CNS Flight Stack for Reproducible, Customizable, and Fully Autonomous Applications

Martin Scheiber[1*], Alessandro Fornasier[1*], Roland Jung[1], Christoph Böhm[1], Rohit Dhakate[1], Christian Stewart[2], Jan Steinbrener[1], Stephan Weiss[1], and Christian Brommer[1*]

***Abstract*—While low-level auto pilot stacks for aerial vehicles focus on robust control, sensing, and estimation, the continuous advancement of higher-level autonomy for aerial vehicles requires much more complex higher-level flight stacks in order to enable safe, fully autonomous long-duration missions. Rather than focusing on the low-level control, high-level flight stacks are required to monitor the system's integrity continuously, initiate contingency plans, execute mission plans and adapt them in non-nominal situations, allow for proper data logging, and provide standardized interfaces and integrity verification for external mission planners and localization modules.**

To that end, we present our freely available, high-level flight stack (dubbed CNS Flight Stack) that meets the above requirements and at the same time a) is platform-agnostic through a generalized (embedded) hardware abstraction layer, b) uses low compute complexity for online use on embedded hardware, and c) can be extended with other sensor modalities, integrity checks, and mission modules. These additional properties make it reproducible on a variety of different platforms for safe and fully autonomous applications.

We tested the proposed flight stack in over 450 real-world flights and report the failure modes our framework detected and also mitigated to avoid crashes of the aerial system.

***Index Terms*—Software Architecture for Robotic and Automation, Autonomous Vehicle Navigation, Aerial Systems: Perception and Autonomy**

The software components for the CNS Flight Stack , a configuration tutorial, and a descriptive video is available at https://github.com/aau-cns/flight_stack.

I. Introduction

TODAY, generalized autopilot stacks for low-level control of unmanned aerial vehicles (UAVs) such as PX4, ArduPilot, or proprietary solutions are widely employed in research and industry. The integrated controls (from rate, over attitude, to position), its tuning, and mission design tools have greatly facilitated and boosted the design and development process of different UAV platforms across communities and in industry. With additional low-level pre-flight and in-flight checks (e.g., battery level, geo-fencing), rudimentary safety elements have become commonly available.

Nowadays, the development has moved towards supporting higher-level autonomy. In order to boost the development of fully autonomous applications of aerial vehicles in research and industry, a generalized, high-level flight stack for safe system management, including health monitoring, issue mitigation, and mission management, is required. Several flight stacks, methods, and heuristics (cf. Sec. II) are being developed to oversee the health of the autonomous system, to detect non-nominal conditions, and to initiate contingency plans upon failure detection. This also includes extensions of existing low-level autopilot flight stacks to handle these tasks. To our knowledge though, no flight system currently available provides a generalized, customizable and reproducible framework for high-level autonomy control.

With our proposed flight stack, the *CNS Flight Stack*, we aim at a generalized support for UAV autonomy development to conduct safe long-duration fully autonomous missions minimizing crashes and damage to systems, persons, and the environment.

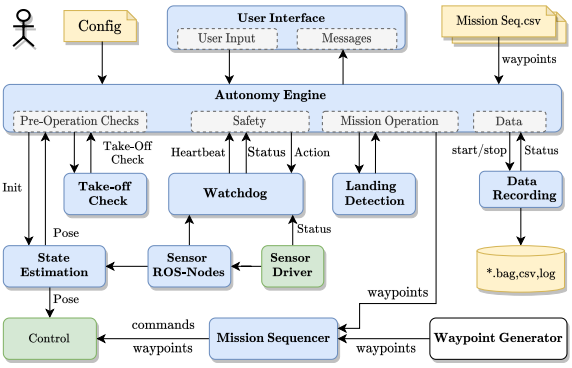

Fig. 1. The proposed system's modules and their interaction. With a single configuration file and multiple mission sequence files the user interacts through a visual user interface with the autonomy engine.

II. Related Work and Contribution

The necessity of advanced flight stacks that monitor the system's mission and health and provide contingency actions is known to the research community and to industry. Auto pilot stacks such as from Ascending Technologies[A] and Intel

Manuscript received: February, 24, 2022; Revised June, 08, 2022; Accepted July, 08, 2022.

This paper was recommended for publication by Editor Tamim Asfour upon evaluation of the Associate Editor and Reviewers' comments. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 871260, from the Austrian Ministry of Climate Action and Energy (BMK) under the grant agreement 879666 (TAUBE2), and was sponsored by the Army Research Office and was accomplished under Cooperative Agreement Number W911NF-21-2-0245. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

[1]All authors are with Institute of Smart Systems Technologies, Group Control of Networked Systems, University of Klagenfurt, Klagenfurt, Austria {first.lastname}@ieee.org

[2]This author is with Aperture Robotics, LLC. christian@aperture.us

*C. Brommer, A. Fornasier, and M. Scheiber contributed equally.

Pre-print version, accepted July/2022, DOI: 10.1109/LRA.2022.3196117

[A]http://wiki.ros.org/asctec_mav_framework

provided rudimentary safety elements through redundancy. A fully open-source autopilot named PX4 [1] was released, using their previously developed hardware and sensor suite. QGroundControl[B] was added to the stack, including control tuning, system calibration, mission planning tools, and rudimentary pre-flight and in-flight system checks. ArduPilot[C] [2], [3] provides a similar open-source flight stack with similar capabilities on system checks. Further, [4] recently tackled a flight stack design on the guidance, navigation, and control site. However, this framework lacks a high-level overarching autonomy engine performing system health checks during and before flight.

Other auto pilot stacks like BetaFlight[C], CleanFlight[E], or LibrePilot[E] all do not provide functionality for autonomous UAVs but are rather designed for first-person view (FPV) flying. Industrial systems from, e.g., DJI[F] or Auterion[G], have become popular to test control and navigation tasks. Naturally, their focus lies on industry partners rather than on the research community that require fast adaptation cycles, scalable interfaces to a variety of different sensors, and different contingency plans per test-flight.

