E-Rocket Report 1 - Using PX4 hardware to read sensor data and actuate servos and motors

Master in Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa

Pedro Maria da Costa Almeida Martins — 99303*
pedromcamartins@tecnico.ulisboa.pt

Advisor: Prof. Paulo Oliveira
Co-advisor: Pedro Santos

*I declare that this document is an original work of our own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa (https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/).

Contents

1 Objective

The E-Rocket project is a research project developed at Instituto Superior Técnico (IST), University of Lisbon. The project aims to develop a drone with thrust vector control (TVC[1]) capabilities, in order to validate control algorithms and navigation systems developed by control researchers. A drone is used as an alternative to a rocket, as it is significantly easier, cheaper, reliable, and safer to perform validation.

The drone is based on a Pixhawk running PX4 flight controller, which are hardware and software open-source platforms for drones (respectively). It is equipped with a set of sensors[2], including an inertial measurement unit (IMU) and a GPS. The drone is also equipped with a set of actuators, including servos and motors, which are used to control the drone's attitude and thrust.

In other to evaluate if the drone can be used to validate the control algorithms, the team needs to access sensor data and directly control the servos and motors. The team is considering the use of differential thrust, which is a technique used to control the attitude of a drone by varying the thrust of the motors. Meaning that we would need to be able to control the motors independently of one another.

The system has the following requirements:

- The drone must be able to run and validate controllers developed by the researches in the team.
- The drone must have a thrust-to-weight ratio of at least 2.
- The drone shouldn't weigh more than 2 kg.
- The team should use COTS hardware and software, in order to reduce costs and development time.

In the context of this report, we will tackle the first requirement.

[1] https://en.wikipedia.org/wiki/Thrust_vectoring
[2] https://docs.px4.io/main/en/getting_started/px4_basic_concepts.html#sensors

2 Background

PX4[3] is an open-source flight control software that runs on a variety of hardware platforms, including PixHawk[4]. It is designed to be used in drones, but it can also be used in other vehicles, such as cars and boats[5]. PX4 is a modular system, meaning that it can be extended to add new features. It is composed of a set of modules that communicate with each other using a publish-subscribe model, with the help of a uORB communication bus. The team decided to use a PX4 flight controller, as it is an open-source platform that is widely used in the industry and academia. Additionally, other researchers in the department are already using it, which means that the team can benefit from their experience and knowledge.

PX4 currently doesn't provide native support for a drone with a TVC module[6]. This means that the team cannot natively use some parts of the ecosystem. These includes simulators like gazebo[7], airframes[8], autonomous flight modes[9], existing controllers and navigation algorithms with RC control[10]. This constraint had to be taken into account when designing the system.

There are two major ways of extending the functionality of a vehicle running PX4[11]: Modifying the firmware to add features (called modules), or to use an offboard computer to control PX4. The team decided to use an offboard computer, as it seemed simpler, faster and more intuitive.

PX4 uses uORB[12], which is an asynchronous `publish()` / `subscribe()` messaging API used for inter-thread/inter-process communication. This protocol allows communication between the modules in PX4, and is a key part of the PX4 architecture, allowing it to be modular and extensible. The offboard computers send and receive messages from PX4 through a middleware connected to the uORB bus. This allows the offboard computer to act as an internal module of PX4, and to use the same messages as the rest of the system.

In order to send and receive uORB messages from the offboard computer, PX4 supports the MAVLink and ROS2 interfaces[13] (which uses uXRCE-DDS as a middleware). By default, some messages are not mapped through the MAVLink interface or ROS2, but this can be changed by modifying the PX4 firmware[14]. The PX4 firmware uses a mapping file to convert the uORB messages to uXRCE-DDS, which is converted to ROS2 messages on the offboard computer[15]. The team chose to use ROS2 interface for its powerful framework and large ecosystem of packages and libraries.

The offboard computer can send messages to PX4 at any chosen rate. But the rate of receiving messages depends on the rate of messages sent by PX4. It is possible to change some of the message rates in the PX4 firmware. More on this in https://docs.px4.io/main/en/concept/architecture.html#update-rates.

To validate if the demo works, the two options were to use a simulation (as it is used in PX4 offboard controller example[16]), or the actual hardware. The team decided to validate the demo in the actual hardware, as we already had the vehicle hardware in hand, and as mentioned in section 1, there's no native support for our drone in the simulation.

In the final product, the offboard computer should be aboard the drone but for convenience a laptop was used. This should not affect the conclusions taken from this report, as both the laptop and the Raspberry Pi (used on the final product) use the same Operating Systems and packages.

[3] https://px4.io/
[4] https://pixhawk.org/
[5] https://docs.px4.io/main/en/getting_started/px4_basic_concepts.html#drone-types
[6] https://docs.px4.io/main/en/airframes/
[7] https://docs.px4.io/main/en/sim_gazebo_gz/vehicles.html
[8] https://docs.px4.io/main/en/airframes/airframe_reference.html
[9] https://docs.px4.io/main/en/flight_modes/
[10] https://docs.px4.io/main/en/concept/architecture.html#flight-stack
[11] https://docs.px4.io/main/en/concept/px4_systems_architecture.html
[12] https://docs.px4.io/main/en/middleware/uorb.html
[13] https://docs.px4.io/main/en/robotics/
[14] https://docs.px4.io/main/en/middleware/uorb.html#adding-a-new-topic
[15] https://docs.px4.io/main/en/middleware/uxrce_dds.html#supported-uorb-messages
[16] https://docs.px4.io/main/en/ros2/offboard_control.html

In summary, the objectives of this demonstration are to:

- Use real hardware
- Use ROS2 to control PX4
- Read sensor data
- Send servo and motor PWM signals

This demo was accomplished by using the assembled TVC mechanism, with 2 counter-rotating motors and 2 servos controlling the outer ring and inner beam, the PX4 software running on PixHawk 6c mini, and an offboard computer which in this case was a Laptop running Linux.

