Спеціалізація шаблонів — це механізм, що дозволяє перевизначити поведінку шаблону для конкретних типів або значень. Спеціалізація буває повною, коли надано всі шаблонні параметри (по суті, спеціалізація одного конкретного випадку), і частковою. Яскравим прикладом є std::vector<bool>, який є частковою спеціалізацією std::vector<T>. Замість того, щоб зберігати кожне булеве значення в окремому байті (що призвело б до 700% надлишкових витрат пам'яті), спеціалізована версія використовує бітові поля, зберігаючи 8 значень в одному байті.

Спеціалізація також відіграє роль умови зупинки в рекурсивних метафункціях. Розглянемо метафункцію для обчислення чисел Фібоначчі:

$$Fibonacci_n = Fibonacci_{n-1} + Fibonacci_{n-2}, n = 2,3,\ldots$$

Основний шаблон реалізує рекурсивне правило, а спеціалізації для 0 та 1 надають кінцеві значення для зупинки рекурсії.

```
template<int n> struct Fibonacci {
   static consteval int fib() {
      return Fibonacci<n - 1>::fib() + Fibonacci<n - 2>::fib();
   }
};
template<> struct Fibonacci<1> { static consteval int fib() { return 1; } };
template<> struct Fibonacci<0> { static consteval int fib() { return 0; } };
```

Ефективність таких обчислень можна підвищити за допомогою кешування, зберігаючи вже обчислені результати у статичних членах класу, що дозволяє оптимізувати алгоритмічну складність з експоненційної до лінійної.

Сила метапрограмування полягає у вирішенні реальних задач. У 2001 році Андрей Александреску [3] сформулював задачу реалізації шаблону проектування «*Абстрактна Фабрика*» з єдиним шаблонним методом Create<T>. На той час елегантне рішення було неможливим, і доводилося використовувати громіздкі макроси препроцесора. Сучасне метапрограмування дозволяє вирішити цю задачу за допомогою структури даних списку типів (type_list). Така фабрика може перевіряти на етапі компіляції, чи входить певний тип до списку продуктів, які вона може створювати, і чи можливо його сконструювати з наданих аргументів, використовуючи SFINAE.

```
type_list<Triangle, Rectangle, Circle> factory{};
Triangle* t = factory.create<Triangle>(3, 4, 5);
```

Такий підхід робить фабрику «дружньою до шаблонів» і дозволяє легко її розширювати, не змінюючи існуючий інтерфейс.

Таким чином, метапрограмування шаблонами в C++ виводить нас за рамки чисто імперативного програмування, дозволяючи виконувати складні обчислення на етапі компіляції, генерувати ефективний код та реалізовувати складні патерни проектування елегантно та безпечно з точки зору типів.

Список джерел:

1. Veldhuizen T. L. C++ Templates are Turing Complete, University of Indiana Technical Report, 2003.
2. Vandevoorde D. C++ Templates: The Complete Guide / David Vandevoorde and Nicolai M. Josuttis, Addison-Wesley, Boston, MA, 2002
3. Alexandrescu A. Modern C++ design: generic programming and design patterns applied / Andrei Alexandrescu. Addison-Wesley Professional, 2001. 352 p.

DATA STREAMING PIPELINE FOR THE QUADCOPTER FLIGHT CONTROL STACK

T. Zavaliy, N. Shakhovska, V. Iatsyshyn

Lviv Polytechnic National University

79000, Lviv, Kn. Romana str., 5, Department of Artificial Intelligence

E-mail: taras.i.zavalii@lpnu.ua, nataliya.b.shakhovska@lpnu.ua, volodymyr.p.yatsyshyn@lpnu.ua

Abstract: Modern UAV and UGV technologies are highly competitive dynamic field of applied research and development. Mainstream software stacks being used include PX4, ArduPilot, Betaflight, as well as ROS2 – a modular computing environment for processing sensor data streams, running perception and control algorithms, as well as software-in-the-loop simulations. In our research, ROS2 integration with Betaflight for on-board sensor data streaming is implemented and tested.

Keywords: *data streaming, digital twin, ROS2, UAV, Betaflight.*

Introduction

The ultimate goal in unmanned aerial vehicles (UAV) research is to enable high-speed perception, mapping and planning, which requires software-in-the-loop and hardware-in-the-looop simulation and training. It will allow full autonomy in multiple applications – surveilance, military, disaster recovery, agriculture and logistics.

Robot operating system (ROS2) is one of the tools for achieving the goal. It's data streaming model consists of topics and nodes as publishers/subscribers, allowing one-to-many distributed communication between system components. We focused on the integration of ROS2 middleware with the mainstream flight control stack to enable data collection and sensor fusion. Our expected outcome was to understand the limitations, data rates, processing latency and standard deviation that can be achieved with the off-the-shelve hardware.

Related work

The open-source tooling and the ecosystem has greatly improved, fuelling wide spectrum of research in UAV flight simulation, perception and control. Two popular flight control software stacks, both PX4 and Ardupilot, are targeted at RnD, industrial and commercial use. Survey [1] has brief introduction into the ecosystem, and there are multiple examples of PX4 integration with companion computers [2], [3].

Betaflight-based flight control stack gained it's reputation as affordable, user-friendly and effective in Ukraine, though not optimal for autonomous applications. However, it was shown in [4], that despite limited support for MAVLink, Betaflight integration with companion computer is possible via MSP protocol. Another example is indoor racing platform running Betaflight with NVIDIA Orin NX companion, ROS2 middleware, sending control commands and reading IMU data via MSP [5].

The gap in Betaflight's integration options presents a good research opportunity, as design and implementation of autonomous flight control algorithms is a crucial task today.