A. Improvements beyond the State-of-the-Art

Although historically grown auto pilot stacks are inherently limited in tackling the full system analysis required for safe aerial missions, they are well-established, tested, and open-source frameworks for low-level control and provide a well-documented interface, such as the MAVLink protocol[H] [5], to which our high-level flight stack can connect to.

All the above existing works do not provide an overarching high-level system integrity verification, issue detection and mitigation pipeline, and adaptive mission scheduler to ensure fully safe and long-duration autonomy. Our flight stack provides these capabilities and, in addition, the following benefits:

- Due to its modularity, different autonomy sub-modules (cf. Fig. 1) can be run on different processing boards.
- The cross-compiled system can run on any Linux-compatible processor with reproducible behavior. Quality-of-service (QoS) features in Linux are leveraged to guarantee adequate compute time for mission-critical components.
- A minimal in-RAM ephemeral host operating system (OS) managing containerized environments attached to persistent storage [6] masks the platform specifics and allows for cross-platform reproducible autonomy container-images.
- The same in-RAM ephemeral host OS guarantees system (re-)boot into a defined and fixed initial state from where the flight stack can start, with the ability to upgrade/roll-back the host and container system remotely.

The above benefits and contributions are at system/software level and are built upon existing theory, processes, and models. That being said, our proposed flight stack can drastically speed-up development cycles, reduce maintenance time and human-triggered failures, and prevent damage to the UAV, persons, and the environment.

[B] qgroundcontrol.com [C] ardupilot.org [D] betaflight.com [E] cleanflight.com [E] librepilot.org [F] dji.com [G] auterion.com [H] mavlink.io

III. Flight Stack Description

Introducing full autonomy to UAVs requires several components to replace human-decision making and monitoring in-flight. A fully autonomous flight stack requires modules for

1) state estimation, mission navigation, and vehicle control,
2) monitoring the health of the system,
3) detecting deviation from nominal conditions,
4) initiating safety mechanisms in case of failures, and
5) communicating the UAV's flight state to the (potentially remote) user

The CNS Flight Stack handles these tasks in different individual modules under the management of the *autonomy engine* displayed in Fig. 1 and described in the subsections below.

A. Hardware Abstracted Operating System (SkiffOS)

A core strength of the CNS Flight Stack is its minimal in-RAM ephemeral host OS – SkiffOS [6] – that masks the underlying compute platform specifics and therefore builds a unified base for our flight stack software. This allows for cross-platform reproducible container-images of our flight stack, guarantees identical boot sequences independent of the preceding shut-down reasons, and simplified over-the-air software management.

In the state-of-the-art workflow, manufacturers of single-board computers (SBCs) typically provide a reference system as a full-disk snapshot. Developers copy and manually adjust the OS for their purposes. Package managers (e.g., apt) upgrade the system state over time with a rolling release model. Full-disk images lack reproducible behavior [7], cannot be replicated from declarative configuration, and have no simple remote upgrade mechanism.

SkiffOS [6] uses the Buildroot [8] OS cross-compiler to re-target the flight stack to any Linux-compatible compute platform with reproducible results. The immutable "host" system is booted as an in-RAM OS, ensuring networking and stable SSH access for remote debugging with automatic repair of any persistent storage filesystem errors. The flight stack is then managed as a set of Docker containers attached to persistent storage. The host system can be upgraded independently from the containerized workloads.

The combination of an immutable in-RAM host OS with containerized workloads results in strongly reproducible system behavior, even in the face of common interruptions such as power brown-outs and flash storage failure. Container images are portable across machines with the same CPU architecture, and do not need platform-specific drivers or firmware. Quality-of-service (QoS) controls are used to allocate resources appropriately to critical flight components.

Developers expect a mutable system with a traditional package manager based workflow for development and system management. SkiffOS replicates this workflow by forwarding incoming user sessions (e.g., ssh) to a mutable OS distribution instance running in a container. The forwarding is invisible to the user and all tools behave as if the containerized distribution was booted directly. Init managers (e.g., systemd) work as expected, and all container isolation features are disabled to prevent any performance impact [9].

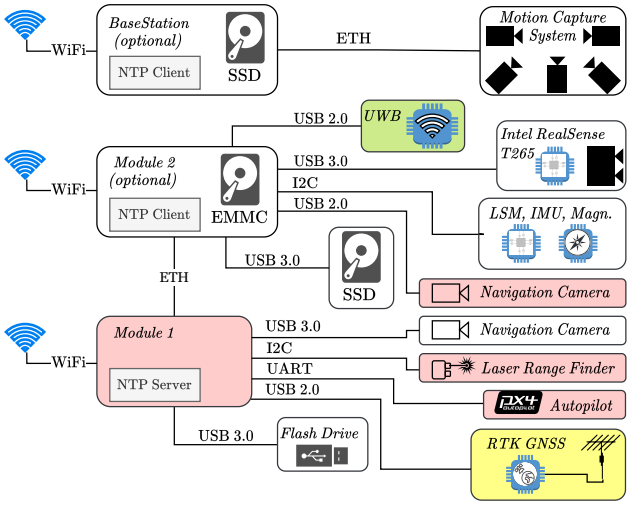

Fig. 2. Schematic for possible flight platform setups. It is important to note that the CNS Flight Stack can operate on single or multi-device computing platforms and different sensor inputs. Thus, the provided schematic is an example, and sensor interfaces, as well as software module deployment, can be designed for the task at hand. However, this schematic is optimized to the tasks performed in Sec. IV with aspects to data interface bandwidth and computational load balancing. The colors represent the setups used for the experiments in this paper (cf. Sec. IV red+green, cf. Sec. V red+yellow), with the full schematic being the most complex setup used to date.

