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 evaluate if we can 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 the team needs 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. To test if this is possible given the current setup, the team decided to start with a demonstration that read sensor data and used it to control the servos and motors.

[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 Micro-XRCE-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 Micro-XRCE-DDS, which is converted to ROS2 messages on the offboard computer[15]. The team chose to use ROS2 interface for its very powerful framework, and large ecosystem of packages and libraries.

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.

In summary, the objectives of this demonstration is to:

- Use real hardware
- Control servo and motor PWM signals.

[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

- Read sensor data, and be able to use it.

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

3 Architecture

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

In order to control PX4, PX4's internal state machine mode has to change to offboard (`VehicleCommand`).

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 1: OffboardControlMode Message Table

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.

[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/

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:

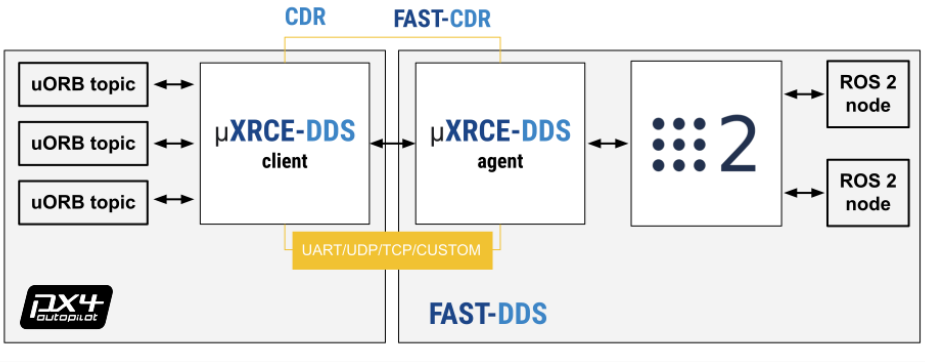

Figure 2: Software architecture connecting PX4 to ROS2 using uXRCE-DDS

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

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[20].

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[21], with a custom generic geometry[22].

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, 2 motors were configured to the outputs FMU Aux 3 and 4, and 2 servos were configured to the outputs ...

Note that PWM AUX outputs are preferred over the PWM MAIN outputs for controlling motors (as they have lower latency)[23].

4.1.4 Safety checks

...

4.1.5 uXRCE-DDS client setup

The team followed the official tutorial[24]. `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`)[25].

[20] https://docs.px4.io/main/en/config/airframe.html
[21] https://docs.px4.io/main/en/airframes/airframe_reference.html#quadrotor-x
[22] https://docs.px4.io/main/en/config/actuators.html#geometry
[23] https://docs.px4.io/main/en/config/actuators.html#actuator-outputs
[24] https://docs.px4.io/main/en/middleware/uxrce_dds.html#starting-the-client
[25] https://docs.px4.io/main/en/middleware/uxrce_dds.html#starting-the-agent

4.2.2 ROS2 setup

The team followed the official tutorial, installing ROS2 humble edition on Ubuntu 20.04[26].

4.2.3 PX4 ROS2 messages

[26] 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 PX4 ROS2 examples[27].

The following uORB messages are used in the demo:

- `ActuatorMotors`[28]: Sets PWM values of motors.
- `ActuatorServos`[29]: Sets PWM values of servos.
- `VehicleAttitude`[30]: Gets the attitude data as a quaternion, from the attitude estimator. The attitude estimator uses the Extended Kalman Filter (EKF) present in PX4.
- `VehicleCommand`[31]: Sends commands to PX4, specifically Arming, Disarming, and change mode to Offboard.
- `VehicleControlMode`[32]: Sets the control algoritms to be used by PX4. In this case, it disables all control algorithms, except for the offboard control.
- `OffboardControlMode`[33]: 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.

```
18 /**
19  * @brief Demo Node for offboard control using actuator servos and attitude
        readings
20  */
21 class OffboardControl : public rclcpp::Node
```

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

```
{
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> roll_;    //!< roll position for servos
    std::atomic<float> pitch_;    //!< pitch position for servos

    //!< 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.

```
            [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]
```

```
78                  roll_.store(std::max(-1.0f, std::min(1.0f, roll / 45.0f)), std
                        ::memory_order_relaxed);
79                  pitch_.store(std::max(-1.0f, std::min(1.0f, pitch / 45.0f)),
                        std::memory_order_relaxed);
80
81                  if (is_offboard_mode_) {
82                      publish_actuator_servos();
83                      publish_actuator_motors();
84                  }
85              }
```

Listing 2: Attitude subscriber callback

`roll_` and `pitch_` are atomic variables that store the roll and pitch values, 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[34]. The roll and pitch values are mapped from the 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 mapped roll and pitch 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[35].