3 Architecture

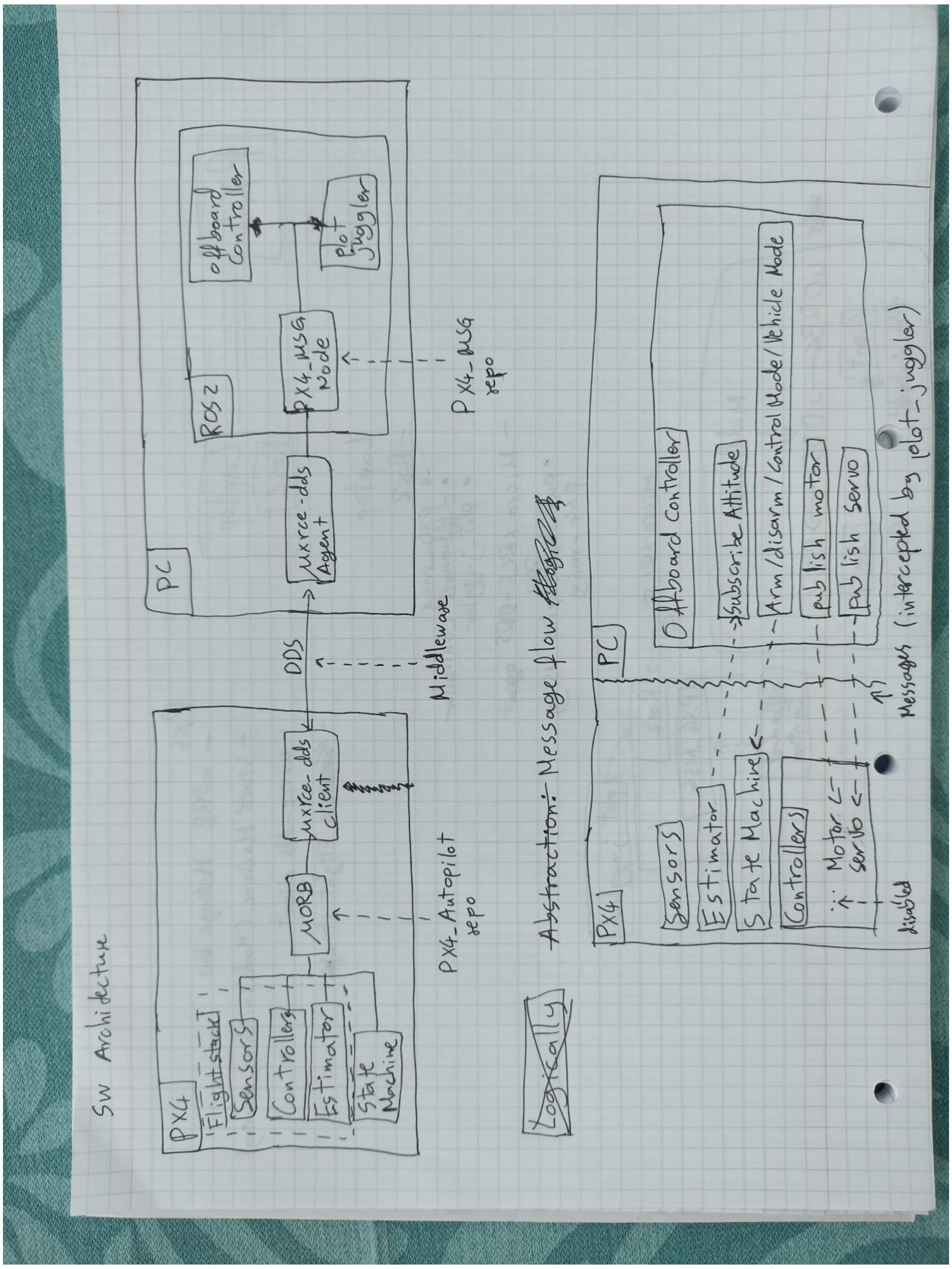


Figure 1: Software Architecture

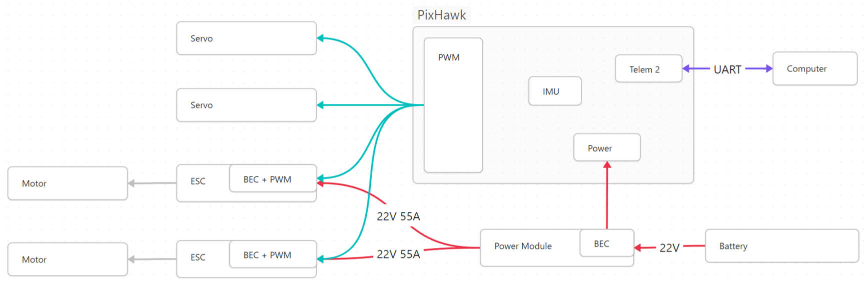


Figure 2: Hardware Block Diagram

The system is composed of two main components: the PX4 flight controller and the offboard computer. The PX4 flight controller is responsible for the hardware drivers, sensors readings, and position and attitude estimation. The offboard computer is responsible for using the sensor data to actuate the servos and motors.

3.1 PX4

To enable offboard mode, PX4's internal state machine mode has to be changed to offboard (using `VehicleCommand` Message).

In addition, the offboard computer has to send a heartbeat message (`OffboardControlMode`) to PX4 with a frequency of, at least, 2 Hz, so that it knows that the offboard computer is connected. Additionally, it enables offboard control only after receiving the signal for more than a second, and will regain control if the signal stops, which will switch to failsafe mode, currently configured to disarm the vehicle. This behaviour is part of PX4's safety features. This is explain in greater detail in https://docs.px4.io/main/en/flight_modes/offboard.html.

To control the servos and motors, the offboard computer has to send a message (`OffboardControlMode`), informing the level of the PX4 control architecture at which offboard setpoints will be injected. In this case, only direct actuation is enabled. Another message (`VehicleControlMode`) has to be sent to disable (i.e. bypass) the default controller algorithms. In this case, all default controllers are disabled, and offboard control is enabled.

desired control quantity	position field	velocity field	acceleration field	attitude field	body_rate field	thrust_and_torque field	direct_actuator field	required estimate	required message
position (NED)	✓	-	-	-	-	-	-	position	TrajectorySetpoint
velocity (NED)	✗	✓	-	-	-	-	-	velocity	TrajectorySetpoint
acceleration (NED)	✗	✗	✓	-	-	-	-	velocity	TrajectorySetpoint
attitude (FRD)	✗	✗	✗	✓	-	-	-	none	VehicleAttitudeSetpoint
body_rate (FRD)	✗	✗	✗	✗	✓	-	-	none	VehicleRatesSetpoint
thrust and torque (FRD)	✗	✗	✗	✗	✗	✓	-	none	VehicleThrustSetpoint and VehicleTorqueSetpoint
direct motors and servos	✗	✗	✗	✗	✗	✗	✓	none	ActuatorMotors and ActuatorServos

where ✓ means that the bit is set, ✗ means that the bit is not set and - means that the bit is value is irrelevant.