B. Hardware and Flight Stack Configuration

Although the flight stack software container for SkiffOS is easily replicable for multiple UAVs and across platforms, the user initially needs to define the actual mission(s) and the behavior the UAVs should have upon specific trigger signals. For our proposed flight stack, this initial *configuration* is done in a simple, single configuration file per mission. The configuration file requires the following input:

First, the user specifies the sensor setup of the UAVs. This includes the type of sensors, their calibration (intrinsic and extrinsic), measurement noise values, measurement-based initial uncertainties of corresponding estimated system states, and other localization method-specific settings that can be forwarded to the chosen algorithms linked to the flight stack (cf. Sec. III-G and III-H).

Second, the file contains information on the mission sequence, including take-off, hover spots, potential interaction spots, and landing. This sequence is accompanied by mission paths, path segments and criteria when a waypoint (3D position, orientation, and an optional holdtime) is reached.

Third, the user includes information for contingency plans. This includes the safety checks and thresholds, the mitigation plans and their execution, and the different levels of severeness connected to failures of specified sensors and signals.

Of course, certain functionalities require a specific sensor type in order to be properly executed. The landing detection, for instance, needs a sensor which provides altitude or distance to ground information. This may be a barometer, visual-inertial odometry (VIO) source, general pose sensor, or a laser-range finder (LRF), like the one used in our experiments.

To illustrate the components of the flight stack in a less abstract form and how they might be used on a real system, the platform shown in Fig. 2 (schematically) and 3 (real system)

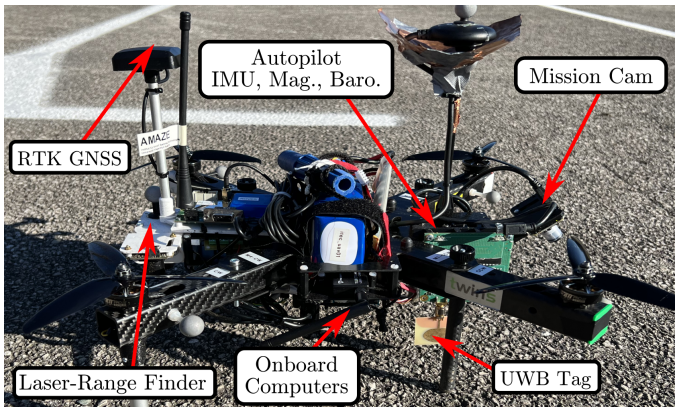

Fig. 3. The vehicle which is used in the experiment and field tests (Sec. IV and V) and hosts the system outlined by Fig. 1.

is introduced as an example. In the remainder of the paper, we will refer to this platform for more tangible explanations.

This platform can operate in two configurations, both with adequate sensing redundancy, which are demonstrated in the field-tests in Sec. IV and Sec. V. The first mode features a real-time kinematic (RTK) global navigation satellite system (GNSS) sensor, a tilted navigation camera, an LRF, and the full PixHawk4 sensor suite [10] (including an inertial measurement unit (IMU), a magnetometer, and a barometer). The platform also features two embedded computation boards (Odroid XU4) to show the option of splitting the computational load between multiple entities if needed. The second configuration excludes GNSS and magnetometer but includes an ultra-wideband (UWB) module with external anchors. These anchors are placed in the environment to support vehicle localization.

Sec. IV will show scenarios where different system failures are intentionally introduced and attributed contingency plans in said configuration file are executed.

C. Autonomy Engine

The autonomy engine is the core component of the CNS Flight Stack and is responsible for making high-level decisions to ensure the full functionality of an autonomous system. The autonomy engine is a fully configurable module (e.g., via the previously described configuration file). It allows the user to enable or disable other described modules, especially the watchdog, data recording, safety checks, or landing detection. Moreover, it provides a set of parametric background timers to account for maximum flight time and stalling of processes such as the watchdog or failure handling scripts. The main functionalities of the autonomy engine are based on the underlying state-machine shown in Fig. 4.

The autonomy engine's state-machine always starts at the state "*Uninitialized*". There, it waits for terminal-based user input to select a mission from the available mission set defined in the configuration file described above. Once the mission is selected, the autonomy engine switches to the "*Init*" state, which performs an initialization routine to initialize the other modules (e.g., watchdog). The "*Nominal*" state is set when the system is initialized and running but waiting for a signal to

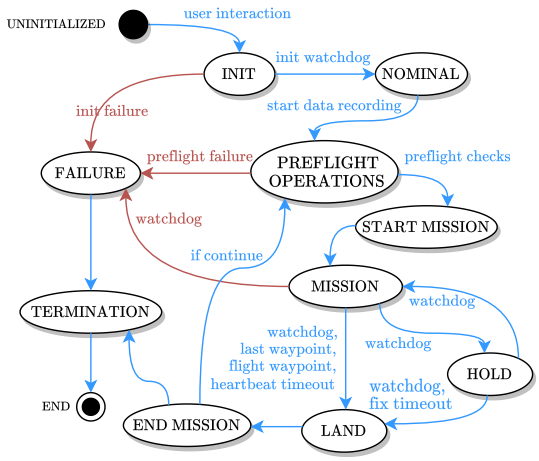

Fig. 4. State diagram of the autonomy engine: Most state changes are triggered by the system's status reported to the autonomy engine (e.g., watchdog status or landing detection) or triggered through timeouts. In each state the autonomy engine performs a predefined list of tasks, i.e., communicate actions to other modules. Upon receiving a failure from any other module (mostly from the watchdog) the autonomy engine also decides the corresponding action to safely land the vehicle or perform countermeasures.

start the mission. The "*PreflightOperations*" state is the state where all the pre-flight setup and safety checks are performed. With the success of the pre-flight operation, a transition to the "*StartMission*" state is triggered. Here the autonomy engine interacts with the mission sequencer, ensuring a correct arming and take-off of the platform. Once the mission sequencer reported that a correct take-off is performed, the autonomy engine communicates the mission's waypoints to the mission sequencer and switches to the "*Mission*" state.

Multiple state transitions are possible from the "*Mission*" state. Whenever the watchdog module reports a failure, the autonomy engine performs a state transition based on the reported severity of the failure (cf. Sec. III-E2 for more details on failure detection). "*Hold*" is the state triggered upon a non-severe failure that does not affect the ability to fly. In such a case, the autonomy engine first interacts with the mission sequencer module requiring the platform to hold in-place. Further, the autonomy engine delegates a pre-defined action to the watchdog to try and resolve the occurred failure. If the failure is resolved within a given maximum duration, the autonomy engine switches back to the "*Mission*" state and instructs the mission sequencer to resume the mission. If the failure could not be resolved in time or if the severity of the failure does not allow to keep the UAV airborne, the autonomy engine transitions to the "*Land*" state.

Reaching the last waypoint of the mission, reaching the maximum flight time, or failing to receive the watchdog's heartbeat message can also trigger a transition to the "*Land*" state. In this state, the autonomy engine tries to properly land the platform by communicating a land request to the mission sequencer. Upon receiving a confirmation of a successful landing from the landing detection module, the autonomy engine switches to the "*EndMission*" state, instructing the mission sequencer to disarm the vehicle. Depending on the configuration, the autonomy engine then either ends the mission ("*Termination*" state) or continues with the next trajectory in the list (transition and continue with "*PreflightOperations*").

In case of an unrecoverable failure (e.g., initialization failure, safety-checks failure, autopilot failure, or estimation error), a state transition to the "*Failure*" state is triggered. If the UAV is in-flight a message is displayed to signal the safety pilot to take manual control (in most cases before it would be humanly possible to detect the fault). If the vehicle is on the ground, it is disarmed, and the autonomy engine stops all modules of the flight stack.

The autonomy engine is mission-independent and fully customizable. All of the aforementioned states inherit from an abstract state interface, allowing the ability to easily add desired functionalities and individual custom states for a specific platform or mission.

D. Mission Sequencer

The mission sequencer is a complementary module of the autonomy engine and is responsible, once requested by the autonomy engine, for performing the mission in the form of a sequence of actions (e.g., takeoff, land, hold, etc., cf. Fig. 5). That is, the mission sequencer receives flight state requests from the autonomy engine. It then executes user pre-defined actions and communicates them to the autopilot via MAVLink[H] (specifically using MAVROS, a MAVLink extendable communication node for the robot operating system (ROS)). Therefore, this module is usable with any autopilot allowing MAVLink-based communication.

Further, to ensure safe operations of the UAV during a mission, the mission sequencer employs a state-machine adhering to the autonomy engine's request and current pose information of the UAV provided by the navigation state estimator. Fig. 5 depicts the different state-machine states and possible transitions of the mission sequencer. Upon receiving a state transition request, the mission sequencer performs an internal check to ensure the feasibility of the requested transition. It performs the associated actions and communicates the completion to the autonomy engine if feasible.

Before taking off (i.e., in the "*Idle*" state), the mission sequencer can "*Arm*" the UAV, which refers to starting the motors and spinning them at idle speed. Then, in the "*Takeoff*" state, a vertical takeoff to a parametric height is performed. After that, the mission sequencer automatically transits into the "*Hover*" state, awaiting navigation waypoints from any module (by default, the autonomy engine's waypoint list). Once any are received, transitioning to the "*Mission*" state, the mission sequencer communicates each waypoint to the autopilot.

Any module can communicate a list of waypoints to the mission sequencer at any time and replace, insert, or append the existing buffer. Further, the mission sequencer exposes a getter service such that any other module, e.g., obstacle detection and replanning modules, can acquire the current list of waypoints for possible modification.

Waypoints are interpreted with respect to either a global reference or the starting pose of the UAV. They also can be limited to a predefined bounding box (geofence) for safety reasons in limited environments, e.g., indoors. Any waypoint exceeding the bounding box will be skipped and not communicated to the autopilot. Once the last waypoint of the buffered

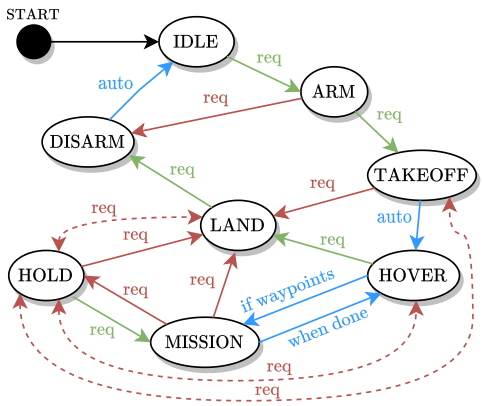

Fig. 5. State diagram of the mission sequencer. Green arrows indicate a regular request from the autonomy engine. Arrows in blue indicate automatic state transitions that depend on reaching the waypoint position. Arrows in red indicate error requests from the autonomy engine to resolve high-level errors. Red-dashed arrows are "*Hold*" requests from the autonomy engine. These can theoretically be issued in any flight state, and the mission sequencer continues afterward with the previous state upon a resume request.

list is reached, the mission sequencer switches to the "*Hover*" state. It communicates this transition to the autonomy engine, which decides the following action (either to land or hover).

The mission sequencer has a particular state called "*Hold*". As previously mentioned, the autonomy engine might issue a transition to the "*Hold*" state of the mission sequencer upon a minor sensor failure. The action and behavior of the "*Hold*" state look the same as the "*Hover*" state to the user. However, the mission sequencer requires a request to either resume its previous state or land when holding. By default, the latter will be issued if the failure could not be resolved in time.

