ARL-TR-10151 ● AUG 2025

Autonomous Drop-Drone System

by Nikolas Vale and Joshua Gray

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-10151 ● AUG 2025

Autonomous Drop-Drone System

Nikolas Vale
DEVCOM Army Research Laboratory

Joshua Gray
Fibertek Inc.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
August 2025	Technical Report	10/2024	03/2025

4. TITLE AND SUBTITLE
Autonomous Drop-Drone System

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Nikolas Vale and Joshua Gray

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-PB Adelphi, MD 20783	ARL-TR-10151

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCID: Joshua Gray, 0009-0006-2615-0287

14. ABSTRACT
This report details the design and implementation of both software and hardware developed for an autonomous drop-deployable unmanned aerial vehicle (UAV). The primary challenge addressed is the development of a system capable of detecting a free-fall or tumble state and executing an automated arming sequence to achieve level flight. The solution centers around a Teensy 4.1 microcontroller acting as a high-level "man-in-the-middle" logic processor. This processor reads data from the primary flight controller's inertial measurement unit, interprets inputs from physical switches and a remote-control receiver, and executes a multistage state machine. The software uses accelerometer and gyroscope data to reliably detect deployment. Upon detection, it executes a timed arming sequence, sending manipulated SBUS control signals to the flight controller. The system's functionality was validated through a series of drop tests. This work demonstrates the capability for a UAV to be deployed in air and arm and stabilize prior to higher-level autonomy or pilot control.

15. SUBJECT TERMS
drones, autonomous arming, air deploy, marsupial release and arm in air, Photonics, Electronics, and Quantum Sciences

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	19

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Nikolas Vale	(520) 691-5113

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Introduction

This project addresses the operational need for an unmanned aerial vehicle (UAV) that can be deployed from a significant height and autonomously arm and stabilize in air. The core problem is enabling a UAV to safely manage its own deployment and arming sequence, recovering to a stable, controllable state from which a mission can be initiated. This capability is not supported by standard off-the-shelf flight controller software.

The developed solution is a software-based logic controller hosted on a Teensy 4.1 microcontroller. This system successfully integrates with a primary flight controller to read sensor data, detects a free-fall or tumble state using a custom algorithm, and executes a multistage arming sequence. Upon completion of the sequence, the UAV stabilizes and is ready to be controlled.

The result is a functional prototype that demonstrates the viability of the "man-in-the-middle" processor approach for creating advanced deployment capabilities. The system reliably detects the drop event and successfully transitions the UAV to a flyable state.

It is concluded that this firmware architecture is an effective method for implementing complex, mission-specific behaviors. It is recommended that future work focuses on simplifying and optimizing the existing arming software, with the goal of integrating this deployment capability into a fully autonomous system that can arm, stabilize, and then execute a preprogrammed mission.

1.1 Problem Statement

In search-and-rescue, defense, or remote surveillance scenarios, there is often a need to deploy a UAV into an area that is not accessible for a traditional ground takeoff. This project addresses the challenge of creating a UAV that can be safely dropped from a parent aircraft or elevated position and autonomously transition into a stable, controlled flight, ready to accept commands from a pilot or another onboard system.

1.2 Purpose and Scope

The purpose of this project is to design, build, and test a software system that enables a standard quadcopter to perform an autonomous drop-deployment and achieve a stable, controllable state.

The scope of this work encompasses the physical design of the UAV chassis as it relates to the testing methodology, and the development of the high-level logic and control software. This includes the following:

- Interfacing with a primary flight controller to acquire sensor data.
- Developing algorithms for reliable free-fall and tumble detection.
- Creating a state machine for a safe, multistage arming sequence that results in a stable hover.
- Manipulating and passing control signals to the flight controller.
- Verifying system performance through empirical testing.

This report does not cover subsequent mission-specific autonomous behaviors beyond the initial stabilization but considers the successful integration of the physical and electronic components to achieve a mission-ready state.

1.3 Report Development Plan

This report first outlines the hardware and software architecture, detailing the methods, procedures, and assumptions made during development. It then presents the results from system testing, with a discussion of their significance. Finally, it draws conclusions based on those results and provides recommendations for future development.