Figure 3: OffboardControlMode Message Description

Afterwards, the offboard computer can send the desired PWM values to the motors (`ActuatorMotors`) and servos (`ActuatorServos`). This is also explained in greater detail in https://docs.px4.io/main/en/flight_modes/offboard.html#ros-2-messages.

Due to PX4's safety features, the motors are only enabled when the vehicle is armed[17]. This menas that the offboard computer has to arm the vehicle (`VehicleCommand`) before sending the PWM values to the motor.

All of these messages are part of the uORB communication bus, and are mapped to ROS2 messages[18]. The section 5 describes the uORB messages used in this demo.

3.2 uXRCE-DDS

To connect PX4 to ROS2, PX4 uses uXRCE-DDS, which is a lightweight implementation of DDS. This acts as a middleware between the uORB, used between modules in PX4, and the ROS2 topics, used between ROS2 nodes. To use uXRCE-DDS, a uXRCE-DDS client runs in PX4, and a uXRCE-DDS agent runs in the offboard computer. This is shown in the diagram below:

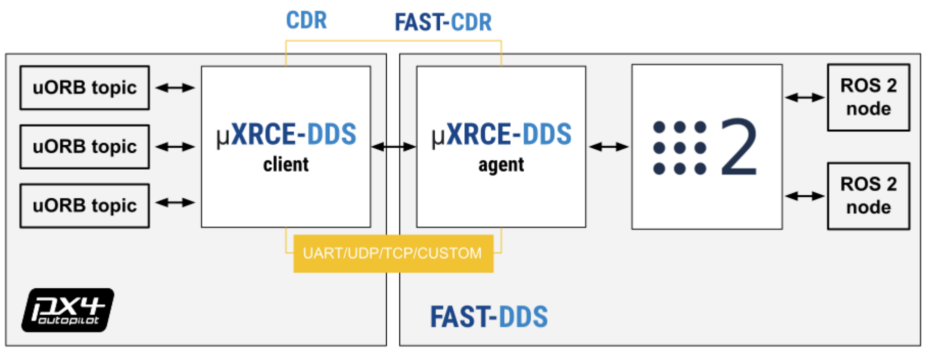


Figure 4: Software architecture connecting PX4 to ROS2 using uXRCE-DDS

[17] https://docs.px4.io/main/en/advanced_config/prearm_arm_disarm.html#arm-disarm-prearm-configuration
[18] https://docs.px4.io/main/en/msg_docs/

In our case, the connection between client and agent is done using a serial connection. This is also explained in greater detail in https://docs.px4.io/main/en/middleware/uxrce_dds.html.

3.3 ROS2

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications[19]. ROS 2 is a middleware based on a strongly-typed, anonymous publish/subscribe mechanism that allows for message passing between different processes. At the heart of any ROS 2 system is the ROS graph. The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate.

The offboard computer ROS2 workspace, which is a set of packages that can be built and run together, available in https://github.com/PedromcaMartins/e-rocket.

The demo workspace uses 3 packages:

- `px4_msgs\footnote{\url{https://github.com/PX4/px4_msgs}}`: This package contains a node that connects to the uXRCE-DDS agent, and subsctibes to the PX4 uORB messages, exposing ROS2 topics that converting the PX4 messages to and from ROS2 messages.
- `offboard_control`: This package contains the offboard control node that subscribes to the sensor messages, and publishes the servo and motor messages.
- `plot_juggler\footnote{\url{https://plotjuggler.io/}}`: This package is used to visualize ROS2 topics and messages in real time, with the help of a GUI.

[19] https://www.ros.org/

4 Setup

4.1 PX4

Before being able to test this demo, a number of configurations are required. Most of these steps follow the official PX4 documentation, and are not specific to this project, available in `https://docs.px4.io/main/en/dev_setup/getting_started.html`.

4.1.1 Firmware

The first step when using hardware like PixHawk, which runs the PX4 software, is to flash the PX4 firmware. The team chose the latest stable version, version v1.15.2[20].

4.1.2 Airframe

After installing firmware you need to select a vehicle type and frame configuration. This applies appropriate initial parameter values for the selected frame, such as the vehicle type, number of motors, relative motor position, and so on. The team followed the official tutorial[21].

The team started by using a baloon frame, because the vehicle is uncontrolled (baloon frame has disabled the native flight controller algorithms). This turned out to not be unfeasable, as this frame also disables the ability to configure servos, which our system requires. Instead, the team used the generic quadcopter frame[22], with a custom generic geometry[23].

The team always intends to control this system using the offboard computer. This means that the relative motor positions configured in the geometry, which are used for PX4 controller allocation don't need to be configured. Additionally, as mentioned in Section 1, the PX4 existing controllers don't support the TVC mechanism, and by extension, our system.

4.1.3 Actuator setup

Using a generic geometry, the 2 motors were configured to the outputs PWM AUX (FMU Board output) 3 and 4, and the 2 servos were configured to the outputs PWM MAIN (I/O Board output) 5 and 6.

Note that for controlling motors, PWM AUX outputs are preferred over the PWM MAIN outputs (as they have lower latency)[24].

4.1.4 uXRCE-DDS client setup

The team followed the official tutorial[25]. `TELEM2` Serial Port was the port used by the client. It uses the uart protocol configured to 115200 baud rate, 8 data bits, no parity, and 1 stop bit.

4.2 Offboard computer

4.2.1 uXRCE-DDS agent setup

The team followed the official tutorial, and configured the agent to connect to the serial port where PixHaw was connected (`/dev/ttyACM0`)[26].

4.2.2 ROS2 setup

The team followed the official tutorial, installing ROS2 humble edition on Ubuntu 20.04[27].