Finally, in the "*Land*" state, the mission sequencer descends the UAV until it is on the ground. If pose estimates are available, they are used to navigate to the desired landing position by lowering the desired height. Otherwise, the mission sequencer gradually reduces the vehicle's thrust. Note that this selection and safety procedure, similar to the transition to the "*Hold*" state, happens without user input and inherently chooses the safest flight mode to prevent crashes and damage. In any case, once the landing detector confirms a landing, the autonomy engine might request a transition to the "*Disarm*" state, which then turns off the UAV's motors.

E. Safety Checks

The CNS Flight Stack provides multi-stage safety features. Different pre-flight checks, a fault detection system (watchdog) monitoring the system's status during the operational phase, and a landing detection module.

1) Pre-Flight Checks: The first of the pre-flight checks is to check if the platform is level before arming: let ${}_I\bar{\mathbf{a}}$ be the average measured acceleration from the IMU over a predefined, configurable window of time, and $\mathbf{e}_1 = \begin{bmatrix}1, 0, 0\end{bmatrix}^\mathsf{T}$ be a reference vector, then we compute the axes $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, $\hat{\mathbf{z}}$ of the IMU frame expressed in the world frame as follows

$$\hat{\mathbf{z}} = \frac{{}_I\bar{\mathbf{a}}}{\|{}_I\bar{\mathbf{a}}\|}, \qquad \hat{\mathbf{x}} = \frac{\mathbf{e}_1 - \hat{\mathbf{z}}\hat{\mathbf{z}}^\mathsf{T}\mathbf{e}_1}{\|\mathbf{e}_1 - \hat{\mathbf{z}}\hat{\mathbf{z}}^\mathsf{T}\mathbf{e}_1\|}, \qquad \hat{\mathbf{y}} = \lfloor\hat{\mathbf{z}}\rfloor_\times\hat{\mathbf{x}}.$$

The pitch θ and the roll ϕ of the platform are extracted from the rotation matrix $\mathbf{R}_{WI} = \begin{bmatrix}\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}\end{bmatrix}$ and then compared with a predefined threshold to ensure flatness. Please note that this threshold is set in the aforementioned configuration files and should include bias-induced inclination within the check period. Furthermore, when a sensor providing height information is present, averaged height measurements are compared with a configurable height threshold to ensure the platform is on the ground.

The second pre-flight check is to ensure that the external pose estimator is converged. This check compares the norm of the difference between the position estimate with respect to the world frame ${}_W\mathbf{r}_{WI}$ at time t and at time $t + \Delta t$ given a predefined difference threshold. If both checks succeed, the autonomy engine instructs the UAV to be armed.

2) Watchdog: The watchdog's responsibilities are twofold. First, to observe and detect faulty entities (such as sensors, battery, or ROS nodes/topics) by checking if a predefined behavior is violated. Second, to perform a sequence of corrective actions to re-establish a nominal condition. Such actions are defined in the aforementioned human-readable configuration files. Each entity is given an action set, expected restore duration, and a unique ID that is known to the autonomy engine and is used to communicate its status across the flight stack.

For the sensors, often separate (sometimes proprietary) driver code is executed. Thus, the watchdog monitors their system processes according to the user-defined configuration file entries. The entries can even include script calls allowing monitoring and acting on system components even outside the ROS universe. A user-specified maximum number of automated correction trials is used to elevate the severity of the failure under consideration.

For ROS-related entities, the configuration file allows defining sections with a unique ROS node name and the number of restart attempts. If, after restarting the node, the restore duration is exceeded, an elevated node failure is triggered. Similarly, the configuration for ROS topics allows the definition of a section per (unique) ROS topic name, the expected message rate with a certain tolerance, and a restore duration for the observation in case the corresponding ROS node (or driver) was restarted. The watchdog can also observe sensor measurements (e.g., battery voltage level) and report a failure to the autonomy engine if a pre-defined range is exceeded.

The watchdog parses its configuration file containing all the above-mentioned information upon obtaining a start request by the autonomy engine. Then, a so-called *Observer* for each entity is created. Every *Observer* follows a state-machine that is polled periodically at a predefined rate by the watchdog to track the status of each individual entity. At every poll, faulty entities are reported to the autonomy engine, which makes a decision based on the current system status. In case of failures, it authorizes the watchdog to perform the pre-defined action for a recovery attempt (e.g., restart a sensor's driver or ROS node). In addition, the watchdog communicates its own heartbeat at a predefined rate to the autonomy engine to allow the detection of a stalled process.

3) Landing detection: The landing detection is an extra safety module that can be used whenever a sensor providing altitude above ground level (AGL) information is present, such

as an LRF. The landing detection is only active during flight to either detect a landing while in the autonomy engine's "*Landing*" state and therefore disarm the UAV, or to recognize an unexpected touchdown. The latter case assumes a detection due to unforeseen structure. Thus to avoid a crash, the autonomy engine instructs the mission sequencer to properly land and then disarm the UAV.

F. Data Recording and Log Analysis

In order to perform post-analysis of the flight and autonomous decisions made, it is necessary to record all data and communication. All our developed components provide their communication and decision-making redundantly via log files and ROS messages. Hence, one can recreate each flight and inspect each module's decision offline. Depending on the underlying hardware, storing the measurements on different media devices or communicating them through a network connection may be required. Our system already provides the tools to perform either option and allows each sensor measurement to be stored accordingly, extending available tools for the flight stack's needs [6], [11]. Hence, one can easily adjust the data recording and storing for their needs.

G. ROS-PX4 Bridge

Our UAV platform used in this work features a PixHawk 4 embedded autopilot [10] with PX4 flight software [1]. This autopilot includes its own state estimation modules, which are tightly interwoven into the PX4 software. This makes it difficult to connect safety-observers and -actions and include custom-made probabilistic state estimators. For R&D in academia and industry, safe testing of novel state estimators, trajectory generators, and mission planners is, however, of high importance. Rather than daisy-chain novel algorithms with the gray-box heuristics of PX4, we propose to bypass these heuristics altogether with our ROS-PX4 bridge.

For this, the CNS Flight Stack provides an additional module to the PX4 firmware which bypasses said heuristics (e.g., such as the PX4 internal extended Kalman filter (EKF)) and provides an external state estimation directly to the PX4 controller. Besides removing the effect of introducing inconsistencies by daisy-chaining probabilistic estimators, our bridge module enables a well-defined, reproducible interaction with the PX4 autopilot.

The bridge module includes two components, our new MAVLink/MAVROS plugin for communication between an external state estimator and the PX4 firmware and our new PX4 firmware module that receives this information and passes the estimate directly to the internal controller. Our internal PX4 module also communicates status flags related to the state estimator to the commander for pre-flight checks in order to maintain the communication requirements on the PX4 side.

H. Pose & Velocity Estimation

Although the CNS Flight Stack would allow the inclusion of any external localization module, for demonstration purposes, we include a specific probabilistic navigation-state estimator described in the following. We include the MaRS framework [12], a state-of-the-art modular EKF-based multi-sensor fusion framework developed in-house, and OpenVINS [13] as an open-source state-of-the-art VIO framework which was further developed for its application in the CNS Flight Stack. The output is fed directly to the onboard PX4 controller using our ROS-PX4 bridge described above. We choose MaRS due to its modularity and simplicity to add and remove common UAV sensors.

In our setup of MaRS, we use the IMU for propagation, and the core states are: Position expressed in the world frame ${}_W\mathbf{r}_{WI}$, velocity w.r.t. the world frame ${}_W\mathbf{v}_{WI}$, orientation of the robot w.r.t. the world frame ${}_W\mathbf{q}_I$ as well as IMU biases for the gyroscope ${}_I\mathbf{b}_\omega$ and accelerometer ${}_I\mathbf{b}_a$. The core state is described as follows and corresponding dynamics can be found in previous work [12]:

$$\mathbf{x}_{\text{core}} = \begin{bmatrix} {}_W\mathbf{r}_{WI}^\mathsf{T}, & {}_W\mathbf{v}_{WI}^\mathsf{T}, & {}_W\mathbf{q}_I^\mathsf{T}, & {}_I\mathbf{b}_a^\mathsf{T}, & {}_I\mathbf{b}_\omega^\mathsf{T} \end{bmatrix}^\mathsf{T}$$

Compared to other EKFs, MaRS does not only provide modularity and on-demand system retrofitting with different sensors, but it also minimizes the mathematical operations to a bare minimum for online, onboard operation. Thus, the system is more computationally efficient even if the sensor suite is not synchronized and supports asynchronous update requests. Real-world examples of included sensor modules, e.g., GNSS, UWB, barometer, and VIO (used as a pose sensor), are provided in Sec. V. The modularity of MaRS is a handy feature in combination with our proposed CNS Flight Stack and its safety processes since sensors can dynamically be removed and added to the estimation process according to the requests of the autonomy engine (via watchdog triggers).

We extended the original OpenVINS implementation with an advanced initialization routine that uses available sensors (e.g., GNSS, magnetometer, barometer, etc.) to initialize the VIO part in a common world reference frame together with MaRS. In addition, we introduced a tightly-coupled magnetometer and UWB update routine, making the core navigation state fully observable and aligned with MaRS.

IV. Experiments

To test the proposed flight stack, we performed various experiments in which we purposefully failed different sensors and let the flight stack recover or safely land. The following tests were performed using the hardware previously described as mode 2 (cf. Fig. 2: red-green setup including UWB, LRF, and VIO navigation camera).

1) Nominal Flight: We first show the proposed flight stack's communication and state changes in a nominal autonomous flight. As shown in Fig. 6, the first item to start is the watchdog to monitor and check the system in the background. To circumvent any sensor startup delays, it then observes the system for a given time of $10\,\mathrm{s}$ (blue). Once the system is reported to be nominal, the data recording is started and the autonomy engine switches to the "*PreflightOperations*" state (orange). Upon confirmation by the user (red line), the pose estimation is observed for a given time ($10\,\mathrm{s}$), and the takeoff condition is checked. If successful, the vehicle is armed

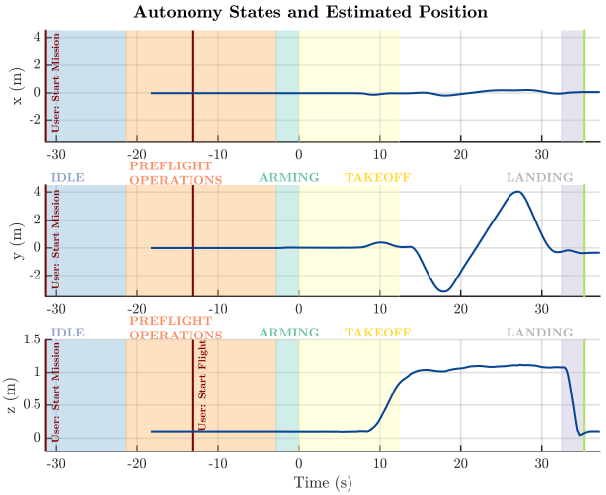

Fig. 6. In a nominal flight the CNS Flight Stack autonomously starts its background monitoring (blue), performs pre-flight checks (orange), arms and spins the motors (green), performs takeoff (yellow), performs a pre-defined waypoint mission (white), lands (purple), and stops/disarms the rotors (olive green). The blue line is the estimated position by MaRS and the red vertical lines the user input to start the autonomy engine and flight, respectively.

