2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

ROSflight: A Lean Open-Source Research Autopilot

James Jackson[1], Daniel Koch[1], Trey Henrichsen[1], Tim McLain[1]

***Abstract*— ROSflight is a lean, open-source autopilot system developed with the primary goal of supporting the needs of researchers working with micro aerial vehicle systems. The project consists of firmware designed to run on low-cost, readily available flight controller boards, as well as ROS packages for interfacing between the flight controller and application code and for simulation. The core objectives of the project are as follows: maintain a small, easy-to-understand code base; provide high-bandwidth, low-latency communication between the flight controller and application code; provide a straightforward interface to research application code; allow for robust safety pilot integration; and enable true software-in-the-loop simulation capability.**

I. Introduction

In recent years, there has been a tremendous amount of research in autonomous operation of micro aerial vehicles (MAVs) supporting a broad spectrum of use cases, including disaster response, inspection, monitoring, and mapping to name a few. To aid academic institutions and other organizations in this research and in the rapid development of MAV technology, we present ROSflight as a minimalistic, lean, open-source autopilot developed specifically with the needs of researchers in mind.

A large variety of autopilot technologies are already available to researchers. Some of these have been developed and maintained by commercial entities [1]–[5], while others have been developed by communities of volunteers [6]–[10]. Most of these autopilots are designed to enable out-of-the-box operation of MAVs in open-air situations with GPS, or under tight control from a human pilot. As a result of both the intense competition and the incredible energy in the autopilot domain, many of these autopilots have become quite feature-rich and demonstrate state-of-the-art capabilities in terms of autonomy and real-time performance.

[1]Multiple Agent Intelligent Coordination and Control (MAGICC) Lab, Brigham Young University, Provo, UT, USA

* This work has been funded by the Center for Unmanned Aircraft Systems (C-UAS), a National Science Foundation Industry/University Cooperative Research Center (I/UCRC) under NSF award Numbers IIP-1161036 and CNS-1650547 along with significant contributions from C-UAS industry members. This work was also supported in part by Air Force Research Laboratory Science and Technology (AFRL S&T) sponsorship. This research was conducted with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a.

Corresponding author:
Daniel Koch, Brigham Young University
Email: daniel.p.koch@gmail.com

Fig. 1: The ROSflight project logo. Links to the documentation, discussion forum, and source code can be found at `https://rosflight.org/`.

The performance and capabilities of these autopilot systems is outstanding, and we recognize and praise the excellent work that has gone into them. Unfortunately, however, this very richness of features sometimes becomes a burden to the researcher. One way this manifests is that some of the leading open-source autopilots, such as [7], [9] have become so feature-rich that it takes an unreasonable amount of time for a researcher to understand the implications of changing aspects of the code they may wish to customize. Another common manifestation is that sometimes the interfaces provided to the autopilot are not flexible enough, either in terms of the types of control setpoints that can be sent to the autopilot, or in terms of insufficient band width in the sensor data streaming or control setpoint transmission. Finally, it can be very difficult to operate such autopilot systems under conditions that differ from those for which they were designed, such as trying to fly in GPS-denied environments with autopilots that require GPS reception to operate.

To ease MAV research and development efforts, we offer ROSflight as an alternative, bare-bones autopilot system designed primarily with the needs of researchers in mind. To meet those needs, the core objectives of the ROSflight project are to provide the following:

- A small, easy-to-understand code base
- High-bandwidth, low-latency communication between the flight controller and application code
- A straightforward interface to application code
- Robust safety pilot integration
- True software-in-the-loop simulation capability

The rest of the paper is organized as follows: First, we will describe the overall vision, intended use case and organization of the ROSflight project. Next, we will describe the design of the autopilot, including the various components and the communication between them. Third, we will discuss integration of the autopilot for both hardware experimentation and simulation, and finally, we will discuss a few research projects that have used ROSflight to successfully complete and publish

978-1-7281-6211-9/20/$31.00 ©2020 IEEE

novel research.

A previous publication [11] described an earlier implementation of ROSflight. While continuing with similar objectives, this paper describes a new, from-scratch implementation that replaces the one described in the previous publication.