[20] https://github.com/PX4/PX4-Autopilot/releases/tag/v1.15.2
[21] https://docs.px4.io/main/en/config/airframe.html
[22] https://docs.px4.io/main/en/airframes/airframe_reference.html#quadrotor-x
[23] https://docs.px4.io/main/en/config/actuators.html#geometry
[24] https://docs.px4.io/main/en/config/actuators.html#actuator-outputs
[25] https://docs.px4.io/main/en/middleware/uxrce_dds.html#starting-the-client
[26] https://docs.px4.io/main/en/middleware/uxrce_dds.html#starting-the-agent
[27] https://docs.px4.io/main/en/ros2/user_guide.html#installation-setup

5 Demo

This demo aligns the TVC mount with the vehicle's attitude, using the servos, and actuates the motors using a sinusoidal waveform out of sync (with the two motors). This demo is heavily inspired by the official PX4 ROS2 examples[28].

The following uORB messages are used in the demo:

- `ActuatorMotors`[29]: Sets PWM values of motors.
- `ActuatorServos`[30]: Sets PWM values of servos.
- `VehicleAttitude`[31]: Gets the attitude data as a quaternion, from the attitude estimator. The attitude estimator uses the Extended Kalman Filter (EKF) present in PX4.
- `VehicleCommand`[32]: Sends commands to PX4, specifically Arming, Disarming, and change mode to Offboard.
- `VehicleControlMode`[33]: Sets the control algoritms to be used by PX4. In this case, it disables all control algorithms, except for the offboard control.
- `OffboardControlMode`[34]: Sets the control mode the offboard computer will use, specifically direct actuation. It also serves as the hearbeat message to remain in Offboard mode.

5.1 Demo Code

The code consists of a ROS2 node that contains the logic of the demo. It includes the subscribers / publishers for the PX4 Messages, auxiliary functions and variables. There are two main components: the constructor and the callbacks. The full code can be found as part of the appendix in 8.

The constructor of the ROS2 node is executed once, when the node is created (using `spin` function in main).

The code includes two main callbacks, which get called to respond to specific events. These callbacks include:

- The subscriber to the `VehicleAttitude` message, that is called when a new message is received. It uses the attitude data to control the servos, and actuates the motors.
- A timer callback function, called every 100ms, that changes the mode of the vehicle, arms and disarms the vehicle, and sends the heartbeat message.

5.1.1 Class OffboardControl

The class called `OffboardControl` which extends the `Node` class, is the main class of the demo. It contains the variables and functions used.

The function `OffboardControl()` is the constructor of the class, which initializes the node, the publishers, subscribers subscribers, and both callbacks: the timer and the attitude subscriber.

```
/**
 * @brief Demo Node for offboard control using actuator servos and attitude
     readings
 */
class OffboardControl : public rclcpp::Node
```

[28] https://github.com/PX4/px4_ros_com
[29] https://docs.px4.io/main/en/msg_docs/ActuatorMotors.html
[30] https://docs.px4.io/main/en/msg_docs/ActuatorServos.html
[31] https://docs.px4.io/main/en/msg_docs/VehicleAttitude.html
[32] https://docs.px4.io/main/en/msg_docs/VehicleCommand.html
[33] https://docs.px4.io/main/en/msg_docs/VehicleControlMode.html
[34] https://docs.px4.io/main/en/msg_docs/OffboardControlMode.html

```cpp
{
private:
    rclcpp::TimerBase::SharedPtr timer_;

    //!< Publishers and Subscribers
    rclcpp::Publisher<ActuatorMotors>::SharedPtr actuator_motors_publisher_;
    rclcpp::Publisher<ActuatorServos>::SharedPtr actuator_servos_publisher_;
    rclcpp::Publisher<VehicleCommand>::SharedPtr vehicle_command_publisher_;
    rclcpp::Publisher<OffboardControlMode>::SharedPtr
        offboard_control_mode_publisher_;
    rclcpp::Publisher<VehicleControlMode>::SharedPtr
        vehicle_control_mode_publisher_;
    rclcpp::Subscription<VehicleAttitude>::SharedPtr
        vehicle_attitude_subscription_;

    uint64_t timer_callback_iteration_ = 0;   //!< counter for the number of
        setpoints sent

    bool is_offboard_mode_ = false; //!< flag to check if the vehicle is in
        offboard mode

    std::atomic<float> inner_beam_pwm_;   //!< inner beam servo position
    std::atomic<float> outer_ring_pwm_;   //!< outer ring servo position

    //!< Auxiliary functions
    void quaternionToEuler(const Quaternion& q, float& roll, float& pitch,
        float& yaw);
    void publish_actuator_servos();
    void publish_actuator_motors();

    void publish_offboard_control_mode();
    void publish_vehicle_control_mode();
    void publish_vehicle_command(uint16_t command, float param1 = 0.0, float
        param2 = 0.0);

public:
    explicit OffboardControl() : Node("offboard_control")
```

Listing 1: OffboardControl constructor

5.1.2 Attitude subscriber callback

In the code, the function is implemented as an anonymous function, but for convenience, it is shown here as a normal function.

```cpp
            [this](const VehicleAttitude::SharedPtr msg) {
                // Process the vehicle attitude message
                Quaternion q;
                q.w = msg->q[0];
                q.x = msg->q[1];
                q.y = msg->q[2];
                q.z = msg->q[3];

                float roll, pitch, yaw;
                quaternionToEuler(q, roll, pitch, yaw);

                // Map roll and pitch from [-45, 45] to [-1, 1] constrained to
                    [-1, 1]
```

```
                inner_beam_pwm_.store(std::max(-1.0f, std::min(1.0f, roll / 45.0f)), std::memory_order_relaxed);
                outer_ring_pwm_.store(std::max(-1.0f, std::min(1.0f, pitch / 45.0f)), std::memory_order_relaxed);

                if (is_offboard_mode_) {
                    publish_actuator_servos();
                    publish_actuator_motors();
                }
            }
```

Listing 2: Attitude subscriber callback

The `outer_ring_pwm_` and `inner_beam_pwm_` are atomic variables that store the pwm values for the servos connected to the outer ring and inner beam, respectively. Atomic variables are used for they are shared between threads, and need to be accessed in a thread-safe manner.

