RaspiCar Documentation & User Manual
Version 1.1

Student: BSc Fynn Luca Maaß, f.l.maass@student.tugraz.at
Supervisors: Dipl.-Ing. Dr.techn. BSc MBA Georg Macher
BSc MSc Romana Blažević

Contents

August 2024

1 Introduction

This documentation and user manual presents an embedded system architecture to transform the RaspiCar into a versatile research platform. The source code can be accessed over a repository [1]. This architecture is efficient and feature-rich, including functionalities such as gamepad control, data recording, video live-streaming, and providing a framework to embed other control algorithms or neural networks. It is a node-based publish/subscribe architecture based on ZeroMQ [2].

The proposed software architecture is explained in 2. 3 explains the principles of using inter-process communication using ZeroMQ between different nodes in the system. 4 gives an overview of such Nodes with a brief explanation of their functionalities.

Finally, 5 gives a guidance on how to start and use the RaspiCar. Along with 6 to avoid some pitfalls and useful tips when using the RaspiCar.

1.1 Main Features of the RaspiCar

- Steering the vehicle demonstrator with gamepad controls.
- View the camera live-feed over the network.
- Easy deployment of self-implemented control algorithms as nodes to control the vehicle.
- Intervene the self-implemented control algorithms anytime with the connected gamepad.
- Record and save the camera feed and label images with current steering commands to create custom data sets.
- Handling errors by automatically entering emergency states if nodes terminate unexpectedly, the gamepad or camera disconnects or if the communication between the Raspberry Pi 3 and Raspberry Pi Pico fails.
- Automated scripts to upload source code to the Raspberry Pi 3 and Raspberry Pi Pico.
- Efficient architecture by utilizing multicore-processing, bypassing problems such as Python's Global Interpreter Lock [1]

2 System Architecture

The RaspiCar uses a Raspberry Pi 3 and a Raspberry Pi Pico. A component diagram of the hardware and software components is provided in Figure 1.

[1] https://realpython.com/python-gil/

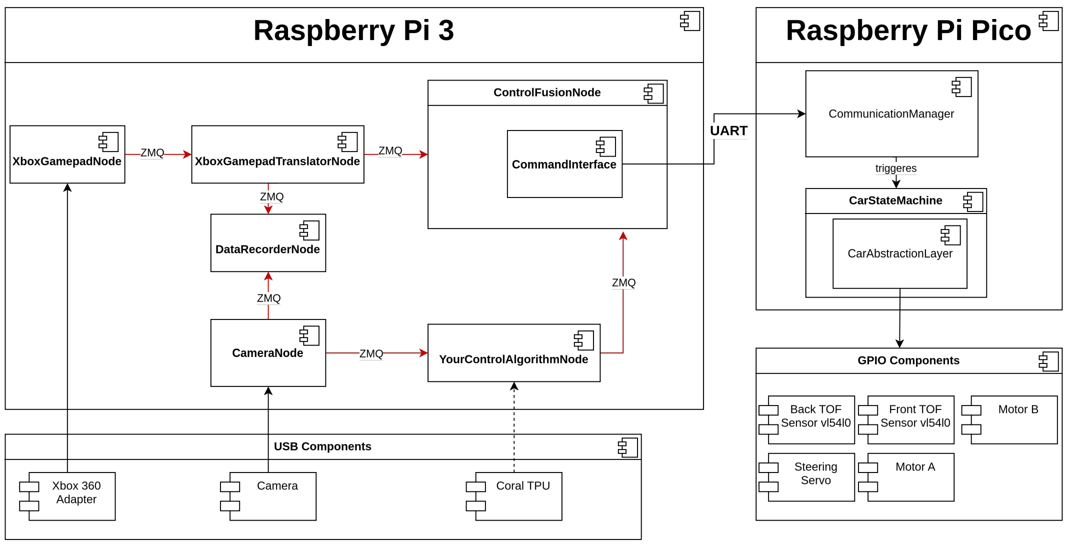


Figure 1: Component diagram of the RaspiCar. Nodes on Raspberry Pi 3 communication over network sockets using ZeroMQ (ZMQ). The communication between the Raspberry Pi 3 and the Raspberry Pi Pico is implemented using UART. The three components on the Raspberry Pi Pico are simple MicroPython classes.

2.1 Raspberry Pi 3

The software on the Raspberry Pi 3 is primarily organized in ZeroMQ nodes which can publish and subscribe data to other nodes using network sockets. Those nodes access USB devices, such as the *CameraNode* using the webcam or the *XboxGamepadNode* using the Xbox gamepad adapter. Both nodes publish messages, images and gamepad button states respectively, that are received by other nodes.

The *YourControlAlgorithmNode* resembles a node in the system that can be exchanged by any other node containing a control algorithm for the RaspiCar. A template is provided that can be filled with a custom implementation, which is described in section 5.3. For instance, *YourControlAlgorithmNode* can be replaced with the *PilotNetCNode*, which is also based on this template. The *PilotNetCNode* can be found in the repository and implements a neural network for autonomous steering.

ControlFusionNode is a node implementing logic to fuse controls from different sources. This node subscribes steering commands from a control algorithm node and the *GamepadCommandNode* to dynamically change between both steering inputs. *ControlFusionNode* further holds an instance of *CommandInterface*, which is a class dedicated for serial communication over UART. All steering commands are sent through the *CommandInterface* to the Raspberry Pi Pico. A heartbeat message is also exchanged to detect connection issues.

