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 Background

One requirement of this system is to be able to run controllers developed by the team. 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.

PX4 is an open-source flight control software that runs on a variety of hardware platforms. It is designed to be used in drones, but it can also be used in other vehicles, such as cars and boats. 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.

PX4 currently doesn't provide native support for a drone with a TVC module. This means that the team cannot natively use some parts of the ecosystem. These includes simulators like gazebo, airframes, autonomous flight modes, existing controllers and navigation algorithms, and RC control. 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: 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.

To communicate with the PX4, the two options that PX4 provides are MAVLink and ROS2 interface. The team chose to use ROS2 interface, because MavLink although simpler to use, is limited to MAVLink services, meaning some needed information may not be exposed through it. Additionally, ROS2 is a more powerful framework, with a large ecosystem of packages and libraries.

To validate if the demo works, we could either use simulation (as it is used in PX4 offboard controller example), or use the actual hardware. The team decided to validate the demo in the actual hardware, as we already had the functional hardware, and as mentioned before, 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.
- Read sensor data, and be able to use it.

This demo was accomplished by using the assembled TVC mechanism and connecting an offboard computer, which in this case was a Laptop running Linux.

2 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 reading the sensor data, and actuate the servos and motors.

2.1 PX4

In order to control PX4, PX4 has to change its internal state machine mode to offboard. In addition, the offboard computer has to send a heartbeat message to PX4, so that it knows that the offboard computer is connected. If the offboard computer is not sending a specfic offboard mode message, PX4 will switch to the failsafe mode (configured as disarm mode).

To control the servos and motors, the offboard computer has to send a message, informing the level of the PX4 control architecture at which offboard setpoints must be injected, while disabling the bypassed controllers. In this case, the desired control quantity is direct motors and servos. Afterwards, the offboard computer can send the desired PWM values to the motors and servos.

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

2.2 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:

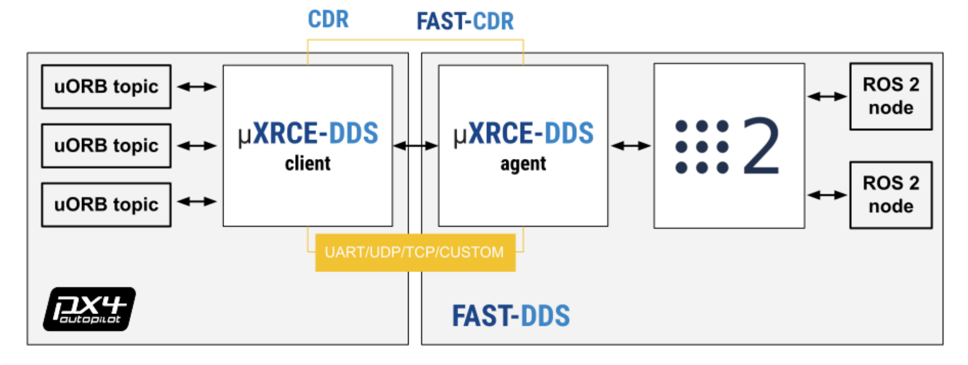

Figure 1: Software architecture connecting PX4 to ROS2 using uXRCE-DDS.

In our case, the connection between client and agent is done using a serial connection.

2.3 ROS2

ROS2 is a set of software libraries and tools that help you build robot applications. It is a modular system, meaning that it can be extended to add new features. It is composed of a set of nodes that communicate with each other using a publish-subscribe model, with the help of a DDS communication bus.

The offboard computer contains a ROS2 workspace, which is a set of packages that can be built and run together.

The demo workspace uses 3 packages:

- `px4_msgs`: This package contains all the messages used by PX4, and a node that connects to the uXRCE-DDS agent, and subsctibes to the PX4 uOrb 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`: This package is used to visualize ROS2 topics and messages in real time, with the help of a GUI.

3 Setup

3.1 PX4

Before being able to test this demo, a number of configurations are required.

3.1.1 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.

3.1.2 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 started by using a baloon frame, because the vehicle is uncontrolled (adeactivates the native flight controller algorithms). This turned out to not be the feasable, as this frame also disables the capacity to configure servos, which our system requires. Instead, the team used the generic multicopter frame, with a custom generic geometry.

3.1.3 Servo setup

Using a generic geometry, 2 motors 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).

3.1.4 Safety checks

...

3.1.5 uXRCE-DDS client setup

The team followed the official tutorial TELEM 2 Serial Port was configured as the port used by the client.

3.2 Offboard computer

3.2.1 uXRCE-DDS agent setup

The team followed the official tutorial, and configured the agent to connect to the serial port where PixHaw was connected.

3.2.2 ROS2 setup

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

3.2.3 PX4 ROS2 messages

4 Demo

The demo only uses sensor data from the attitude estimator, which uses the Extended Kalman Filter present in PX4. It was heavily inspired by the PX4 ROS2 examples.

The demo uses the following messages:

- `px4_msgs::msg::ActuatorMotors`: Sends PWM values to the motors.
- `px4_msgs::msg::ActuatorServos`: Sends PWM values to the servos.
- `px4_msgs::msg::SensorAttitude`: Gets the attitude data as a quaternion, from the attitude estimator.

- `px4_msgs::msg::VehicleCommand`: Sends commands to PX4, such as arming and disarming the vehicle.
- `px4_msgs::msg::VehicleControlMode`: Sends the control mode to PX4, such as offboard mode.
- `px4_msgs::msg::OffboardControlMode`: Sends the control mode to PX4, such as direct motor and servo control.

The code consists of a Ros2 node that contains the main logic of the demo. It includes the subscribers/publishers for the PX4 Messages, and auxiliary functions and variables. 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 means they get called to respond to an event. These callbacks include:

- The subscriber to the `px4_msgs::msg::SensorAttitude` message, that is called when a new message is received. This function uses the attitude data to control the servos, and actuates the motors.
- A timer callback function, which is called every 100ms by a software timer.

4.1 Demo Code

4.1.1 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 routines: the timer and the attitude subscriber.

```cpp
class OffboardControl : public rclcpp::Node
{
public:
    explicit OffboardControl() : Node("offboard_control");
private:
    rclcpp::TimerBase::SharedPtr timer_;

    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;

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

    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);
};
```

Listing 1: Offboard control example

4.1.2 Attitude subscriber callback

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

```
void vehicle_attitude_callback(const px4_msgs::msg::SensorAttitude::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 [-90, 90] to [-1, 1] but constrained to [-0.75,
        0.75]
    roll_.store(std::max(-0.75f, std::min(0.75f, roll / 90.0f)), std::
        memory_order_relaxed);
    pitch_.store(std::max(-0.75f, std::min(0.75f, pitch / 90.0f)), std::
        memory_order_relaxed);

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

Listing 2: Attitude subscriber callback

`roll_` and `pitch_` are atomic variables that store the roll and pitch values, respectively. Atomic variables are used because they are shared between threads, and they need to be accessed in a thread-safe manner.

The values are mapped from the range [-90, 90] to the range [-1, 1], but constrained to the range [-0.75, 0.75]. This is done to ensure that the values are within the range of the servos.

If the vehicle is in offboard mode, the function `publish_actuator_servos()` and `publish_actuator_motors()` are called. The `publish_actuator_servos()` function creates a message with the current roll and pitch values, and publishes it to the servo topic. The `publish_actuator_motors()` function uses a sinusoidal wave to send two sinusoidal curves out of phase to the motors.

4.1.3 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 failsafe mode.

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.

```cpp
void timer_callback()
{
    // 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_++;
};
```

Listing 3: State machine / Timer callback

4.1.4 Publishers

The message `px4_msgs::msg::OffboardControlMode` is used to set the control mode of the vehicle. It is sent with the following values:

```
    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;
```

Listing 4: OffboardControlMode message

The message `px4_msgs::msg::VehicleControlMode` is used to set the control mode of the vehicle. It is sent with the following values:

```
    VehicleControlMode msg{};
    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;
```

Listing 5: VehicleControlMode message

4.1.5 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 node initializes all publishers and subscribers 2. The timer callback continuously sends heartbeat messages 3. After 15 iterations, 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, 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.

5 Results

6 Conclusion