The servo control message accepts values in the range [-1, 1], -1 being the maximum negative position, and 1 the maximum positive[35]. The PWM values come from the roll and pitch angles and are mapped with a range [-45, 45] to [-1, 1], but constrained to [-1, 1]. This is done to ensure that the values are within the range of the servos.

If the vehicle is in offboard mode, the servo and motor messages are published.

The `publish_actuator_servos()` function uses the pwm values, and publishes them to the servo topic.

The `publish_actuator_motors()` function uses a sinusoidal wave to send two sinusoidal curves out of phase to the motors.

5.1.3 State Machine / Timer callback

This function is responsible for armind, desarming, and changing the modes of the PX4 flight controller, acting as the offboard computer's state machine.

The offboard control message serves as the heartbeat message. It has to be sent with a frequency of at least 2 Hz, otherwise PX4 will switch to the set failsafe mode[36].

```
        auto timer_callback = [this]() {
            // PX4 will switch out of offboard mode if the stream rate of
            // OffboardControlMode messages drops below approximately 2Hz
            publish_offboard_control_mode();

            // PX4 requires that the vehicle is already receiving OffboardControlMode messages
            // before it will arm in offboard mode,
            // or before it will switch to offboard mode when flying
            if (timer_callback_iteration_ == 15) {
                // Change to Offboard mode
                this->publish_vehicle_command(VehicleCommand::VEHICLE_CMD_DO_SET_MODE, 1, 6);
                RCLCPP_INFO(this->get_logger(), "Offboard mode command send");

                // Confirm that we are in offboard mode
                is_offboard_mode_ = true;
                RCLCPP_INFO(this->get_logger(), "Offboard mode confirmed");

                // Arm the vehicle
```

[35] https://docs.px4.io/main/en/msg_docs/ActuatorServos.html

[36] https://docs.px4.io/main/en/flight_modes/offboard.html

```cpp
                this->publish_vehicle_command(VehicleCommand::VEHICLE_CMD_COMPONENT_ARM_DISARM, 1.0);
                RCLCPP_INFO(this->get_logger(), "Arm command send");

                // change the vehicle control mode
                this->publish_vehicle_control_mode();
                RCLCPP_INFO(this->get_logger(), "Vehicle control mode command send");
            }

            // disarm the vehicle
            if (timer_callback_iteration_ == 300) {
                // Disarm the vehicle
                this->publish_vehicle_command(VehicleCommand::VEHICLE_CMD_COMPONENT_ARM_DISARM, 0.0);
                RCLCPP_INFO(this->get_logger(), "Disarm command send");

                // change to Manual mode
                this->publish_vehicle_command(VehicleCommand::VEHICLE_CMD_DO_SET_MODE, 1, 1);
                RCLCPP_INFO(this->get_logger(), "Manual mode command send");

                is_offboard_mode_ = false;
                RCLCPP_INFO(this->get_logger(), "Offboard mode disabled");
            }
            timer_callback_iteration_++;
        };

        timer_ = this->create_wall_timer(100ms, timer_callback);
```

Listing 3: State machine / Timer callback

The variable `timer_callback_iteration_` is used to count the number of iterations of the timer callback. It also controls the duration of the demo.

The function `RCLCPP_INFO` is used to log messages to the console.

5.1.4 OffboardControlMode Message

The message `OffboardControlMode`[37] is used to set the control mode of the vehicle. It is sent with the following values:

```cpp
    OffboardControlMode msg{};
    msg.position = false;
    msg.velocity = false;
    msg.acceleration = false;
    msg.attitude = false;
    msg.body_rate = false;
    msg.thrust_and_torque = false;
    msg.direct_actuator = true;
    msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
    offboard_control_mode_publisher_->publish(msg);
```

Listing 4: OffboardControlMode message

The values were chosen according to the table in https://docs.px4.io/main/en/flight_modes/offboard.html#ros-2-messages.

[37] https://docs.px4.io/main/en/msg_docs/OffboardControlMode.html

5.1.5 VehicleCommand Message

The message `VehicleControlMode`[38] is used to set the control mode of the vehicle. It is sent with the following values:

```
    VehicleControlMode msg{};
    msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
    msg.flag_armed = true;

    msg.flag_multicopter_position_control_enabled = false;

    msg.flag_control_manual_enabled = false;
    msg.flag_control_auto_enabled = false;
    msg.flag_control_offboard_enabled = true;
    msg.flag_control_position_enabled = false;
    msg.flag_control_velocity_enabled = false;
    msg.flag_control_altitude_enabled = false;
    msg.flag_control_climb_rate_enabled = false;
    msg.flag_control_acceleration_enabled = false;
    msg.flag_control_attitude_enabled = false;
    msg.flag_control_rates_enabled = false;
    msg.flag_control_allocation_enabled = false;
    msg.flag_control_termination_enabled = false;

    msg.source_id = 1;

    vehicle_control_mode_publisher_->publish(msg);
```

Listing 5: VehicleControlMode message

Since all control is done by the offboard computer, all control modes are disabled, except for the offboard control.

5.1.6 Summary

In summary, the demo code consists of a ROS2 node that interacts with PX4 through various messages. The timer callback functions as a state machine, sending heartbeat messages and managing the vehicle's mode and arm status according to a predefined sequence. The attitude subscriber captures the vehicle's orientation from sensor data, processes it, and uses it to control servo positions and motor speeds when in offboard mode.

The main components work together in the following sequence:

1. The main function initializes the ROS2 node and creates an instance of the `OffboardControl` class, initializing all publishers and subscribers.
2. The timer callback continuously sends heartbeat messages every 100ms.
3. After 15 iterations (1.5 seconds), the vehicle is switched to offboard mode and armed.
4. While armed, the attitude subscriber processes sensor data and controls the servos and motors in real-time.
5. After 300 iterations (30 seconds), the vehicle is disarmed and returned to manual mode.

This architecture is capable of demonstrating that an offboard computer can successfully read PX4 sensor data and control actuators, fulfilling the demo's objectives.

[38] https://docs.px4.io/main/en/msg_docs/VehicleControlMode.html

6 Results

The demo was successfully executed. One execution of the demo was recorded and is available in https://www.youtube.com/shorts/j01qkGAldi8.