2. Methods, Assumptions, and Procedures

2.1 Chassis and Physical Design

A key requirement for this project is the ability to perform rapid, iterative software tuning, which involves numerous physical deployment tests. Consequently, the UAV chassis was designed with durability and impact resistance as a primary driver. As seen in Figure 1, the chassis consists of two 1/8-inch carbon-fiber frames connected to each other via aluminum standoffs, thus creating a cage that holds the motors and propellers inside. The electronic components and battery were mounted on the bottom.

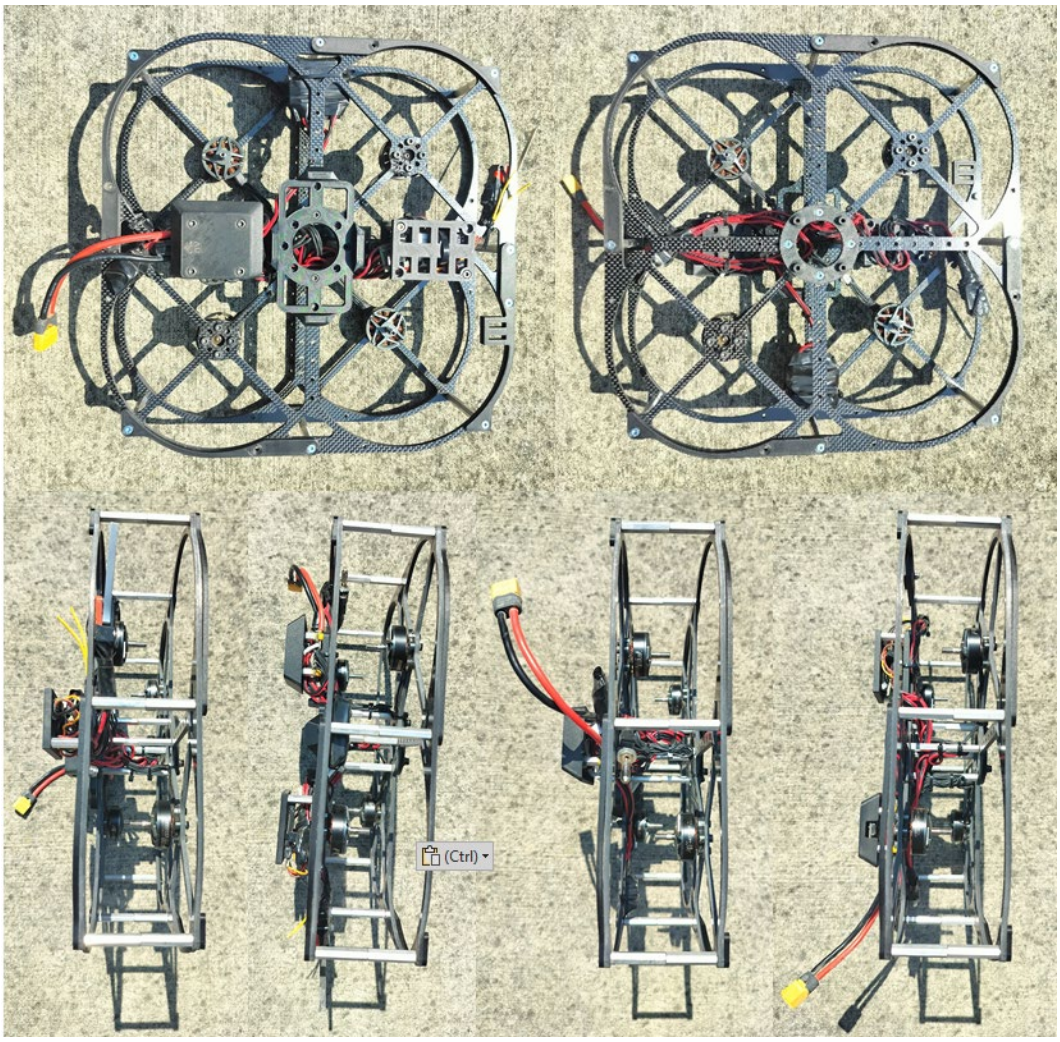

Figure 1. UAV chassis with electronics.

