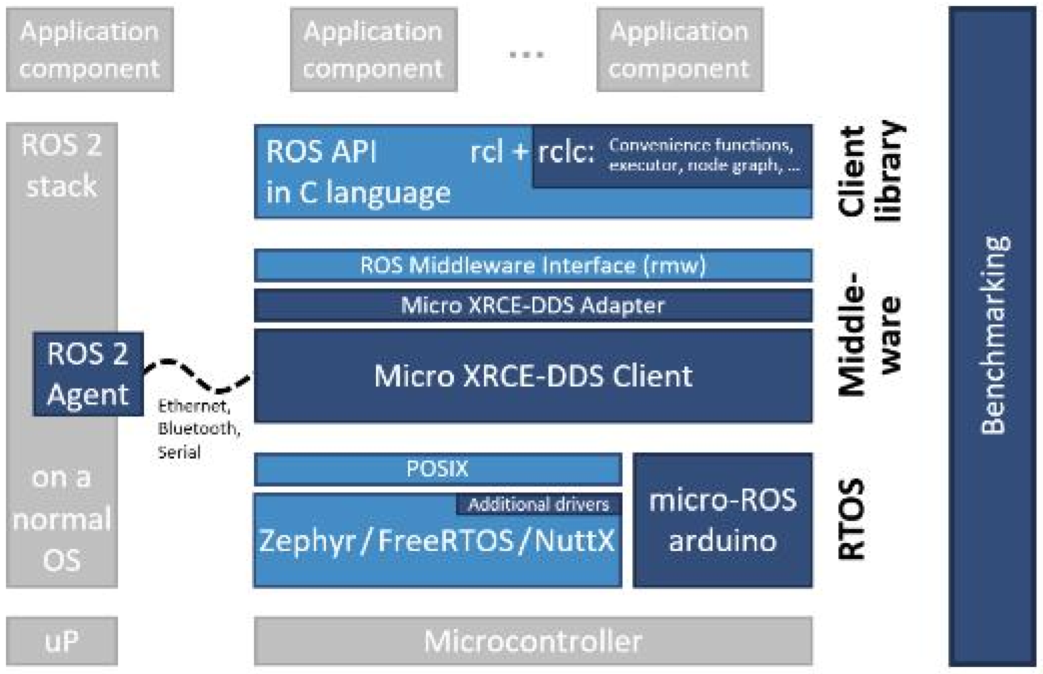
Micro – ROS

In this blog entry, I will share the progress of my work at the CPS and explain how I worked with Micro-ROS. I will start with an introduction to the basics of Micro-ROS, followed by a detailed explanation of how to set up the framework of the Foxy distribution. Additionally, I will discuss two ways of uploading code onto an ESP32 - Wroom microcontroller, and then conclude with the operation of a multi-servo setup. Throughout the blog, I will also provide solutions to some of the problems that encountered during my installation process.

