TECHNOLOGY

VYTVOŘENÍ HOMOGENNÍHO HEJNA ROBOTŮ

HOMOGENEOUS ROBOT SWARM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — **JAN BĚLOHLÁVEK**

AUTHOR

VEDOUCÍ PRÁCE — **mjr. Ing. VÁCLAV KŘIVÁNEK, Ph.D.**

SUPERVISOR

BRNO 2024

Abstrakt
Cílem této práce je umožnit využití robota DJI Robomaster S1 jako drona v robotickém hejnu. To je v práci docíleno analýzou a zdokumentováním vnitřního komunikačního protokolu robota a následnou implementací protokolu v systému, který robota ovládá. Robomaster S1 je robot určený pro vzdělávací účely, který plánuje Univerzita obrany v Brně využít pro experimentování s formacemi robotických hejn. Design robota však neumožňuje jeho ovládání pomocí externího kódu a jeho zdrojový kód není zveřejněn. Hlavní počítač robota ovládá ostatní části robota prostřednictvím sběrnice CAN, tato komunikace byla analyzována a jednotlivé příkazy v ní používané byly zdokumentovány. Hlavní počítač robota byl nahrazen deskou ESP32, která je nyní schopna robota ovládat. Dále bylo implementováno bezdrátové ovládání, které umožňuje uživateli ovládat robota například prostřednictvím klávesnice a které může sloužit jako základ pro budoucí komunikaci mezi roboty. Díky provedeným úpravám se také výrazně prodloužila výdrž baterie. Robot je nyní připraven k použití v hejnu robotů.

Abstract
The subject of this thesis is enabling the DJI Robomaster S1 robot to be used as an autonomous drone in a robotic swarm. This is achieved by an analysis and documentation of the robot's internal communications protocol and the implementation of this protocol into the system. The Robomaster S1 is an educational robot, that the University of Defense in Brno wants to use for experimenting with swarm formations. However, the design of the robot does not allow for it to be controlled by external code and its source code is not public. The robot's main computer controls the rest of the robot via CAN bus, this communication was reverse engineered and the individual commands were documented. The robot's main computer was replaced with an ESP32, which now sends the commands itself and controls the robot. A wireless control method, which utilizes the ESP-NOW protocol, was also implemented. This enables the user to control the robot manually, but also serves as a foundation to enabling several robots to communicate with one another. With the applied modifications the battery life also significantly improved. The robot can now be used in a robotic swarm.

Klíčová slova
Robot, DJI Robomaster S1, CAN bus, reverzní inženýrství, hejno robotů, ESP32

Keywords
Robot, DJI Robomaster S1, CAN bus, reverse engineering, robotic swarm, ESP32

BĚLOHLÁVEK, J. *Vytvoření homogenního hejna robotů.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2024. 26 s. Vedoucí diplomové práce mjr. Ing. Václav Křivánek, Ph.D.

Contents

List of Figures

List of Tables

1. Introduction

In recent years, the field of robotics has witnessed significant advancements, driven to a large extent by the military sector. There is a growing demand for autonomous robotic systems to increase effectiveness and reduce risk to human personnel in combat environments. Swarm robotics offers a promising solution to address these needs.

The University of Defence in Brno is exploring the potential of robotic swarms. To conduct their research, they require suitable hardware for experimentation. Among the options considered, the Robomaster S1 would be considered viable option due to its relatively low cost and advanced movement capabilities. However, the options of controlling it are very limiting as it lacks an interface which could be used to interact with the robot using custom code.

The Robomaster EP, the younger brother of the Robomaster S1, is more open, but is also much more expensive. To balance cost and function, it was decided to modify the Robomaster S1 and reverse engineer its communication protocols so that it can be controlled with custom software and used as a drone in a swarm formation.

1.1. Relevant methodological approaches

1.1.1. Reverse engineering

Reverse engineering [5] is the process of analysing existing engineering designs in order to gain information about the inner workings, so that they may be then pondered or replicated. This meticulous process can be time-consuming and may not always yield successful results, but it plays a crucial role in various fields.

Generally all reverse engineering processes involve three basic steps:

- Information Extraction: This step involves gathering all necessary details about the target system. This can include disassembling hardware, analyzing software code, and documenting all components and their interactions.
- Modeling: Using the extracted information, the next step is to create a model that replicates the target system. This model can be a physical replica, a software simulation, or a detailed schematic.
- Review: Finally, the model is tested and reviewed to ensure it accurately replicates the original system.

Military applications

Reverse engineering is an important area of study in the military sector. The costs of reverse engineering a captured piece of technology are far lesser than the costs of research and design of a new technology. One of the most significant areas where reverse engineering is applied is in the military sector. The cost and time required to reverse engineering a captured piece of technology are far lesser than that which is required to develop similar technologies from scratch [12]. This can lead to significant strategic advantages.

Reverse engineering has played a pivotal role in military history. During World War II, several notable instances of reverse engineering significantly impacted the war's outcome:

- The German anti-tank weapon Panzerschreck was developed after the Germans captured an American bazooka.
- The Enigma machine was a device used by the Axis to encrypt their communications. Its encryption was based on a key, which was generated each day. By inputting words they knew would appear daily, such as "weather", into a captured enigma machine, a British team led by Alan Turing created a device called the Bombe, which was capable of computing the encryption key very quickly, enabling the Allies to translate much of the German's communications.

Some examples from more modern times are:

- The Wine project creates a compatibility layer that allows Windows applications to run on Unix-like operating systems such as Linux and macOS
- In the early 1980s, Compaq reverse-engineered the IBM PC BIOS to create their own IBM-compatible personal computers. Even though IBM released the source code of the BIOS, copying it would violate copyright, so the researchers had to use a method called "clean-room design" [8].

1.1.2. Swarm robotics

Swarm robotics [6] is a method of robot coordination, creating a network of simple autonomous units. The network operates in a decentralized manner, where instructions usually originate from the robots themselves as a result of interaction with their environment and themselves . Swarm robotics draws inspiration from natural systems, such as flocks of birds or ant colonies. This approach is predicted to be particularly useful in military applications, reducing the need for manpower and is already in use today.

An important feature of swarm robotics is its scalability, once the initial design is completed, it is easier to create and coordinate a large number of robots, than it is in traditional robotics. For military applications, a large advantage of swarm robotics is its redundancy; should a part of the swarm be incapacitated, the rest of the swarm can function without significant hindrance [11].

1.2. Goals

The goal of this thesis is to enable the DJI Robomaster S1 to be used in more complex tasks, namely in the creation of robot swarms. This thesis serves as a documentation of the work done, but also as a reference manual to the modifications done on the Robomaster S1. The goals accomplished in this thesis are the following:

- Analyse the current design of the DJI tank and extend its capabilities
- Design means of communication

Originally, it was also intended to create basic swarm movement formations to demonstrate the capabilities of the modified robot. However, due to unforeseen difficulty in finding a way to control the robot, after a consultation with the thesis supervisor it was decided to only focus on the first two tasks. The method that was originally intended to be used to control the S1 proved unreliable, so a more complex solution had to be used. The details of the shortcomings of the original method are described in section 2.3.

2. Background

2.1. Hardware of the Robomaster S1

The Robomaster S1 is a robot made by the DJI company, depicted in figure 2.1. It is used as an educational tool to teach the basics of programming and robot control. Users can program movement sequences, control its movement via a joystick and even deploy basic facial recognition.

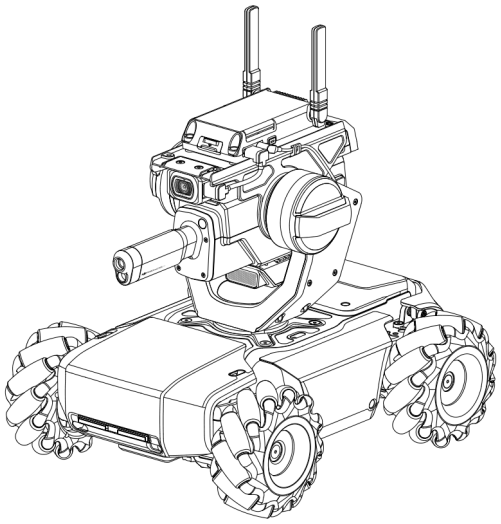

Figure 2.1: DJI Robomaster S1

Its hardware components include the chassis, which houses the majority of its electronic components. The robot is controlled via the Intelligent Controller[1] mounted atop a mechanical gimbal. The Intelligent Controller is a computer running android and serves as the brains of the robot. A camera is connected to the front of the gimbal, which allows users to stream 1080p point-of-view video. Also on the gimbal is a blaster, which is capable of firing either small gel beads or infrared beams from its blaster allowing users to engage in robot battles.

Four omnidirectional Mecanum wheels give the RoboMaster S1 good maneuverability, making it capable of performing complex movements and maneuvers. This functions based on a cancellation of forces. The wheels have rollers attached to their circumference at a 45° angle. The combinations of wheel rotations and their resulting movement is demonstrated in figure 2.2. The front wheels also feature suspension system, necessary for high speeds in rough terrain. The wheel motors are powerful M3508I brushless motors with built-in closed-loop speed control. They also overvoltage, overtemperature and short-circuit protection and have built in soft-starting.

The robot is powered by a 12V LiPo battery located at the back end of the chassis. Its capacity is 2400 mAh. It contains a chip, which communicates with the chassis. The robot will only power on, if it recognizes the battery, making it impossible to use the robot without it.

[1] official name given by DJI

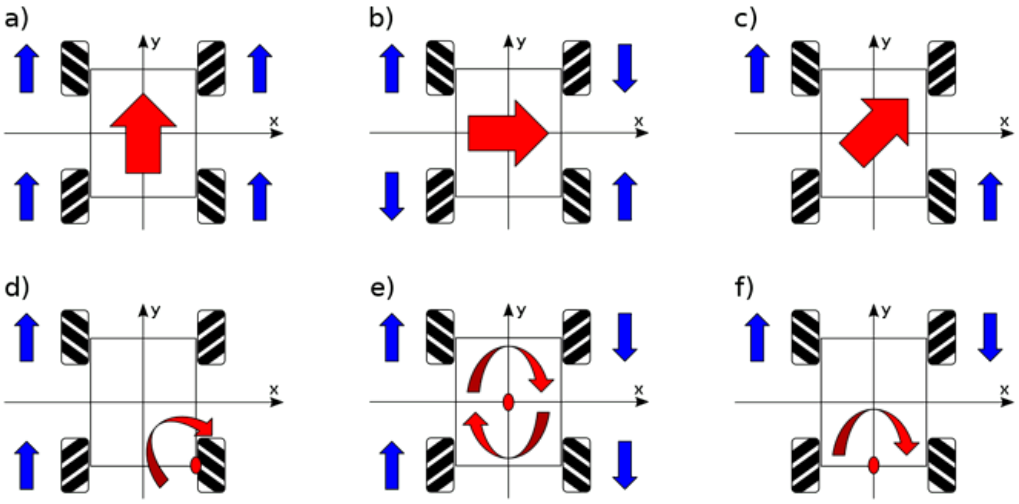

Figure 2.2: The movement capabilities of a robot equipped with mecanum wheels

Power is supplied to the robot via a four-core cable, which runs throughout the robot. They connect to a port with four pins depicted in figure 2.3. The ports are located in many places, but when fully assembled, there is a free port on the right side of the gimbal.

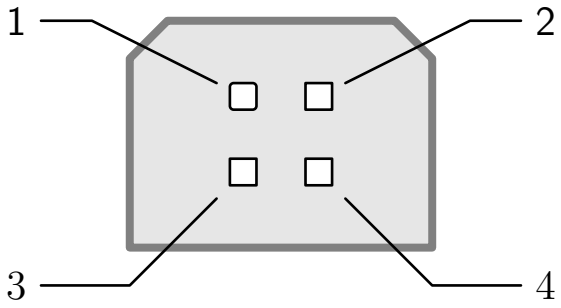

Figure 2.3: The port used in the Robomaster S1

The pinout of the port is as follows:

1. CANH - CAN bus High
2. CANL - CAN bus Low
3. 12V
4. GND

2.2. Means of communication

There are two ways in which the user can connect to the Intelligent Controller – either the user connects their device directly to the robots WiFi access point, or they connect the robot and their device to a router.

The standard way to control the robot is via the Robomaster desktop or mobile application. From there the user can either control the robot directly via a joystick, or

program the robot in the programming language Scratch or Python. However, programming is restricted to the application's editor and there is no interface through which external commands could be passed, therefore is necessary to use a different approach.

2.3. Alternate means of communication

Robomaster hack

Users of the Robomaster S1 have found a way to upload files from the significantly more open Robomaster EP to the S1's Intelligent Controller [2]. This enables several the features of the EP, the primary of which is the ability to control the robot via an SDK. This works, because the chassis and Intelligent Controller is the same on both devices.

While this would be a solution, newer versions of S1 firmware have patched this hack. Upon our attempt to upload the EP files to an updated S1, the Intelligent Controller was bricked and it was necessary to send it to Germany be repaired. This method is also quite restrictive, because the robot has to be intact in order to function, restricting further modification. Therefore, this route was not taken.

CAN Bus

The main avenue through which the Intelligent Controller communicates with the rest of the robot is the CAN bus. CAN bus is a protocol used to connect microcontrollers (nodes) to one another using a twisted pair of wires. The main nodes found on the Robomaster S1 are the following:

- Intelligent Controller
- Chassis controller
- Gimbal
- Blaster
- Armor

A diagram showing the Robomaster's CAN nodes is depicted in figure 2.4. Details about CAN communications can be found in section 2.4.

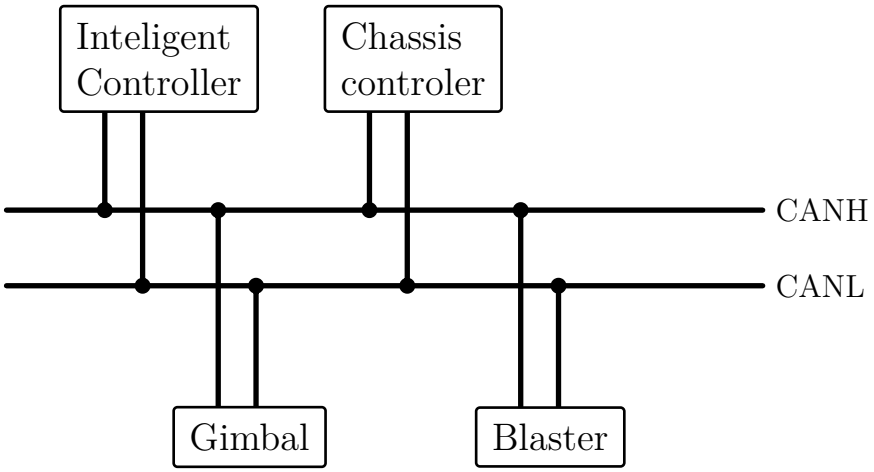

Figure 2.4: Diagram of main nodes found on the Robomaster S1

By replicating the messages of the Intelligent Controller, it would be possible to replace it entirely and even remove unnecessary components that use a lot of power, such as the gimbal. This is the method used in this thesis.

S-BUS

The microcontroller on the chassis has 3 S-BUS pins. S-BUS is a closed source protocol developed by Futaba Corporation, a remote control device manufacturer. Attaching a receiver allows the S1 to be controlled by a remote control. The protocol has been reverse engineered to some extent to work for Arduino boards. Due to its closed nature, it is not well documented and therefore this method not used.

2.4. Details of CAN communication

The Control Area Network bus is a message-based protocol primarily used in automobile and industrial applications. It connects nodes (microcontrollers, FPGAs, etc.) via two wires (CAN high and CAN low), which terminate with 120 Ω resistors at each end.

The CAN bus uses differential signals. Traditionally data is sent over only one line, with the other acting as a reference, but then using differential signals, the data is sent over both lines, the magnitude of both signals is the same, but the polarity is opposite, which increases noise immunity [9]. An example of differential signals on the can bus can be seen in figure 2.5. A logical *1* occurs, when the voltages on the two lines are the same and a *0* occurs when the difference in voltages is different, typically 3.3 V.

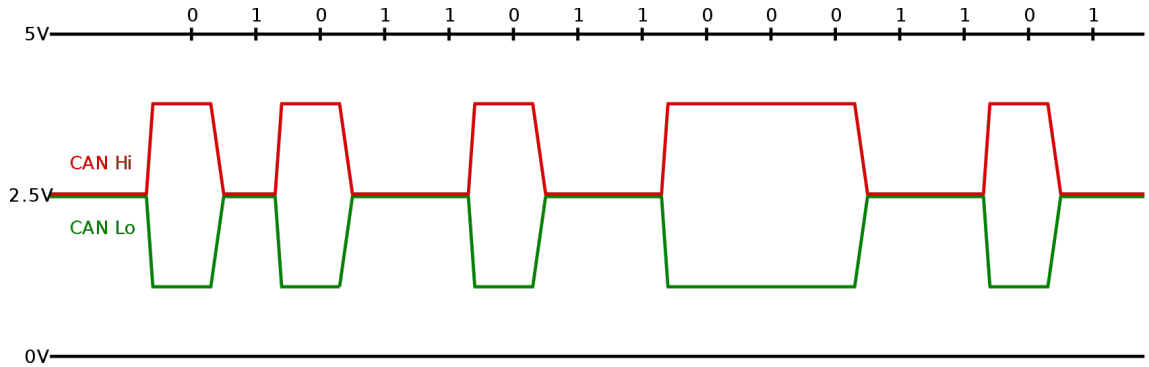

Figure 2.5: CAN bus differential signal

The baud rate of the CAN bus can be up to 1 Mbit/s for classical CAN and up to 5 Mbit/s for CAN FD, which is a newer version of the CAN protocol. The CAN bus of the Robomaster operates at 1 Mbit/s. The general structure of CAN messages (or frames) is described in table 2.1

For the purposes of this project, the important bits are the identifier (ID) and the data length code (DLC). In fact, most of the fields are handled by the CAN transceiver.

2.5. Useful sources

DJI has open sourced the SDK for the Robomaster EP [3]. Despite being software for a different robot, many parts of the code produce similar results as the S1 (such as CAN message structure) and are therefore useful in reverse engineering the CAN bus protocol.