(green), and the takeoff is performed (yellow). Upon reaching the predefined takeoff height, the mission is performed using the predefined flight path (white). Finally, upon reaching the last waypoint, the UAV autonomously lands (purple). Then, signaled by the landing detection, the vehicle is disarmed (olive green) and, the user can select a new mission.

2) Failure Handling: Next, we perform a visual-inertial-based flight and trigger several failures with increasing severity one after another. As outlined in Sec. III-B, the user chooses each sensor failure's severity, which should reflect the mission's goal. For this test, we chose to fail (i) the UWB module – irrelevant for the pose estimation and thus flight continues – (ii) the LRF – required only for takeoff and landing detection, triggering a "*Hold*" and action to fix it in-flight – and (iii) the navigation camera – forces our state-estimation to fail and thus requires immediate landing.

As presented in Fig. 7, the UWB failure is triggered after 5 s. Since it is irrelevant for pose estimation, the flight continues. Then the LRF is failed twice at seconds 10 and 15. Immediately a "*Hold*" is issued and a sensor recovery attempt is communicated. Here, the fix was to restart the LRF's driver, which successfully worked in both cases, and thus the flight continued. Finally, the camera is deactivated at second 21, which causes the vision-based pose estimation to fail. We are able to detect this failure almost immediately and land the vehicle in a safe manner using MaRS' IMU propagation.

V. Field Tests

With the first field test, we show the proposed system handling a camera sensor failure, resulting in a vision-based pose estimation failure. Since our platform is equipped with other, low-rate or less accurate sensors (cf. Fig. 2: red-yellow setup including RTK GNSS and an LRF), this outage can be bridged using their measurements within MaRS. It is important to note that even though an RTK-grade GNSS signal is available, we deliberately use this signal only at 1 Hz. At this frequency, it provides the necessary information to eliminate long-term

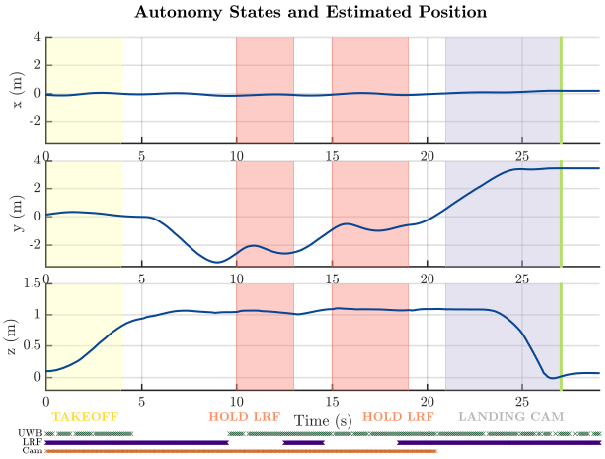

Fig. 7. In this vision-based navigation experiment we manually triggered sensor failures to test the autonomous system recovery and failure modes. First we triggered a non-relevant sensor failure (at second 5) which had no influence on the navigation performance and thus flight continued. Then we failed a minor mission-relevant sensor, such as the LRF, twice (at seconds 10 and 15) which triggered a "*Hold*" by the autonomy engine (red). Meanwhile, the watchdog was trying to perform a sensor fix and upon each successful recovery, the flight was continued. Finally, we triggered a major navigation sensor failure, i.e., the camera, which requires immediate landing for vision-based state estimation (at second 22). This was also detected in time and the vehicle safely landed using limited estimation capabilities (purple).

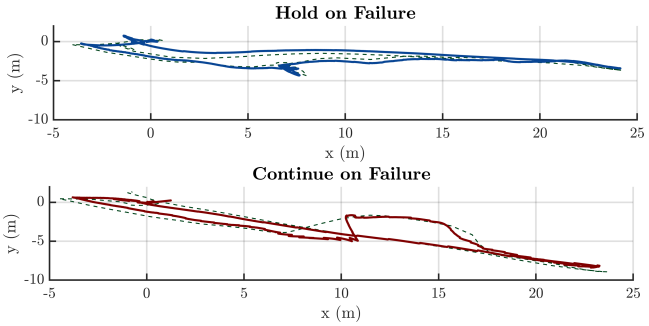

Fig. 8. Aligned comparison of 15 Hz-VIO fused with 1 Hz-GNSS pose estimation and closed-loop navigation in two scenarios: holding (blue) or continuing (red) the flight plan when the camera fails. In both cases the failure duration is bridged in MaRS by fusing the low-rate GNSS measurement with the IMU measurements. The respective ground-truth from a higher rated RTK GNSS sensor is shown in green. Comparing the two resulting closed-loop trajectories, it is clearly visible, that holding the UAV in-place while recovering the visual sensor is preferable compared to continuing with the other sensors. Thus the CNS Flight Stack provides a high-level systematic approach on handling sensor failures and the resulting estimation dropouts.

visual-inertial drift but is in itself unsuitable for pure GNSS-inertial-based navigation that goes beyond position-hold. Thus, whenever the primary navigation information, the camera, is lost, the best strategy is to hold the position best possible until the camera is available again. Holding the positing reduces the accumulated offset, and thus the control jump when the visual-inertial framework is re-initialized and the mission is continued.

Fig. 8 displays the resulting position estimate and flight path when the UAV holds (in blue) or continues (in red) its flight plan upon having a camera sensor (and thus VIO) failure. It is clearly visible that (without altering the VIO estimation algorithm) holding the vehicle in-place results in no visible position jumps and thus VIO propagation error.