II. Overview

In this section we describe the vision and long-term goals for the ROSflight project, then provide an overview of how we envision the system typically being used. We also provide a brief overview of the organization of the ROSflight project.

A. Vision and Objectives

One of the primary goals of the project is that the code base will remain lean and easy to understand. In our experience, complex black-box systems have not been conducive to research activities because they make it difficult to debug the full-system behavior when the details of the inner-loop operation are not well-understood. In addition, highly complex systems can be difficult to configure for the unique requirements of research applications, and can be difficult to modify when required.

To avoid these pitfalls, we have chosen to adopt the philosophy that the embedded ROSflight firmware will implement only the minimum functionality required to achieve safe and stable flight. This functionality includes sensor and actuator input/output, high-bandwidth communication with a companion computer, attitude or attitude-rate control when operating a multirotor aircraft, and supporting functionality such as configuration management and safety features. All higher-level functionality is left for the user to implement, typically on a companion Linux computer. While this requires effort from the user, we believe that it makes ROSflight a much more flexible and easy-to-use tool for researchers who often already write highly-customized application code. For example, GPS-denied operations are inherently supported because the firmware makes no assumptions about the presence or quality of GPS, as opposed to other flight controllers that require a GPS lock before arming to support their large feature set. We have also chosen to adopt a hard stance against feature creep; we welcome users who wish to incorporate additional functionality into the embedded flight controller to fork our project and to share their successes, but in general we will not merge these extensions into the core code base.

Another key goal of the project is to enable useful simulation capabilities, including true software-in-the-loop (SIL) simulation. Some autopilot systems use flags in the firmware to change the behavior of the code when running in SIL mode. The ROSflight firmware instead uses a hardware abstraction layer to implement SIL, so that the core flight-stack code that runs in SIL is identical to the code that runs in hardware, and has no knowledge of which mode it is running in. This approach also allows ROSflight be be incorporated into a variety of simulation environments. Additionally, the interface with application code is identical between simulation and hardware, which allows many integration issues to be debugged in simulation before moving to hardware. These features are discussed in more depth in Section IV.

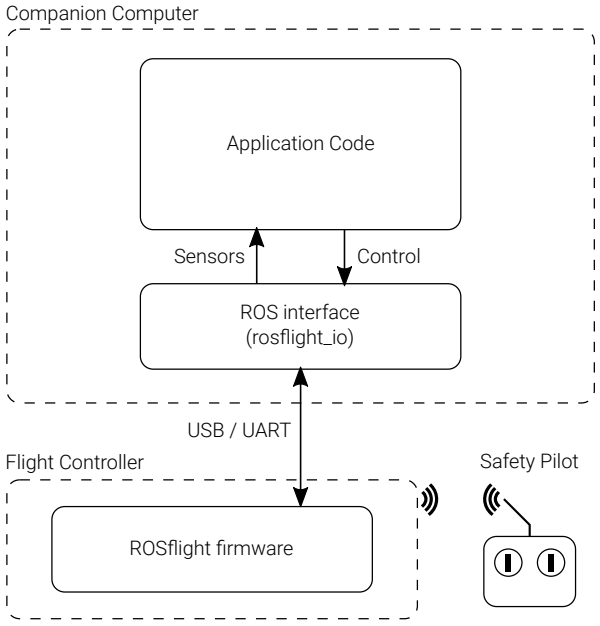


Fig. 2: Intended use case. The embedded flight controller provides sensor and actuator input/output. The application code runs on a companion computer, and communicates with the flight controller through a provided ROS interface. A safety pilot is able to override computer control if needed.

B. Typical Use Case

The intended use case for ROSflight is illustrated by the diagram in Figure 2. There are three main components to the system: the embedded flight controller, the companion computer, and the safety pilot.

The *flight controller* runs the ROSflight firmware,[1] and provides low-level input/output for sensors and actuators (servos and electronic speed controllers (ESCs)). For multirotor vehicles, the flight controller also performs attitude or attitude-rate control. For most research applications, the embedded firmware should not need to be modified, unless new low-level multirotor attitude or attitude-rate controllers are being developed. The ROSflight firmware is targeted to run on low-cost, readily available flight controller boards.