A detailed overview of all nodes describing their exact behaviour and possible configurations is provided in 4.

2.2 Raspberry Pi Pico

The Raspberry Pi Pico receives higher level steering commands such as throttle or steer over the UART interface and converts those messages into actual action on the hardware. The Raspberry Pi Pico controls two motors using PWM and H-Bridges, a steering servo using PWM and two TOF sensors using I2C. The software on the Raspberry Pi Pico is organized into three primary classes.

The *CommunicationManager* implements the receiving end of the UART interface to receive higher level steering commands from the Raspbbery Pi 3 as well as the heartbeat message. Received messages and missing heartbeats drive the *CarStateMachine*. The *CarStateMachine* holds an instance of *CarAbstractionLayer* and executes the actions from the state machine.

2.2.1 CarStateMachine

The *CarStateMachine* is illustrated in Figure 2. This state machine is implemented using the state pattern (software design pattern) and controls different states of the car and defines possible actions in those states. The state machine has four states in total. The onboard LED of the Raspberry Pi Pico is used to indicate different states.

The *IdleState* is the initial state when the Raspberry Pi Pico is powered on and remains in this state until a steering command is received. When a steering or throttling command is received, the state machine transitions to *DrivingState*, indicating that the Raspberry Pi Pico receives steering commands from the Raspberry Pi 3.

Two emergency states are implemented that will both stop the motors and disable the steering, but are entered and left with different triggers.

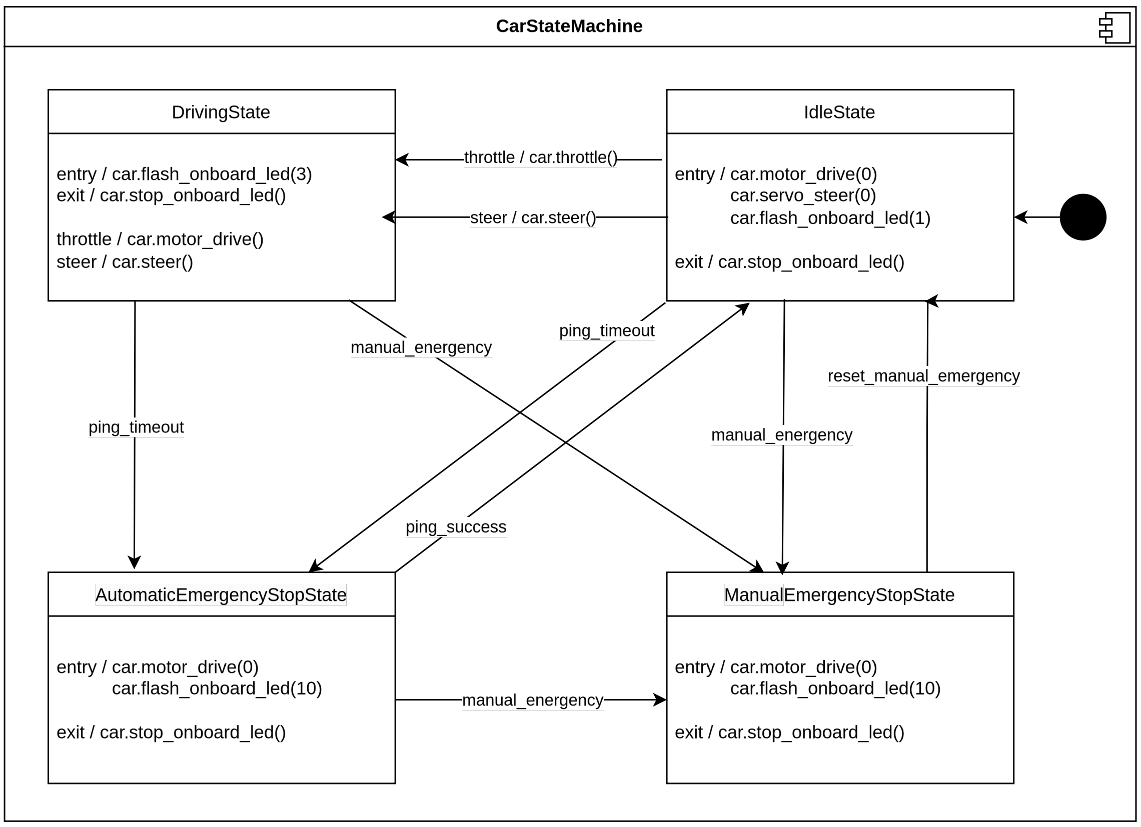
The *AutomaticEmergencyStopState* is entered by missing a heartbeat message from the Raspberry Pi 3. The heartbeat is sent every 100ms, the threshold for considering a miss is 200ms. This state indicates that the communication between Raspberry Pi 3 and Raspberry Pi Pico is interrupted or severely impacted. The state is automatically left, once the heartbeat resumes, hence the name *AutomaticEmergencyStopState*.

ManualEmergencyStopState is triggered manually over the gamepad, allowing to enter the emergency state manually on purpose. This state can then only be left by pressing a different button on the gamepad again. Further, this state will also be triggered when the Gamepad disconnects. In this case, the *XboxGamepadNode* running on the Raspberry Pi 3 will detect a disconnect and publishes this information, which is ultimately sent over UART as a *manual_emergency* and triggers the state machine. If the gamepad reconnects, the *ManualEmergencyStopState* must still be left manually by pressing the corresponding button on the gamepad.

If the state is already *ManualEmergencyStopState*, the state *AutomaticEmergencyStopState* can not be reached directly, even if the heartbeat is missed during that time.

However, *AutomaticEmergencyStopState* can transition to *ManualEmergencyStopState*. A reasonable scenario would be that the Raspberry Pi 3 lags behind due to heavy CPU usage, delaying all messages sent over UART. In this scenario, the delayed *manual_emergency* still triggers *ManualEmergencyStopState* eventually and is not lost. If the heartbeat should then arrive within it's 200ms window again, the car will not continue operation but requires a manual button press on the gamepad.

For future use-cases, this state machine can easily be extended to include triggers of the TOF sensors. Code skeletons are already provided in the state machine implementation for TOF measurements but not used yet.



CarStateMachine
DrivingState
entry / car.flash_onboard_led(3)
exit / car.stop_onboard_led()
throttle / car.motor_drive()
steer / car.steer()
IdleState
entry / car.motor_drive(0)
car.servo_steer(0)
car.flash_onboard_led(1)
exit / car.stop_onboard_led()
throttle / car.throttle()
steer / car.steer()
manual_energency
ping_timeout
ping_timeout
reset_manual_emergency
manual_energency
ping_success
AutomaticEmergencyStopState
entry / car.motor_drive(0)
car.flash_onboard_led(10)
exit / car.stop_onboard_led()
manual_energency
ManualEmergencyStopState
entry / car.motor_drive(0)
car.flash_onboard_led(10)
exit / car.stop_onboard_led()

Figure 2: Finite State Machine running on the Raspberry Pi Pico.

3 Inter-Process Communication with ZeroMQ

For the RaspiCar, Inter-Process Communication (IPC) between nodes on the Raspberry Pi 3 is achieved using the publish/subscribe pattern by ZeroMQ (ZMQ). ZMQ is a networking library that provides a simple API for socket communication and allows publishing or subscribing messages under given addresses and ports. Further, ZMQ allows filtering messages by topics. In ZeroMQ, topics are strings used to categorize messages. When a publisher sends a message, it prefixes the message with a topic. Subscribers can then filter the messages they receive by subscribing to one or more topics. Nodes can use the localhost network interface, which allows them to send and receive messages efficiently without the overhead of network latency. Alternatively, nodes can be easily distributed across the local network if required.

ZeroMQ only offers the socket API and communication patterns such as Pub/Sub, but does not define actual message contents or message types that will be sent.

3.0.1 Message Types

For the RaspiCar, we differentiate message into JSON messages and image messages which are exchanged using ZeroMQ.

- JSON messages: Transmit Python dictionaries between nodes. This messages are used to transmit steering commands from the node implementing a control algorithm to the *ControlFusionNode* which executes them. Or by the *XboxGamepadNode* to publish its raw inputs.
- Image messages: Transmit OpenCV images between nodes. This message is used to transmit camera images to the control algorithm or other nodes requiring camera images.

Utility functions provided in *utils.message_utils.py* ease the creation and parsing of these messages on the sender receiver side. This utility function will also prefix the messages with the specified topic.

3.0.2 Publishing and Subscribing JSON Messages

JSON messages contain structured data serialized in JSON format, including a timestamp and a payload (a dictionary of data). Below is a small code example on how to publish and subscribe some data by using the *create_json_message()* and *parse_json_message()* function from *utils.message_utils.py*. This function will create a string with a leading prefix as a ZMQ topic followed by an utf-8 encoded JSON string which includes the payload and an additional timestamp.

```
# Do the initial ZMQ setup and bind a publisher socket...

# Define a python dict with data.
# In this case steering_command message
payload = {
    "steer" : 0.75, #between -1 (max. left), 1 (max. right)
    "throttle" : 0, #between -1 (max. reverse), 1 (max. forward)
    # Define other values, see steering_command in code listings...
}
message = create_json_message(payload, topic="steering_commands")

# Send the JSON message
pubblisher_socket.send(message)
```

Listing 1: JSON Message Publishing Side Code Snippet

```
# Do the initial ZMQ setup and bind a subsciber socket...

# Receive the JSON message
message = subsciber_socket.recv_string()
topic, timestamp, payload = parse_json_message(message)
# now payload can be accessed as normal python dict again...
steer = payload["steer"]
throttle = payload["throttle"]
```

Listing 2: JSON Message Subscribing Side Code Snippet

For implementation details of the *create_json_message()* and *parse_json_message()* functions, please look into the source code in *utils.message_utils.py*. Further, the listing 6 and 7 provides a complete example including the ZMQ setup that is missing in the code snippet below. A list of all available payload-"python dictionaries", especially the payload that needs to be send to steer the car can be found in listing 10.

3.0.3 Publishing and Subscribing Image Messages

The following shows some simple example code of how to publish an image message using the *create_image_message()* and *parse_image_message()* from *utils.message_utils.py* and ZeroMQ. The code shows how to set up a publisher socket and bind it to an address and port. Further, it reads a camera image from a connected USB-webcam and converts it into an image message, which is then published under a given topic.

```
# Do the initial ZMQ setup and bind a publisher socket...

# Capture and send a single frame
_, image = cv2.VideoCapture(camera_source).read()
message = create_jpg_image_message(image, topic="camera_topic",
    quality=95)

# Send the image as ZMQ multipart message
publisher_socket.send_multipart(message)
```

Listing 3: Image Message Publishing Side Code

The following shows some example code of how to subscribe the image message.

```
# Do the initial ZMQ setup and bind a subsciber socket...

# Receive and display a single frame
message = subsciber_socket.recv_multipart()
topic, image, timestamp = parse_jpg_image_message(message)

# now image is a normal cv2 image type again...
cv2.imshow('Received Image', image)
```

Listing 4: Image Message Subscribing Side Code

4 Node Descriptions and Messages

A comprehensive overview of all nodes in the system is provided below. Unless necessary, it is recommended to keep their default parameters. A list of the different messages published (the python dictionaries) is provided in (Listing 10) in the appendix.

4.1 CameraNode

CameraNode

Captures video frames from a camera and publishes the frames to a ZeroMQ topic. Initializes the camera and captures frames at a specified rate across the network. Frame rates or resolution can be increased, but add higher latencies when published in across the whole network. If it is intended to significantly increase the resolution (up to 1080p possible), consider setting this node to only publish images to localhost. Then, the camera feed can not be live-streamed any more to a different.

Parameters (default values):

- *frame_rate* (10): Specifies the frame rate in frames per second
- *camera_source* (0): Specifies the camera index or source
- *frame_width* (640): Specifies the frame width
- *frame_height* (360): Specifies the frame height
- *zmq_pub_url* ("tcp://*:5550") Specifies where the video frames are published
- *pub_topic* ("camera"): Specifies the topic of the published message

Publishes: Video frames as image message on the specified ZMQ topic.
Subscribes: Nothing

4.2 XboxGamepadNode

XboxGamepadNode

Reads raw input from an Xbox gamepad and publishes gamepad state to a ZeroMQ topic. Initializes the gamepad and polls the state at a specified rate defined by *input_freq*. Handles disconnects by reconnecting the gamepad automatically. In case of a disconnect, the default button states are published under the "{pub_topic}_disconnect" topic.

Parameters (default values):

- *joystick_index* (0): Specifies a gamepad/controller
- *input_freq* (20): Specifies the polling rate of the controller in Hz
- *zmq_pub_url* ("tcp://localhost:5540"): Specifies where the gamepad state is published
- *pub_topic* ("gamepad"): Specifies the topic of the published message.

Publishes: *raw_gamepad_inputs* as JSON message. Publishes in case of a disconnect under the same URL and "{pub_topic}_disconnect" topic.
Subscribes: Nothing

4.3 DataRecorderNode

DataRecorderNode

Records images and labels them with steering commands, by default from the webcam and the Xbox gamepad, and saves the data to disk. Subscribes to image and steering data topics to pair each camera image with one steering command based on timestamps, dropping frames if timestamps don't align within a margin of *max_time_diff*. Images and steering commands are received asynchronously and at different rates. If the *CameraNode* publishes 10 frames per second, then frames are saved at 10 frames per second as well. When dropping frames, a warning on the console is issued.
The capturing process is started and stopped with a command from the *GamepadCommandNode*. While the car is in operation, the capturing process can be started and stopped multiple times. Each time, a new directory is created, named with a timestamp.
Parameters (default values):

- *image_sub_url* ("tcp://localhost:5550"): Specifies the URL to subscribe to images
- *image_sub_topic* ("camera"): Specifies the topic for the images
- *gamepad_function_sub_url* ("tcp://localhost:5541"): Specifies the URL to subscribe to gamepad data. Function commands are received here to start and stop data recording with the gamepad
- *gamepad_function_sub_topic* ("gamepad_function_commands"): Specifies the topic for gamepad function commands
- *steering_commands_sub_url* ("tcp://localhost:5551"): Specifies the URL to receive steering commands that are used to label the images
- *steering_commands_sub_topic* ("gamepad_steering_commands"): Specifies the topic for steering commands
- *max_time_diff* (0.1): Specifies the slack in seconds for which a pair of steering commands and an image are considered to match.
- *save_dir* ("/home/pi/data/"): Specifies the directory where data is saved

Publishes: Nothing
Subscribes: Images, *gamepad_function_commands* and *steering_commands*

4.4 GamepadCommandNode

GamepadCommandNode

Translates raw gamepad input into higher-level function commands and steering commands. Maps raw gamepad inputs to functions such as start/stop data recording and steering commands such as throttle, steer and emergency stop. This node abstracts the physical Xbox gamepad and defines the key binds to the function and steering commands in the system. Further, it dead zones the left thumb-stick drift with the *joystick_deadzone*. If the forward throttle and backward throttle triggers are activated simultaneously, it averages the raw values. Publishes the translated steering and function commands to ZMQ topics.