This robust physical design was a critical enabler for the empirical testing methodology. It ensured that unsuccessful test deployments, where the UAV might fail to arm and subsequently impact the ground, were nondestructive. This survivability allowed immediate software parameter adjustments and retesting, dramatically accelerating the development and tuning cycle without the need for lengthy and costly repairs.

2.2 Electronic and Control System Architecture

The system integrates a suite of commercially available components, each selected to fulfill a specific role within the control architecture. The Teensy 4.1 microcontroller serves as the core of the custom logic, interfacing with the BrainFPV Radix 2 HD flight controller. A detailed list of primary components is provided in Table 1.

Table 1. Primary hardware component specifications.

Component	Specification	Purpose
Flight controller	BrainFPV Radix 2 HD	In combination with the microcontroller, central processing unit for flight control and stabilization
Microcontroller	Teensy 4.1	Handles sensor input, release detection logic, and communication between components. It also plays a critical role in stabilized flight control.
Receiver	Jeti Duplex EX R5	Receives control signals from the transmitter and feeds them into the microcontroller
Transmitter	Jeti Duplex DS-12	Real-time control of UAV flight parameters (throttle, pitch, roll, yaw) along with telemetry display (transmitter voltage, signal strength), flight mode selection, and fail-safe configuration.
Electronic speed control	T-Motor F55A ProIII 6S 4-in-1	Controls motor speed based on flight controller commands
Motors	T-Motor F90 kV 1300	Provides thrust for flight
Battery	LiPo 6S 1850 mAh 150C	Powers the system
Switches	SPDT toggle switch	Provide manual control input to indicate it should be in "arm in air" mode
	Momentary snap-action switch	Detect release event from payload holder
Power regulators	Pololu 12V, 15A step-down voltage regulator D24V150F12	Provide stable power for the flight controller
	Pololu 5V, 5.5A step-down voltage regulator D36V50F5	Provide stable power for the microcontroller

The Teensy interfaces with these components as detailed in Table 2 and Figure 2. This "man-in-the-middle" configuration allows the Teensy to augment the flight controller's capabilities.

Table 2. System hardware and Teensy pinout configuration.

Component	Teensy pin	Signal type	Function
RC receiver	9	PPM input	Reads pilot commands
Flight controller	0 (RX1), 1 (TX1)	UART (MSP)	Reads sensor data from flight controller
Flight controller	8 (TX2)	UART (SBUS)	Sends control commands to flight controller
Toggle switch	2, 3	Digital input	Arming sequence trigger
Bumper switch	4, 5	Digital input	Arming sequence trigger
Status LED	13	Digital output	Provides visual feedback on system state

Notes: RC = remote control; PPM = pulse-position modulation; RX = receive; TX = transmit; UART = universal asynchronous receiver/transmitter; MSP = MultiWii Serial Protocol; SBUS = Serial Bus; LED = light-emitting diode