The *companion computer* is a Linux computer—such as an Intel NUC, NVIDIA Jetson, Odroid, or other small-form-factor computer—that is mounted on the vehicle

[1] `https://github.com/rosflight/firmware`

and is connected to the flight controller via USB. Connections over UART serial are also supported if required. For researchers who use the Robot Operating System (ROS)[2], the `rosflight_io` node in the `rosflight` package[3] provides a ROS interface to communicate with the flight controller. All configuration of the flight controller is also performed via service calls provided by `rosflight_io`.

It is intended that most research code will run on the companion computer, as indicated by the *application code* block in Figure 2. The application code can use the high-rate sensor data streams exposed by `rosflight_io`, and sends control setpoints to the flight controller. These control setpoints can consist of attitude and throttle commands, attitude rate and throttle commands, or direct throttle and servo commands. We refer to these setpoints as *offboard control* setpoints, because they are "offboard" from the perspective of the flight controller (even though the companion computer is also mounted on the vehicle).

The *safety pilot* is an integral part of the system, and interacts with the flight controller using a standard radio control (RC) transmitter. Due to the nature of research code, it is important that the safety pilot have the ability to quickly override the offboard control setpoints at any time. The flight-controller firmware makes three mechanisms available to accomplish this:

1) The safety pilot may lock out the offboard setpoints completely by flipping a switch on the transmitter,
2) The flight controller will follow the minimum of the throttle setpoints coming from the safety pilot and offboard setpoints, allowing the safety pilot to quickly kill the throttle if necessary,
3) The safety pilot may temporarily and independently override the roll, pitch, or yaw-rate channels by deviating the corresponding transmitter stick from center.

These override mechanisms have proven to be valuable in our experience, although they may be independently disabled if desired. For safety reasons, the flight controller can only be armed from the safety pilot's RC transmitter.

We note that while ROSflight has proven effective for operating smaller research UAS, and substantial effort has been made to make it as safe and dependable as possible, the design process did not include formal safety analysis or redundant design to mitigate potential software or hardware faults. We therefore recommend that ROSflight be used only on smaller UAS, and not on larger vehicles that pose significant risks in the case of failure.

C. Project Organization

The ROSflight project is hosted on GitHub[4], and consists of two separate but related code bases: the embedded flight-controller firmware[5], and the ROS interface[6]. Documentation is maintained as part of the project. Links to the code repositories, documentation, and other resources are provided at the project website, `https://rosflight.org/`.

We also wish to point out the ROSplane project [12], which provides a good example of application code that provides GPS-waypoint following capabilities for fixed-wing MAVs. The ROSplane project implements the algorithms and architecture described in [13].

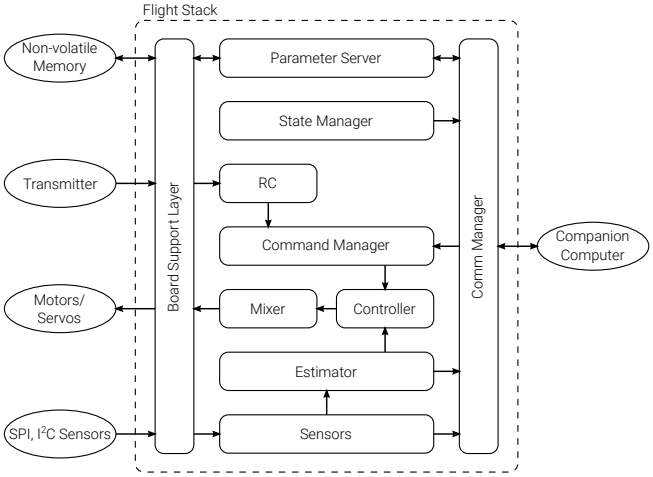


Fig. 3: Flight stack architecture. Rectangles represent modules in the flight stack, and ellipses represent inputs or outputs. The nominal flow of data is represented by the arrows.

III. Design

The core ROSflight firmware flight stack is made up of several modules. This design is intended to limit the scope of each module so that the implications of changes to any one module can be easily understood. The module-level architecture of the flight stack is shown in Figure 3. In this section, we will describe each module, its role in the flight stack, and the relevant algorithms used in its runtime processing. Additional details beyond the overview given in this section are provided in the online documentation.

A. Flight Management Modules

The first three modules we will discuss are the state manager, the parameter server, and the communications manager. These modules are responsible for providing an efficient and safe environment for the control and state estimation to occur. Because of their role in managing the operating environment, they tend to have larger scope than the other modules, and can be a little less straight-forward in implementation. However, significant

[2] `http://www.ros.org/`
[3] `http://wiki.ros.org/rosflight`, `https://github.com/rosflight/rosflight`
[4] `https://github.com/rosflight`
[5] `https://github.com/rosflight/firmware`
[6] `https://github.com/rosflight/rosflight`

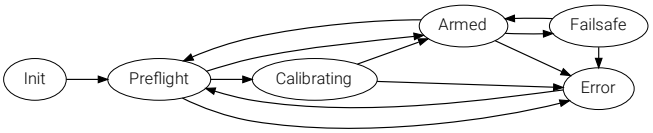


Fig. 4: Simplified finite state machine diagram for the state manager module

attempts have been made to limit their complexity while still maintaining an efficient implementation.

a) State Manager: The state manager is responsible for handling external events and system errors, and managing the system state. "State" in this context refers to whether the system is experiencing errors and whether the system is armed or in a failsafe condition, as opposed to the current attitude and angular rate of the MAV, which will be discussed later. The state manager also informs all other modules of the current state of the flight controller so they can behave appropriately. The state manager implements a finite state machine, a simplified depiction of which is diagrammed in Figure 4.

b) Parameter Server: The flight controller behavior is controlled by a number of parameters that are configurable by the user while in the setup phase, but remain constant during operation. These parameters include controller gains, motor and remote control configuration, and sensor stream rates. Parameters are accessible to all modules in the flight stack at any time; however, the parameter server includes a mechanism that informs the other modules of changes to parameter values to improve efficiency during runtime. The parameter server is also responsible for interacting with the communications manager to provide the user interface for viewing and setting parameter values.

Parameters are also saved to non-volatile memory so they persist through reboots. If a corrupt parameter set is detected via a mismatched checksum, or if the stored firmware code version hash does not match that of the current firmware, the stored parameters are discarded and default values are used, and an associated warning is displayed to the user.

c) Communications Manager: The communications manager is responsible for receiving commands from the companion computer and streaming sensor data. While the actual communication protocol is abstracted to enable easy replacement, the current implementation uses MAVLink [14] as the serial protocol. An associated MAVLink parser is supplied for the companion computer to decode and encode the sensor information and commands.

Communications between the computer and the flight controller take one of three forms. First, data such as sensor readings or motor commands are streamed to the companion computer at configurable rates. Second, commands can be sent to the flight controller, which trigger actions such as calibrating sensors and changing parameters, and which are followed by an acknowledgment report. Finally, control setpoints (such as desired roll, pitch, yaw rate, and throttle) and external attitude updates can be streamed to the flight controller.

All three of these communication modes can take place simultaneously. Priority has been given to enabling very high streaming rates, including streaming IMU sensor measurements at up to 1000 Hz. To ensure safety and that streaming messages are prioritized over other communications during flight, many acknowledgment-type commands are disabled while the aircraft is in the armed state.

B. Flight Control Modules

The rest of the modules shown in Figure 3 are primarily responsible for the real-time state estimation and control of the aircraft. At a high-level, control commands are received from both the RC transmitter and the companion computer. These commands are merged together by the command manager, taking into account safety pilot integration and other user-specified constraints, and given to the controller. Meanwhile, sensor information is collected from the board abstraction layer and provided to the state estimator, which provides the current state estimate to the controller. The controller determines the desired forces and torques in the body frame, and the mixer takes these body-frame commands and transforms them into individual motor commands, based on the geometry of the MAV. These commands are passed back down through the board layer to the actuators on the aircraft.

a) Command Manager: The command manager is responsible for combining commands from the companion computer and the RC operator, enabling safety pilot integration and safe testing of unproven control and estimation algorithms.