Parameters (default values):

- *joystick_deadzone* (0.15): Specifies the dead zone to handle joystick drift.
- *gamepad_sub_url* ("tcp://*:5540"): Specifies the URL to subscribe to raw gamepad input
- *gamepad_sub_topic* ("gamepad"): Specifies the topic for raw gamepad input. Based on that automatically takes the {gamepad_sub_topic}_disconnected as well.
- *gamepad_command_pub_url* ("tcp://*:5541"): Specifies where the translated steering and function commands are published
- *gamepad_function_commands_pub_topic* ("gamepad_function_commands"): Specifies the topic for publishing translated gamepad function commands
- *gamepad_steering_commands_pub_topic* ("gamepad_steering_commands"): Specifies the topic for publishing translated gamepad steering commands

Publishes: *gamepad_function_commands* and *steering_commands*
Subscribes: *raw_gamepad_inputs*

4.5 ControlFusionNode

ControlFusionNode

Fuses *steering_commands* messages from the gamepad (received from GamepadCommandNode) and a control algorithm (e.g. PilotNetCNode) and executes them on the *CommandInterface* which sends them over UART to the Raspberry Pi Pico.
If an input is made with the gamepad (such that it deviates from its default state), it disables steering commands from the control algorithm and the gamepad command is executed. This is done independently for all actions in the *steering_commands* message, but mainly throttle and steer. With that, it is possible to manually throttle the car, but let it be steered by the control algorithm or vice versa. It always allows to fully intervene with the gamepad. Once a gamepad action is detected and the corresponding action from the control algorithm is disabled, it will stay disabled until *override_duration* (seconds) have passed. If *override_duration* is 0, then the control algorithm takes back control immediately after the human stopped making an input on the gamepad. If it is set to i.e. 3, the control algorithm waits 3 seconds until it takes back control.
Parameters (default values):

- *control_sub_url* ("tcp://*:5560"): Specifies the URL to subscribe to steering commands from the control algorithm
- *control_sub_topic* ("steering_commands"): Specifies the topic for the control algorithm
- *gamepad_sub_url* ("tcp://*:5541"): Specifies the URL to subscribe to gamepad steering commands
- *gamepad_sub_topic* ("gamepad_steering_commands"): Specifies the topic for gamepad steering commands
- *zmq_pub_url* ("tcp://*:5570"): Specifies where the fused commands are published
- *zmq_pub_topic* ("fused_steering_commands"): Specifies the topic for fused control commands
- *override_duration* (3): Specifies the duration after which the control algorithm takes back control after

Publishes: Fused *steering_commands* as JSON message on the specified ZMQ topic. Note: Publishing those commands has no purpose other than logging them with another node. The steering_commands are also sent over UART and therefore executed by the Pico.
Subscribes: *steering_commands* from one algorithm and the gamepad

4.6 PilotNetCNode

PilotNetCNode

A control algoriithm that uses a neural network and the Coral TPU to predict steering commands from camera images. Publishes the predicted *steering_commands* along with further information to a ZMQ topic. The training and architectural details of this neural network are described in [3].

Parameters (default values):

- *zmq_pub_url* ("tcp://*:5560"): Specifies where the predicted steering commands are published
- *zmq_pub_topic* ("steering_commands"): Specifies the topic for predicted steering commands
- *camera_sub_url* ("tcp://*:5550"): Specifies the URL to subscribe to camera images
- *camera_sub_topic* ("camera"): Specifies the topic for camera images

Publishes: Predicted *steering_commands*
Subscribes: Images

4.7 LaneDetectionNode

LaneDetectionNode

A control algorithm that detects lanes in camera images and publishes *steeing_commands* to a ZMQ topic. Further, the processed image with annotated lanes and edges is published. Implementation details are described in [3].

Parameters (default values):

- *zmq_pub_url* ("tcp://*:5560"): Specifies where the *steering_commands* is published
- *zmq_pub_topic* ("steering_commands"): Specifies the topic for *steering_commands*
- *zmq_camera_pub_url* ("tcp://*:5551"): Specifies where the processed camera images are published
- *zmq_camera_pub_topic* ("camera_lane_detection"): Specifies the topic for processed camera images
- *camera_sub_url* ("tcp://*:5550"): Specifies the URL to subscribe to camera images
- *camera_sub_topic* ("camera"): Specifies the topic for camera images

Publishes: *steering_commands* as JSON messages. Also publishes camera images with annotated lanes and edges that are detected by the algorithm on a dedicated URL and topic.
Subscribes: Camera images

5 Usage Examples

The usage of the RaspiCar is primarily tested on Linux and recommended. Windows and MacOS are of course also possible, but may require some workarounds.

5.1 Setting Up the RaspiCar and Workflow

1. Clone the repository from [1] to your PC:
   git clone https://github.com/GGArdrey/RaspiCar.git

2. Install necessary python package requirements on your PC found in the `requirements_pc.txt` file in the repository. Hint: can be done conveniently in a Linux terminal via:
   pip install -r /path/to/requirements_pc.txt

3. Connect the Raspberry Pi 3 with the Wifi by editing the `/etc/wpa_supplicant/wpa_supplicant.conf` file on the Raspberry Pi's filesystem (SD card). The Raspberry Pi and your PC should be in the same network.

4. If an internet connection already exists and another network should be added:

   (a) Open terminal/PowerShell and run:
   ssh pi@raspberrypi.local
   the password is: **tugraz**

   (b) In the ssh session type:
   sudo nano etc/wpa_supplicant/wpa_supplicant.conf

   (c) Add network SSID and password to the file

5. If no internet connection exists:

   (a) Remove SD card from Raspberry Pi 3 and plug into your PC

   (b) Locate the `wpa_supplicant.conf` file in the following directory:
   `/etc/wpa_supplicant/wpa_supplicant.conf`

   (c) Manually edit the network SSID and password

   (d) Put SD card back into the Raspberry Pi 3

6. Optional: An SSH key can be deposited on the PC/Raspberry Pi to automatically authenticate. This is especially useful when code is uploaded to the Raspberry Pi often, because it will prompt for passwords every time.

5.1.1 Upload Code and do other Utilities

A `Makefile` is provided in the top-level directory of the repository. This makefile contains some useful commands or macros to easy data uploading, opening a REPL session with the Raspberry Pi Pico or copying record images from the Raspberry Pi 3 back to the PC. The `Makefile` can be used when using Linux and when `make` is installed. For Windows, coping the commands from the `Makefile` manually is required.

To upload code from your PC to the Raspberry Pi, the code will first be uploaded to the Raspberry Pi 3 and then parts of the code are uploaded to the Raspberry Pi Pico from there.

- **For Linux**:

  1. Open a terminal
  2. Navigate to the `Makefile` in the top-level of the repository:
     cd path/to/repository
  3. Run **make** in the terminal.

- **For Windows**:

  1. Open the PowerShell
  2. Navigate to the inside top-level directory of the repository:
     cd path/to/repository
  3. Manually open the `Makefile` in an editor and copy&paste the command at the top of the file into the PowerShell

  This uploads both `src_pi` and `src_pico` to the Raspberry Pi 3 and then only the contents from `src_pico` further to the Raspberry Pi Pico. It will ask passwords multiple times.

- After the code is uploaded, the Raspberry Pi Pico needs to be power cycled by pressing the small black button soldered to the PCB next to it.

5.2 Starting the RaspiCar

In order to start the RaspiCar and drive around, the following steps have to be done:

1. Power the RaspiCar.
2. Power the Xbox gamepad by pressing the Xbox logo
3. Optional: Upload Source Code as described in subsubsection 5.1.1
4. Open a terminal/PowerShell and SSH into the Raspberry Pi 3 with:
   ssh pi@raspberrypi.local
   the password is: **tugraz**
5. In the terminal of the Raspberry Pi 3 run the uploaded code:
   python3 ./raspicar/src_pi/main.py –log=INFO
   All nodes will start after a couple of seconds. The log level can be specified.

To stop the RaspiCar, the `main.py` can be simply terminated by hitting ctrl+c in the ssh terminal where the script was previously started. It can be run again any time by executing `main.py` again.

Warning: Please don't use the RaspiCar in battery mode for too long. The RaspiCar has no mechanism to detect battery charging status. If left for too long, the battery might get damaged. Periodically check battery voltage by plugging it into the charger. Below 7V, consider recharging.

When the RaspiCar is just powered on but not driving, the battery can last a couple of hours. Pure driving, around 15-20 minutes.

5.3 Writing and Integrating Control Algorithms

A template *YourAlgorithmTemplateNode.py* is provided in the `src_pi` directory to simplify the development and integration of custom control algorithms. The template provides a default configuration to act as a node in the system. It will automatically subscribe the webcam images can publish steering commands. The default publishing and subscribing IP addresses and ports are set up to match the default configurations of the other nodes. To launch a node together with all other nodes, it needs to be added to the `node_configs` in the `main.py`, which is already done for the *YourAlgorithmTemplateNode* example.

The *PilotNetCNode* is based on the same template and implements a neural network for autonomous steering, and can also be used as a reference. To run *PilotNetCNode*, it needs to replace *YourAlgorithmTemplateNode* in the `main.py`.

When publishing a steering command from within the *YourAlgorithmTemplateNode*, the payload (a python dict) of the message must contain a dictionary with at least the following keys provided in 5. All fields in the dictionary can be set by the *YourAlgorithmTemplateNode* and will be executed on the car when published. If the control algorithm can only steer, then throttle has to be set to zero. Then, it must be throttled manually using the connected gamepad.

```
steering_commands = {
    "steer": 0,
    "throttle": 0,
    "emergency_stop": 0,
    "reset_emergency_stop": 0,
    "sensors_enable": 0,
    "sensors_disable": 0
}
```

Listing 5: Steering Command Payload

5.4 Streaming RaspiCar's Webcam to a PC in the same Network

It can be very useful to stream the webcam of the RaspiCar to a PC to see what the car can see. Since the architecture is node based using ZeroMQ and network sockets, nodes can communicate across the network. A *CameraSubscriberNode* is provided, which can be executed on any PC running python and ZeroMQ and OpenCV (pip packages, see *requirements_pc.txt*) to receive the image messages from the camera node as well.

Theoretically, other nodes can be distributed across the network as well. For instance, the *YourAlgorithmTemplateNode* could be run on a separate PC and receive camera images across the network and publish it's steering commands across the network back to the RaspiCar. This would be particularly useful if the Raspberry Pi 3 can not handle the computational load of the algorithm running. However, this particular use case has not been tested and latencies of transmitting messages, especially large images, must be considered.

5.5 Record Camera Images and Gamepad Inputs

The *DataRecorderNode* runs by default (see `main.py`). To start a data recording session:

1. Start the RaspiCar as described in subsection 5.2
2. Press the **A**-Button on the Xbox gamepad to start the recording. A logging message will pop up in the ssh terminal.
3. Now the *DataRecorderNode* will save frames at the rate in which it receives those frames and labels them with the steering command which is applied at that time. By default, the webcam from the *CameraNode* and the steering input from the *GamepadCommandNode* are used.
4. Stop the recording session by pressing the **Y**-Button on the Xbox gamepad.
5. Copy the images from the `/home/pi/data/` directory of the Raspberry Pi 3. A *Makefile* macro **make copy** is provided. For Windows users: Manually copy&paste the corresponding command from the `Makefile` in the PowerShell. Alternatively, plug the Raspberry Pi's SD card into your computer and copy manually.

6 Troubleshooting

6.1 I can't SSH into the Rapberry Pi/ Find it on the network

- Make sure the Network is configured correctly on the Raspberry Pi 3. See subsection 5.1
- Make sure both your PC and the Raspberry Pi are in the same Network
- If a mobile hotspot is used, try to switch it off and on again, then connect the Raspberry Pi 3 first, then the PC.

Especially when using a mobile hotspot, connection issues were observed in the past. The recommended way is to use a dedicated router and not a smartphone. From past experiences, a mobile hotspot works 50 percent of the time. The problem is not the connectivity but somewhere in the configuration of the mobile hotspot.

6.2 I started the RaspiCar, but it doesn't respond to control inputs

1. Check in the SSH terminal where the `main.py` was executed, if all nodes started or if they terminated already due to an internal error

2. In the same terminal, check if the Xbox gamepad connected
3. Press the black reset button on the PCB to reset the Raspberry Pi Pico
4. Check if you correctly used the *YourControlAlgorithmTemplateNode*, see subsection 5.3. Are the Pub/Sub topics and addresses specified correctly? Do you publish a proper *steering_commands* python-dict? Do you launch all nodes properly in the *main.py*?
5. If nothing helps, reboot the Raspberry Pi 3

6.3 I can drive the RaspiCar with the gamepad, but only sometimes the control does not automatically change back to the ControlAlgorithmNode

If this particular case happens, try to give the thumb-stick on the gamepad another wiggle. The thump-stick sometimes has a small drift. This causes the system to sometimes think that a control input from the gamepad is present. The threshold for "dead zoning" the thumb-stick can be adjusted in the *GamepadCommandNode*.

6.4 The Raspberry Pi Pico blinks rapidly (10Hz), Emergency State

This indicates an emergency state. Please look into subsubsection 2.2.1 for details. This happens either by interrupted communication between the Raspberry Pi 3 and Raspberry Pi Pico or because the gamepad disconnected or triggered this state manually. To enter this state manually, press **Back**-button on the gamepad, to exit the emergency state press **Start**-button.

6.5 Sometimes in the middle of driving, the RaspiCar stops and the Pico blinks Rapidly (10Hz)

The car sometimes enters the automatic emergency state (see subsubsection 2.2.1) when throttling or steering and then automatically leaves again shortly after. This can be caused by low battery voltage or will occur very frequently when using the external power supply. When the voltage or supply current drops too low, the Raspberry Pi 3 will freeze, causing all peripherals (like UART) to be paused. This triggers the emergency state automatically and leaves the state when the Raspberry Pi 3 recovers to normal operation.

If this happens while driving with the battery, consider recharging it.

6.6 I get a charging-error with the LiPo battery charger

This error might be related to the accidentally undercharged LiPo battery. The battery voltage can be measured with the battery charger as well. If it is found that the battery is indeed undercharged (below 3.2V per cell and 6.4V for the whole battery), it can be recovered if its not too low.

Please refer to this video [4] to recover the battery. These instructions have been tested two times to recover both cells from 2.8V to again 3.2V. Please don't use 0.5A charging current as in the video, but only 0.1A until 3.2V are reached. Then proceed with normal charging.

CAUTION!

- **I don't take any responsibility for possible damages.**
- **Do not leave a recovering battery unattended!**
- **If the battery starts to puff or deform in any way during charging you should stop.**

7 Appendices

7.1 Gamepad Keybinds

Button	**Function**
start	reset manual emergency stop
back	manual emergency stop
A	start data recording
Y	stop data recording
X	TOF sensors enable
B	TOF sensors disable
left thumb stick	steering
right shoulder trigger	forward throttle
left shoulder trigger	reverse throttle

Table 1: Gamepad Key Bindings

7.2 Benchmarks

Table 2 provides a brief overview of end-to-end latencies of the system. End-to-end latency measures the total processing time on the Raspberry Pi 3 from capturing a frame to sending the final steering command to the Raspberry Pi Pico. Inference measures the inference time of *PilotNetC*. CPU usage reflects the total load on the Raspberry Pi 3 CPU. The measurements were conducted by comparing timestamps within various stages of the processing steps. The provided numbers are only serving as a guidance to understand the magnitude of latencies to expect.