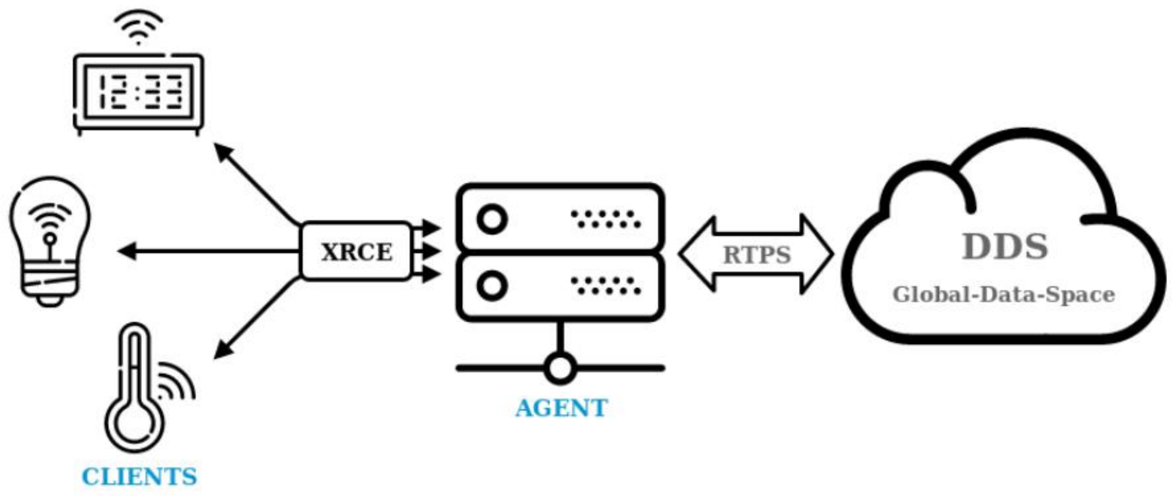
Basics

Micro-ROS is an open-source software framework for building real-time robotic systems. It is based on the Robot Operating System (ROS), which is a widely used framework for developing robotic software. Micro-ROS is designed specifically for micro controllers and other embedded devices, making it ideal for building low-power, low-cost, and real-time robotic systems.

The Micro-ROS framework is built on top of two main components: the Micro-XRCE-DDS client and the ROS 2 client library. Micro-XRCE-DDS is a lightweight and efficient implementation of the Data Distribution Service (DDS) protocol, which provides a standard mechanism for sharing data between different components in a distributed system. The ROS 2 client library runs upon the real time operation system of the microcontroller. Therefore, the basic installation supports several real time operating Systems as FreeRTOS, Zephyr and NuttX and is part of the base layer. (https://micro.ros.org/docs/concepts/client_library/introduction/, https://micro.ros.org/docs/overview/rtos/)



The Middleware is used for the communication. There the microcontroller unit represents an XRCE - Client and communicates based on a client – server architecture with the XRCE - Agent which is located in the hosting ROS 2 system. This Agent builds the bridge between the DDS system which is used within ROS 2 and the Micro-XRCE-DDS which runs on the microcontrollers (https://micro.ros.org/docs/concepts/middleware/Micro_XRCE-DDS/). The communication is quite similar to MQTT and has also Topics which can be subscribed and published to.



To simplify the programming and maintenance the already existing libraries RCL of ROS2 is extended for Micro - ROS and a new RCLC is implemented.

To sum up, this means that it is necessary to build the firmware of micro – ROS in a ROS 2 workspace. Within this environment the executed code will be located, and all the dependencies and settings will be defined. After the firmware is build it can be flashed to the microcontroller. In the end the Agent which is responsive for the communication needs to be started.

Installation

Before setting up a micro - ROS environment it is necessary to install a ROS 2 Distribution. In this case the **Foxy Distribution** will be installed on **Ubuntu 20.04.6 LTS** and a workspace with the name “microros_ws” will be generated.

To avoid any problems during the installation it is advisable to update the system and install docker.

```
# Assuming that ROS2 is already installed on your computer

source /opt/ros/foxy/setup.bash


# Create a workspace and download the micro-ROS tools

mkdir microros_ws

cd microros_ws

git clone -b foxy https://github.com/micro-ROS/micro_ros_setup.git
src/micro_ros_setup


# Install rosdep2

sudo apt install python3-rosdep2


# Update dependencies using rosdep

sudo apt update && rosdep update

rosdep install --from-paths src --ignore-src -y


# Install pip & colcon

sudo apt-get install python3-pip

sudo apt install python3-colcon-common-extensions



# Build micro-ROS tools and source them

colcon build

source install/local_setup.bash
```

After following the instructions to this point, the basic workspace of Micro - Ros should be setup.

Since the system is based on the DDS protocol every code need to be placed into an app. Each app is represented by a folder containing the “main.c” file and the “CMakeLists.txt”. The “main.c” file contains the executed code while the “CMakeLists.txt” file defines the script to complie the application is defined.

Within this Blog post I am going to describe two different ways to create such an app. In the first part I will focus on the supposedly less powerfull one based on “PlatformIO” and afterwards the from my perspective more complex method, which will be referred to as the "ROS-Way,"

PlatformIO

“PlatformIO is a cross-platform, cross-architecture, multiple framework, professional tool for embedded systems engineers and for software developers who write applications for embedded products.” [https://docs.PlatformIO.org/en/latest/what-is-PlatformIO.html]

By choosing this way of uploading, it is predefinded that the “micro-ROS Arduino” operating systems is used. This brings the huge benefit that the familiar coding style of the ArduinoIDE can be used, as well as its libraries.

PlatformIO is very common and therfore available as an extension of Visual Studio Code (VSCode). The installation is quite straightforward. More informations can be found at the following link (https://docs.PlatformIO.org/en/stable/integration/ide/vscode.html#installation).

Afterwards 4 Steps are required.

1. Step: Setup a new project

On “PlatformIO Home” a new project can be created. There a usefull name should be chosen, the board and the framework needs to selected. Since in this case an ESP WROOM 32 is used the “Espressif ESP32 Dev Module” is chosen as board and the “Arduino Framework” is used.

2. Step: Overwrite platfomio.ini file:

The auto-generated .ini file needs to be adapted to be able to use Micro - ROS and the ESP32 boards. In this case the following lines are used:

```
[env:esp32dev]

platform = https://github.com/PlatformIO/platform-espressif32.git#feature/arduino-upstream

board = esp32dev
```

```
framework = arduino

lib_deps =

   https://github.com/micro-ROS/micro_ros_arduino.git


build_flags =

   -L ./.pio/libdeps/esp32dev/micro_ros_arduino/src/esp32/

   -l microros

   -D ESP32


platform_packages =

  toolchain-xtensa32 @ ~2.80400.0

  framework-arduinoespressif32@https://github.com/espressif/arduino-esp32.git#2.0.2
```

3. Step: write code

In this step the code for the application which will run on the microcontroller needs to be written into the "main.cpp" file.

Third-party libraries, such as libraries written for the Arduino IDE, can be implemented by copying the ".cpp" file into the "src" directory of the PlatformIO project (same directory where the main file ist located) and the ".h" file into the "include" directory of the project.

4. Upload

First of all, a PlatformIO terminal need to be opened, which can be found in the Platfomio extension. Afterward, it is important to navigate to the project directory and then run the following commands with the serial-connected ESP32 mounted at /dev/ttyUSB0 -v6. To run the commands, use a **terminal** that can be started in the **Platfromio extention**. Therfore the commands will be executed in the source directory of the project. Otherwise navigate first to the right directory:

```
# Install libaries and dependencies  ( old command "pio lib install")

pio pkg install

# Build the firmware

pio run
```

```
# Flash the firmware

pio run --target upload

# Activate the micro - ROS Agent

sudo docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --
net=host microros/micro-ros-agent:foxy serial --dev /dev/ttyUSB0 -v6
```

Other options for the Agent are.

```
# UDPv4 micro-ROS Agent

docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --
net=host microros/micro-ros-agent:$ROS_DISTRO udp4 --port 8888 -v6



# TCPv4 micro-ROS Agent

docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/shm --privileged --
net=host microros/micro-ros-agent:$ROS_DISTRO tcp4 --port 8888 -v6



# CAN-FD micro-ROS Agent

docker run -it --rm -v /dev:/dev -v /dev/shm:/dev/mshm --privileged --
net=host microros/micro-ros-agent:$ROS_DISTRO canfd --dev [YOUR CAN
INTERFACE] -v6
```

During the first upload I encountered the following error: “esptool write_flash: error2”. A solution for this problem can be found at the end of the blog post.

After pusshing the Rest - Button on the ESP32 board the agent should log in again and several messages will be received. If everything went right, the agent is now up and running. As long as the agent is running, the communication between the microcontroller and the ROS2 should work fine.

Custom Message

It is recommended to create custom messages so that the communication can be precisely adapted to your needs and add a lightweight encryption layer. With custom messages, only the network partners who know the message format can read the data directly.

To create a custom message by using PlatformIO, the process is similar to the ROS-Way, except that a new folder named "extra_packages" needs to be created in the source folder of the projec. The message can then be built in this folder by following the same instructions until the building step afterward use insted the following ones.

```
# clean library

pio run --target clean_microros

# Install libaries and dependencies

pio pkg install

# Build the firmware

pio run
```

The continue with flashing, and aktivating the agent.

Third Party Libraries

Using third-party libraries in PlatformIO is straightforward. Simply place the ".cpp" file in the source folder and add the header file (".h file") to the include folder. After importing the header file in the application code, it can be used as usual. In this project, the "Adafruit PWMServoDriver" library was implemented to control the PCA9685 breakout board, which can control up to 16 servo motors or other PWM-based components. Since communication is via I2C, multiple PCA9685 breakout boards can be attached to one pin pair, with each board controlling up to 16 servo motors or other PWM-based components. This results in a total number of 992 servos which can be controled with a single ESP32 microcontroller. (https://dronebotworkshop.com/esp32-servo/, https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library)

ROS - Way

For using the “ROS - Way” it is necesarry to have the “**micro_ros_setup”** package installed. Which is already done when following the previously described installation process. Otherwise it can be installed like any other ROS2 package.

The setup process is a four-step procedure. In the first step, the user can create a new micro-ROS application by configuring the target hardware and RTOS. A very helpful video can be found at the follwoing link. (https://www.youtube.com/watch?v=t-ovNiyWbDg, https://asciinema.org/a/JzmqYAOStptH2UF4WENKBq5Yx)

1. Step: create firmware

By following this methode of uploading the code to the microcontroller, the user can choose the RTOS system. The available RTOS systems can be found in the documentation (https://micro.ros.org/docs/tutorials/core/first_application_rtos/). In the following steps the FreeRTOS will be used and therfore the firmeware needs to be installed.

```
# Create step

ros2 run micro_ros_setup create_firmware_ws.sh freertos esp32
```

After installing the FreeRTOS, a new folder called "firmware" will appear. Every created application is build up of two files and needs to be stored at very specific location to be executable. The application that will be created in the next step must be located within the workspace at **“firmware/freertos_apps/apps/”**, where the example applications are located as well. The actual application constists of an **“app.c”** and an **“app-colcon.meta”** file. The easiest way to generate a new application is to copy the files from one of the examples into a new directory and edit the app.c afterwards. (For this project the files from the “int32_publisher” were copied and edited).

```
# naviagte into the apps folder ~ microros_ws/firmware/freertos_apps/apps/

mkdir servo_subscriber

scp int32_publisher/app.c servo_subscriber

scp int32_publisher/app-colcon.meta servo_subscriber
```

Afterwards the actual code can be written in it, while the part of the code which needs be executed frequently will be written into the “timer_callback” function. However, remember that the coding style needs to be FreeRTOS compliant, which means that the pin definition and the available functions differ to the Arduino style.

Before continuing with the next step, it is important to know the IP address of the host machine on which the Agent will run in future, assuming that WiFi communication needs to be established. The IP address can be determined by opening a new terminal and running the following command, where the required address is listed under **“wlp2s0: inet”**

```
# New Terminal --> wlp2s0: inet adress

ifconfig
```

In my case the IP address is “192.168.20.35”. After identifying the IP address, return back to the other terminal and continue with configuring the firmware. Choose the previously created application or one of the examples, as well as the transmission settings. For more information about other transport options, please visit the website. (https://micro.ros.org/docs/tutorials/core/first_application_rtos/freertos/)

```
# Choose application & transport option

ros2 run micro_ros_setup configure_firmware.sh int32_publisher -t udp -i 192.168.20.35 -p 8888
```

Since the microcontroller will transmit data via WiFi, the credentials must be defined. Many settings, including this, can be configured in the user interface (UI) that will open after executing the following command. Within the Ui, navigate to “micro-ROS Transport Settings --> WiFi Configuration” and edit the SSID as well as the WiFi Password. Afterwards, save the changes and exit.

```
# Transport and Credentials

ros2 run micro_ros_setup build_firmware.sh menuconfig

# After setting the credentials press “s” to save, then confirme with “enter key” twice and then leave with “q”
```

2. Step: Build step

Build the firmware on the local machine and check that no errors appear at the ouput.

```
# Build step
ros2 run micro_ros_setup build_firmware.sh
```

3. Step: Flash step

Before flashing the firmware to the microcontroller it is necessary to connect the ESP32 with a USB cable to the computer. Then execute the following command.

```
# Flash step
ros2 run micro_ros_setup flash_firmware.sh
```

If the “esptool write_flash: error2” error occurs during the flashing process, it means that you do not have the permition to write to the USB. To solve this, have a look at Error: “esptool write_flash: error2” section at the end of the blog.

4. Step: Agent step

To finally reach a running system it is required to set up the Agent which will translate between the two DDS systems.

```
# Create Agent
ros2 run micro_ros_setup create_agent_ws.sh
# Build Agent
ros2 run micro_ros_setup build_agent.sh
source install/local_setup.bash
# Run Agent
ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888
```

After restarting the microcontroller by pushing the “RST - Button” some new messages should appear in the terminal. If not, something went wrong with the WiFi connection. Therfore, double-check the IP address of the host machine and make sure that all commands were executed without raising any errors.

Custom Message

When creating a custom message on micro - ROS, it is essential to create the message in a specific location. The correct place to put the message is in the “**firmware/mcu_ws”** directory. Afterwads, you can follow the instructions form the micro - ROS page or execute the next code lines. In this case the message “ServoMessage” will be generated and will be placed in the “servo_message” directory within the “mcu_ws” directory.
(https://micro.ros.org/docs/tutorials/advanced/create_new_type/)

```
# Navigate into the firmware/mcu_ws directory
ros2 pkg create --build-type ament_cmake servo_message
cd servo_message
mkdir msg
# Create an empty message
touch msg/ServoMessage.msg
```

In the autogenerated **CMakeLists.txt** file you should add the following lines just before **ament_package().**

```
#  Add the following lines before the “ament_package()” command
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/ServoMessage.msg" )
#next line is the “ament_package()” command
```

In the autogenerated **package.xml** file you should add the following lines after the license declaration:

```
#  Add the following lines after the "<license>TODO: License declaration</license>"
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Now you can specify the content of your message. A custom message can be easily built up based on the standard message types. A list of the **standard message types** can be found at the following page (http://docs.ros.org/en/melodic/api/std_msgs/html/index-msg.html). For this use case, the message is quite simple, as it just needs an integer value for the servo number and an float for the value that will be written to the PWM board. Furthermore, it is important that the **name** of the stored variable **starts** with a **lower case** letter**.** To define the message, open the .msg file and create your custom message.

```
#  Open ServoMessage.msg
nano msg/ServoMessage.msg
```

The content of ServoMessage is:

```
#  Build your message by writing the messag type and your name for it into the file and
int32 servo_number
float64 servo_value
# save and leave the file
```

Then go back in the workspace and run the following command, which will build the message.

```
#  In the workspace -->  microros_ws
ros2 run micro_ros_setup build_firmware.sh
```

After a successful built, it is possible to use the message in the application file by including the message type and generating a message of the type ServoMessage which the values will be assigned to.

```
#include <servo_message/msg/servo_message.h>
...
servo_message__msg__ServoMessage msg;

msg.servo_number = 3;
msg.servo_value = 42.0;
...
```

When using it with a publisher remember to set the right types there too.

Third Party Libraries

To use a third party library within micro - ROS based on FreeRTOS, it is again important to place it in the right location, similar to the message. In the following steps, the “pca9685 library” will be implemented. Therfore, navigat to **“firmware/mcu_ws”** and create a new package.(https://github.com/brainelectronics/esp32-pca9685, https://www.youtube.com/watch?v=S7AjhNKHpvs)

```
# navigate to “~ firmware/mcu_ws” and create a new package
ros2 pkg create pca9685_library --build-type ament_cmake
```

Afterwards, it is required to place the **header file** within the new package under **“<~/include/pca9685_library”** and the **“.c”** or **“.cpp”** into the **“~/src”** folder. **Update** the **path** to the header file in the “.c” file. In most cases the path is defined as ”#include pca9685.h” and needs to be changed to **“#include "pca9685_library/pca9685.h""**. Finally, the **“CMakeList.txt”** needs to be adapted. Therefor, add the following lines of code bevor the ament_package() command.

```
# add at the bottom directly before ament_package ()
# Add the "include" directory to the include path for the compiler
include_directories(include)

# Create a library target called "pca9685_library" from the source file "src/pca9685.c"
add_library(pca9685_library src/pca9685.c)

# Export the library target so it can be used by other packages
ament_export_targets(pca9685_library HAS_LIBRARY_TARGET)

# Install the header files in the "include/pca9685_library" directory to the system
"include" directory
install(
  DIRECTORY include/pca9685_library
  DESTINATION include
  )