In the case of pure RC operation, the command manager simply passes through the RC commands. If offboard control setpoints are provided by the companion computer, then the command manager will pass those commands through unless one of the safety-pilot override mechanisms described in Section II-B is activated, in which case it reverts to RC control. The command manager will also revert to RC control if the offboard control setpoints time out. The command manager additionally applies a default failsafe command if the RC signal is lost.

When sending offboard control setpoints, the user may specify which channels contain valid setpoints. This can be useful for scenarios such as implementing an altitude-hold controller while still allowing for RC attitude control. The stick-deviation safety-pilot override mechanism also operates on a per-channel basis.

b) Estimator: The state estimator is responsible for using the accelerometer and rate gyroscope to estimate the current attitude, angular rate, and rate gyroscope biases of the MAV. ROSflight implements the quaternion-

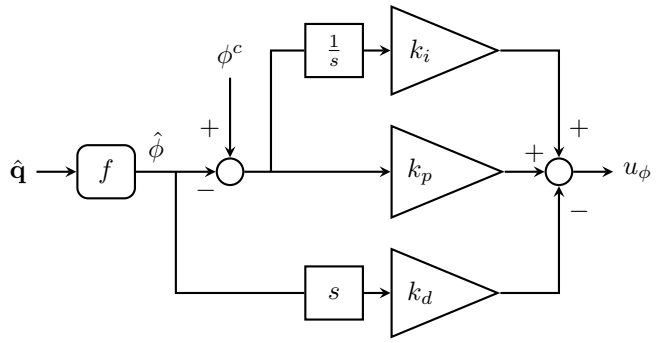


Fig. 5: Block diagram for PID angle controller design. The function f converts the quaternion attitude estimate to Euler angles. The estimated angle $\hat{\phi}$ is compared to the commanded angle ϕ^c. The output of the controller is the normalized torque u_ϕ.

based complementary filter from [15] with some modifications from [16].

Without external position or velocity measurements, attitude is only observable under unaccelerated conditions and the yaw-rate gyroscope bias is completely unobservable. To compensate for these limitations, the companion computer may optionally provide an external attitude measurement update that is used to update the complementary filter estimate. If this application-specific measurement is accurate, then ROSflight performs much better in accelerated conditions, and the yaw-rate gyroscope bias can converge. A complete description of the algorithm used to perform state estimation is detailed in the online documentation.

c) Controller: The controller is responsible for driving the MAV to its desired state. It consumes both the state estimate and the combined command from the command manager, and produces normalized desired torques and forces. The controller can be configured to operate in one of three modes: angle mode, rate mode, or passthrough mode. All relevant control loops use a PID-like structure as shown in Figure 5.

In attitude mode, the user supplies the desired roll and pitch angle, the desired yaw rate, and desired throttle values. Because the controller operates directly on the Euler angles, the current roll and pitch angles are first extracted from the current estimated attitude before being consumed by the PID control loops. Due to the Euler decomposition step, commanding extreme pitch angles in angle mode is not advised, because the Euler decomposition will encounter a singularity at 90 degrees pitch. Rate mode control does not have this problem, but it is considerably more difficult to operate from a safety-pilot perspective. The block diagram for the attitude-mode controller is shown in Figure 5.

In rate mode, the user supplies values of desired angular rate in all three axes, plus throttle. The angular rate control loops have the same structure as the attitude controllers, with the exceptions of omitting the Euler-angle decomposition step and operating on angular rates rather than on the Euler angles. However, they use a separate set of gains, with the k_i and the k_d gains typically set to zero.

In passthrough mode, the controller is completely bypassed, allowing direct access to the motor mixer. This allows for direct actuator control from the companion computer or RC if required, and is a common use case when operating fixed-wing aircraft that have slower dynamics than multirotors.