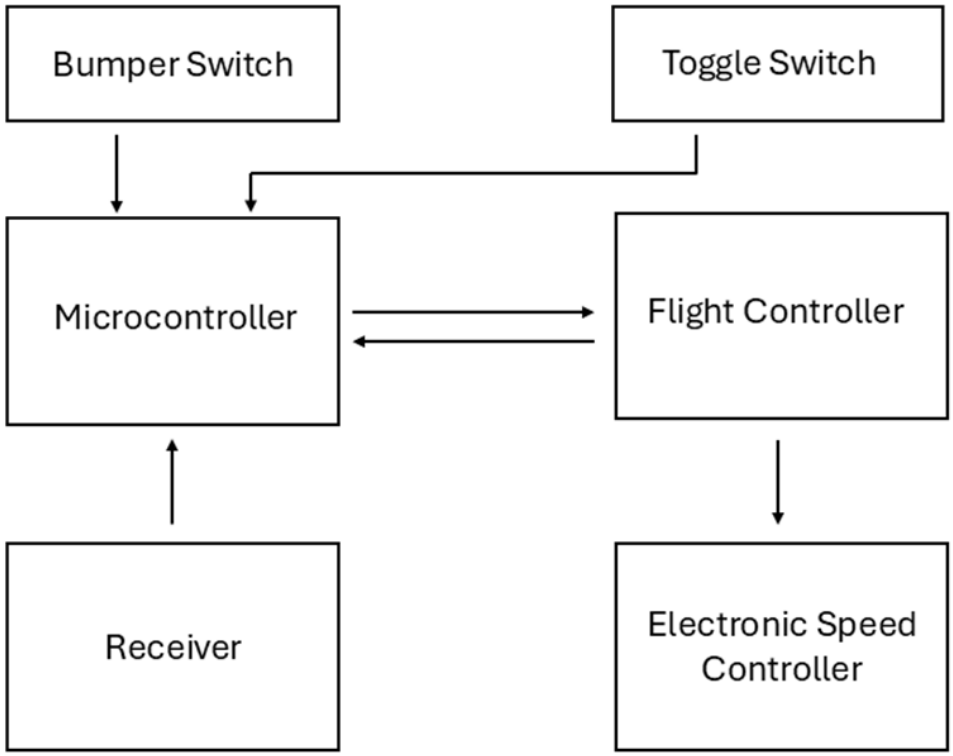

Figure 2. System hardware flow chart.

2.3 Firmware Design and Implementation

The firmware was written in C++ using the Arduino IDE with the Teensyduino add-on. The logic is contained within a single main loop that continuously calls functions for each subsystem. The firmware's logic was validated throughout the development process via iterative empirical testing.

2.3.1 System Initialization

The `setup()` function (Figure 3) configures all hardware peripherals and software libraries at boot.

```
void setup() {
  Serial1.begin(115200);
  Serial.begin(115200);
  myIn.begin(9);
  sbus_tx.Begin();
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(4, INPUT);
  pinMode(5, INPUT);
  pinMode(13, OUTPUT);
  SD.begin(chipSelect);
}
```

Figure 3. Code snippet for system initialization.

2.3.2 Sensor and Signal Processing

The firmware's main loop continuously processes data from the flight controller's sensors and the pilot's RC transmitter.

2.3.2.1 MultiWii Serial Protocol (MSP) Sensor Data Acquisition

The firmware polls the flight controller for real-time sensor data using the MSP, a well-defined request-response protocol (Figure 4). A request message is constructed with a specific command ID, and the firmware parses the subsequent reply. A delay(25) is used to provide the flight controller sufficient time to process the request and transmit a full response before the Teensy attempts to read the serial buffer.

```
void msp_attitude_request() {
  // ... (timing logic to ensure periodic requests) ...
  uint8_t checksum_attitude = 0;
  // MSP message preamble ($M<), payload length (0), and command ID (108)
  Serial1.write((byte *) "$M<", 3);
  Serial1.write(0);
  checksum_attitude ^= 0;
  Serial1.write(108); // MSP_ATTITUDE command
  checksum_attitude ^= 108;
  Serial1.write(checksum_attitude);
  delay(25); // Wait for the flight controller to respond
  while(Serial1.available()){
    byte attitude[12];
    for(uint8_t nv = 0; nv < 12; nv++) {
       attitude[nv] = Serial1.read();
    }
    // Values are 16-bit, little-endian. They are divided by 10 to get
degrees.
    angx = ((attitude[6] << 8 | attitude[5]) & 0xFFFF) / 10;
    angy = ((attitude[8] << 8 | attitude[7]) & 0xFFFF) / 10;
    heading = ((attitude[10] << 8 | attitude[9]) & 0xFFFF);
  }
}
```

Figure 4. Code snippet for MSP attitude request.

2.3.2.2 Inertial Measurement Unit (IMU) Data Processing and Gravity Compensation

To reliably detect a free-fall, the raw accelerometer data must be compensated for the effect of gravity (Figure 5). The firmware uses trigonometric functions based on the drone's current roll (angx) and pitch (angy) angles to subtract the gravity vector from the accelerometer readings. The resulting gacc value represents the true acceleration of the drone, which will approach zero during a free fall. This value is passed through a moving average filter to smooth out sensor noise.

```
gaccx = (accx * sin(abs(angy) * (PI/180)));
gaccxz = (abs(accz) * cos(abs(angy) * (PI/180)));
gaccxz = abs(gaccx) + abs(gaccxz);
// ... (similar calculation for y-axis) ...
gacc = acc_average((gaccxz / (2-sin(abs(angy)*(PI/180)))) +
(gaccyz / (2-sin(abs(angx)*(PI/180)))));
```

Figure 5. Code snippet for IMU gravity compensation.