6.1 Attitude

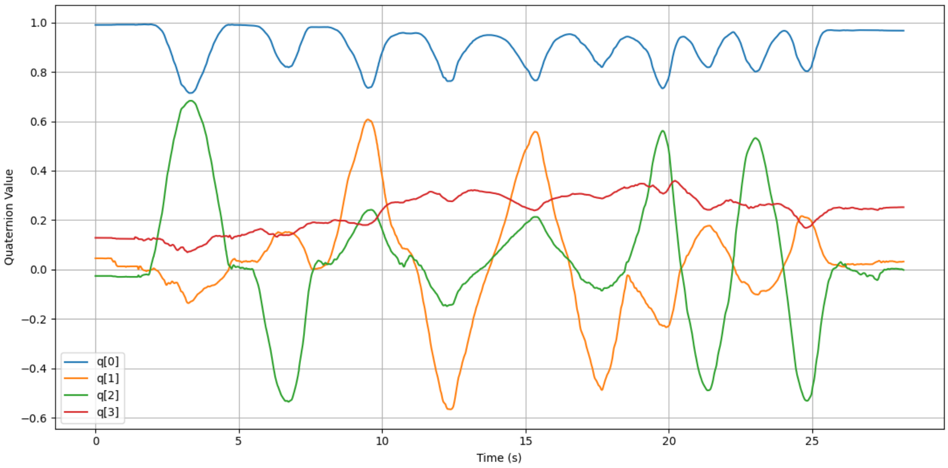
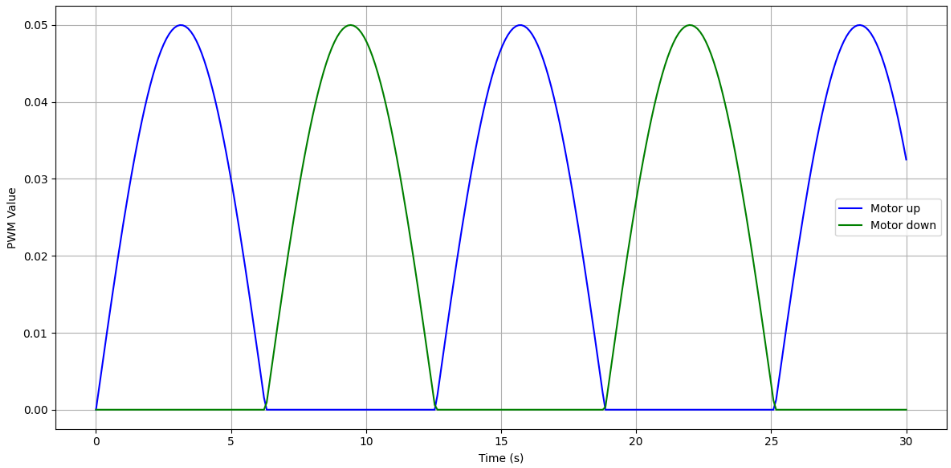


Figure 5: Quaternion Components Over Time

As we can see, the quaternion values show the change in attitude of the vehicle.

The following figures show plots from the same demo.

6.2 Motors



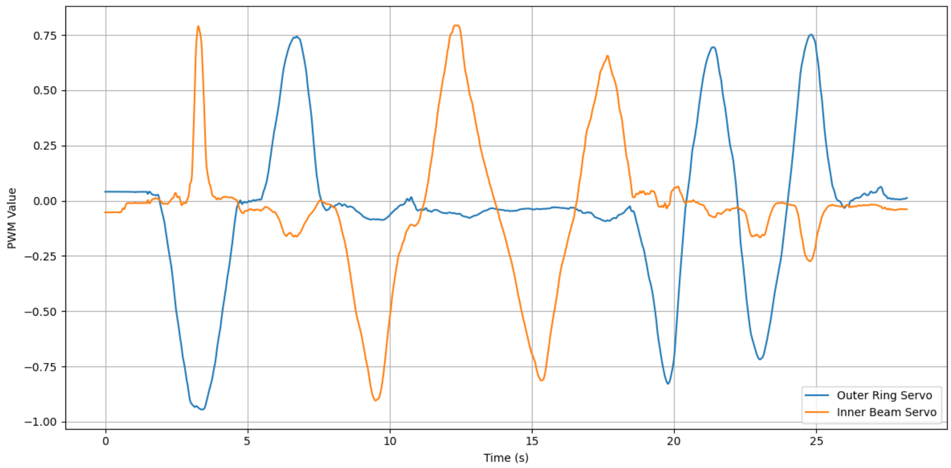

(a)

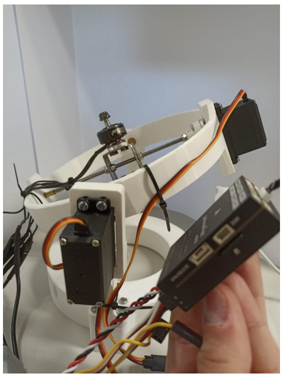

(b)

Figure 6: Upper and Bottom motors spinning independently, following the sinusoidal waveform

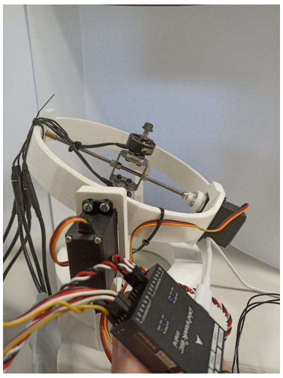
This validates that the motors are spinning independently, and following the sinusoidal waveform.

6.3 Servos

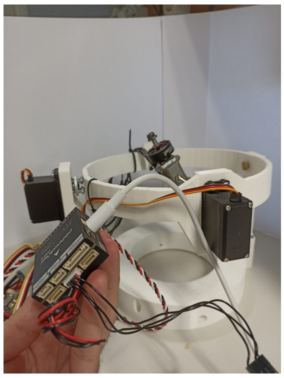




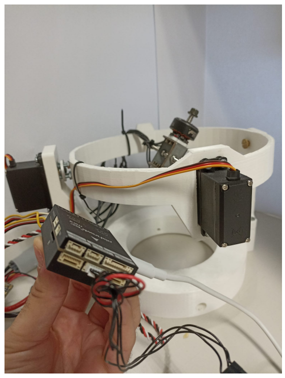
(a)



(b)



(c)



(d)

Figure 7: Outer Ring and Inner Beam corresponding to the PX4 attitude

The servos are following the attitude of the vehicle, and are able to control the TVC mechanism.