Results

An optimal companion computer in terms of price/productivity in our task was the Orange Pi Zero 3 series. Flight controller GOKU F405S was running Betaflight 4.5.2. Implementation steps needed for our experimental set up on the Companion include but not limited to:

- configure ROS2 docker container with USB serial link to flight controller,
- research MSP protocol structure and message format,
- implement MSP bridge node to poll Betaflight and publish to ROS2 topics,
- insert timestamps into the data stream for latency measurement,
- implement data sink node for ROS2 bag storage and MQTT streaming.

We found the upper rate limit of Betaflight's MSP implementation to be at 100 Hz. Increasing request rate did not increase response rate over this limit. Read rate measurements in Table 1 were obtained at the bridge node using standard ROS2 topic monitoring tools. File write latency was calculated at the Data sink node by inspecting timestamps in ROS2 Bag file and comparing them with original read timestamps. Barometer and IMU packets mean write latency was 2.47 and 8 ms respectively, with standard deviation of 1.6 and 9.5 ms. P95 latency was around 5 ms for barometer and 26 ms for IMU datastream. Note, that IMU packet is larger, as it contains both accelerometer and gyroscope sensor readings.

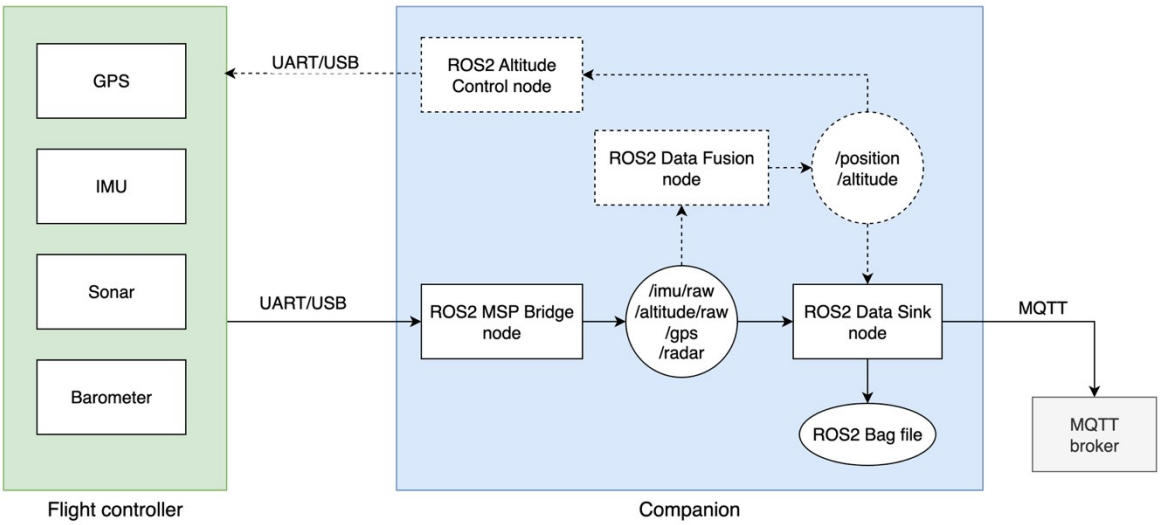

Figure 1. ROS2–MSP integration schema for Betaflight telemetry and control.

Finally, the MQTT data sink was implemented and tested over Wi-Fi connection. Due to the networking overhead the latency was higher: P95 latency of 27 ms for barometer and 125 ms for the IMU. In this case, the MQTT broker (EMQX) on the laptop was used to receive data from the companion computer for analisys, visualization and storage.

Sensor	MSP read rate, Hz	MSP read rate, min, ms	MSP read rate, max, ms	MSP read rate σ, ms	File write latency, mean, ms	MQTT write latency, mean, ms
IMU	99.4	0.1	5.2	0.57	8	105.7
Barometer	99.6	0.1	3.5	0.71	2.47	17.7

Table 1. Metrics for sensor data streams over MSP and MQTT protocols.

Conclusion

Data streaming integration with popular flight control stack was implemented and tested. We showed that MSP protocol in Betaflight has rate limitation. Using pull mode for querying single sensor, we achieved the maximum frequency of up to 100 Hz and measured the latencies for MQTT target. Future research will focus on sensor data fusion for reliable navigation in complex environments, as well as running quadcopter control algorithms on companion computer.

References

1. N. Aliane, “A Survey of Open-Source UAV Autopilots” *Electronics*, vol. 13, no. 23, p. 4785, Jan. 2024, doi: 10.3390/electronics13234785.
2. L. Meier, D. Honegger, and M. Pollefeys, “PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 6235–6240. doi: 10.1109/ICRA.2015.7140074.
3. J. Galvez-Serna, F. Vanegas, S. Brar, J. Sandino, D. Flannery, and F. Gonzalez, “UAV4PE: An Open-Source Framework to Plan UAV Autonomous Missions for Planetary Exploration,” *Drones*, vol. 6, no. 12, p. 391, Dec. 2022, doi: 10.3390/drones6120391.
4. D. Sazonov, “FPV autonomous operation with Betaflight and Raspberry Pi,” ILLUMINATION. Accessed: Sept. 16, 2025. [Online]. Available: https://medium.com/illumination/fpv-autonomous-operation-with-betaflight-and-raspberry-pi-0caeb4b3ca69
5. M. Bosello *et al.*, “Race Against the Machine: a Fully-annotated, Open-design Dataset of Autonomous and Piloted High-speed Flight,” Feb. 24, 2024, *arXiv*: arXiv:2311.02667. doi: 10.48550/arXiv.2311.02667.