2.3.2.3 RC Signal Input and Failsafe

The rc_read_in_ppm() function reads the pulse-position modulation (PPM) signal, which encodes all RC channels into a timed sequence of pulses on a single wire (Figure 6). A critical safety feature is implemented here: a failsafe timer is reset every time a valid RC signal is received. If the time since the last valid signal exceeds 2 s, the firmware assumes a loss of connection and overrides all pilot inputs, forcing the throttle channel to its minimum value and the kill switch channel to its maximum (disarmed) value to prevent a flyaway.

```
void rc_read_in_ppm() {
  // ... (timing logic to run periodically) ...
  if(myIn.available()) {
    // ... (loop to read ppm channels 1-8) ...
    rc_in = 1; // Flag that a new signal was received
  }

  if(rc_in == 1) {
    wait_safety_new = micros(); // Reset failsafe timer on new signal
    rc_in = 0;
  }

  // If 2 seconds (2,000,000 microseconds) have passed since the last signal
  if(wait_safety_new + 2000000 < micros()) {
    ppm[5] = 2000; // Set kill switch channel to max (disarmed)
    ppm[0] = 900;  // Set throttle channel to min
  }
}
```

Figure 6. Code snippet for RC signal failsafe.

2.3.3 Core Logic: Drop Detection and Arming

The core of the firmware is a state machine that manages the arming sequence. The system progresses through several stages based on physical switch inputs. The final automated arming is triggered only when the system is in the correct stage, the pilot has enabled the auto-arm mode via RC, and the drop-detection algorithm has been triggered (Figure 7). The detection algorithm uses a dual-condition approach, monitoring for either a near-zero G condition (free fall) or a high-rotation condition (tumble) to ensure robust detection across different deployment scenarios.

```
void accel_drop_detect(){
  accel_threshold = 700;
  ang_threshold = 2;
  gyr_threshold = 2;

  // Condition 1: Free-fall detection
  if(gacc <= accel_threshold) {
    if(micros() > accel_wait_new) { // Check if condition persists
      accel_trig = 1;
      free_fall_trig = 1;
    }
  } else {
     accel_wait_new = micros() + 150000; // Reset timer
     free_fall_trig = 0;
  }

  // Condition 2: Tumble detection
  if(gyro_accel_x >= gyr_threshold && ang_avg >= ang_threshold) {
    if(micros() > gyr_wait_new) { // Check if tumble has persisted
      accel_trig = 1;
      tumble_trig = 1;
     }
  } else {
    gyr_wait_new = micros() + 250000; // Reset timer
    tumble_trig = 0;
  }
}
```

Figure 7. Code snippet for dual-condition drop detection.

The arming sequence itself is a timed, two-step process for safety (Figure 8). First, a 250-ms timer is started to arm the flight controller. Then, after an additional 125 ms, a second timer allows the pilot's throttle input to pass through. This staggered sequence was necessary since the flight controller firmware would not let the system arm if the throttle is too high.

```
void drop_arm() {
  // ... (logic for stages 1-4) ...
  if(stage == 5 && auto_arm_mode > 1750 && accel_trig == 1) {
    if(trig_trig == 0) {
      trig_wait_new = micros() + 250000; // Arm after 250ms
      trig_trig = 1;
    }
    if(thro_trig == 0) {
      thro_wait_new = trig_wait_new + 125000; // Throttle up after 375ms
      thro_trig = 1;
    }
  }
  // ... (logic to execute timed triggers and handle manual disarm) ...
}
```

Figure 8. Code snippet for automated arming sequence.