Field name	Length (bits)
Start-of-frame	1
Identifier	11
Stuff bit	1
Remote transmission bit	1
Identifier extension bit	1
Reserved bit	1
Data length code	0-64
Cyclic redundancy check	15
CRC delimiter	1
ACK slot	1
ACK delimiter	1
End of frame	7
Inter-frame spacing	3

Table 2.1: CAN bus frame format

Another useful resource is the detailed hardware analysis of the S1 on hackday.io by Duane Deng [4]. It includes a full disassembly of the Robomaster S1 and details of some of the chips used. Some assumptions about the software are not precise.

There are two Github repositories that attempt to achieve the same goal as this project, one for the Raspberry Pi [1], written in Micropython and one for the STM32 [10], written in C. They contain a list of observed CAN commands, which is very helpful, however, in testing they were not completely accurate.

3. Solution

The method chosen for controlling the S1 is to replace the Intelligent Controller and replicate the commands used to control the movement. The Intelligent Controller communicates with the robot via CAN bus, so it is necessary to first reverse engineer the messages sent and then replicate using on a custom board. It was also decided to remove the gimbal, as it added unnecessary bulk. However, gimbal control has also been implemented, should the user wish to keep it.

3.1. Reverse engineering the Robomaster's CAN bus

The Robomaster has an additional protocol, used atop the CAN bus protocol. The data (which we will call commands) it sends is longer than the 64 bits that one standard CAN message allows, it splits the data into multiple CAN messages, which are then sent as the data length code (DLC) of CAN messages.

3.1.1. CAN bus sniffing

Sniffing in the context of reverse engineering is the process of capturing low-level data in an effort to understand and utilize it. A common way of sniffing the CAN bus is by using a USB CAN bus adapter with software capable of interpreting this data such as Wireshark.

Typically the rate at which messages are sent makes it impractical to read one by one, many of the messages also repeat periodically (to update some value, like the speed of the vehicle), so what Wireshark does is print out the last message sent per CAN ID, then watches for changes in the individual bytes of messages and marks them. The user can then for instance turn on the turn signal lights and watch, which of the message's bytes have changed and how.

This works for most purposes, however in the case of the S1, where the data sent is split into several messages and where one CAN ID sends completely different messages, it does not. The solution is to use a command line tool – candump. Candump is a part of the Linux kernel (SocketCAN) and it, much like Wireshark, prints out individual CAN messages received by it. But because it is a command line tool, we can then process and filter the stream with *grep* and get a display of complete messages. A Raspberry Pi 4 and a Waveshare RS485 CAN hat were used in this project.

3.1.2. Filtering the stream of CAN messages

To create the filters, we have to know the general structure of the 8-byte messages. Luckily, much of that is provided to us by the SDK from the developers [3]. Not all of it is accurate, so it is also necessary to review the output of the sniffs.

A good starting point is always the first three bytes; the first byte 0x55[1], which marks the beginning of a message. The second byte, which is the length of the message, and the

[1] strictly speaking, this not always the case, as various bytes, such as the message count and checksum bytes sometimes also output 0x55

third byte, which is always 0x04[2]. Filtering with this sequence is a good way of getting the first 8-byte message. Then with grep's -Ax option, we can pass through x amount of following lines, which include the rest of the message. However, because several messages have the same length, it may also be necessary to filter the second 8-byte message as well. The three byte are again a good starting point.

With several of these filters in place we can then join the lines together to better visualize the changes in bytes and determine, which bytes react to the action we are trying to replicate.

3.1.3. Example

Because this process is a crucial step in understanding the Robomaster's CAN communications, to better illustrate the method of identifying and understanding a command, an example is included.

Once the chosen sniffing tool (in our case the Raspberry Pi 4 with the Waveshare RS485 CAN hat) have been connected to one of the Robomaster S1's CAN bus pins, we can turn on the robot. In a terminal on the Raspberry Pi, we enter the following command to initialize the CAN interface:

```
# ip link set up can0 type can bitrate 1000000
```

This creates an can interface running at 1Mbit/s. To display the messages being sent on the CAN bus, we enter the following command:

```
candump can0
```

The output of this command will look similarly to:

```
...
can0 203 [5] 3D E5 BB 4E 81
can0 203 [8] 55 11 04 92 04 03 D9 6F
can0 203 [8] 55 0F 04 A2 04 C3 DB 6F
can0 203 [7] 00 3F 2E 80 00 90 46
can0 202 [8] 55 3D 04 1E 03 09 C0 8A
can0 202 [8] 20 48 08 00 01 B8 61 04
can0 202 [8] 00 CA B8 1D 11 00 00 00
can0 202 [8] 00 00 00 91 CC 3D 3A 26
can0 202 [8] 8F ED B9 00 00 00 00 3A
can0 202 [5] 00 00 00 C3 0C
can0 201 [8] 55 0E 04 66 09 03 C0 8A
can0 201 [6] A0 48 08 01 A7 29
can0 213 [8] 55 10 04 56 78 28 01 00
can0 213 [8] 00 00 F1 01 79 BC B7 77
can0 201 [8] 55 0F 04 A2 F1 C3 8D 0B
can0 201 [7] 00 0A 53 32 00 52 F7
can0 202 [8] 55 31 04 53 03 04 C1 8A
can0 202 [8] 20 48 08 00 00 BD 61 04
```

[2] again, based on the information from the SDK, this is theoretically not true, but in testing, the value of the byte was always 0x04

```
can0 202 [8] 3F 01 B3 BD 3B AE 10 96
can0 202 [8] 3C 2A 26 08 BC C8 2F DF
can0 202 [1] 83
...
```

The first column contains the interface name, the second column contains the CAN ID, the third column contains the number of bytes sent in the given CAN message and the rest are the byte values themselves. Because we are replacing the Intelligent Controller, we are only interested in the bytes sent by it. The CAN ID of the Intelligent controller is 201, so to filter out the unwanted IDs (gimbal, chassis commands), we pipe the output of the command into grep:

```
candump can0 | grep 201
```

The output:

```
...
can0 201 [8] 55 0E 04 66 09 03 AB 8A
can0 201 [6] A0 48 08 01 33 83
can0 201 [8] 55 1B 04 75 09 C3 10 6B
can0 201 [8] 00 3F 60 00 04 20 00 01
can0 201 [8] 08 40 00 02 10 04 00 00
can0 201 [8] 04 16 6B 00 04 69 08 05
can0 201 [7] 00 00 00 00 6C FB 91
can0 201 [8] 55 0D 04 33 0A 38 28 0B
can0 201 [5] 40 00 01 66 26
can0 201 [8] 55 0E 04 66 09 03 AD 8A
can0 201 [6] A0 48 08 01 C9 9B
can0 201 [8] 55 1B 04 75 09 C3 11 6B
can0 201 [8] 00 3F 60 00 04 20 00 01
can0 201 [8] 08 40 00 02 10 04 00 00
can0 201 [8] 04 17 6B 00 04 69 08 05
can0 201 [8] 55 1B 04 75 09 C3 12 6B
can0 201 [8] 00 3F 60 00 04 20 00 01
can0 201 [8] 08 40 00 02 10 04 00 00
can0 201 [8] 04 18 6B 00 04 69 08 05
can0 201 [8] 55 0E 04 66 09 03 AF 8A
can0 201 [6] A0 48 08 01 9F 93
...
```

We know that all commands start with 0x55, the next byte is the length and the third is always 0x04, To guarantee that these three bytes are at the beginning of the CAN message, we can also use the "] " that comes before 55 in each message. We create a filter, which passes only the first eight bytes (the first CAN message) of the command:

```
candump can0 | grep 201 | grep "] 55 .. 04"
```

The "." character means match with any character. The output will be:

```
...
can0 201 [8] 55 0F 04 A2 F1 C3 81 0B
```

```
can0 201 [8] 55 1B 04 75 09 C3 B3 6A
can0 201 [8] 55 1B 04 75 09 C3 B4 6A
can0 201 [8] 55 0E 04 66 09 03 3B 8A
can0 201 [8] 55 1B 04 75 09 C3 B5 6A
can0 201 [8] 55 0D 04 33 0A 38 1F 0B
can0 201 [8] 55 14 04 6D 09 04 BC 6A
can0 201 [8] 55 0E 04 66 09 03 3D 8A
can0 201 [8] 55 1B 04 75 09 C3 B8 6A
...
```

In the output above we see the first 8 bytes of several commands. We choose any of these, for instance the `55 14 04 6D 09 04 BC 6A` command. The length in bytes of this command is 0x14, which is 20 in decimal. This means that this command will be split into 3 CAN messages, the first will be 8 bytes long, the second as well and the third will be 4 bytes long. To isolate these 3 CAN messages, we can use the -A option of grep, which prints not only the line with matched characters, but also a number of lines after:

```
candump can0 | grep 201 | grep -A2 "] 55 14 04 .. 09 04"
```

We get the output:

```
can0 201 [8] 55 14 04 6D 09 04 EE 65
can0 201 [8] 00 04 69 08 05 00 00 00
can0 201 [4] 00 6C 02 15
--
can0 201 [8] 55 14 04 6D 09 04 73 68
can0 201 [8] 00 04 69 08 05 00 00 00
can0 201 [4] 00 6D 32 82
--
can0 201 [8] 55 14 04 6D 09 04 A1 69
can0 201 [8] 00 04 69 08 05 00 00 00
can0 201 [4] 00 6C E9 28
--
can0 201 [8] 55 14 04 6D 09 04 BC 6A
can0 201 [8] 00 04 69 08 05 00 00 00
can0 201 [4] 00 6C 4D 49
--
can0 201 [8] 55 14 04 6D 09 04 C6 6A
can0 201 [8] 00 04 69 08 05 00 00 00
can0 201 [4] 00 6C 12 28
```

With the information we have gathered, we can now search the Robomaster SDK source code to see, if this command is defined there. If it is, then the task is finished and we can use it in our code, if it is not, we have to do more work. We will now connect the robot to the Robomaster application and control the robot in various ways in an attempt to illicit a change in the bytes of the command. If the bytes do not react to any input via the Robomaster application, it may be prudent to send the command anyway, as seen in the output of the dumps. The robot generally doesn't mind receiving unknown

commands. The vast majority of commands are sent periodically, so it is necessary to know the frequency of

To check the frequency of a command, the -tz option of candump may be used. With this option candump will include a timestamp to its output. Apply a filter to only display the first 8 bytes of a command (for instance "] 55 1B 04)" and observe the output:

```
...
(000.573636) can0 201 [8] 55 1B 04 75 09 C3 5F E8
(000.583573) can0 201 [8] 55 1B 04 75 09 C3 60 E8
(000.593602) can0 201 [8] 55 1B 04 75 09 C3 61 E8
(000.603852) can0 201 [8] 55 1B 04 75 09 C3 62 E8
(000.613602) can0 201 [8] 55 1B 04 75 09 C3 63 E8
(000.633550) can0 201 [8] 55 1B 04 75 09 C3 65 E8
(000.643575) can0 201 [8] 55 1B 04 75 09 C3 66 E8
(000.653570) can0 201 [8] 55 1B 04 75 09 C3 67 E8
(000.663605) can0 201 [8] 55 1B 04 75 09 C3 68 E8
(000.673619) can0 201 [8] 55 1B 04 75 09 C3 69 E8
(000.683560) can0 201 [8] 55 1B 04 75 09 C3 6A E8
(000.693582) can0 201 [8] 55 1B 04 75 09 C3 6B E8
(000.703801) can0 201 [8] 55 1B 04 75 09 C3 6C E8
(000.713553) can0 201 [8] 55 1B 04 75 09 C3 6D E8
(000.723582) can0 201 [8] 55 1B 04 75 09 C3 6E E8
(000.733585) can0 201 [8] 55 1B 04 75 09 C3 6F E8
...
```

We can see a new column, which contain the timestamp of the respective CAN message in seconds, counting from the moment the command was called. This command has a transmission interval of 10ms.

Not all commands are transmitted periodically, to identify those, it is useful to first filter for the first CAN message, then use a negative filter to filter known commands.

3.2. Hardware solution

ESP32

The device chosen as a replacement to the Intelligent Controller is the ESP32 microcontroller from Espressif. The ESP32 is a low-cost SoC microcontroller. It features a dual-core CPU, WiFi and Bluetooth capability and importantly a CAN interface, meaning that with a transceiver, it can send CAN commands. The logic level of the ESP32 is 3.3V. The exact model used is the *ESP32-WROOM-32U* development board.

The ESP32 supports the ESP-NOW protocol – a wireless communication protocol designed by Espressif, which allows for low-latency, low-power communication between two ESP devices. It is peer-to-peer, not requiring a central router, exactly what is needed for a dynamically expandable robot swarm.

SN65HVD230

Beside the ESP32, a CAN transceiver is needed. The SN65HVD230 is a 3.3V CAN transceiver capable of up to 1Mb/s baud rate, sufficient for this project. It is not necessary to add a 120 Ω resistor to the CANH and CANL output of the transceiver, if using a breakout board, it is necessary to desolder the resistor. The wiring diagram for connecting the SN65HVD230 to the ESP32 and the CAN bus ports of the S1 is depicted in figure 3.1.

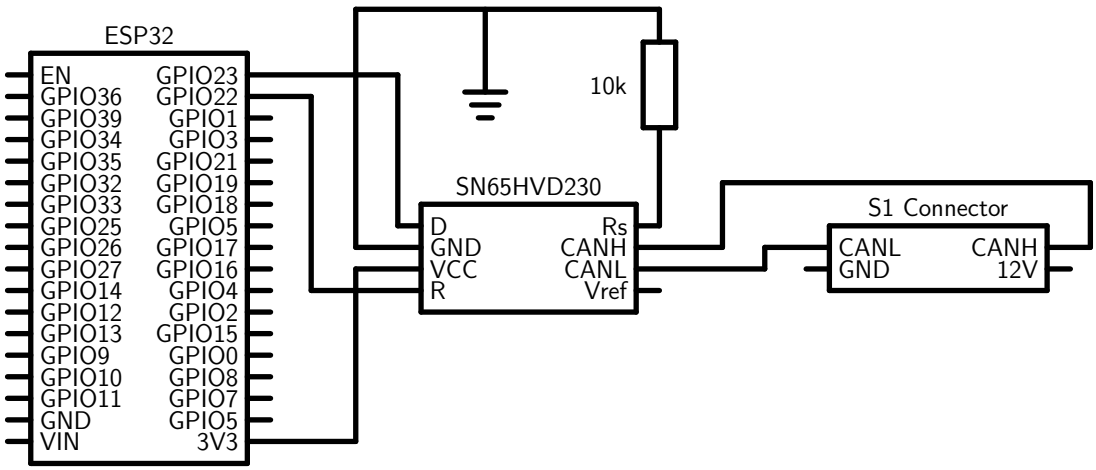

Figure 3.1: Wiring diagram of the ESP32, SN65HVD230 and the S1's CAN bus

Both of the S1 connectors are located on the chassis's microcontroller. Some ESP32 development boards have a on-board voltage regulator that takes 5 - 12V making it possible to power the ESP32 from the 5V pins on the chassis (they are the two right most of the pins marked S-BUS) or directly from the battery.

3.3. Reference manual

This section is a manual to be used along the code provided with this thesis. This is also a useful resource for anyone attempting to analyse the commands of the Robomaster S1, as no such document is currently available.

The Robomaster has an additional protocol, used atop the CAN bus protocol. The data (which we will call commands) it sends is longer than the 8 bytes that one standard CAN message allows, so it splits the data into multiple CAN messages, which are then sent up to 8 bytes at a time. The structure of the Robomaster commands is described in table 3.1

In order for the for the chassis to operate normally, several commands need to be sent periodically. These commands are generally used to update an important variable, such as movement information.

The symbol `--` denotes either a CRC8 byte, counter of command bytes, or CRC16 bytes. CRC bytes are discussed in detail in section 3.3.4.

The symbol `XX` denotes a byte, into which a variable is encoded.

Byte	Value	Description
1	0x55	indicates the beginning of a new message
2	-	LSB of message length
3	0x04	MSB of message length or 0x04
4	-	8-bit cyclic redundancy check CRC8
5	0x09	internal ID of sender
6	-	internal ID of receiver
7	-	LSB of counter of commands
8	-	MSB of counter of commands
9	-	attributes
10	-	SDK command set ID
11	-	SDK command ID
12+	-	individual command data
last − 1	-	16-bit cyclic redundancy check CRC16
last	-	16-bit cyclic redundancy check CRC16

Table 3.1: Robomaster S1 command structure

All other bytes are for the purposes of this thesis constant[3]. The following is a summary of important commands the Robomaster S1 sends.

3.3.1. Periodic commands

These commands are sent periodically. They are sent at a set frequency, however, in testing, the frequency does not need to be exact. In fact, the robot will still work with sending some commands at a frequency ten times smaller than the measured frequency. In the following commands, their frequency is expressed as transmission interval, meaning the time elapsed between each transmission of the command.

Chassis movement command

This command controls the movement of the chassis. This command is sent with a transmission interval of 10ms. The form of the command is the following:

```
55  1B  04  --  09  C3  --  --
00  3F  60  XX  XX  XX  00  01
XX  XX  00  02  10  04  XX  00
04  --  --
```

There are three variables, that determine movement:

x	speed forwards/backwards
y	speed left/right
z	rotation to the left/right

[3]the bytes change, but they do not in any observed way impact the behaviour of the robot

The x and y movement data is packed into the 12th, 13th and 14th bytes and the z movement data into the 17th and 18th byte. The following is code written in C, which demonstrates the exact method of packing the variables into the bytes:

```
bytes[11] = y & 0xFF;
bytes[12] = ((x << 3) & 0xF8) | ((y >> 8) & 0x07);
bytes[13] = (x >> 5) & 0x3F;
bytes[16] = ((z << 4) & 0xF0) | 0x08;
bytes[17] = (z >> 4) & 0xFF;
```

Zero speed is achieved by setting by setting x, y and z to 1024. The 23rd byte dictates, whether the robot moves only in the x and y direction (0x04), only rotate around the z axis (0x08), both x and y and z (0x0C) or no movement at all (0x00).

The S1 has two different movement modes, one is active when the chassis is being given movement instructions from the programming interface, the other when the free-drive mode is used (movement using joystick for android or keyboard for PC). We are using the command, that is sent when using the free-drive mode.

Gimbal movement command

This command controls the movement of the gimbal. It is sent with a 10ms transmission interval. The form of the command is the following:

```
55  14  04  --  09  04  --  --
00  04  69  08  05  XX  XX  XX
XX  6C  --  --
```

Two variables determine the movement:

yaw	turn left/right
roll	tilt up/down

Yaw is packed into the 14th and 15th byte, roll into the 16th and 17th. The exact packing method written in C is shown below:

```
bytes[13] = roll & 0xFF;
bytes[14] = (roll >> 8) & 0xFF;
bytes[15] = yaw & 0xFF;
bytes[16] = (yaw >> 8) & 0xFF;
```

Zero speed is achieved by setting yaw and roll to 0.

Other periodic commands

These are commands, that were sent periodically and were observed to be needed for the robot to function.

The following command is sent with a transmission interval of 20ms. The form of the command is the following:

```
55  0F  04  --  F1  C3  --  --
00  0A  53  32  00  FF  FF
```

The following command is sent with a transmission interval of 100ms. The form of the command is the following:

```
55  0D  04  FF  0A  FF  FF  FF
40  00  01  --  --
```

The following command is sent with a transmission interval of 100ms. The form of the command is the following:

```
55  0E  04  FF  09  03  FF  FF
A0  48  08  01  --  --
```

The following command is sent with a transmission interval of 5000ms. The form of the command is the following:

```
55  12  04  FF  F1  C3  FF  FF
40  00  58  03  92  06  02  00
--  --
```

The following command is sent with a transmission interval of 1000ms. The form of the command is the following:

```
55  49  04  FF  49  03  FF  FF
00  3F  70  B4  0F  66  03  00
00  D3  03  3D  08  10  00  08
00  08  00  08  00  08  00  08
00  08  00  00  00  00  00  00
00  00  00  00  00  00  00  00
00  00  00  00  00  7E  0E  F3
0B  D9  07  0E  07  3D  07  6A
08  62  0A  05  0B  D6  0B  FF
--
```

3.3.2. Fire commands

These commands control the blaster, which can be fired in two modes.

Bead blaster fire command

The following command fires the blaster in bead mode.

```
55  0E  04  --  09  17  --  --
00  3F  51  01  --  --
```

Infrared blaster fire command

The following command fires the blaster in infrared light mode.

```
55  16  04  --  09  17  --  --
00  3F  55  73  00  FF  00  01
28  00  00  00  --  --
```

3.3.3. Initialization commands

These commands are sent once to set certain values in the chassis microcontroller.

Chassis accelerator enable command

The following command sets enables the chassis accelerator, allowing wheels to accelerate.

```
55  16  04  --  09  17  --  --
00  3F  55  73  00  FF  00  01
28  00  00  00  --  --
```

Speed mode command

The following command sets the speed mode. This sets a maximum speed limit.

```
55  0E  04  FF  09  C3  FF  FF
40  3F  3F  XX  --  --
```

The twelfth byte dictates the mode, set the byte to:
`01` for fast mode
`02` for medium mode
`03` for slow mode
`04` for custom mode, where the speed is determined by the speed set via the Robomaster applications

Other initialization commands

These are commands, that were observed to be necessary for the robot to function.

```
55  15  04  --  F1  C3  --  --
00  03  D7  01  07  00  02  00
00  00  00  --  --
```

```
55  12  04  --  09  03  --  --
40  48  01  09  00  00  00  03
--  --
```

```
55  1C  04  --  09  03  --  --
40  48  03  09  00  03  00  01
FB  DC  F5  D7  03  00  02  00
01  00  --  --
```

```
55  12  04  --  09  03  --  --
40  48  01  09  00  00  00  03
--  --

55  24  04  --  09  03  --  --
40  48  03  09  01  03  00  02
A7  02  29  88  03  00  02  00
66  3E  3E  4C  03  00  02  00
32  00  --  --
```

3.3.4. Robomaster cyclic redundancy check

The protocol used by the Robomaster S1's CAN communication includes several cyclic redundancy check (CRC) bytes as a part of the command. This is a method used in digital networks to detect message corruption. It is done by calculating a value based on the content of the message. The value is then added to the message, usually as a additional byte. Note that the CRC algorithm used by the CAN bus is not the same as the algorithm used by the Robomaster.

The Robomaster uses two algorithms: CRC8 and CRC16, the number represents the number of bits the respective algorithm generates. These algorithm would be difficult to reverse engineer, fortunately they are defined in the SDK. Despite being called CRCs by the SDK, the algorithms used by conventional CRCs (polynomial division [7]). Instead it calculates the CRC for a given set of data using a precomputed lookup table. The following are the C implementations of the two algorithms.

CRC8

This algorithm takes the first three bytes of the command and returns a single byte, which is stored in the fourth byte of the command. Because the first three bytes do not change (at least in testing), it would be possible to save on computation by pre-calculating these bytes.

```
unsigned char crc8_calc(const unsigned char *data, int len) {
    unsigned char crc = 0x77;
    for (int i = 0; i < len; i++) {
        crc = crc8_table[crc ^ data[i]];
    }
    return crc;
}
```

`crc8_table` is an array of 256 unsigned chars, `len` is the amount of bytes in `data[]`. The initial value `0x77` of `crc` is a magic number given by the SDK.

CRC16

This algorithm takes the entire command except for the last two bytes and returns two bytes to be used as the last two bytes of the command.

```
unsigned short crc16_calc(const unsigned char *data, int len) {
    unsigned short crc = 0x3692;
    for (int i = 0; i < len; i++) {
        crc = ((crc >> 8) & 0xff) ^ crc16_table[(crc ^ data[i]) & 0xff];
    }
    return crc;
}
```

`crc8_table` is an array of 256 unsigned chars, `len` is the amount of bytes in `data[]`. The initial value `0x3692` of `crc` is a magic number given by the SDK.

3.4. Code description

The subject of this section is a description of the code presented in the archive submitted along with this thesis. There are two files, *robomaster*, which is the main program, and *controller*, which is a basic wireless controller program, which takes keyboard input to control the Robomaster. The board used, the ESP32, allows for two development frameworks, the ESP Arduino Core and the ESP-IDF. Because this is a low level application and fine-tuning is needed, the ESP-IDF is used. The code is compiled using ESP-IDF's `idf.py build` command, which uses CMake under the hood, but it is possible to use CMake directly.

FreeRTOS

FreeRTOS is a real-time operating system kernel made to run on microcontrollers. Because Vanilla FreeRTOS is a single-core RTOS, Espressif heavily modified it in order to support multiple cores and created the ESP-IDF FreeRTOS implementation. It allows simple implementation of semaphores, tasks and software timers, all of which are used in the code.

Main loop

The main loop contains the following:

- Variable definition and initialization
- ESP-NOW initialization
- CAN bus initialization
- CAN send loop initialization
- Send Robomaster boot commands task
- Create semaphore
- Software timer creation
- `/while(1)`

The CAN messages are sent on by 3 different software timers: 10ms, 100ms and 1000ms, which call the function `send_command(uint8_t command_number)`. Based on the value of `command_number`, this function loads the command bytes into a one dimensional array, the size of which is determined by the second byte loaded. The array's bytes are then modified based on what the command it contains requires (for the movement command, the movement variables are packed into the bytes) and CRC8 and CRC16 is applied. The bytes of the array are then loaded into `can_data_buffer`, which is a buffer of bytes ready to be sent. It also loads the total length of the command into `can_data_length_buffer`. Both of these buffers are protected by semaphores.

A task watches for changes in the `can_data_length_buffer`, when it detects a change, based on the data length in `can_data_length_buffer` it loads that amount of bytes, splits them into groups of up to eight (the last message can be smaller) and sends them via the function `twai_send` into the CAN bus.

Controller

The *controller* folder of the archive submitted contains code used to create a controller for the Robomaster, also using an ESP32. The program watches for keyboard input, interprets it and sends the values via the ESP-NOW wireless protocol to the ESP32 on the Robomaster. It then reads it onto movement variables and uses them in the `send_command()` function.

4. Conclusion

This thesis undertook the task of reverse engineering the DJI Robomaster S1 to enable its integration into a homogeneous robot swarm. The primary objective was to bypass the constraints of the Robomaster S1's proprietary software and modify its hardware solution to enable future custom control and enable its use as an autonomous agent in a robotic swarm.

The reverse engineering process centered on the S1's communication protocols, particularly its CAN bus communications. The Robomaster S1's computer, the Intelligent Controller, sends instructions via CAN bus messages, which are read by other devices connected to the bus. By analyzing these messages it was possible to replicate and control the robot's functions independently of the original software. A crucial modification involved replacing the Intelligent Controller with a microcontroller, which then sent the analyzed messages itself.

The device chosen to replace the Intelligent Controller is the ESP32 from Espressif Systems. It is a popular dual-core microcontroller that is, if connected to a CAN transceiver, capable of sending CAN messages. Using the data collected by analysing the commands of the S1, the ESP32 was successfully programmed to replicate the Intelligent Controller. The source code for the ESP32 is available in the archive submitted with this text and also at https://github.com/HayBudden/robomaster_s1_esp32.

The achieved result is very simple, inexpensive[1] and easy to deploy. This is an important aspect in swarm robotics, where the design is replicated many times. In addition to the software modifications, practical hardware adjustments led to notable improvements in performance. Primarily:

- Battery life: The robot's battery life increased significantly from 1 hour to 4 hours in standby mode (powered on with no motion). This enhancement is useful for swarm operations, where prolonged operational periods without intervention are often required.

- Gimbal removal: The modifications also allowed for the removal of the gimbal, significantly reducing the weight and height of the robot, expanding its movement potential. Other parts of the robot may also be removed to reduce weight further.

All of the robot's original movement capabilities have been preserved, including gimbal movement. The range at which the modified robot could be controlled reliably was measured around 40 metres in an open field and the maximum range at which it received commands was 70m. This is with the ESP32's integrated antennas, which can be upgraded (distances over 1000 m) if needed. ESP-NOW, the protocol used for wireless communication in this project, also supports bridging, allowing robots that would be too far from each other to communicate, can route their traffic via a middle-man robot, which is a realistic scenario in a swarm of robots. There is also zero noticeable latency when controlling the robot.

By successfully reverse engineering and modifying the Robomaster S1, the project demonstrated that commercially available robots could be repurposed for swarm robotics research, offering a cost and time-effective alternative to developing custom robots. The modified DJI Robomaster S1 can now be used effectively in a swarm.

[1] The cost of an ESP32 development board is 200 CZK (eq. 8.8 USD), the cost of the SN65HVD230 breakout board is 100 CZK (eq. 4.4 USD)

Bibliography

[1] Braaf, J. *Robomaster-Micropython*. 2020. Github. https://github.com/JohnieBraaf/Robomaster-Micropython

[2] BRUNOGA. *Robomaster*. 2024. Github. https://github.com/brunoga/robomaster

[3] DJI-SDK. *RoboMaster-SDK*. 2024. Github. https://github.com/dji-sdk/RoboMaster-SDK

[4] Degn, D. *DJI Robomaster S1 Hacks*. 2019. Hackaday.io. https://hackaday.io/project/167276-dji-robomaster-s1-hacks.

[5] Eilam, E. *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, 2011.

[6] Hamann, H. *Swarm Robotics: A Formal Approach*. Springer, 2018.

[7] Koopman, P. *Understanding Checksums and Cyclic Redundancy Checks*. in publishing, 2024

[8] Ljungqvist, B., Reinmuller B. *Clean Room Design: Minimizing Contamination Through Proper Design*. Routledge, 2018.

[9] Ott, H. W. *Noise Reduction Techniques in Electronic Systems*. Wiley-Interscience, 1988.

[10] RoboMasterS1Challenge. *Robomaster_s1_can_hack*. (2024). Github. https://github.com/RoboMasterS1Challenge/robomaster_s1_can_hack

[11] Sangeetha, M., Srinivasan, K. Swarm Robotics: A New Framework of Military Robots. *Journal of Physics: Conference Series*, vol. 1717, no. 1, Jan. 2021, p. 012017, https://doi.org/10.1088/1742-6596/1717/1/012017.

[12] Saunders, P. C., Wiseman, J. K. *Buy, Build, Or Steal: China's Quest for Advanced Military Aviation Technologies*. Createspace Independent Publishing Platform, 2011.