```
88          auto timer_callback = [this]() -> void {
89              // PX4 will switch out of offboard mode if the stream rate of
90              // OffboardControlMode messages drops below approximately 2Hz
91              publish_offboard_control_mode();
92
93              // PX4 requires that the vehicle is already receiving
                    OffboardControlMode messages
94              // before it will arm in offboard mode,
95              // or before it will switch to offboard mode when flying
96              if (timer_callback_iteration_ == 15) {
97                  // Change to Offboard mode
98                  this->publish_vehicle_command(VehicleCommand::
                        VEHICLE_CMD_DO_SET_MODE, 1, 6);
99                  RCLCPP_INFO(this->get_logger(), "Offboard mode command send");
100
101                 // Confirm that we are in offboard mode
102                 is_offboard_mode_ = true;
103                 RCLCPP_INFO(this->get_logger(), "Offboard mode confirmed");
104
105                 // Arm the vehicle
106                 this->publish_vehicle_command(VehicleCommand::
                        VEHICLE_CMD_COMPONENT_ARM_DISARM, 1.0);
107                 RCLCPP_INFO(this->get_logger(), "Arm command send");
```

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

```
108
109                 // change the vehicle control mode
110                 this->publish_vehicle_control_mode();
111                 RCLCPP_INFO(this->get_logger(), "Vehicle control mode command
                        send");
112             }
113
114             // disarm the vehicle
115             if (timer_callback_iteration_ == 300) {
116                 // Disarm the vehicle
117                 this->publish_vehicle_command(VehicleCommand::
                        VEHICLE_CMD_COMPONENT_ARM_DISARM, 0.0);
118                 RCLCPP_INFO(this->get_logger(), "Disarm command send");
119
120                 // change to Manual mode
121                 this->publish_vehicle_command(VehicleCommand::
                        VEHICLE_CMD_DO_SET_MODE, 1, 1);
122                 RCLCPP_INFO(this->get_logger(), "Manual mode command send");
123
124                 is_offboard_mode_ = false;
125                 RCLCPP_INFO(this->get_logger(), "Offboard mode disabled");
126             }
127             timer_callback_iteration_++;
128         };
129
130         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 Publishers

The message `OffboardControlMode`[36] is used to set the control mode of the vehicle. It is sent with the following values:

```
206     OffboardControlMode msg{};
207     msg.position = false;
208     msg.velocity = false;
209     msg.acceleration = false;
210     msg.attitude = false;
211     msg.body_rate = false;
212     msg.thrust_and_torque = false;
213     msg.direct_actuator = true;
214     msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
215     offboard_control_mode_publisher_->publish(msg);
```

Listing 4: OffboardControlMode message

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

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

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

```
227
228     msg.flag_multicopter_position_control_enabled = false;
229
230     msg.flag_control_manual_enabled = false;
231     msg.flag_control_auto_enabled = false;
232     msg.flag_control_offboard_enabled = true;
233     msg.flag_control_position_enabled = false;
234     msg.flag_control_velocity_enabled = false;
235     msg.flag_control_altitude_enabled = false;
236     msg.flag_control_climb_rate_enabled = false;
237     msg.flag_control_acceleration_enabled = false;
238     msg.flag_control_attitude_enabled = false;
239     msg.flag_control_rates_enabled = false;
240     msg.flag_control_allocation_enabled = false;
241     msg.flag_control_termination_enabled = false;
242
243     msg.source_id = 1;
244
245     vehicle_control_mode_publisher_->publish(msg);
```

Listing 5: VehicleControlMode message

5.1.5 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 demonstrates that an offboard computer can successfully read PX4 sensor data and control actuators, fulfilling the demo's objectives.

6 Results

The demo was successfully executed. It was recorded and is available in . The following figures show the results of the demo.

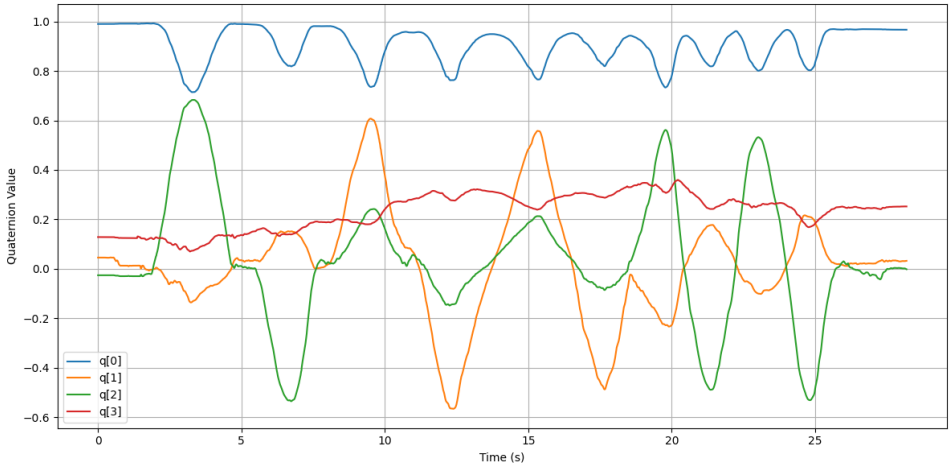

Figure 3: Quaternion Components Over Time

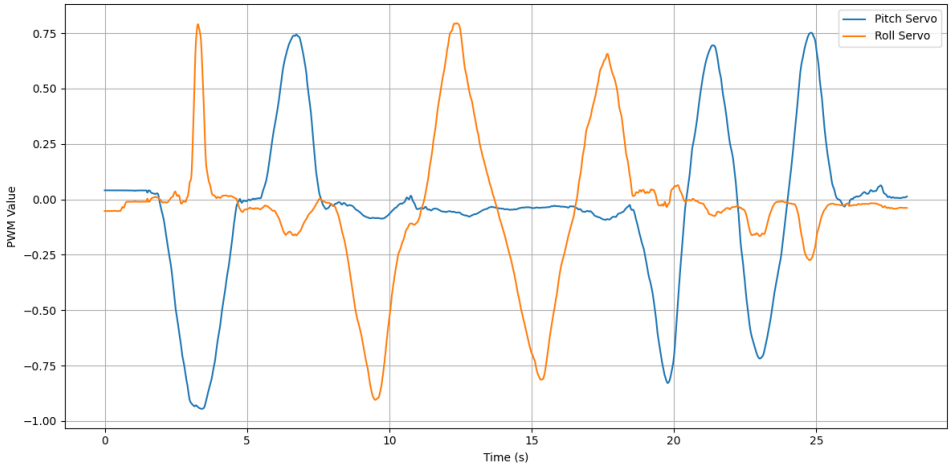

Figure 4: Servos PWM Over Time

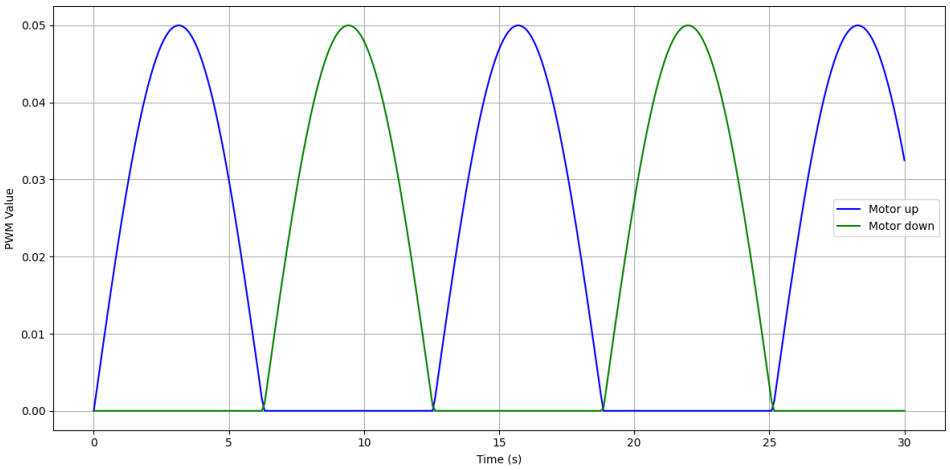

Figure 5: Motors PWM Over Time

7 Conclusion

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> roll_;    //!< roll position for servos
    std::atomic<float> pitch_;    //!< pitch position for servos

    //!< 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();
```

```
48      void publish_vehicle_command(uint16_t command, float param1 = 0.0, float
            param2 = 0.0);
49
50  public:
51      explicit OffboardControl() : Node("offboard_control")
52      {
53          actuator_motors_publisher_ = this->create_publisher<ActuatorMotors>("/
                fmu/in/actuator_motors", 10);
54          actuator_servos_publisher_ = this->create_publisher<ActuatorServos>("/
                fmu/in/actuator_servos", 10);
55          vehicle_command_publisher_ = this->create_publisher<VehicleCommand>("/
                fmu/in/vehicle_command", 10);
56          offboard_control_mode_publisher_ = this->create_publisher<
                OffboardControlMode>("/fmu/in/offboard_control_mode", 10);
57          vehicle_control_mode_publisher_ = this->create_publisher<
                VehicleControlMode>("/fmu/in/vehicle_control_mode", 10);
58
59          roll_.store(0.0, std::memory_order_relaxed);
60          pitch_.store(0.0, std::memory_order_relaxed);
61
62          rmw_qos_profile_t qos_profile = rmw_qos_profile_sensor_data;
63          auto qos = rclcpp::QoS(rclcpp::QoSInitialization(qos_profile.history,
                5), qos_profile);
64
65          vehicle_attitude_subscription_ = this->create_subscription<
                VehicleAttitude>("/fmu/out/vehicle_attitude", qos,
66              [this](const VehicleAttitude::SharedPtr msg) {
67                  // Process the vehicle attitude message
68                  Quaternion q;
69                  q.w = msg->q[0];
70                  q.x = msg->q[1];
71                  q.y = msg->q[2];
72                  q.z = msg->q[3];
73
74                  float roll, pitch, yaw;
75                  quaternionToEuler(q, roll, pitch, yaw);
76
77                  // Map roll and pitch from [-45, 45] to [-1, 1] constrained to
                        [-1, 1]
78                  roll_.store(std::max(-1.0f, std::min(1.0f, roll / 45.0f)), std
                        ::memory_order_relaxed);
79                  pitch_.store(std::max(-1.0f, std::min(1.0f, pitch / 45.0f)),
                        std::memory_order_relaxed);
80
81                  if (is_offboard_mode_) {
82                      publish_actuator_servos();
83                      publish_actuator_motors();
84                  }
85              }
86          );
87
88          auto timer_callback = [this]() -> void {
89              // PX4 will switch out of offboard mode if the stream rate of
90              // OffboardControlMode messages drops below approximately 2Hz
91              publish_offboard_control_mode();
92
```

```
93              // PX4 requires that the vehicle is already receiving
                    OffboardControlMode messages
94              // before it will arm in offboard mode,
95              // or before it will switch to offboard mode when flying
96              if (timer_callback_iteration_ == 15) {
97                  // Change to Offboard mode
98                  this->publish_vehicle_command(VehicleCommand::
                        VEHICLE_CMD_DO_SET_MODE, 1, 6);
99                  RCLCPP_INFO(this->get_logger(), "Offboard mode command send");
100
101                 // Confirm that we are in offboard mode
102                 is_offboard_mode_ = true;
103                 RCLCPP_INFO(this->get_logger(), "Offboard mode confirmed");
104
105                 // Arm the vehicle
106                 this->publish_vehicle_command(VehicleCommand::
                        VEHICLE_CMD_COMPONENT_ARM_DISARM, 1.0);
107                 RCLCPP_INFO(this->get_logger(), "Arm command send");
108
109                 // change the vehicle control mode
110                 this->publish_vehicle_control_mode();
111                 RCLCPP_INFO(this->get_logger(), "Vehicle control mode command
                        send");
112             }
113
114             // disarm the vehicle
115             if (timer_callback_iteration_ == 300) {
116                 // Disarm the vehicle
117                 this->publish_vehicle_command(VehicleCommand::
                        VEHICLE_CMD_COMPONENT_ARM_DISARM, 0.0);
118                 RCLCPP_INFO(this->get_logger(), "Disarm command send");
119
120                 // change to Manual mode
121                 this->publish_vehicle_command(VehicleCommand::
                        VEHICLE_CMD_DO_SET_MODE, 1, 1);
122                 RCLCPP_INFO(this->get_logger(), "Manual mode command send");
123
124                 is_offboard_mode_ = false;
125                 RCLCPP_INFO(this->get_logger(), "Offboard mode disabled");
126             }
127             timer_callback_iteration_++;
128         };
129
130         timer_ = this->create_wall_timer(100ms, timer_callback);
131     }
132 };
133
134 void OffboardControl::quaternionToEuler(const Quaternion& q, float& roll, float
        & pitch, float& yaw) {
135     // Roll (x-axis rotation)
136     float sinr_cosp = 2 * (q.w * q.x + q.y * q.z);
137     float cosr_cosp = 1 - 2 * (q.x * q.x + q.y * q.y);
138     roll = std::atan2(sinr_cosp, cosr_cosp) * 180.0 / M_PI;
139
140     // Pitch (y-axis rotation)
141     float sinp = 2 * (q.w * q.y - q.z * q.x);
142     if (std::abs(sinp) >= 1)
```

```
143         pitch = std::copysign(90.0, sinp); // Use 90 degrees if out of range
144     else
145         pitch = std::asin(sinp) * 180.0 / M_PI;
146 
147     // Yaw (z-axis rotation)
148     float siny_cosp = 2 * (q.w * q.z + q.x * q.y);
149     float cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z);
150     yaw = std::atan2(siny_cosp, cosy_cosp) * 180.0 / M_PI;
151 }
152 
153 /**
154  * @brief Publish the actuator motors.
155  *        For this example, we are generating sinusoidal values for the
        actuator positions.
156  */
157 void OffboardControl::publish_actuator_servos()
158 {
159     static double time = 0.0;
160     ActuatorServos msg{};
161 
162     msg.control[0] = pitch_.load(); // Pitch value between -0.75 and 0.75
163     msg.control[1] = -roll_.load(); // Roll value between -0.75 and 0.75
164 
165     msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
166     actuator_servos_publisher_->publish(msg);
167 
168     time += 0.1; // Increment time for the next wave
169 }
170 
171 /**
172  * @brief Publish the actuator motors.
173  *        For this example, we are generating sinusoidal values for the
        actuator positions.
174  */
175 void OffboardControl::publish_actuator_motors()
176 {
177     static double time = 0.0;
178     ActuatorMotors msg{};
179 
180     // Generate sinusoidal values for actuator positions
181     float value_1 = 0.05f * (sin(time / 10.0f)); // Sinusoidal wave between 0
        and 0.10
182     float value_2 = -value_1;
183 
184     if (value_1 < 0.0f) {
185         value_1 = NAN;
186     }
187     if (value_2 < 0.0f) {
188         value_2 = NAN;
189     }
190 
191     msg.control[0] = value_1;
192     msg.control[1] = value_2;
193 
194     msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
195     actuator_motors_publisher_->publish(msg);
196 
```

```
197     time += 0.1; // Increment time for the next wave
198 }
199
200 /**
201  * @brief Publish the offboard control mode.
202  *        For this example, only direct actuator is active.
203  */
204 void OffboardControl::publish_offboard_control_mode()
205 {
206     OffboardControlMode msg{};
207     msg.position = false;
208     msg.velocity = false;
209     msg.acceleration = false;
210     msg.attitude = false;
211     msg.body_rate = false;
212     msg.thrust_and_torque = false;
213     msg.direct_actuator = true;
214     msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
215     offboard_control_mode_publisher_->publish(msg);
216 }
217
218 /**
219  * @brief Publish the vehicle control mode.
220  *        For this example, we are setting the vehicle to offboard mode.
221  */
222 void OffboardControl::publish_vehicle_control_mode()
223 {
224     VehicleControlMode msg{};
225     msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
226     msg.flag_armed = true;
227
228     msg.flag_multicopter_position_control_enabled = false;
229
230     msg.flag_control_manual_enabled = false;
231     msg.flag_control_auto_enabled = false;
232     msg.flag_control_offboard_enabled = true;
233     msg.flag_control_position_enabled = false;
234     msg.flag_control_velocity_enabled = false;
235     msg.flag_control_altitude_enabled = false;
236     msg.flag_control_climb_rate_enabled = false;
237     msg.flag_control_acceleration_enabled = false;
238     msg.flag_control_attitude_enabled = false;
239     msg.flag_control_rates_enabled = false;
240     msg.flag_control_allocation_enabled = false;
241     msg.flag_control_termination_enabled = false;
242
243     msg.source_id = 1;
244
245     vehicle_control_mode_publisher_->publish(msg);
246 }
247
248 /**
249  * @brief Publish vehicle commands
250  * @param command   Command code (matches VehicleCommand and MAVLink MAV_CMD
        codes)
251  * @param param1    Command parameter 1
252  * @param param2    Command parameter 2
```

```
253  */
254 void OffboardControl::publish_vehicle_command(uint16_t command, float param1,
        float param2)
255 {
256     VehicleCommand msg{};
257     msg.param1 = param1;
258     msg.param2 = param2;
259     msg.command = command;
260     msg.target_system = 1;
261     msg.target_component = 1;
262     msg.source_system = 1;
263     msg.source_component = 1;
264     msg.from_external = true;
265     msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
266     vehicle_command_publisher_->publish(msg);
267 }
268
269 int main(int argc, char *argv[])
270 {
271     std::cout << "Starting offboard control node..." << std::endl;
272     setvbuf(stdout, NULL, _IONBF, BUFSIZ);
273     rclcpp::init(argc, argv);
274     rclcpp::spin(std::make_shared<OffboardControl>());
275
276     rclcpp::shutdown();
277     return 0;
278 }
```

Listing 6: OffboardControl.cpp