2.3.3.1 Flight Correction Logic

In a semi-autonomous mode, the correction() function implements a form of "angle-assist" stabilization (Figure 9). It calculates the error between the pilot's desired angle (derived from the joystick position) and the drone's actual angle, then uses this error in a simple proportional control loop to calculate a corrective command.

```
void correction(){
  // Calculate desired angle based on joystick input (range -45 to +45 degrees)
  desired_roll = (((ppm[1] - 1500) / 500.0)) * max_angle;
  desired_pitch = (((ppm[2] - 1500) / 500.0)) * max_angle;

  // Calculate the error between desired and actual angle
  roll_x = desired_roll - angx;
  pitch_y = desired_pitch - angy;

  // Scale the error to a percentage for the output command
  roll_x_percentage = (roll_x / 90.0) + 0.10;
  // ... (clamping logic to keep percentage between -1.0 and 1.0) ...

  // Convert percentage back to a 1000-2000 PPM-style value for SBUS output
  roll_out = (500 * roll_x_percentage) + 1500;
  pitch_out = (500 * pitch_y_percentage) + 1500;
}
```

Figure 9. Code snippet for flight correction logic.

2.4 Assumptions

The development of this system was based on the following assumptions:

- The BrainFPV Radix 2 HD is running firmware that supports MSP communication (e.g., Betaflight).
- The flight controller firmware is in Acro mode (as opposed to being in angle or horizontal mode).
- The drone airframe is inherently stable and well-tuned.
- The physical switches provide clean, debounced signals.

- The drone is deployed from a high enough altitude to complete the arming sequence before ground impact.

3. Results and Discussion

Following the initial benchtop validation, the system's performance was confirmed through an iterative, empirical testing process. This process was critically enabled by the robust physical design of the UAV chassis (as described in Section 2.1), which could withstand repeated impacts from unsuccessful tests without sustaining critical damage.

Initial values for key software parameters, such as the accelerometer and gyroscope thresholds, were hypothetically estimated. The system was then subjected to a series of low-altitude, controlled deployment tests where the drone was manually tossed to simulate a drop event. The primary method of verification was the qualitative observation of the system's behavior. Based on these observations, the software parameters were incrementally adjusted and retested until the desired reliable and stable performance was achieved.

Once tuned, the system was validated in a final series of operational tests involving deployment from a parent aircraft. After the initial tuning phase, the system demonstrated exceptional reliability. Over the course of approximately 20 consecutive deployment tests, the system performed flawlessly, achieving a 100% success rate. In every test, the drop was correctly identified and the UAV successfully armed and stabilized in mid-air, ready to accept pilot commands.

The high success rate validates the robustness of the dual-condition detection logic and the overall firmware architecture. The simplicity of the detection algorithm proved to be an asset, avoiding edge cases and potential failure points that more complex systems might encounter. Given the perfect success rate under nominal conditions, further reliability testing was deemed a lower priority. Future work could involve testing the system's performance envelope in adverse weather conditions, such as high winds, to determine at what point the physical limitations of the airframe and motors, rather than the software logic, become the limiting factor.

4. Conclusions

Based on the design and testing of the firmware, the following can be concluded:

1. The man-in-the-middle architecture using a Teensy 4.1 microcontroller is an effective method for implementing custom deployment and recovery behaviors without modifying the primary flight controller's source code.
2. The dual-condition (free-fall and tumble) drop-detection algorithm provides a robust framework for reliably identifying a deployment event and initiating the stabilization sequence.
3. The multistage, timed arming sequence successfully transitions the UAV from a passive, falling state to a stable, controllable flight state ready for mission execution.

The implications of this work are that complex autonomous capabilities can be developed as modular hardware and software add-ons for a wide range of drone platforms. This system serves as a successful proof of concept for a platform that can be air-deployed and autonomously prepare itself for its subsequent mission objective, significantly expanding the operational envelope for UAVs in challenging environments.

List of Symbols, Abbreviations, and Acronyms

ID	identification
IDE	integrated development environment
IMU	inertial measurement unit
LED	light-emitting diode
MSP	MultiWii Serial Protocol
PPM	pulse-position modulation
RC	remote control
SBUS	Serial Bus
SPDT	single pole, double throw
UAV	unmanned aerial vehicle

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB