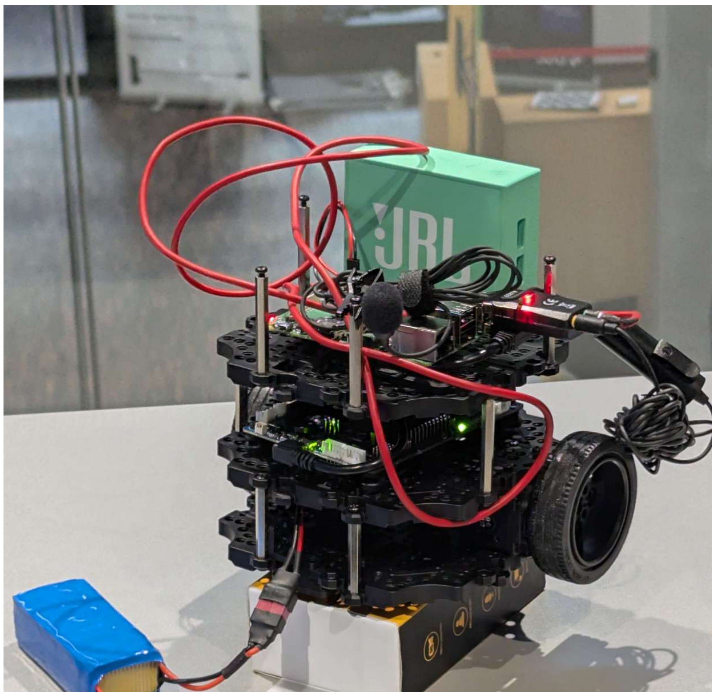
University of Southern Denmark

Mobile Robot

TURTLEBOT3 - CRC - DMTF



Robotics 3rd Semester

23rd of December 2025

Group 6

Asmus Snede Rise (27/09/02) (asris24)

August Maximillian Rosford Tranberg (06/09/04) (autra24)

Elias Ottar Alstrup (25/03/04) (elals24)

Emil Gramstrup (25/01/04) (egram24)

Frederik Nørregaard Wilkens (23/03/04) (frwil24)

Abstract

This project explores the use of Dual-Tone Mono-Frequency (DTMF) signals to wirelessly control the trajectory of a TurtleBot3 mobile robot. By leveraging a layered software architecture, the system integrates signal processing, reliability mechanisms, data logging, and motor control to achieve robust communication between a Linux-based sender and the robot. The protocol enables users to issue high-level commands (e.g., drive, turn, stop) via audio signals, which are decoded and executed in real time.

The system's design includes five distinct layers: the Application Layer for user interaction, the Command Abstraction Layer for command encoding and state management, the Reliability Layer for error detection using Cyclic Redundancy Check (CRC), the Signal Encoding/Decoding Layer for converting binary data to/from DTMF frequencies, and the Physical Layer for audio transmission and reception. Testing under various conditions—ideal, noisy, and long-range—demonstrated a high success rate (98.46%–100%) in command transmission, with minimal false positives and reliable error detection.

The project successfully validates the feasibility of using DTMF for mobile robot control, offering a flexible and modular framework adaptable to diverse applications. While latency and occasional confirmation failures were observed, the system's overall reliability and efficiency confirm its potential for wireless, audio-based communication in robotics and beyond.

Titlepage

Project title	RB-PRO3 - Turtlebot3 (Semester Project in Mobile Robotsystems)
Educational institution	University of Southern Denmark, Faculty of Engineering
Educational program	BSc in Robotics
No. of attachments	8
Project period	1st of September 2025 - 23rd of December 2025
Project coordinaters	Leon Bodenhagen (lebo@mmmi.sdu.dk)
	Rupam Singh (rupsi@mmmi.sdu.dk)
Project group no.	6
Project participators (examnumber)	Asmus Snede Rise (4139364)
	August Maximillian Rosford Tranberg (215661973)
	Elias Ottar Alstrup (215662371)
	Emil Gramstrup (4139374)
	Frederik Nørregaard Wilkens (4139312)
GitHub link	`https://github.com/Gubi0609/Semesterprojekt-i-mobile-robotsystemer`
Demovideo of system	`https://youtu.be/oCFVBX-PZiQ`
Responsibilities:	
Asmus R.	ROS, Robot handling, Thread handling
August T.	Error handling, Integration of sound & database
Elias A.	Sound rx/tx, Signal Processing, Command Protocol
Emil G.	Data Logging (SQL), Signal Processing
Frederik W.	ROS, Robot handling

Contents

Glossary

bps	Bits per second.
CLI	Command Line Interface.
CRC	Cyclic Redundancy Check.
DDS	Data Distribution Service.
DFT	Discrete Fourier Transform.
DTMF	Dual-Tone Mono-Frequency.
FFT	Fast Fourier Transform.
FSK	Frequency-shift keying.
MSB	Most Significant Bit.
RCLCPP	ROS Client Library C++.
ROS	Robot Operating System.
SQL	Structured Query Language.
uint16_t	Unsigned Integer 16-bit.
uint8_t	Unsigned Integer 8-bit.
XOR	Exclusive OR.

1 Introduction

Wireless data transmission is widely adopted in industrial applications due to its flexibility and mobility, advantages that wired connections cannot provide. Although wired communication ensures reliability and speed, it inherently restricts movement. This limitation is particularly critical for mobile robots, which require unconstrained operation to function effectively. A wired connection, for example, would physically constrain a mobile robot with trailing cables.

An alternative approach to communication between dynamic systems is the use of sound waves. Inspired by human communication, where vocal cords transmit information and ears receive it, this project establishes a communication link between a computer and a mobile robot using speakers and microphones. Given the robot's mobility, sound-based communication presents a natural and practical solution.

This project implements data transfer through audio signals, specifically DTMF tones. However, wireless communication is susceptible to interference, which can degrade signal quality and make it challenging to distinguish meaningful data from noise. To address this, digital audio processing and a robust communication protocol are employed to mitigate noise and enhance reliability.

The system is designed using a layered communication protocol, optimized for fast and reliable robot control. This modular architecture allows for clear separation of system components, improving flexibility and maintainability. Each layer can be independently tailored to meet specific requirements, ensuring adaptability to diverse applications. The primary objective of this project is to achieve control of a TurtleBot3 using DTMF tones.

2 Project description

The purpose of this semester project is to utilize DTMF sounds to communicate between two devices, one of which should be a mobile robot. In this context, a communication protocol should be designed.

The following criteria are given for the project:

- Utilize the concept of DTMF sound for communication between two or more devices.
- Use at least one mobile robot.
- Develop a communication protocol.
- Use a layered software architecture.

A use-case for the project incorporating these criteria was then established. A communication protocol enables a user to issue control commands to a mobile robot in a fast and reliable manner. The protocol provides a simple command interface that ensures consistent execution of user inputs.

3 System Description

This project implements a layered communication protocol for controlling a TurtleBot3 mobile robot using Dual-Tone Mono-Frequency (DTMF) signals. The system enables users to issue high-level commands (e.g., drive, turn, stop) to the robot via audio signals, which are decoded and executed in real time. The protocol is designed to be reliable and lightweight, with a focus on error detection and efficient data transmission.

3.1 System overview

The system consists of two main components:

Sender: A Linux computer running a Command Line Interface (CLI) for user input, signal encoding, and transmission with speakers.

Receiver: A TurtleBot3 robot equipped with a microphone and speakers, and a Raspberry Pi running Ubuntu[5] and Robot Operating System (ROS) Jazzy[4] for signal decoding, command interpretation, and motor control.

Communication is structured in five layers (see Fig. 1):

Application Layer: User interface and high-level command abstraction. The TurtleBot3 utilizes ROS2 with a velocity provider.

Command Abstraction Layer: Conversion of commands into binary and state management including the reverse process.

Reliability Layer: Error detection using CRC-4 and acknowledgment handling.

Signal Encoding/Decoding Layer: Conversion of binary data to/from DTMF frequencies, Fast Fourier Transform (FFT) and bandpass filters.

Physical Layer: Transmission and reception of DTMF signals via audio hardware.

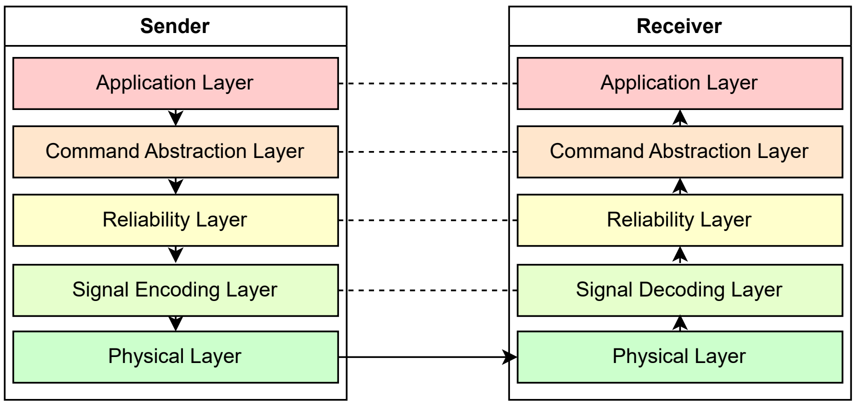


Figure 1: Overview of communication protocol layers showing communication from sender to receiver

3.2 Hardware and Software Setup

3.2.1 TurtleBot3 Platform

The robot platform uses a TurtleBot3 Burger[5] with a Raspberry Pi 3B+ as its computer. ROS handles the necessary program for receiving commands and acting upon them. For audio transmission and reception, a microphone and speaker are connected through a 3.5 mm audio jack. The ROS program uses only one ROS node as a publisher. This handles setting velocities from received commands and publishing them to the `cmd_vel` topic.

3.2.2 Communication Hardware

The communication hardware used by the user is a standard Linux computer. The program was developed using Linux Ubuntu 24.04. For the audio interface, the native speaker and microphone can be used. For development, however, external speakers were used.

3.3 Limitations

Range: Effective communication is limited to <10 meters in most environments.

Throughput: The protocol achieves 16–18 Bits per second (bps), suitable for simple data-transfer.

False Positives: Ambient noise can trigger false positives, though CRC mitigates this.

4 Method

This project uses methods and technologies that were curated through exploration of the different options and their suitability for the intended application. This approach led to a refined communication protocol that combines knowledge from different fields – including networking protocols and signal processing – to enable fast and reliable control of a mobile robot.

Git served as the primary version control system. Feature development was carried out on dedicated branches, which were merged into the main branch through pull requests once completed. This workflow supported parallel development, maintained a clear project history, and made it possible to revert to earlier versions if needed.

Weekly meetings were held to review progress and coordinate upcoming work. These meetings helped identify technical issues and ensured that tasks were clearly assigned and that all group members understood the broader system design. A kanban board was used throughout development to distribute tasks and track progress, providing a continuous overview of the project's state.

Work distribution followed a parallel structure in which individual members focused on specific subsystems, such as robot motion control or DTMF implementation. These contributions were developed in parallel to increase efficiency.

The communication protocol is based on DTMF tones. This allowed for encoding of multiple bits into a single audio signal. Four bits were mapped to each frequency, and four frequencies were combined into a chord, resulting in sixteen bits transmitted per signal. Error detection was handled through CRC, allowing the system to identify and discard corrupted transmissions and improving overall reliability.

Robot movement was implemented using ROS, while data logging and test result storage were handled using Structured Query Language (SQL). The ROS setup provided a structured interface between the communication protocol and the robot control software, ensuring that incoming commands were translated reliably into motor actions. The SQL database enabled logging of test results, which were used to compare transmitted- and received data and identify deviations during development and testing.

5 Application Layer

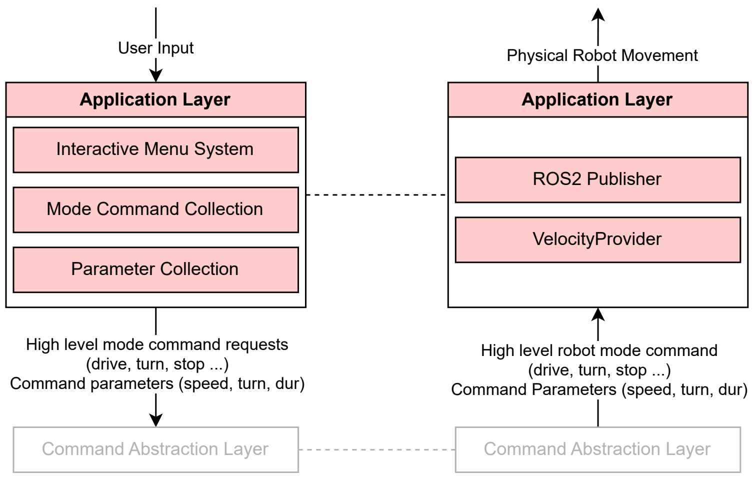


Figure 2: Overview of the Application Layer

In order for the user to interact with the system, a CLI was used. Displaying relevant options enabled the user to issue commands accordingly. Options include robot behavior such as linear or rotational movement, stop or reset commands and also a predefined test sequence, which were used during the testing of the robot.

ROS is used to publish control commands to the robot, allowing it to move. ROS is a modular software layer that helps bind together multiple programs through ROS-nodes. The ROS architecture consists of multiple independent ROS-nodes connecting into one big network called a ROS graph. ROS consists of several libraries, tools and communication techniques that make it easier for multiple programs to cooperate. This is especially useful for robotics due to its ability to bind together motor-drivers, signal processing, navigation-modules etc.

Data Distribution Service (DDS) is a middleware-protocol which ROS uses for its communication between its nodes. DDS handles data-exchange. Using DDS the nodes register/discover each other and communicate via a publisher/subscriber protocol.

ROS-nodes are individual executable programs that contain either subscriber, publisher, actions, services or parameters. Publishers send structured message types to a topic. The message could be a `TwistStamped` message that includes linear and angular x, y and z coordinates together with a header and a timestamp. Subscribers listen on a topic and reads the structured messages sent to that topic.[4] A topic is considered a data-pipeline. It accepts a certain type of message and connects publisher and subscriber.

For this project ROS was used by feeding the ROS-publisher with appropriate velocities depending on the signals from the receiver.

The ROS Client Library C++ (RCLCPP) object `publisher` is used to publish rotational and linear velocities to the Raspberry Pi in order to drive the robot. These values are floats. The publisher publishes linear velocities in the range -0.22 to 0.22, corresponding to full reverse or full forward. It also publishes the rotational velocities in the range -2.84 to 2.84, corresponding to full left or full right turn. When sending instructions with DTMF from the computer, it would be possible simply sending these two floats, then publishing these values to the topic. Instead, to have better resolution and faster transmission of commands, the `provider_` of the class `velocityProvider` is implemented. This object holds variables such as linear velocity, rotational velocity and duration, and provides linear and rotational velocities to the publisher in the correct ranges. Providing flexibility in the data we send. This object can take a variety of floating point inputs such as a duration, a linear velocity in the range of 0-100, and a rotational velocities in the range -100 to 100. Methods in the object then convert these values, making them compatible with the publisher.

Within the `provider_` object, a set of states are defined: `IDLE`, `DURATION` and `CONTINUOUS`. These states decide the flow of logic within the provider.

When operating in the `DURATION` state an, `end_time_` is calculated. This is used in combination with an `update()` method, that is called every cycle, to check if the duration parameter given has passed. If this is the case, the velocity attributes are set to zero, and the state is set to `IDLE`.

The following diagram shows the flow of logic being executed in the velocity provider every cycle of the ROS node.

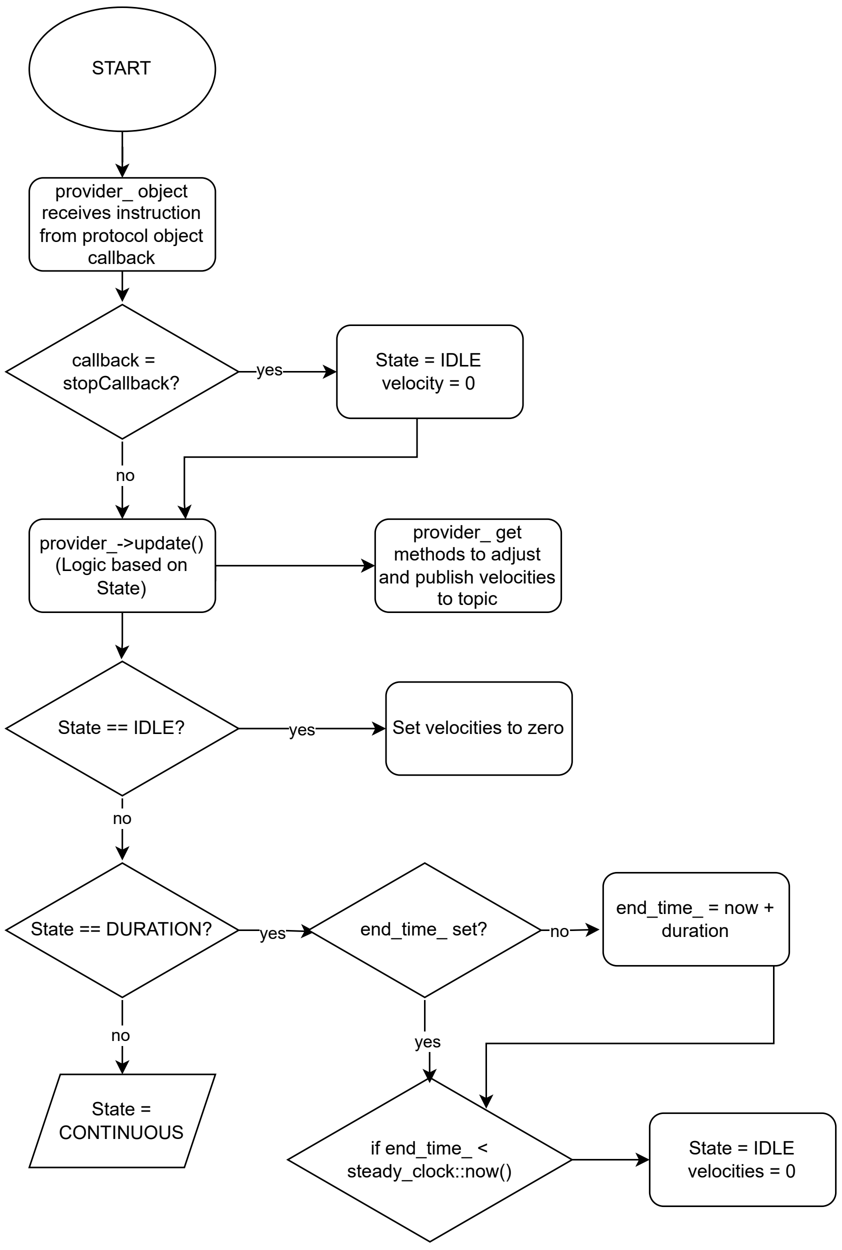

START
provider_ object receives instruction from protocol object callback
callback = stopCallback?
yes
State = IDLE velocity = 0
no
provider_->update() (Logic based on State)
provider_ get methods to adjust and publish velocities to topic
State == IDLE?
yes
Set velocities to zero
no
State == DURATION?
yes
end_time_ set?
no
end_time_ = now + duration
no
State = CONTINUOUS
yes
if end_time_ < steady_clock::now()
State = IDLE velocities = 0

Figure 3: Flowchart of VelocityProvider callback every publish cycle.

6 Command Abstraction Layer

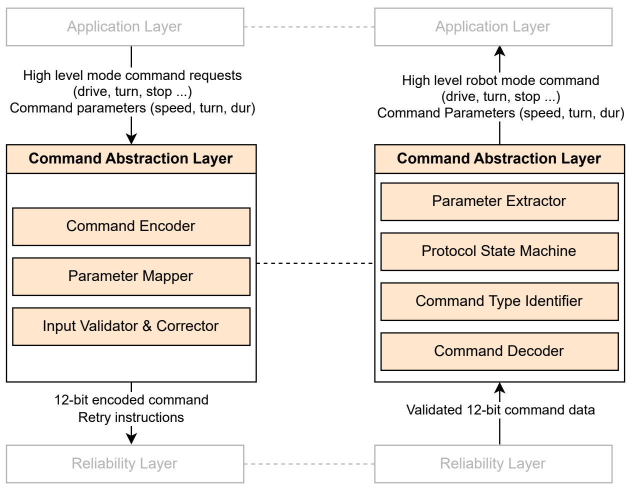


Figure 4: Overview of the Command Abstraction Layer

Right below the application layer is the command abstraction layer. The commands consists of different parameters based on the desired behavior of the system. The command abstraction layer includes the conversion from commands to binary data which essentially encodes the commands.

In order to send data effectively, the size of the data sent is reduced. Since the robot operates through the use of floats and states, an encoding method that works for both is implemented. The commands sent are limited to 12 bit, and since floats are larger datatype of 32 bits [2], reducing the precision and utilizing bit strings would enable fewer amount of commands necessary to operate the robot, thus decreasing transfer times.

The commands sent to the robot consists of either states or floats where the states can represent drive modes, and the floats represent the relevant parameters to operate within a drive mode.

The process of sending a command generally consists of two steps: first send the drive mode as a state, then send the appropriate values for the given state (linear velocity, rotational velocity and/or a duration) to define the behavior of the robot.

In order to operate the robot, six states are defined: `MODE_SELECT`, `DRIVE_FOR_DURATION`, `TURN_FOR_DURATION`, `DRIVE_FORWARD`, `TURN` and `STOP`. To transfer these modes to the robot they are mapped directly to 12-bit, binary strings, ranging from 0000 0000 0000 to 0000 0000 0110 (0-6 in decimal).

After the state is sent a command can now be processed, the different states have a variable amount of precision depending on how the 12 bits of information are split between the three possible parameters, the conversion from floats into binary strings varies for each of the driving modes.

An example of encoding parameters such as duration for the `DRIVE_FOR_DURATION` drive mode goes as follows: The duration, ranging from zero to eight seconds, is converted to a float of maximum 63.5, then cast into an Unsigned Integer 8-bit (uint8_t). The casting ensures the float is mapped to the lower integer bit value, which will at max be 00111111. This essentially creates a six bit integer. The same process is done for the linear velocity, though the computational logic is different based on whether the velocity is rotational or linear as the input for rotational velocity ranges from -100 to 100, and the linear velocity ranges from 0 to 100. Once the two bit strings for duration and linear velocity are computed, they are combined into a 12-bit binary string that then continues through to the next steps of our layered protocol.

Another example is the continuous `DRIVE_FORWARD` state, as it doesn't need the input parameter duration, thereby freeing six bits. This aspect is used to increase the resolution of the bit string representing speed from 6 to 12 bits.

On the receiving end this happens in reverse depending on the drive mode to convert from a 12-bit, binary string into the relevant states or floats depending on the drive mode.

Additionally, the binary string 0000 0000 0000 is used as a `RESET` signal, changing the state to `MODE_SELECT`. It is the only bit string that always gets treated the same regardless of the mode the robot is currently in.

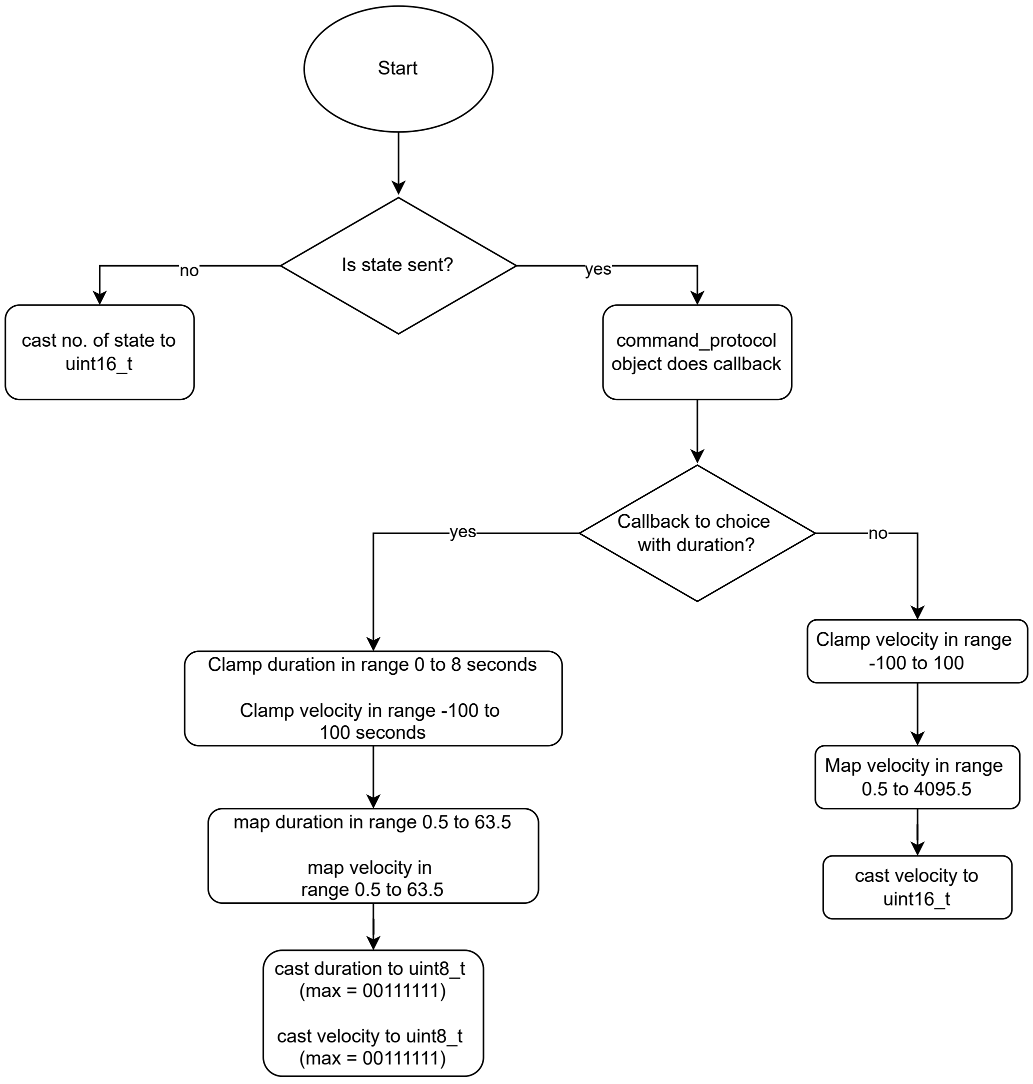

Start
Is state sent?
no
yes
cast no. of state to uint16_t
command_protocol object does callback
Callback to choice with duration?
yes
no
Clamp duration in range 0 to 8 seconds
Clamp velocity in range -100 to 100 seconds
map duration in range 0.5 to 63.5
map velocity in range 0.5 to 63.5
cast duration to uint8_t (max = 00111111)
cast velocity to uint8_t (max = 00111111)
Clamp velocity in range -100 to 100
Map velocity in range 0.5 to 4095.5
cast velocity to uint16_t

Figure 5: Flowchart showing Binary Conversion (example with rotational velocity)

7 Reliability Layer

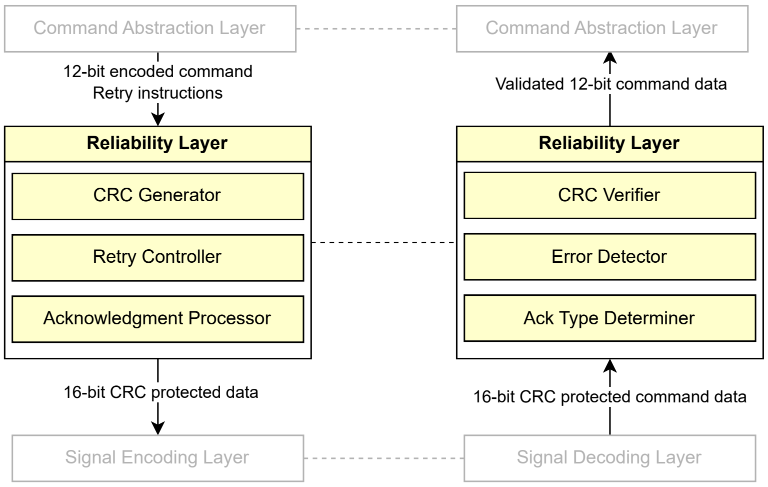


Figure 6: Overview of the Reliability Layer

In order to increase the reliability of transmitted and received binary commands, error handling is implemented in the reliability layer. These measures ensure that corrupt or incorrect information received by the receiver is rejected and requested again. Error handling can be performed in many ways. For this project one of the measures implemented has been CRC, which was chosen because of its robust and lightweight way of verifying data. The project uses CRC-4 on a 12-bit binary data string, meaning that the final transmitted binary string will be 16 bits long with the final 4 bits consisting of the CRC control bits. CRC works by a polynomial generator key represented by binary. So

$$1x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0 = x^4 + x + 1 = 10011$$

is the binary generator key used in computation. To make space for the CRC control-bits, (length of generator key) $-$ 1 = four 0's [3] are appended to the 12-bit data string making it 16 bits. Polynomial division is then performed using the generator key. In practice, this is performed using Exclusive OR (XOR) operations between the generator key and a section of the 16-bit string. For each XOR operation, the Most Significant Bit (MSB) is ignored from the result and the next bit from the data string is appended to the back. If the MSB is zero for the current XOR, 00000 is used instead of the generator key. An example of this can be seen in Fig. 7 below.

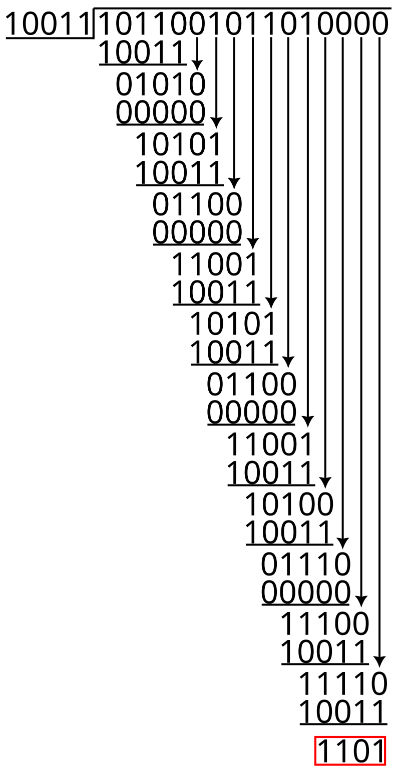


Figure 7: Example of CRC calculation

The resulting CRC control-bits are in the red square (1101). The four 0's appended before are replaced with these bits, resulting in the final CRC-4 data string (1011001011011101), which is ready to be sent. On the receiver side, the same polynomial division is performed on the received CRC-4 data string. If the result is 0, the received data is correct. If it is not 0, the received data is incorrect.[3]

As mentioned, to properly use CRC error detection, the data gets transmitted in 16-bit blocks whereof twelve bits are data bits, and four bits are the CRC control bits. On the sender side, the CRC control bits are calculated and appended to our 12-bit data string. When the receiver has received the now 16-bit data string, it performs the same calculation as mentioned above to check if the received data is intact. Both sender and receiver uses the same binary generator key (10011) to encode and decode the data. This is crucial as else the receiver would verify the received data incorrectly and thus reject otherwise intact data.

To implement this in practice a custom made CRC class has been made to accommodate these needs and ensure ease of use in our communication protocol.

To evaluate the reliability of of the CRC-4 class, a test was conducted in which bit patterns were systematically scrambled within a four-bit window. Since each window contains four bits, there are $2^4 = 16$ possible bit patterns in that window alone. All 16 configurations were tested to determine whether any of these alterations would produce detectable errors.

In this method, a known correct CRC-4 value is used as the reference. For each window position, all 16 possible patterns are substituted into that specific 4-bit segment while the remaining bits are left unchanged. This ensures that every local variation of the original CRC value is examined without altering the rest of the sequence. The procedure was applied to five distinct original CRC data values to assess whether the initial 12-bit data word influences the robustness of the CRC-4 algorithm.

Do note that the data used in these tests was generated and passed directly through the CRC class. Thus circumventing the rest of our system. This was done as these tests were performed to evaluate the effectiveness of the CRC system in isolation.

For each original CRC value, 208 modified binary sequences were tested; the unmodified original value was not included in the acceptance count. The combined results are shown in Tab. 1.

	1011001011011101		0000001011011110		0011110101010001		1111111100011111		0010000000001101	
	Value	Percentile	Value	Percentile	Value	Percentile	Value	Percentile	Value	Percentile
Accepted	13	6.25%	13	6.25%	13	6.25%	13	6.25%	13	6.25%
Rejected	195	93.75%	195	93.75%	195	93.75%	195	93.75%	195	93.75%

Table 1: Combined results from 4-bit window full scramble tests

The results indicate that each originally known binary value yields an identical error-detection rate: 93.75% of manipulated values are correctly rejected, while 6.25% pass through undetected.

To further combat failure in an unreliable environment, an acknowledgment from the receiver and a retry controller for the transmitter was also implemented as part of our reliability layer. Upon receiving a signal, the receiver has the option of transmitting a positive or negative acknowledgment. A positive acknowledgment is sent, if the data received successfully passes the CRC-checksum verification step. Otherwise a negative acknowledgment is sent.

On the senders side, the two types of acknowledgment, or a lack of one, is understood. If the sender does not receive an acknowledgment or receives a negative acknowledgment, the signal is retransmitted, as the sender understands that the receiver either did not receive the command, or that the command was corrupted. For this, the sender uses the retry controller which handles retransmissions as described. The retry controller only retransmits data once automatically, but if a confirmation is not received after the second time, the sender prompts the user to either continue, as if an acknowledgment has been received, or to retransmit the signal again. This prompt is repeated as long as the sender does not receive a positive acknowledgment or until the user chooses to continue without.

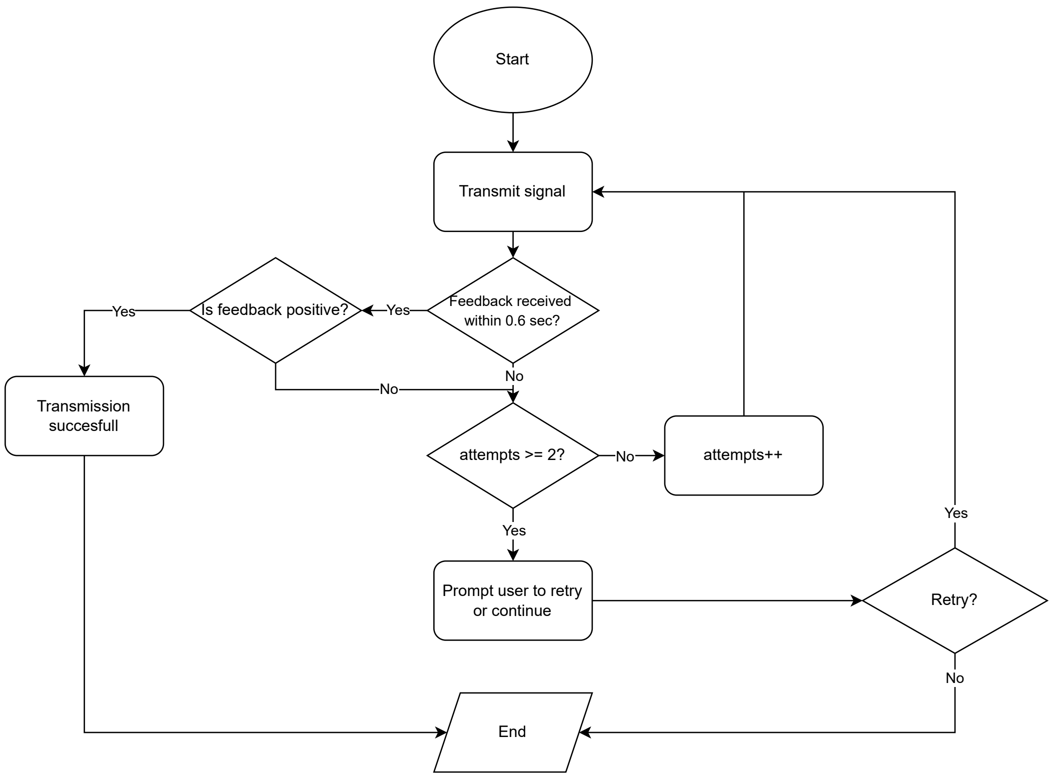


Figure 8: Flowchat showing retry algorithm for Reliability Layer

The user gets this option, as the sender may not have understood the acknowledgment from the receiver. The user is notified that continuing without positive acknowledgment from the receiver, could result in unexpected behavior.

8 Signal Encoding & Decoding Layer

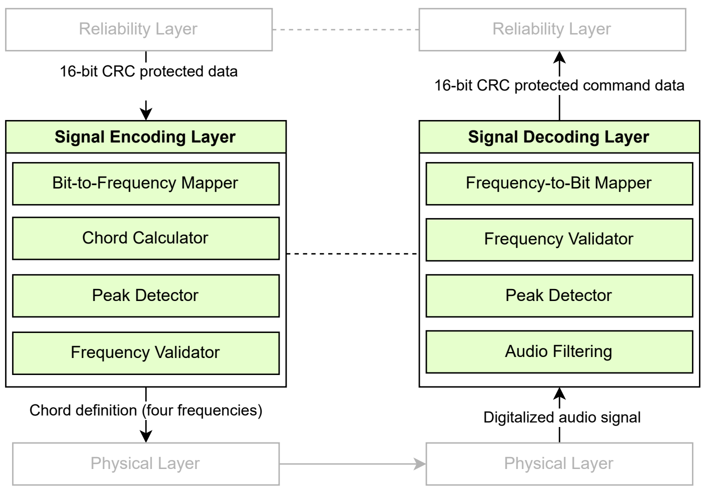


Figure 9: Overview of the Signal Encoding & Decoding Layer

Notice Frequency validator and Peak detector being on both sides in Fig. 9. This is due to the acknowledgment signals that the sender receives from the receiver.

In Sec. 6, the conversion from command to binary data was explained. In order to send information to the robot, the binary data is converted to frequencies. This enables sending and receiving the binary data with the use of a speaker and microphone.

In the project, a chord consisting of four frequencies is sent at an instance. These frequencies lie in different ranges: 4500 Hz to 7000 Hz, 7500 Hz to 10000 Hz, 10500 Hz to 13000 Hz, and 13500 Hz to 16000 Hz. The first three ranges contain the information being sent in a 12-bit format, and the fourth tone consists of the four bit control bits reserved for CRC error correction.

Conversion from binary data to frequencies consists of four iterations of calculations. These are similar in the process, but lie in different ranges. The binary data is processed in the format Unsigned Integer 16-bit (uint16_t), where it is split into four chunks of four bits, one for each iteration. The information is stored as `toneValue`, where 0000 implies the lowest frequency in the range, and 1111 (equal to 15) is the maximum value in the range. The frequency is then calculated by adding the minimum frequency for that range to the product of the `tonevalue` and frequency step size. In the implementation, the range of frequencies are 2500 hz. (e.g. 4500 to 7000 hz). The step size spaces this range into 16 equally sized chunks. This calculated frequency is then stored in a vector. This process repeats for all four sections, until a vector containing four frequencies is returned.

On the receiver side, conversion from tones to binary data is done in a similar fashion. A container of type uint16_t is created. The frequency vector is iterated four times over. A frequency is extracted, then the offset is calculated from the minimum frequency in that range to the received frequency. Then the frequency offset is divided with the frequency step size and the result is rounded to an integer. This integer is then inserted into the correct place in the container, and repeated for all the frequencies. Bellow in Fig. 10 are diagrams showing these processes.

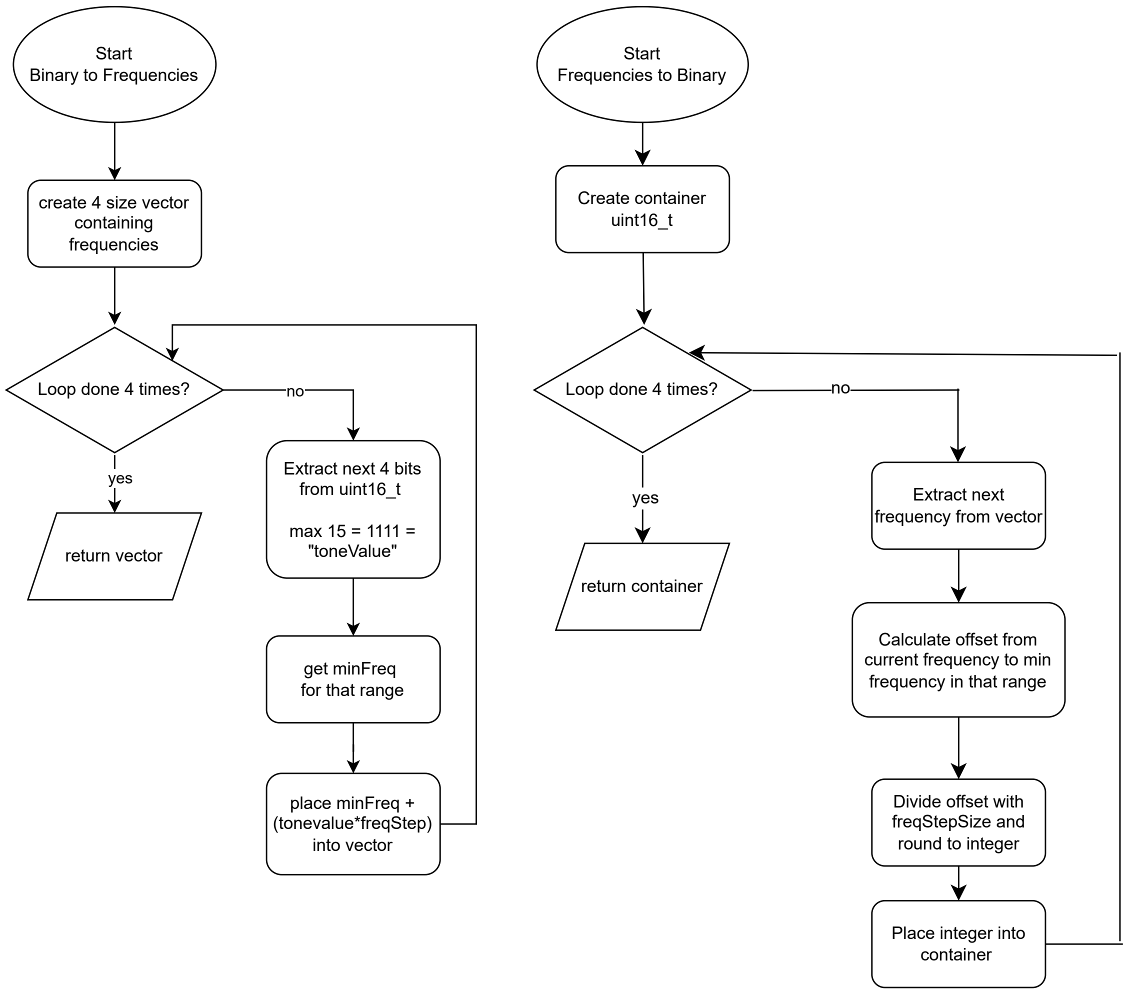

Start
Binary to Frequencies
create 4 size vector containing frequencies
Loop done 4 times?
no
yes
return vector
Extract next 4 bits from uint16_t
max 15 = 1111 = "toneValue"
get minFreq for that range
place minFreq + (tonevalue*freqStep) into vector
Start
Frequencies to Binary
Create container uint16_t
Loop done 4 times?
no
yes
return container
Extract next frequency from vector
Calculate offset from current frequency to min frequency in that range
Divide offset with freqStepSize and round to integer
Place integer into container

Figure 10: Binary to frequency and frequency to binary

In order to identify the relevant tones from the audio received by the microphone, we extract the dominant frequency components using FFT. FFT is an algorithm for computing the Discrete Fourier Transform (DFT) of a digital signal. A Fourier transform converts a signal from the time domain into its frequency-domain representation, allowing the signal to be expressed as a weighted sum of sinusoidal components. Direct computation of the DFT has a computational complexity of $O(n^2)$, which becomes inefficient for large input sizes. The FFT reduces this complexity to $O(n \log n)$ by exploiting symmetries and recursively decomposing the DFT into smaller subproblems, thereby enabling significantly faster analysis of high-resolution signals [7]. In the project, the number of samples every calculation (n), was 4096. Using DFT, this would result in the complexity $O(4096^2) = 16\,777\,216$ while the complexity of FFT would become $O(4096 \log 4096) \approx 14\,796$. Using FFT in this project then reduces the number of computations by more than a hundred times.

Through testing, it was shown, that FFT was performed ∼20 times per second meaning we had a sampling rate of ∼20 Hz. This proved useful as it allowed for multiple detections of the same signal within a time frame, allowing for more advanced signal processing. Because of the quick sampling rate the receiver requires two or more detections of the same signal within a 0.3 second time frame, to pass any detected tones as being a valid signal. Once recognized as a transmitted signal, duplicate signals within a 0.8 second time window is ignored as a lockout period is engaged. It is worth noting, new commands, being detected within this time frame, do pass through to the next layers.

Once the detected frequencies are transformed into the frequency domain, peak detection is applied, by taking the frequencies with the highest amplitudes. If these frequencies corresponds to one of the recognized commands defined previously, according to our described chord, they are accepted as a detected signals. Employing the FFT ensures that these tones can be detected reliably and consistently, enabling stable data transmission between the speakers and the robot. For this step the sender listens to the 5 loudest frequencies whereas the receiver listens to the 10 loudest frequencies. This was chosen, as the acknowledgment signal from the receiver only uses two frequencies, whereas the command signal from the sender uses four frequencies. This system of listening in a wider frequency band allows for more deviation in signal strength.

To suppress irrelevant or spurious frequency components, a band-pass filter is applied in combination with the FFT. A band-pass filter allows frequencies within a specified passband to be retained while attenuating frequencies outside this interval. A margin is typically included at both boundaries of the passband to account for slight deviations in the transmitted signal or imperfections in the filter response [6].

The receiving device (the TurtleBot) is equipped with a band-pass filter that passes frequencies between 4000 Hz and 17000 Hz. The transmitting device, in contrast, employs a narrower band-pass filter for acknowledge signal detection that admits frequencies in the range 1500 Hz to 4000 Hz. The upper bound of the transmitter's passband overlaps with the lower bound of the receiver's passband. This configuration is intentional as the receiver's acknowledgment signal occupies the interval 2000 Hz to 3500 Hz, placing it outside the command frequency range and preventing interference between command tones and acknowledgment responses.

In addition to the band-pass filter a noise gate was implemented. The noise gate filters the FFT results to exclude frequencies under a set amplitude. This will in theory result in less ambient noise being included in the signal, making the peak detection more reliable. The noise gate was tuned individually for the sender and receiver as it has shown to be dependent on specific hardware setups.

9 Physical Layer

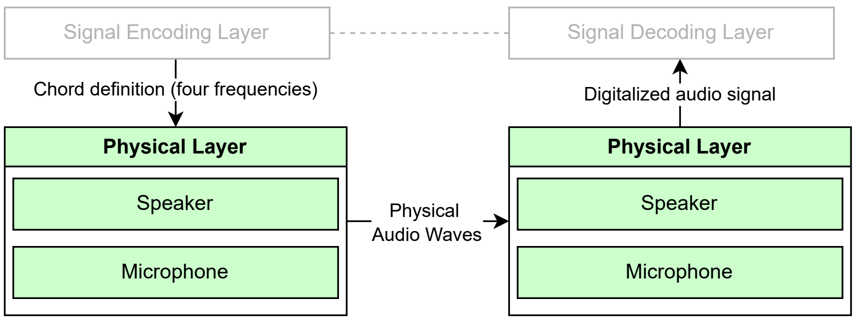


Figure 11: Overview of the Physical Layer

Early in the project it was determined that sending information as a single tone would work, but would eventually be limited in the amount of data sent in an instance. Sending multiple frequencies at a time would mean more information in that time frame, increasing speed of communication. To implement this, the concept of DTMF was utilized. DTMF was originally developed for use in telephone systems to indicate user button presses. Each button would have two frequencies. The combination of these two frequencies is unique for each button, and the system was designed to allow for sixteen unique combinations. When a user presses a button, the DTMF signal is sent over the voice signal, and the receiving system then decodes the signal to allow for the users intent. An example of such telephone DTMF systems can be seen in Tab. 2. [10]

Digit	Value	Percentile
1	697 Hz	1209 Hz
2	697 Hz	1336 Hz
3	697 Hz	1477 Hz
4	770 Hz	1209 Hz
5	770 Hz	1336 Hz
...	...	...

Table 2: Example of DTMF signaling used in telecommunication [10]

This system provides a base-line for communications between systems, and is still used in some industries (including military phones and company telephones).[10] This project expands upon this concept, and combines four different frequencies for each signal that is sent. Thus more data can be delivered in a shorter time-span allowing for faster communication.

From the frequencies calculated in Sec. 8 a chord is constructed, ready for transmission. The chord is transmitted through connected speakers for 0.8 seconds allowing the receiver to detect the signal. Once detected the receiver responds with either a positive or negative acknowledgment. As this is a more simple process than transmitting 16 bits, the receiver utilizes DTMF with only two frequencies instead of four. A positive acknowledgment consist of a 2500 Hz frequency and a 3500 Hz frequency. A negative acknowledgment consists of a 2000 Hz frequency and a 3000 Hz frequency.

10 Testing of the System

To evaluate the system as a whole, the following criteria were set up:

Successful transmission rate: The rate of sent transmissions that are successfully received and acted upon by the robot.

False positive rate: The rate of transmissions that are misinterpreted by the robot, as well as the amount of received transmissions, that are not actually sent.

Error catch rate: The rate that faulty transmissions are successfully discarded by the Reliability Layer.

Noise pollution resistance

Information throughput rate: The rate of information being sent within a given time frame, in our case bps.

Transmission speed/response time: The time between a command being sent and a confirmation tone being received.

Range of communication: The range at which our communication protocol can reliably transmit information.

Three tests were designed in order to evaluate these criteria.

The datasets below are the results of the robot receiving encoded audio commands from the sender. These encoded commands remained consistent between all the tests and were as follows:

- 5 reset commands
- 5 drive for duration commands
- 5 turn for duration commands
- 3 drive continuous commands
- 3 turn continuous commands

All the drive commands were preceded by a reset and mode select command and the continuous commands were also followed by a reset and stop command. Each drive command is followed by the appropriate parameters. This results in a total of 65 encoded audio commands being transmitted during a test, sometimes more due to retransmissions. Both the sender and receiver had their speakers set at maximum volume.

The tests measures which commands were sent, how many were correctly received, how often false positives occurred, and how quickly the robot responded. It also shows the rate at which the receiver successfully reports a positive confirmation. Alongside this, response time and latency tables show how long the robot takes to acknowledge and act on each signal. Together these results paint an extensive picture of the reliability of our communication protocol under the relevant conditions.

10.1 General transmission test

This test focuses on the general aspects of our communication under ideal conditions, and evaluates the criteria explained in the start of Sec. 10.

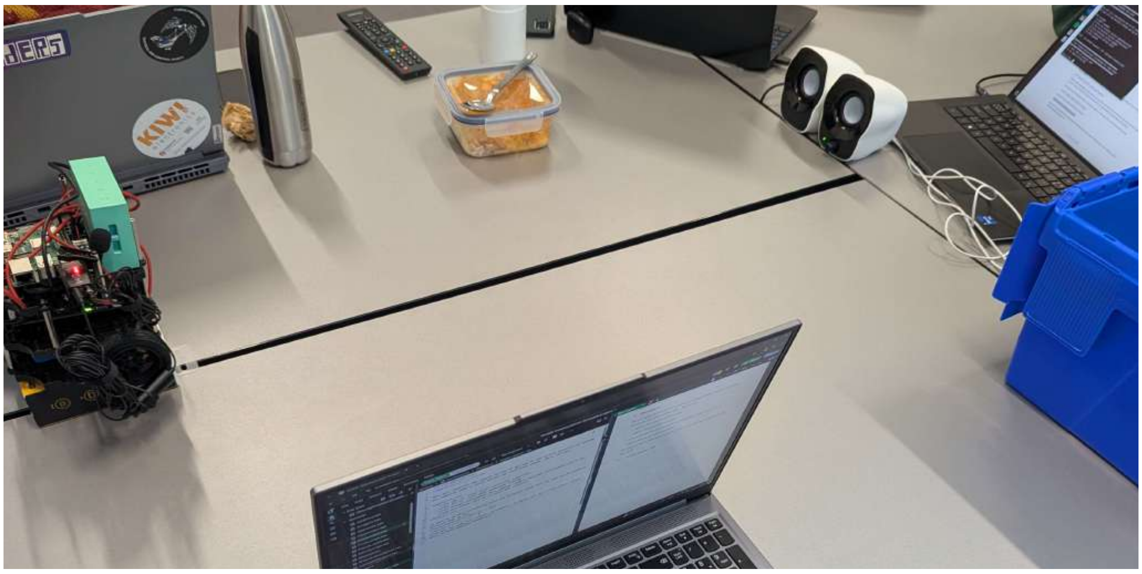


Figure 12: Setup for the General test, the receiver (the robot) is seen on the left and the sender (the computer) on the right

For the first test the robot and the speakers were placed one meter from each other and ran our test program. The following are the results of the test.

Test Results – general			
Total transmission sent	65	Successful transmissions	65
Total transmission received	65	Success rate	100%
Successfully matched	65	Sender confirmed rate	100%
Dropped	0	CRC accepted	65
False positive	0	CRC rejected	0
False positive received	0.00%	Negative confirmation sent	0
		Error rate	N/A

Response Time	**Value**	**Latency**	**Value**
Average response time	887.26 ms	Average latency	198.83 ms
Min response time	874 ms	Min latency	139 ms
Max response time	1424 ms	Max latency	758 ms

This test was done 12/12 - 2025. Link to test video (https://youtu.be/5JsGvIqe7jg)

A total of 65 transmissions were sent, all of which were received and matched with the sender. No packets were dropped, and no false positives were detected. Every transmission passed the CRC check correctly, as there was no corruption observed or any false positives detected. The sender confirmation rate was also 100%.

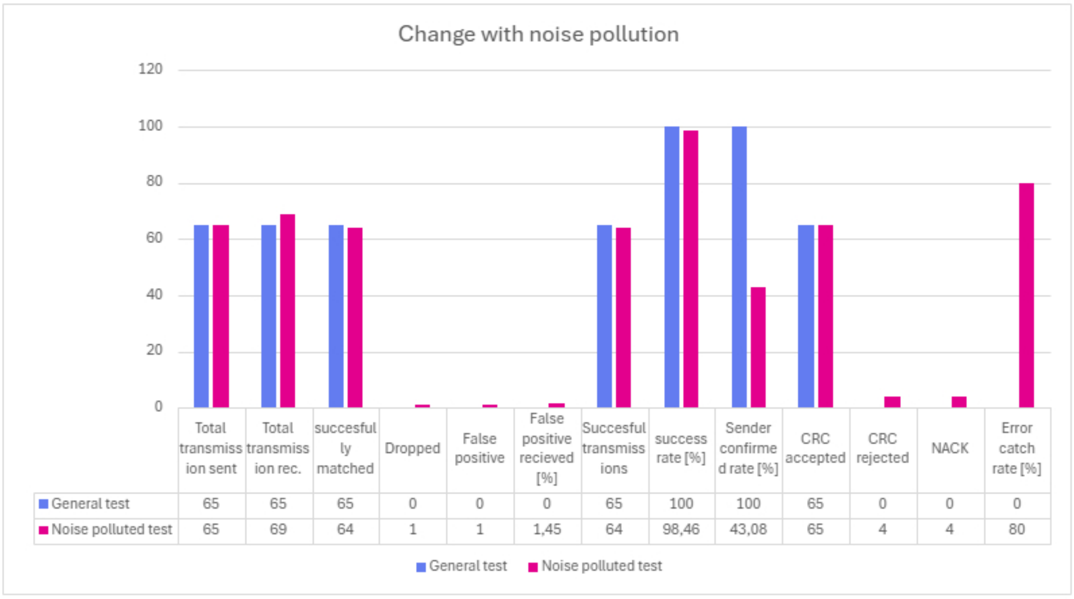
The average response time observed was 887.26 ms, with a minimum of 874 ms and a maximum of 1424 ms. The average latency was 198.83 ms, ranging from 139 ms to 758 ms.

Latency in this project is defined as the time elapsed between a transmission being emitted by the sender and the moment it is successfully detected and processed by the receiver. This metric therefore reflects not only the physical propagation delay of the signal but also the internal processing time required for signal detection, decoding, and validation.

Response time in this project is defined as the total elapsed time from when a transmission is initiated by the sender until the system completes its corresponding response. This metric represents the end-to-end behavior of the system and therefore includes both communication delays and internal processing time.

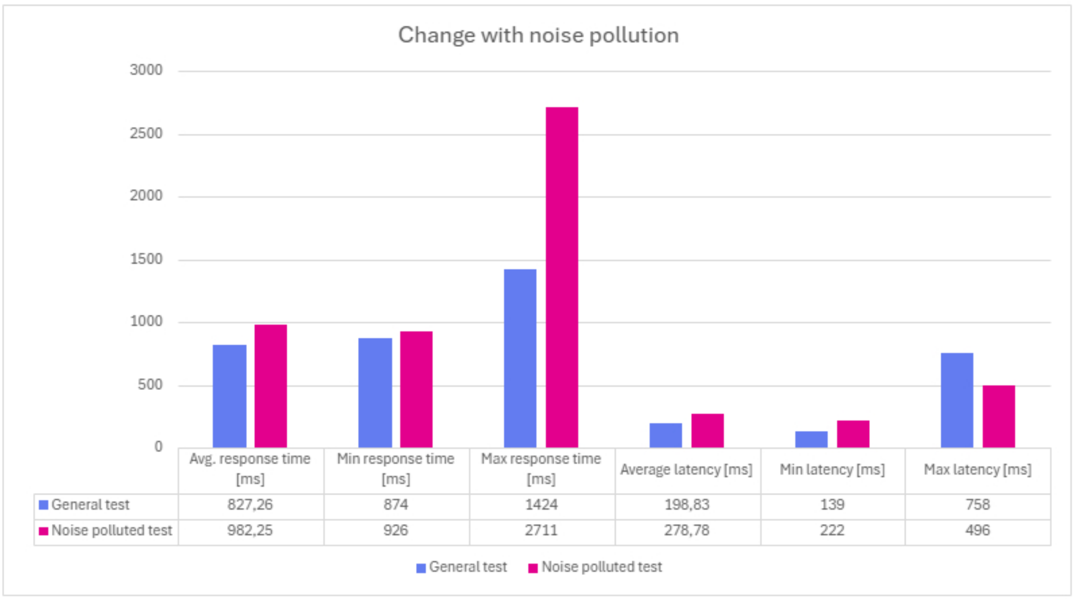
10.2 Transmissions under noise pollutions

This test examines how well the system withstands noise pollution. The setup was identical to the previous test (Fig. 12), but continuous white noise was played through the sender's speakers while the commands were transmitted. The goal of this test is to evaluate the system's overall resilience to noisy environments and its ability to maintain reliable communication under acoustic interference.



	Total transmission sent	Total transmission rec.	succesfully matched	Dropped	False positive	False positive recieved [%]	Succesful transmissions	success rate [%]	Sender confirmed rate [%]	CRC accepted	CRC rejected	NACK	Error catch rate [%]
🟦 General test	65	65	65	0	0	0	65	100	100	65	0	0	0
🟥 Noise polluted test	65	69	64	1	1	1,45	64	98,46	43,08	65	4	4	80

Figure 13: Difference between the general test and the noise pollution test



	Avg. response time [ms]	Min response time [ms]	Max response time [ms]	Average latency [ms]	Min latency [ms]	Max latency [ms]
🟦 General test	827,26	874	1424	198,83	139	758
🟥 Noise polluted test	982,25	926	2711	278,78	222	496

Figure 14: Latency and response difference for noise test

This test was done 12/12 - 2025. Link to test video (https://youtu.be/Ie0afMn7j5w)

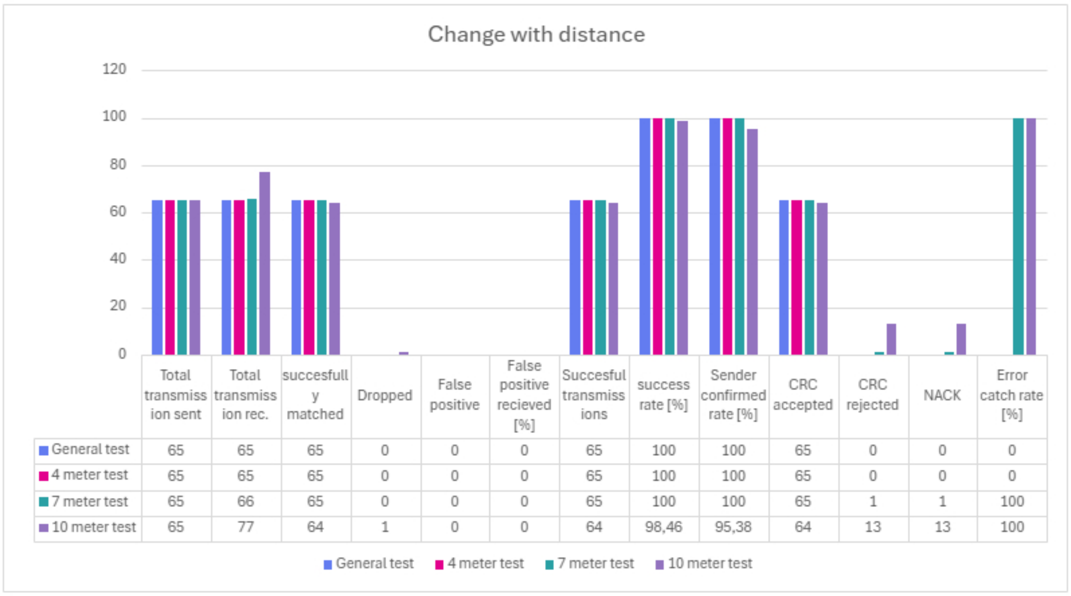
A total of 65 transmissions were sent, with 64 successfully received and matched. This yields a success rate of 98.46%. One transmission was dropped, and the system experienced one false positive, corresponding to 1.54% of the transmissions.

The sender confirmation rate dropped significantly to 43.08%, suggesting that nearly half of the received transmissions were not confirmed by the sender. CRC checks passed for 65 transmissions, but 4 rejections were recorded, and the error catch rate was 80%. This drop in error catch rate stems from the one false positive received, as that was not caught by the reliability layer.

The average response time was increased to 982.85 ms, with a minimum of 926 ms and a maximum of 2711 ms. The average latency was 278.78 ms, ranging from 222 ms to 496 ms. Compared to noise-free conditions, both response time and latency are higher, and the maximum response time exhibits a substantial spike.

10.3 Transmissions at range

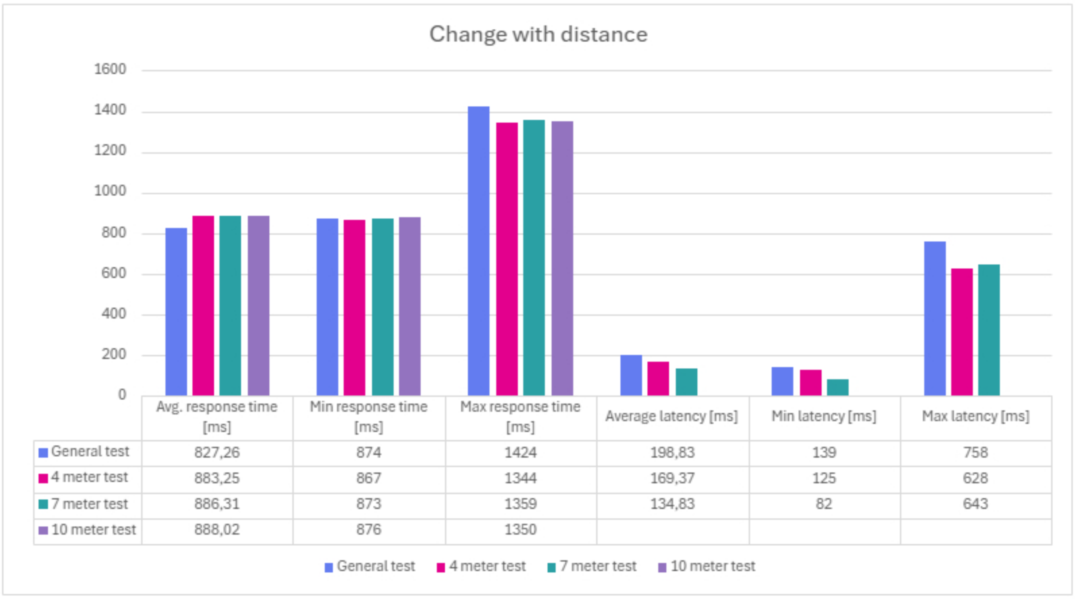
These tests evaluate the Range of Communication criteria by measuring how reliably the robot can interpret transmitted commands at different distances. The experimental setup remained the same as in earlier tests (Fig. 12), with the only variable being the distance between the speakers and the robot. Three distances were tested: 4 meters, 7 meters, and 10 meters, allowing observation of how well the communication protocol worked at range.



	Total transmission sent	Total transmission rec.	succesfully matched	Dropped	False positive	False positive recieved [%]	Succesful transmissions	success rate [%]	Sender confirmed rate [%]	CRC accepted	CRC rejected	NACK	Error catch rate [%]
🟦 General test	65	65	65	0	0	0	65	100	100	65	0	0	0
🟥 4 meter test	65	65	65	0	0	0	65	100	100	65	0	0	0
🟩 7 meter test	65	66	65	0	0	0	65	100	100	65	1	1	100
🟪 10 meter test	65	77	64	1	0	0	64	98,46	95,38	64	13	13	100

🟦 General test 🟥 4 meter test 🟩 7 meter test 🟪 10 meter test

Figure 15: Difference between the general test and the different distance tests



	Avg. response time [ms]	Min response time [ms]	Max response time [ms]	Average latency [ms]	Min latency [ms]	Max latency [ms]
🟦 General test	827,26	874	1424	198,83	139	758
🟥 4 meter test	883,25	867	1344	169,37	125	628
🟩 7 meter test	886,31	873	1359	134,83	82	643
🟪 10 meter test	888,02	876	1350			

Figure 16: Latency and response difference for distance test

This test was done 12/12 - 2025. Link to test video (https://youtu.be/qescPLRHLoc)

At four meters, all 65 transmissions were successfully received and correctly matched, with no dropped packets, CRC rejections, or false positives. This resulted in a sender confirmation rate of 100%. At seven meters, all 65 transmissions were again received successfully, with zero false positives observed. The sender confirmation rate remained at 100%, suggesting that overall reliability was largely unaffected at this range.

At ten meters, system performance began to degrade. Out of 65 transmissions, 64 were successfully matched, yielding a success rate of 98.46%. The number of false positives remained at zero, and CRC rejections became more frequent. Additionally, the sender confirmation rate dropped to 95.38%.

Response times remained relatively consistent across all distances, with average values around 883–888 ms. Latency also remained relatively the same across all distances. While latency values at four and seven meters remained relatively stable.

Error detection performance was optimal at four meters, with no false positives or CRC rejections recorded, and only minor errors were observed at seven meters. At ten meters, CRC rejections increased noticeably, while the error catch rate remained at 100% where errors occurred.

11 Discussion

11.1 Project management

The project was primarily managed using Git and GitHub, as these tools enabled efficient version control and parallel development across multiple features. The use of branching and shared repositories made it possible for several group members to work simultaneously without interfering with each other's progress. In addition, GitHub's built-in Kanban board was used for task management, which helped ensure that all group work was centralized and visible to every member. This created a clear overview of ongoing tasks, responsibilities, and progress throughout most of the project.

Furthermore, the integrated GitHub Wiki was used in the early stages of the project as a shared knowledge base. This made it easier for group members to document and explain relevant theory and technical concepts to one another. However, as the project progressed and the focus shifted towards implementation and testing, the use of the Wiki gradually decreased. In hindsight, continued use of the Wiki could have improved documentation consistency.

The weekly meetings functioned as the primary coordination point for the group, where progress, technical challenges, and task distribution were discussed. These meetings were especially valuable during the development phase, as they helped reduce uncertainty about individual responsibilities and supported faster problem-solving through direct discussion. However, as the project progressed into the later testing and reporting phase, the structure of the meetings became less formal, and the associated logbook documentation was de-prioritized. This reduced the consistency of project tracking in the final phase and made it more difficult to retrospectively trace certain decisions and changes. In hindsight, maintaining a stricter meeting structure throughout the entire project could have improved both documentation quality and overall coordination.

Overall, the chosen project management approach proved effective in supporting collaboration and maintaining project structure. However, more consistent documentation in the final phase and extended use of shared knowledge tools could have further improved transparency and long-term maintainability.

11.2 Evaluation of Application & Command Abstraction Layer

When discussing how ROS should be implemented in our software, it was chosen to only use ROS-nodes for publishing linear and angular velocities. This meant that there would be only one node to put focus on. Additionally, keeping our software mainly ROS-free meant that testing the system would be much easier and the rest of the code could be run as normal cpp. The argument for making `velocityProvider` a node was the added robustness of decoupling the process between the motor control and the rest of the software. The decision to keep `velocityProvider`, as well as the rest of our software, as internal logic was therefore made because we valued keeping the software modular and clear over having the extra certainty of control over the motors.

The resolution of 6 and 12 bit information sent through the command abstraction layer was chosen for the project due to the increased transfer time that would follow a 32 bit information transfer. As previously mentioned, our floats containing appropriate velocity values would take up 32 bits, which meant the size needed to be decreased in order to put it down to 12 bits, with four other bits used for CRC bringing the amount of sent bits up to 16. The low bit count is significantly faster for transfer, but at the cost of precision and control in the resolution of parameters we can send to the robot. If later additions of functionalities would be implemented, the transfer of only 12 bits would quickly bottleneck. However, for this project 12 bits was perfect for the intended purpose and transfer speed outweighed a higher resolution.

11.3 Evaluation of Reliability Layer

Our reliability layer provided much needed error detection and noise rejection that, greatly improved the reliability of the communication protocol over its development. During our research, Hamming Codes were considered, instead of CRC, an error detection and correction method that uses parity checks to locate and correct single-bit errors. However, its effectiveness diminishes with multiple errors: it can detect but not correct even numbers of errors, and it may even introduce additional corruption for odd numbers of errors. For a 16-bit signal, Hamming Codes require five control bits, reducing the usable data to just 11 bits and limiting error detection to approximately 56% of cases [1].

As CRC can detect over 90.00% of errors as shown through the tests in Sec. 7, Tab. 1, and only uses four control bits for a 16 bit signal, this approach was chosen. This resulted in a fast and reliable error detection method, that fit the requirements.

The results in Sec. 7, Tab. 1 show a consistent error-detection rate of 93.75%. This follows directly from the structure of the CRC-4 scheme. The 12-bit data space contains $2^{12} = 4096$ possible values, while the CRC field contains only $2^4 = 16$ possible control-bit combinations. Consequently, each CRC value corresponds to $4096/16 = 256$ distinct 12-bit data patterns. Any manipulated input that falls within the same group of 256 values will produce the same CRC output as the original and therefore cannot be detected.

The proportion of undetectable errors is thus

$$\frac{256}{4096} = 0.0625$$

which matches the observed 6.25% acceptance rate across all tests. The complementary proportion,

$$1 - 0.0625 = 0.9375$$

corresponds to the 93.75% rejection rate. Across all experiments, the error-detection rate remains stable at 93.75%, and it is reasonable to assume that this result generalizes to all 12-bit data values encoded using CRC-4. While testing showed a significantly smaller error catch rate at around 60-70% under challenging conditions. This seems to be caused by duplicate commands being received and processed, due to unreliable confirmation. These errors do not fall under what CRC is designed to do and thus are expected to fall through. These errors should be handled by other parts of the Reliability Layer, specifically the retry system has significant limitations under these conditions. The overall capabilities of the reliability layer fall well within the desired goals especially under ideal conditions.

As seen from the system tests in Sec. 10, the Reliability Layer rejected 100% of erroneous data received under these conditions, proving that CRC-4 along with acknowledgment signals provided the reliability needed for steady data-transfer.

11.4 Evaluation of Signal Encoding & Decoding Layer

In the signal encoding and decoding layer, mapping the combined 16 bit strings to frequencies using separation into four section and thereby producing four tones into a chord proved to be a simple but effective way of encoding and decoding the bit strings, partly because of the simplicity, but also because this separation enabled the mapping of information to a chord.

As mentioned, the range of frequencies sent from the sender was from 4500 to 16000 Hz, while the receiver sent information in the range 2000 to 3500 Hz. These frequencies are in the range of the human hearing range[9], which is practical in testing as it is easy to hear if the speakers are working and sending different chords. However, moving frequency ranges over 20000 Hz would result in a quieter system which theoretically would work the same, although it would need other hardware as the speakers and microphones are intended for the human hearing range. Using two pass-band filters to separate the receiver frequencies from the sender frequencies ensured non-overlap between the two.

When receiving the chords, FFT was used to process the tones, decreasing computational complexity by over a thousand times compared to DFT. Using FFT in a frequency of 20 Hz adds 50 ms delay to system, which is not a lot compared to the delay of the system in general.

After passing the received chords through the FFT calculations, peak detection ensured the collection of the frequencies with the highest magnitudes, effectively processing the signals to a compatible format with the decoding procedure. A noise gate ensures the received frequencies has a minimum magnitude, thereby rejecting potential noise in the same frequency ranges as the sent information.

Together, these processes enable fast and reliable conversion between binary data and audio frequencies, ensuring robust communication between the sender and receiver.

11.5 Evaluation of the physical layer

Sending information with sound could have been done in other ways than the use of the concept DTMF. An example is Frequency-shift keying (FSK), where a single frequency is transmitted at a time, and information is encoded by modulating the signal between two or more predefined frequencies, essentially sending information in serial[8]. While this would reduce the complexity of code needed for encoding, decoding and signal processing, it was not the goal of the project. Instead, sending information in parallel worked well with the implementation of the communication protocol. It allowed for more generalized packets that could be processed cleanly in isolation. This enabled relatively complex commands to be issued through singular, easily identifiable chords, that where more resilient to noise.

Using four tones instead of just two improved reliability greatly, especially under noisy conditions. This is apparent when looking at the test under noisy conditions. Here the primary failure point was the confirmation tone which were exclusively dual tone. This made them significantly more susceptible to noise thus leading to false positives.

11.6 Tests

Throughout testing, the system showed great reliability under various conditions. At seven meters distance, the system reliably communicates back and fourth, while continuing to work at 10 meters using one-way communication. Generally the confirmation tones seems to be the limiting factor as its the first part of the protocol that begins to fail at range or under noisy conditions where one-way communication still seems to be performing reliably. The latency and response times tended to hover around similar values all throughout testing with occasional spikes and increases when the confirmation tone system failed. This indicates that performance was consistently high throughout all testing and performance drops did not lead to significant errors. The limiting factor surrounding speed seems to stem from hardware limitations surrounding how fast the speakers and microphone send and respond to sound although it could possibly be lowered through more filtering before peak detection.

Overall these results are well above our expectations and goals.

For further improvements it could be interesting to look at a potential one way communication mode and performing further tests under increasingly challenging conditions to see when such a mode would fail. An alternative option would be improving the reliability of the confirmation tones, through the use of a full four tone chord or using other frequencies, that might be less susceptible to noise. Both approaches would require significant redesigns as to avoid interference between sender and receiver.

12 Conclusion

This project aimed to demonstrate the feasibility of controlling a mobile robot using DTMF signals, extending the traditional two-tone approach to a four-tone system. By leveraging a layered communication protocol, the project successfully established a robust framework for transmitting commands via audio signals, thereby enabling wireless control of a ROS-based mobile platform.

The system architecture was designed with modularity and clarity in mind, incorporating distinct layers for command abstraction, reliability, signal encoding/decoding, and physical transmission. User commands were abstracted into states and parameters, which were subsequently encoded into binary strings to reduce complexity and optimize transmission efficiency. To ensure data integrity, a CRC-4 error detection mechanism was implemented, enabling the system to identify and discard corrupted transmissions effectively.

On the receiver side, audio signals were captured via a microphone and processed using a combination of band-pass filtering and FFT-based peak detection. This approach facilitated the accurate extraction of transmitted frequencies, which were then decoded into their original command format. The layered architecture not only simplified the implementation of these processes but also ensured that each stage of the communication pipeline could be independently optimized and validated.

Reliability was rigorously evaluated using SQL for data logging. Experimental results showed a success rate of 98.46%–100%, with the reliability layer effectively rejecting all corrupted transmissions. While sender confirmation rates occasionally dropped to 43.08% and error catch rates dropped to 80% in one test, the system maintained high overall reliability. Response times averaged 883.25–982.85 ms, with occasional spikes up to 2711 ms, reflecting the inherent latency of audio-based communication.

In summary, this project successfully validated the use of DTMF signals for mobile robot control, demonstrating a reliable and efficient communication protocol. While the system exhibited occasional latency spikes, its high success rate and error resilience confirm its potential for applications requiring wireless, audio-based command transmission.

13 Future work

This project demonstrated the feasibility of using DTMF to control a TurtleBot3 mobile robot. However, the developed communication protocol has broader applications and opportunities for enhancement. The use of DTMF could extend to industrial automation where wireless communication is necessary but radio frequency interference is problematic. It may also find applications in short distance data transfer, as the Reliability Layer proved a stable method of verifying transferred data.

The layered architecture of the protocol enhances its versatility. Each layer, including signal encoding, reliability and command abstraction, can be independently adapted to suit different requirements. This modularity allows for integration with platforms beyond ROS, such as embedded systems and microcontroller based applications. The current CRC based reliability layer could be further improved by incorporating advanced error correction techniques to enhance robustness in noisy environments.

Several improvements could refine the system further. Optimizing the FFT processing pipeline or implementing real time signal processing could reduce latency. Machine learning based noise suppression techniques might enhance performance in challenging acoustic environments. Reducing the computational overhead would also enable deployment on low power devices, broadening the protocol's applicability.

The principles of this protocol could be adapted for general purpose audio based communication. For example, combining DTMF with encryption could create secure communication channels for sensitive applications. Its simplicity also makes it suitable for educational tools in digital communication and signal processing. In emergency scenarios where traditional communication infrastructure is unavailable, DTMF could serve as a reliable fallback for transmitting critical commands or alerts.

In conclusion, while this project focused on robot control, the underlying architecture offers significant potential for broader applications.

14 References

[1] 3Blue1Brown. *But what are Hamming codes? The origin of error correction.* Accessed: 09/12-2025. URL: https://youtu.be/X8jsijhllIA?.

[2] Deduplicator. *What is the size of float and double in C and C++?* Accessed: 10/12-2025. URL: https://stackoverflow.com/questions/25524355/what-is-the-size-of-float-and-double-in-c-and-c.

[3] kartik. *Cyclic Redundancy Check and Modulo-2 Division.* Accessed: 12/11-2025. URL: https://www.geeksforgeeks.org/dsa/modulo-2-binary-division/.

[4] Open Robotics. *ROS 2 Documentation: Jazzy.* Accessed: 08/12-2025. URL: https://docs.ros.org/en/jazzy/index.html.

[5] Robotis. *TurtleBot3 e-Manual.* Accessed: 12/12-2025. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/.

[6] Unknown. *Band-pass filter.* Accessed: 11/12-2025. URL: https://en.wikipedia.org/wiki/Band-pass_filter.

[7] Unknown. *Fast Fourier Transform.* Accessed: 11/12-2025. URL: https://en.wikipedia.org/wiki/Fast_Fourier_transform.

[8] Unknown. *Frequency-shift keying.* Accessed: 16/12-2025. URL: https://en.wikipedia.org/wiki/Frequency-shift_keying.

[9] Unknown. *Hearing range.* Accessed: 18/12-2025. URL: https://en.wikipedia.org/wiki/Hearing_range.

[10] Gavin Wright. *What is DTMF (dual tone multi-frequency and how does it work?)* Accessed: 09/12-2025. URL: https://www.techtarget.com/searchnetworking/definition/DTMF.

A Results from System Tests

Test Results – noise polluted			
Total transmission sent	65	Successful transmissions	64
Total transmission received	69	Success rate	98.46%
Successfully matched	64	Sender confirmed rate	43.08%
Dropped	1	CRC accepted	65
False positive	1	CRC rejected	4
False positive rate	1.54%	Negative confirmation sent	4
		Error catch rate	80%

Response Time	**Value**	**Latency**	**Value**
Average response time	982.85 ms	Average latency	278.78 ms
Min response time	926 ms	Min latency	222 ms
Max response time	2711 ms	Max latency	496 ms

Test Results – 4 meters			
Total transmission sent	65	Successful transmissions	65
Total transmission received	65	Success rate	100%
Successfully matched	65	Sender confirmed rate	100%
Dropped	0	CRC accepted	65
False positive	0	CRC rejected	0
False positive rate	0.00%	Negative confirmation sent	0
		Error catch rate	N/A

Response Time	**Value**	**Latency**	**Value**
Average response time	883.25 ms	Average latency	169.37 ms
Min response time	867 ms	Min latency	125 ms
Max response time	1344 ms	Max latency	628 ms

Test Results – 7 meters			
Total transmission sent	65	Successful transmissions	65
Total transmission received	66	Success rate	100%
Successfully matched	65	Sender confirmed rate	100%
Dropped	0	CRC accepted	65
False positive	0	CRC rejected	1
False positive rate	0%	Negative confirmation sent	1
		Error catch rate	100%

Response Time	**Value**	**Latency**	**Value**
Average response time	886.31 ms	Average latency	134.83 ms
Min response time	873 ms	Min latency	82 ms
Max response time	1359 ms	Max latency	643 ms

Test Results – 10 meters			
Total transmission sent	65	Successful transmissions	64
Total transmission received	77	Success rate	98.46%
Successfully matched	64	Sender confirmed rate	95.38%
Dropped	1	CRC accepted	64
False positive	0	CRC rejected	13
False positive rate	0%	Negative confirmation sent	13
		Error catch rate	100%

Response Time	**Value**	**Latency**	**Value**
Average response time	888.02 ms	Average latency	N/A
Min response time	876 ms	Min latency	N/A
Max response time	1350 ms	Max latency	N/A