In comparison, if the mission is continued, once the camera is restarted and VIO resumes estimation, the new pose estimate

TABLE I
STATISTICS ON RECORDED FLIGHTS USING THE CNS FLIGHT STACK WITHIN THIS PAPER'S EXPERIMENTS AND OTHER PROJECTS

SUCCESSFUL FLIGHTS		FAILED FLIGHTS
incl. recoveries	safety landings	pilot takeover
326 (89.3 %)	26 (7.1 %)	13 (3.6 %)

differs significantly from the previous estimates. The reason is twofold: First, continuing the flight with the very low rate GNSS signal fused with IMU results in an imprecise trajectory tracking with a large error to the nominal position. Second, the re-initialization of the vision component in a new scenery generally does not allow to link the currently seen scene with the last obtained image before the failure. In our specific test case in Fig. 8, we ensured that OpenVINS was able to maintain at least a sparse set of persistent features to reduce the position jump upon regaining the camera feed, to allow MaRS the online re-calibration of the offset between GNSS and visual reference frame, and for better legibility of the graph.

Note that this issue could also be tackled using scenario and senor suite specific estimation, initialization, and sensor switching routines to bridge the time of visual cue loss. Yet, to the best of our knowledge, the proposed flight stack is the first to solve this issue from a flight system point-of-view in a more abstract and general fashion.

VI. SUMMARY

The CNS Flight Stack was used within several projects[I,J,K] and has so far recorded over 450 runs. Of these, approximately 20 % resulted in the flight stack detecting non-nominal conditions before takeoff and thus preventing in-flight failures before the flight even started. However, most of them were user-introduced with the purpose of testing the flight stack and its pre-flight failure detection.

Testing flight software on real platforms can always yield unforeseen problems and behavior. Nonetheless, the CNS Flight Stack has successfully flown, recovered, or safely landed the UAV in 96.4 % of its flights (cf. statistics in Table I). It was also deemed safe enough for usage in an artistic performance with robot-human interaction[K]. Further, an example recovery scenario after sensor failure with an UWB-vision-based flight is given in Fig. 9. In a small number of flights, external influences such as wind condition changes, unforeseen obstacles, or communication loss, a pilot took over manual control. An IMU-based landing would then have been triggered in the latter case, but the safety pilot took over in those cases since the mission had a different focus. The advanced monitoring and fast warning capabilities provided enough time for the pilot to intervene before any severe damage was caused.

ACKNOWLEDGMENT

The authors would like to express special thanks to all administrative and technical personnel, who made this project possible, specifically to Melissa Aichholzer, Fred Arneitz, and Patrik Grausberg. Further, the authors would like to express their gratitude towards the Austrian Space Forum (OeWF) for the possibility to test and fly the proposed system in the Analog Mars Mission AMADEE-20[I].

[I] oewf.org/amadee-20 [J] bugwright2.eu [K] Woyzeck Panopticon

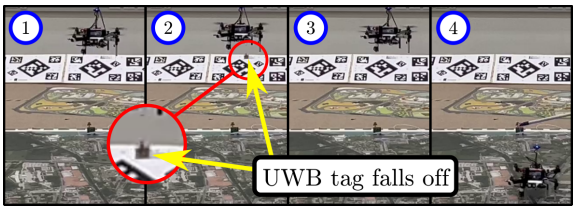

Fig. 9. Real flight scenario of the CNS Flight Stack detecting a hardware failure. While performing a pre-defined mission (1), a UWB antenna was physically dismounted due to vibrations (2), the flight stack trying to fix the failure while holding (3), and finally landing the platform safely as a hardware failure cannot be fixed in-flight (4). This flight is performed using visual-inertial-UWB estimation. The Aruco tags in the background are part of the environment and not used in this flight.

REFERENCES

[1] 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)*. Seattle, Washington, USA: IEEE, 5 2015, pp. 6235–6240.

[2] Z. Luo, X. Xiang, and Q. Zhang, "Autopilot System of Remotely Operated Vehicle Based on Ardupilot," in *Intelligent Robotics and Applications*, H. Yu, , J. Liu, , L. Liu, , Z. Ju, , Y. Liu, , and D. Zhou, Eds. Cham, Switzerland: Springer International Publishing, 2019, pp. 206–217.

[3] S. Staroletov, "Work-in-Progress Abstract: Revealing and Analyzing Architectural Models in Open-source ArduPilot," in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. Houston, TX, USA: IEEE, 8 2021, pp. 207–209.

[4] X. Liu, G. V. Nardari, F. C. Ojeda, Y. Tao, A. Zhou, T. Donnelly, C. Qu, S. W. Chen, R. A. F. Romero, C. J. Taylor, and V. Kumar, "Large-scale Autonomous Flight with Real-time Semantic SLAM under Dense Forest Canopy," *IEEE Robotics and Automation Letters (RA-L)*, 9 2022.

[5] A. Koubaa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui, "Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey," *IEEE Access*, vol. 7, pp. 87 658–87 680, 2019.

[6] C. Stewart, "Skiffos: Minimal cross-compiled linux for embedded containers," 2021.

[7] R. D. Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in FOSS distributions: details and challenges," *CoRR*, vol. abs/0902.1610, 2009.

[8] "Buildroot: Linux system cross-compiler."

[9] R. Morabito, "A performance evaluation of container technologies on internet of things devices," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2016, pp. 999–1000.

[10] L. Meier, P. Tanskanen, L. Heng, G. H. Lee, F. Fraundorfer, and M. Pollefeys, "PIXHAWK: A micro aerial vehicle design for autonomous flight using onboard computer vision," *Autonomous Robots*, vol. 33, pp. 21–39, 8 2012.

[11] O. Tange, "GNU Parallel: the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, 2011.

[12] C. Brommer, R. Jung, J. Steinbrener, and S. Weiss, "MaRS: A Modular and Robust Sensor-Fusion Framework," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 359–366, 2021.

[13] P. Geneva, K. Eckenhoff, W. Lee, Y. Yang, and G. Huang, "OpenVINS: A Research Platform for Visual-Inertial Estimation," in *Proc. of the IEEE International Conference on Robotics and Automation 2020*, 2020.