It should be noted that the RC and the companion computer can each supply angle, rate, or passthrough commands, and that they do not have to match. For example, the safety pilot can be commanding angle-mode commands while the companion computer is operating in rate or passthrough mode without any issue.

d) Mixer: The mixer takes the normalized torque and throttle outputs from the controller and maps them to actuator commands, depending on the location and type of actuator. For example, a quadrotor "+" configuration requires different motor actuation from a quadrotor "×" configuration to achieve the same torques. The motor mixing occurs by multiplying a static matrix $A \in \mathbb{R}^{n\times 4}$, where n is the number of actuators, by the vector of desired normalized torques and throttle $\boldsymbol{\tau}$ to produce the actuator command $\mathbf{u}$ as

$$\mathbf{u} = A\boldsymbol{\tau}. \tag{1}$$

The values of the actuator commands $\mathbf{u}$ vary between -1 and 1 for servos, and between 0 and 1 for motors.

Because both $\boldsymbol{\tau}$ and $\mathbf{u}$ are normalized, each row of A is also typically normalized, and can be calculated for a multirotor MAV as follows: If $\mathbf{r}_i$ is the vector from the center of mass of the MAV to the center of the ith propeller plane, $\mathbf{e}_i$ is the unit vector in the direction of thrust, and s_i is the direction of propeller rotation along $\mathbf{e}_i$ (following the right-hand rule), then

$$M = \begin{bmatrix} \mathbf{k}^\top \mathbf{e}_1 & \dots & \mathbf{k}^\top \mathbf{e}_n \\ \mathbf{r}_1 \times \mathbf{e}_1 + \mathbf{k}s_1 & \dots & \mathbf{r}_n \times \mathbf{e}_n + \mathbf{k}s_n \end{bmatrix}, \tag{2}$$

$$A = \|M^\dagger\|_{\text{row}}, \tag{3}$$

where $\mathbf{k}$ is the unit vector in the z direction, and M is a matrix whose columns represent the thrust and moments contributed by each of the n rotors. The $(\cdot)^\dagger$ operator is the Moore-Penrose pseudo-inverse, and $\|\cdot\|_{\text{row}}$ normalizes each row of its operand matrix.

The mixer is also responsible for incorporating general-purpose setpoints from the companion computer to additional actuators not essential to flight. This allows the user to perform tasks such as raising or lowering landing gear or dropping payloads.

IV. Integration

Beyond a high-quality implementation of the flight control algorithms and associated software, ROSflight is designed to be easily integrated into an existing hardware

Fig. 6: Quadrotor and fixed-wing MAVs using the ROSflight flight controller

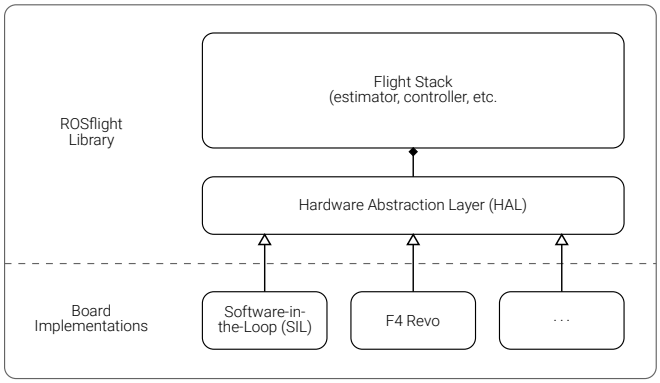


Fig. 7: Hardware abstraction layer and SIL architecture

and software system. As shown in Figure 7, the flight stack described in Section III makes all hardware-level calls through a hardware abstraction layer (HAL). This architecture makes integrating the flight stack into new hardware relatively simple and makes true software-in-the-loop (SIL) simulations possible. The existing hardware and SIL implementations are briefly discussed below. We refer interested readers to the online documentation for additional details on the HAL and SIL architectures.

A. Hardware Implementation

ROSflight is currently distributed with a hardware board interface to STM32F4-based flight control boards derived from the OpenPilot F4 Revolution flight controller [17]. This hardware has become popular in first-person view (FPV) drone racing and is therefore widely available, inexpensive, and made to a good standard of quality. The F4 board layer implements functions for reading sensors, outputting motor commands, providing serial and USB connectivity, and supplying clock-related functions such as getting the current time in microseconds from system boot.

Although STM32F4 is the only board family currently officially supported by the ROSflight project, other flight control boards, such as STM32F1 and STM32F3-based FCUs have been made compatible through implementing the required board-level functions. Figure 6 shows a multirotor MAV and a fixed-wing MAV, which both used ROSflight in visual-inertial state estimation research efforts.

B. Software-in-the-Loop Simulation

Software-in-the-loop simulation is a valuable tool for researchers who wish simulations to be as similar to hardware as possible. While other flight controllers support software-in-the-loop or even hardware-in-the-loop simulation, most of them require configuring the flight control stack into a simulation mode. This mode changes some aspects of how the flight controller operates, and therefore is only a partial SIL simulation. In contrast, a ROSflight SIL implementation simply implements the HAL API, and (depending on processor architecture) exactly the same flight stack library that is run on hardware can be linked to the simulation for accurate simulation testing. The ROSflight flight stack has no knowledge that it is operating in a SIL environment, and runs exactly the same code as it does in hardware. One notable feature of this architecture is that the HAL also defines the means by which the flight stack accesses the current time. This makes faster-than-realtime simulations possible even in SIL, which is useful for applications such as reinforcement learning.

ROSflight is currently distributed with a HAL implementation for the Gazebo robot simulator [18]. However, other, more photorealistic simulations have been developed which support ROSflight SIL, such as the Holodeck simulator [19] based on the Unreal 4 video game engine. Images of a ROSflight SIL simulation in both the Gazebo and Holodeck environments are shown in Figure 8.

V. Conclusion

ROSflight is a lightweight, flexible, easy-to-understand research autopilot designed to meet the needs of researchers in autonomous operation of MAVs. The built-in functionality has by design been limited to a very narrow scope to enable easy understanding and integration into other projects, and serves as a base upon which higher-level functionality can be developed.

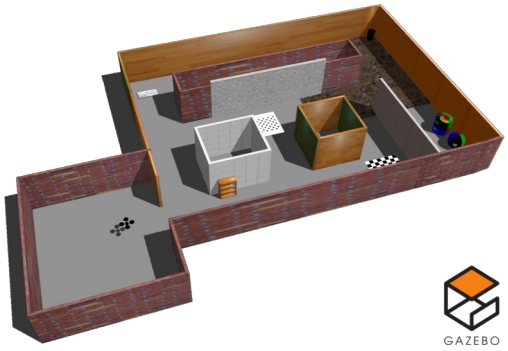
To date, ROSflight has been used in a variety of research efforts resulting in publications [20]–[25], including applications such as model-predictive and LQR control of quadrotors and visual-inertial navigation for quadrotors and fixed-wing vehicles. ROSflight continues to support a number of ongoing research efforts by providing a flexible, high-bandwidth interface for sensor data and actuator or control setpoints. These successes have demonstrated ROSflight to be a useful research platform for MAV state estimation and control.

References

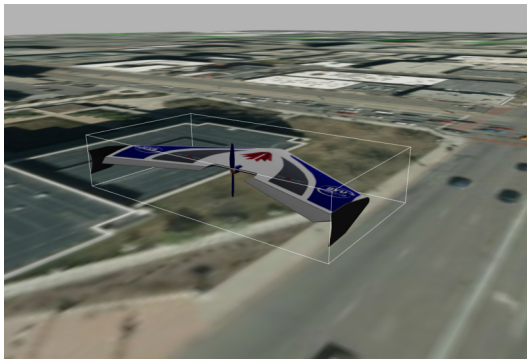
[1] DJI Technology Co., Ltd. (2020). N3, [Online]. Available: `https://www.dji.com/n3` (visited on 02/26/2020).

[2] Cloud Cap Technology. (2020). Piccolo autopilots, [Online]. Available: `http://www.cloudcaptech.com/products/auto-pilots` (visited on 02/26/2020).

[3] HiSystems GmbH. (2020). Mikrokopter, [Online]. Available: `http://www.mikrokopter.de/en/home` (visited on 02/26/2020).



(a) Multirotor aircraft in Gazebo simulation environment



(b) Fixed-wing aircraft in Gazebo simulation environment

(c) Multirotor aircraft in Unreal 4 Holodeck simulation environment

Fig. 8: ROSflight SIL software-in-the-loop simulation implementations

[4] Parrot Drones SAS. (2019). Parrot USA, [Online]. Available: `https://www.parrot.com/us/` (visited on 02/26/2020).