7 Conclusion

In this report we have demonstrated that an offboard computer running ROS 2 can reliably interface with a PX4 flight controller over uXRCE-DDS to read real-time attitude estimates and command both servos and motors. By implementing a simple state machine and heartbeat protocol, the offboard node successfully:

- Switched the flight controller into Offboard mode and armed/disarmed it.
- Subscribed to the VehicleAttitude uORB topic and converted quaternion measurements into servo setpoints.
- Published ActuatorServos and ActuatorMotors messages to drive a thrust-vectoring mechanism.

Experimental results—including live video, PWM logs, and attitude plots—confirm that the servos tracked the vehicle's roll/pitch angles within the expected range and the two counter-rotating motors followed out-of-phase commands as intended. This validates the feasibility of using PX4 + ROS 2 offboard control for custom actuation schemes such as TVC.

Future work will extend this framework to closed-loop control by feeding sensor feedback directly into guidance algorithms and creating a state machine that's syncronized with PX4. Moreover, further safety features (pre-arm checks, failsafe transitions) and higher-frequency control loops may be added to meet more demanding control objectives.

Overall, this demonstration fulfills the primary objective of proving end-to-end offboard control of PX4 hardware.

8 Appendix

```cpp
#include <rclcpp/rclcpp.hpp>
#include <px4_msgs/msg/actuator_motors.hpp>
#include <px4_msgs/msg/actuator_servos.hpp>
#include <px4_msgs/msg/vehicle_command.hpp>
#include <px4_msgs/msg/offboard_control_mode.hpp>
#include <px4_msgs/msg/vehicle_control_mode.hpp>
#include <px4_msgs/msg/vehicle_attitude.hpp>

#include <chrono>

using namespace std::chrono;
using namespace px4_msgs::msg;

struct Quaternion {
    float w, x, y, z;
};

/**
 * @brief Demo Node for offboard control using actuator servos and attitude readings
 */
class OffboardControl : public rclcpp::Node
{
private:
    rclcpp::TimerBase::SharedPtr timer_;

    //!< Publishers and Subscribers
    rclcpp::Publisher<ActuatorMotors>::SharedPtr actuator_motors_publisher_;
    rclcpp::Publisher<ActuatorServos>::SharedPtr actuator_servos_publisher_;
    rclcpp::Publisher<VehicleCommand>::SharedPtr vehicle_command_publisher_;
    rclcpp::Publisher<OffboardControlMode>::SharedPtr offboard_control_mode_publisher_;
    rclcpp::Publisher<VehicleControlMode>::SharedPtr vehicle_control_mode_publisher_;
    rclcpp::Subscription<VehicleAttitude>::SharedPtr vehicle_attitude_subscription_;

    uint64_t timer_callback_iteration_ = 0;   //!< counter for the number of setpoints sent

    bool is_offboard_mode_ = false; //!< flag to check if the vehicle is in offboard mode

    std::atomic<float> inner_beam_pwm_;   //!< inner beam servo position
    std::atomic<float> outer_ring_pwm_;   //!< outer ring servo position

    //!< Auxiliary functions
    void quaternionToEuler(const Quaternion& q, float& roll, float& pitch, float& yaw);
    void publish_actuator_servos();
    void publish_actuator_motors();

    void publish_offboard_control_mode();
    void publish_vehicle_control_mode();
    void publish_vehicle_command(uint16_t command, float param1 = 0.0, float
```

```
            param2 = 0.0);

public:
    explicit OffboardControl() : Node("offboard_control")
    {
        actuator_motors_publisher_ = this->create_publisher<ActuatorMotors>("/fmu/in/actuator_motors", 10);
        actuator_servos_publisher_ = this->create_publisher<ActuatorServos>("/fmu/in/actuator_servos", 10);
        vehicle_command_publisher_ = this->create_publisher<VehicleCommand>("/fmu/in/vehicle_command", 10);
        offboard_control_mode_publisher_ = this->create_publisher<OffboardControlMode>("/fmu/in/offboard_control_mode", 10);
        vehicle_control_mode_publisher_ = this->create_publisher<VehicleControlMode>("/fmu/in/vehicle_control_mode", 10);

        inner_beam_pwm_.store(0.0, std::memory_order_relaxed);
        outer_ring_pwm_.store(0.0, std::memory_order_relaxed);

        rmw_qos_profile_t qos_profile = rmw_qos_profile_sensor_data;
        auto qos = rclcpp::QoS(rclcpp::QoSInitialization(qos_profile.history, 5), qos_profile);

        vehicle_attitude_subscription_ = this->create_subscription<VehicleAttitude>("/fmu/out/vehicle_attitude", qos,
            [this](const VehicleAttitude::SharedPtr msg) {
                // Process the vehicle attitude message
                Quaternion q;
                q.w = msg->q[0];
                q.x = msg->q[1];
                q.y = msg->q[2];
                q.z = msg->q[3];

                float roll, pitch, yaw;
                quaternionToEuler(q, roll, pitch, yaw);

                // Map roll and pitch from [-45, 45] to [-1, 1] constrained to [-1, 1]
                inner_beam_pwm_.store(std::max(-1.0f, std::min(1.0f, roll / 45.0f)), std::memory_order_relaxed);
                outer_ring_pwm_.store(std::max(-1.0f, std::min(1.0f, pitch / 45.0f)), std::memory_order_relaxed);

                if (is_offboard_mode_) {
                    publish_actuator_servos();
                    publish_actuator_motors();
                }
            }
        );

        auto timer_callback = [this]() {
            // PX4 will switch out of offboard mode if the stream rate of
            // OffboardControlMode messages drops below approximately 2Hz
            publish_offboard_control_mode();

            // PX4 requires that the vehicle is already receiving OffboardControlMode messages
```

```
            // before it will arm in offboard mode,
            // or before it will switch to offboard mode when flying
            if (timer_callback_iteration_ == 15) {
                // Change to Offboard mode
                this->publish_vehicle_command(VehicleCommand::
                    VEHICLE_CMD_DO_SET_MODE, 1, 6);
                RCLCPP_INFO(this->get_logger(), "Offboard mode command send");

                // Confirm that we are in offboard mode
                is_offboard_mode_ = true;
                RCLCPP_INFO(this->get_logger(), "Offboard mode confirmed");

                // Arm the vehicle
                this->publish_vehicle_command(VehicleCommand::
                    VEHICLE_CMD_COMPONENT_ARM_DISARM, 1.0);
                RCLCPP_INFO(this->get_logger(), "Arm command send");

                // change the vehicle control mode
                this->publish_vehicle_control_mode();
                RCLCPP_INFO(this->get_logger(), "Vehicle control mode command
                    send");
            }

            // disarm the vehicle
            if (timer_callback_iteration_ == 300) {
                // Disarm the vehicle
                this->publish_vehicle_command(VehicleCommand::
                    VEHICLE_CMD_COMPONENT_ARM_DISARM, 0.0);
                RCLCPP_INFO(this->get_logger(), "Disarm command send");

                // change to Manual mode
                this->publish_vehicle_command(VehicleCommand::
                    VEHICLE_CMD_DO_SET_MODE, 1, 1);
                RCLCPP_INFO(this->get_logger(), "Manual mode command send");

                is_offboard_mode_ = false;
                RCLCPP_INFO(this->get_logger(), "Offboard mode disabled");
            }
            timer_callback_iteration_++;
        };

        timer_ = this->create_wall_timer(100ms, timer_callback);
    }
};

void OffboardControl::quaternionToEuler(const Quaternion& q, float& roll, float
    & pitch, float& yaw) {
    // Roll (x-axis rotation)
    float sinr_cosp = 2 * (q.w * q.x + q.y * q.z);
    float cosr_cosp = 1 - 2 * (q.x * q.x + q.y * q.y);
    roll = std::atan2(sinr_cosp, cosr_cosp) * 180.0 / M_PI;

    // Pitch (y-axis rotation)
    float sinp = 2 * (q.w * q.y - q.z * q.x);
    if (std::abs(sinp) >= 1)
        pitch = std::copysign(90.0, sinp); // Use 90 degrees if out of range
    else
```

```cpp
        pitch = std::asin(sinp) * 180.0 / M_PI;

    // Yaw (z-axis rotation)
    float siny_cosp = 2 * (q.w * q.z + q.x * q.y);
    float cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z);
    yaw = std::atan2(siny_cosp, cosy_cosp) * 180.0 / M_PI;
}

/**
 * @brief Publish the actuator motors.
 *        For this example, we are generating sinusoidal values for the
     actuator positions.
 */
void OffboardControl::publish_actuator_servos()
{
    static double time = 0.0;
    ActuatorServos msg{};

    msg.control[0] = outer_ring_pwm_.load();
    msg.control[1] = -inner_beam_pwm_.load();

    msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
    actuator_servos_publisher_->publish(msg);

    time += 0.1; // Increment time for the next wave
}

/**
 * @brief Publish the actuator motors.
 *        For this example, we are generating sinusoidal values for the
     actuator positions.
 */
void OffboardControl::publish_actuator_motors()
{
    static double time = 0.0;
    ActuatorMotors msg{};

    // Generate sinusoidal values for actuator positions
    float sin_wave = 0.05f * (sin(time / 10.0f)); // Sinusoidal wave between 0
        and 0.10
    float inv_sin_wave = -sin_wave;

    if (sin_wave < 0.0f) {
        sin_wave = NAN;
    }
    if (inv_sin_wave < 0.0f) {
        inv_sin_wave = NAN;
    }

    msg.control[0] = sin_wave;
    msg.control[1] = inv_sin_wave;

    msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
    actuator_motors_publisher_->publish(msg);

    time += 0.1; // Increment time for the next wave
}
```

```
/**
 * @brief Publish the offboard control mode.
 *        For this example, only direct actuator is active.
 */
void OffboardControl::publish_offboard_control_mode()
{
    OffboardControlMode msg{};
    msg.position = false;
    msg.velocity = false;
    msg.acceleration = false;
    msg.attitude = false;
    msg.body_rate = false;
    msg.thrust_and_torque = false;
    msg.direct_actuator = true;
    msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
    offboard_control_mode_publisher_->publish(msg);
}

/**
 * @brief Publish the vehicle control mode.
 *        For this example, we are setting the vehicle to offboard mode.
 */
void OffboardControl::publish_vehicle_control_mode()
{
    VehicleControlMode msg{};
    msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
    msg.flag_armed = true;

    msg.flag_multicopter_position_control_enabled = false;

    msg.flag_control_manual_enabled = false;
    msg.flag_control_auto_enabled = false;
    msg.flag_control_offboard_enabled = true;
    msg.flag_control_position_enabled = false;
    msg.flag_control_velocity_enabled = false;
    msg.flag_control_altitude_enabled = false;
    msg.flag_control_climb_rate_enabled = false;
    msg.flag_control_acceleration_enabled = false;
    msg.flag_control_attitude_enabled = false;
    msg.flag_control_rates_enabled = false;
    msg.flag_control_allocation_enabled = false;
    msg.flag_control_termination_enabled = false;

    msg.source_id = 1;

    vehicle_control_mode_publisher_->publish(msg);
}

/**
 * @brief Publish vehicle commands
 * @param command   Command code (matches VehicleCommand and MAVLink MAV_CMD codes)
 * @param param1    Command parameter 1
 * @param param2    Command parameter 2
 */
void OffboardControl::publish_vehicle_command(uint16_t command, float param1,
```

```
    float param2)
{
    VehicleCommand msg{};
    msg.param1 = param1;
    msg.param2 = param2;
    msg.command = command;
    msg.target_system = 1;
    msg.target_component = 1;
    msg.source_system = 1;
    msg.source_component = 1;
    msg.from_external = true;
    msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
    vehicle_command_publisher_->publish(msg);
}

int main(int argc, char *argv[])
{
    std::cout << "Starting offboard control node..." << std::endl;
    setvbuf(stdout, NULL, _IONBF, BUFSIZ);
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<OffboardControl>());

    rclcpp::shutdown();
    return 0;
}
```

Listing 6: OffboardControl.cpp