Action	End-to-End Latency	Inference	CPU Usage
Running all nodes including *PilotNetCNode* as control algorithm, transmitting camera live-feed over network with 640x360 at 10FPS	50ms	30-35ms	25%
Running all nodes including *PilotNetCNode* as control algorithm, transmitting camera live-feed over network with 640x360 at 10FPS, actively recording and saving camera and steering commands	50ms	30-35ms	32%

Table 2: Benchmarks of Latencies on the RaspiCar

7.3 Code Listings

```
import zmq
import json
import time
from utils.message_utils import create_json_message

# ZMQ setup
zmq_context = zmq.Context()
zmq_publisher = zmq_context.socket(zmq.PUB)
zmq_publisher.bind("tcp://*:5555")
```

```
# Create a Dict with data to publish
payload = {
    "steer" : 0.75,
    "throttle" : 0,
    "emergency_stop" : 0,
    "reset_emergency_stop" : 0,
    "sensors_enable" : 0,
    "sensors_disable" : 0
}
message = create_json_message(payload, topic="steering_command")

# Send the JSON message
zmq_publisher.send(message)

zmq_publisher.close()
zmq_context.term()
```

Listing 6: Full Implementation JSON MEssage Publishing Side Code

```
import zmq
from utils.message_utils import parse_json_message

# ZMQ setup
zmq_context = zmq.Context()
zmq_subscriber = zmq_context.socket(zmq.SUB)
zmq_subscriber.connect("tcp://localhost:5555")
zmq_subscriber.setsockopt_string(zmq.SUBSCRIBE, "steering_command")

# Receive the JSON message
message = zmq_subscriber.recv_string()
topic, timestamp, payload = parse_json_message(message)
# now payload can be accessed as normal python dict again...
print(f"Steer: {payload["steer"]}, Throttle: {payload["throttle"]}"
    )

zmq_subscriber.close()
zmq_context.term()
```

Listing 7: Full Implementation of JSON Message Receiving Side Code

```
import cv2
import zmq
from utils.message_utils import create_jpg_image_message

# ZMQ setup
zmq_context = zmq.Context()
zmq_publisher = zmq_context.socket(zmq.PUB)
zmq_publisher.bind("tcp://*:5555") # Publish across the whole
    Network

# Initialize the camera
```

```
camera_source = 0  # or any other camera source
cap = cv2.VideoCapture(camera_source)

# Capture and send a single frame
ret, frame = cap.read()
if ret:
    message = create_jpg_image_message(frame, topic="camera_topic",
        quality=95)
    zmq_publisher.send_multipart(message)
else:
    print("Error capturing frame.")

cap.release()
zmq_publisher.close()
zmq_context.term()
```

Listing 8: Full Implementation of Image Mesage Publishing Side Code

```
import zmq
import cv2
from utils.message_utils import parse_jpg_image_message

# ZMQ setup
zmq_context = zmq.Context()
zmq_subscriber = zmq_context.socket(zmq.SUB)
zmq_subscriber.connect("tcp://localhost:5555")
zmq_subscriber.setsockopt(zmq.SUBSCRIBE, "camera_topic")

# Receive and display a single frame
message = zmq_subscriber.recv_multipart()
topic, image, timestamp = parse_jpg_image_message(message)
# now image is a normal cv2 image type again...
if image is not None:
    cv2.imshow('Received Image', image)
    cv2.waitKey(0)
else:
    print("Failed to receive image.")

cv2.destroyAllWindows()
zmq_subscriber.close()
zmq_context.term()
```

Listing 9: Full Implementation of Image Message Subscribing Side Code

```
steering_commands = {
    "steer": 0, #between -1 (max. left), 1 (max. right)
    "throttle": 0, #between -1 (max. reverse), 1 (max. forward)
    "emergency_stop": 0,
    "reset_emergency_stop": 0,
    "sensors_enable": 0,
    "sensors_disable": 0
}
```

```
raw_gamepad_inputs = {
    "left_stick_x": 0.0,
    "left_stick_y": 0.0,
    "right_stick_x": 0.0,
    "right_stick_y": 0.0,
    "left_trigger": 0.0,
    "right_trigger": 0.0,
    "A": 0,
    "B": 0,
    "X": 0,
    "Y": 0,
    "left_bumper": 0,
    "right_bumper": 0,
    "back": 0,
    "start": 0,
    "left_stick_click": 0,
    "right_stick_click": 0,
    "dpad_up": 0,
    "dpad_down": 0,
    "dpad_left": 0,
    "dpad_right": 0
}

function_commands = {
    "start_data_recording": 0,
    "stop_data_recording": 0
}
```

Listing 10: Overview of Python-Dictionaries as Payload in JSON Messages

References

[1] F. L. Maaß, "Raspicar repository," 2024, accessed: 2024-08-01. [Online]. Available: https://github.com/GGArdrey/RaspiCar/

[2] Z. Project. (2024) Zeromq. Accessed: 2024-08-01. [Online]. Available: https://zeromq.org/

[3] R. Blazevic, F. L. Maaß, O. Veledar, and G. Macher, "Intelligent decision-making in lane detection systems featuring dynamic framework for autonomous vehicles," 2024.

[4] R. Crawler, "Cell connect error (lipo charging problem howto)," 2020, accessed: 2024-08-01. [Online]. Available: https://www.youtube.com/watch?v=AeygzOtpYwQ&ab_channel=RawangCrawler