[5] Intel Corporation. (2020). Commercial drones from intel, [Online]. Available: `https://www.intel.com/content/www/us/en/drones/drone-applications/commercial-drones.html` (visited on 02/26/2020).

[6] Cleanflight Team. (2017). Cleanflight, [Online]. Available: `http://cleanflight.com/` (visited on 02/26/2020).

[7] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *International Conference on Robotics and Automation*, 2015, pp. 6235–6240.

[8] LibrePilot. (2016). Librepilot, [Online]. Available: `https://www.librepilot.org/site/index.html` (visited on 02/26/2020).

[9] ArduPilot. (2016). ArduPilot, [Online]. Available: `https://ardupilot.org/` (visited on 02/26/2020).

[10] PaparazziUAV. (2020). PaparazziUAV, [Online]. Available: `https://wiki.paparazziuav.org` (visited on 07/30/2020).

[11] J. Jackson, G. Ellingson, and T. McLain, "ROSflight: A lightweight, inexpensive MAV research and development tool," in *International Conference on Unmanned Aircraft Systems (ICUAS)*, 2016, pp. 758–762.

[12] G. Ellingson and T. McLain, "ROSplane: Fixed-wing autopilot for education and research," in *International Conference on Unmanned Aircraft Systems (ICUAS)*, 2017, pp. 1503–1507.

[13] R. W. Beard and T. W. McLain, *Small Unmanned Aircraft: Theory and Practice*. Princeton University Press, 2012.

[14] S. Atoev, K. Kwon, S. Lee, and K. Moon, "Data analysis of the MAVLink communication protocol," in *International Conference on Information Science and Communications Technologies (ICISCT)*, 2017, pp. 1–3.

[15] R. E. Mahony, T. Hamel, and J.-M. Pflimlin, "Complementary filter design on the special orthogonal group SO(3)," *Proceedings of the 44th IEEE Conference on Decision and Control*, pp. 1477–1484, 2005.

[16] R. T. Casey, M. Karpenko, R. Curry, and G. Elkaim, "Attitude representation and kinematic propagation for low-cost UAVs," 2013.

[17] LibrePilot, *Openpilot revolution*, 2019. [Online]. Available: `https://opwiki.readthedocs.io/en/latest/user_manual/revo/revo.html` (visited on 08/03/2019).

[18] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004, pp. 2149–2154.

[19] J. Greaves, M. Robinson, N. Walton, M. Mortensen, R. Pottorff, C. Christopherson, D. Hancock, and J. M. D. Wingate, *Holodeck: A high fidelity simulator*, 2018. [Online]. Available: `https://holodeck.cs.byu.edu/`.

[20] E. Small, P. Sopasakis, E. Fresk, P. Patrinos, and G. Nikolakopoulos, *Aerial navigation in obstructed environments with embedded nonlinear model predictive control*, 2018. arXiv: `1812.04755 [math.OC]`.

[21] M. Farrell, J. Jackson, J. Nielsen, C. Bidstrup, and T. McLain, "Error-state LQR control of a multirotor UAV," in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2019.

[22] J. Ellingson, G. Ellingson, and T. McLain, "Deep RC: Enabling remote control through deep learning," in *International Conference on Unmanned Aircraft Systems (ICUAS)*, 2019, pp. 1161–1167.

[23] G. Ellingson, K. Brink, and T. McLain, "Relative navigation of fixed-wing aircraft in GPS-denied environments," [Online]. Available: `https://scholarsarchive.byu.edu/facpub/3226/`.

[24] J. Jackson, "Enabling autonomous operation of micro aerial vehicles through GPS to GPS-denied transitions," PhD dissertation, Brigham Young University, 2019.

[25] D. Koch, D. Wheeler, R. Beard, T. McLain, and K. Brink, "Relative multiplicative extended Kalman filter for observable GPS-denied navigation," [Online]. Available: `https://scholarsarchive.byu.edu/facpub/1963/`.