# Install the library target to the system library directory
install(
  TARGETS pca9685_library
  EXPORT pca9685_library
  LIBRARY DESTINATION lib
  ARCHIVE DESTINATION lib
  RUNTIME DESTINATION bin
  INCLUDES DESTINATION include
  )
```

Last but not least, the library has also be added to the application in the "firmware/freertos_apps/apps/servo_message" as an include statement **"#include "pca9685_library/pca9685.h""**.

For this particular library, some more definition needs to be made to attach the PCA9685 module to the right GPIO of the ESP32, as well as implementing a setup loop where the board gets initialized, just like in the example code. (https://github.com/brainelectronics/esp32-pca9685/blob/master/main/pca9685Test.c)

Communication

To set up a communication apart form the subscriber on the microcontroller, a publisher is required. Therfore, a new ROS2 workspace was initialized and a Python based publisher was implemented following the instruction in (https://www.theconstructsim.com/ros2-qa-215-how-to-use-ros2-python-launch-files/)

```
# Create a workspace where you want

mkdir my_ros2_ws

cd my_ros2_ws

# execute universal build tool and create basic structure

colcon build

# source ROS distribution

source /opt/ros/foxy/setup.bash

# source the workspace

source install/setup.bash

mkdir src

cd src

# create a package named py_pubsub (python publisher and subscriber)

ros2 pkg create --build-type ament_python py_pubsub

cd py_pubsub

cd py_pubsub
```

Create a python file within that directory and call it "publisher_member_function.py"

```
import rclpyfrom rclpy.node import Node

from std_msgs.msg import String
```

```
class MinimalPublisher(Node):


    def __init__(self):

        super().__init__('minimal_publisher')

        self.publisher_ = self.create_publisher(String, 'topic', 10)

        timer_period = 0.5  # seconds

        self.timer = self.create_timer(timer_period, self.timer_callback)

        self.i = 0


    def timer_callback(self):

        msg = String()

        msg.data = 'Hello World: %d' % self.i

        self.publisher_.publish(msg)

        self.get_logger().info('Publishing: "%s"' % msg.data)

        self.i += 1


def main(args=None):

    rclpy.init(args=args)


    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)


    # Destroy the node explicitly

    # (optional - otherwise it will be done automatically

    # when the garbage collector destroys the node object)

    minimal_publisher.destroy_node()

    rclpy.shutdown()


if __name__ == '__main__':

    main()
```

“Open **package.xml** file and add rclpy and std_msgs as dependencies, by adding the two lines **below ament_python**”

```
    <exec_depend>rclpy</exec_depend>

    <exec_depend>std_msgs</exec_depend>
```

“Let’s now open the **setup.py** file and add our publisher_member_function.py script to the entry_points list. Since we want out publisher to be started in the main function on the publisher_member_function file inside the py_pubsub package, and we want our publisher to be called talker, we have to add the entry_points at the end of the setup.py file”. It works when adding the following code after **tests_require=['pytest']**, which is at the very bottom of the code.

```
    entry_points={

            'console_scripts': [

                    'talker = py_pubsub.publisher_member_function:main',

            ],

    },
```

Now the implementation of the publisher is finished. The last step is now to build our workspace.

```
# Check dependencies

rosdep install -i --from-path src --rosdistro foxy -y

# execute universal build tool and create basic structure

colcon build

# source the workspace

source install/setup.bash
```

Finally the publisher can be executed by:

```
ros2 run py_pubsub talker
```

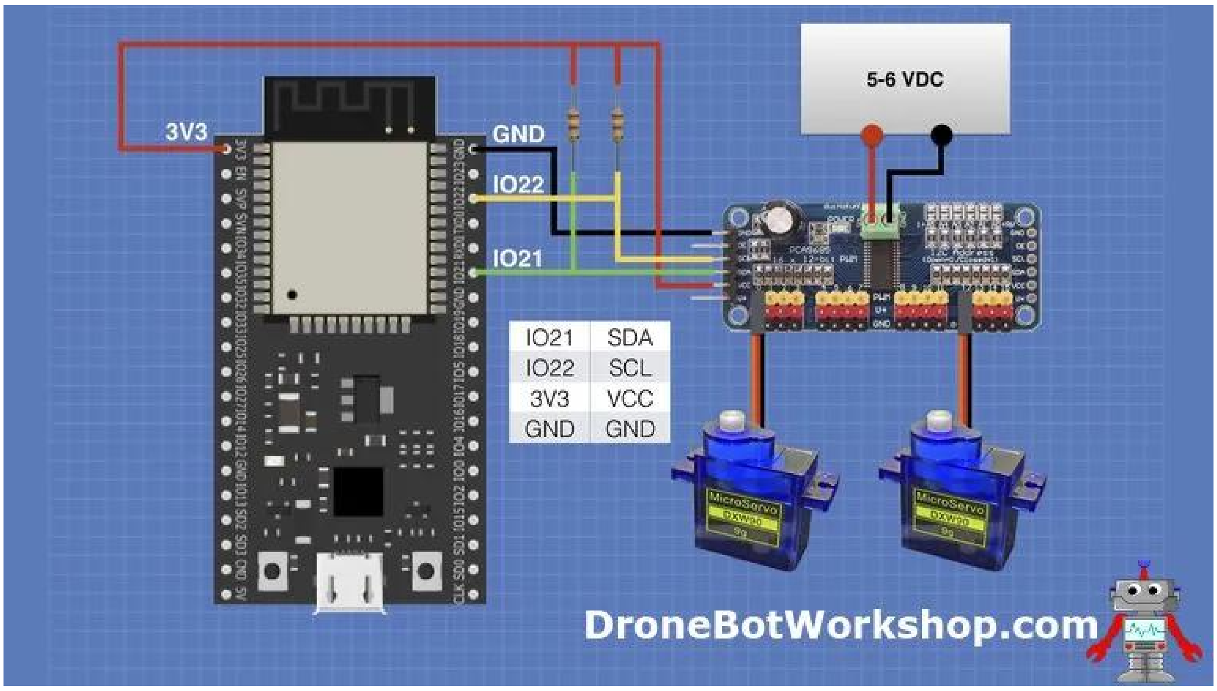
All changes regarding the publisher need to be done in the “publisher_member_function.py” file. By creating multiple “publisher_member_function_XX.py” files it is possible to prepare publishers with different messages. Afterwards only the filename definition in the “setup.py” file needs to be changed. (https://www.theconstructsim.com/ros2-qa-215-how-to-use-ros2-python-launch-files/)

Usage

To used the ServoMessage or the third party PCA9685 board library, copy the corresponding folder into the “~firmware/mcu_ws” directory and include it in the apllication (e.g. #include "pca9685_library/pca9685.h”). To run the application, it is necessary to load the “servo_subscriber” into the “firmware/freertos_apps/apps/” direktory, or to copy the entire repository. The Python based publisher can be plased in any ROS2 workspace, but make sure that the servo_message package is also available in that workspace.

When using the Python based publisher, the default setting will publish messages based on the ServoMessage. By changing the definition of the talker in the “setup.py” file to a different predefined “publisher_member_function”, other messages than the current ServoMessage can be published.

The setup follows that schematics:



IO21	SDA
IO22	SCL
3V3	VCC
GND	GND

Tipps:
After opening or restarting the terminal, it is always a good practice to source the distribution and the local setup in the workspace, which is the directory where Micro - ROS was installed into.

```
# source distribution

source /opt/ros/foxy/setup.bash

# source local setup

source install/local_setup.bash
```

Error: “esptool write_flash: error2”

This error occures when trying to flash to the ESP32 over USB. Based on the research I conducted, this issue has noting to do with the written code. n my case, Linux denied permission to write to the ESP32, which was mounted on '/dev/ttyusb0'. To check if it's a permission problem, you can execute the following command in a new terminal:

```
stat /dev/ttyUSB0
```

The response should be something similar to 'Gid: (20/dialout)'. To solve this problem, it's necessary to grant permission to the current user to access the USB. You can grant permission by adding the user to the dialout group with the following command:

```
sudo usermod -a -G dialout $USER
```

After that, a restart of the computer is required or you can log the user out and back in again. (https://askubuntu.com/questions/133235/how-do-i-allow-non-root-access-to-ttyusb0)

Write protected directory

If a created folder becomes read- and write-protected (e.g. marked in green in the command line), you can use the following command to deactivate write protection of the folder and the files within it:

```
# deactivate read and wirte protection of folder and files within it

sudo chmod -R a+rwx folder
```

RQT_Graph

To understand the setup of the nodes and the communication, the following command executed in a ROS2 workspace can be very helpful and therfor opens a graphical visalisation of the node connections in your network.

```
# Opens RQT Graphic visualisation

rqt_graph
```