# Command & Data Handling Software with cFS-Inspired Architecture

**Lab:** Ω-Space Group, ORION Lab, National Technical University of Athens

**Duration:** 6 months

**Expected Start:** January 2026

## Background

The Ω-Space Group of the ORION lab is developing a CubeSat FlatSat testbed for end-to-end software testing and AI algorithm validation. This thesis focuses on implementing the core Command & Data Handling (C&DH) subsystem using lightweight custom flight software inspired by NASA's proven core Flight System[1] (cFS) architectural patterns.

The C&DH serves as the central flight computer managing satellite operations, subsystem coordination, and telemetry/telecommand handling. This work adopts a hybrid approach similar to JAXA's RACS[1] initiative: implementing C&DH with custom software following cFS organizational patterns while enabling seamless integration with subsystems including an AI payload that uses SpaceROS[2].

## Objectives

Develop a functional C&DH software system that:

1. Implements core satellite control functionality inspired by cFS patterns (mode management, telecommand handling, telemetry generation, housekeeping)
2. Communicates with EPS subsystem via CubeSat Space Protocol (CSP) over CAN bus
3. Communicates with AI payload subsystem via CSP for commands and optional interface for high-bandwidth data
4. Provides time synchronization across subsystems
5. Demonstrates end-to-end command execution and telemetry collection
6. Follows NASA cFS architectural patterns (modular apps, software bus concept, table-driven configuration)
7. Enables straightforward migration from Raspberry Pi 4 (Ubuntu) to STM32 (FreeRTOS)

[1] https://etd.gsfc.nasa.gov/capabilities/core-flight-system/

## Scope of Work

## Core C&DH Implementation

### Development Environment Setup

- Configure Raspberry Pi 4 with Ubuntu 22.04 Server (standard Linux)
- Install Python 3.10+ and development tools
- Setup CAN interface (python-can library + SocketCAN)
- Install and configure libcsp (CubeSat Space Protocol library)
- Configure Git for version control and contribution to project repository

### Investigate cFS Architecture and open-source SW implementations

- Install and run OpenSatKit to understand cFS patterns
- Study cFS application organization (modular apps, software bus, events)
- Understand table-driven configuration approach
- Review cFS command and telemetry paradigms
- Document key patterns to adopt in custom C&DH implementation
- Explore existing open-source implementations for reusability (e.g. SpaceDot, Libre Space Foundation)

### C&DH Software Architecture (cFS-Inspired, Custom Implementation)

Implement the following modules as Python classes

#### Mode Manager:

- Implements satellite operational state machine
    - Safe Mode (minimal operations, fault recovery)
    - Nominal Mode (standard housekeeping, waiting for commands)
    - Payload Mode (AI processing active)
- Mode transitions based on commands or system conditions
- Inspired by cFS Executive Services mode management

#### Telecommand Handler:

- Receives CSP packets over CAN from ground/test interface
- Command validation and parsing
- Command execution (e.g., START_PAYLOAD, STOP_PAYLOAD, REQUEST_TELEMETRY, CHANGE_MODE)

- Command acknowledgment via telemetry
- Following cFS Command Ingest patterns

**Telemetry Manager:**

- Collects telemetry from all C&DH modules
- Packages telemetry in defined format (JSON or binary)
- Sends telemetry via CSP over CAN
- Logs telemetry locally for analysis
- Following cFS Telemetry Output patterns

**Housekeeping Module:**

- Periodic collection (every 5-10 seconds) of system health data
- CPU temperature, load, memory usage (e.g. via psutil)
- Disk space, uptime, CAN bus status
- Subsystem status (EPS responding, payload alive)
- Publishes as telemetry packets
- Following cFS Health & Safety app patterns

**Time Synchronization Module:**

- C&DH as time master (testbed assumes offline operation)
- Periodic broadcast of current time via CSP
- Time synchronization message format
- Following cFS Time Services patterns

**Event Messaging System:**

- Unified logging across all C&DH modules
- Event severity levels (INFO, WARNING, ERROR, CRITICAL)
- Event IDs and descriptions
- Events sent as telemetry and logged locally
- Following cFS Event Services patterns

**Configuration System (cFS Table Pattern)**

Implement table-driven configuration using YAML files that can be modified without code changes:

- System parameters (telemetry rates, command timeouts, mode transition rules)
- Subsystem configuration (CAN addresses, CSP ports, payload interface settings)
- Threshold values (temperature limits, voltage ranges)
- Hot-reloadable configuration (can update parameters without restart)
- Following cFS table management patterns

## Communication Interfaces

### CAN Bus / CSP Interface (Primary):

- CSP over CAN for communication with EPS
- CSP packets for commands to/from payload
- Reliable packet delivery, routing, addressing

### Payload Interface:

**CSP over CAN for command/control messages**

- Send telecommands to payload (START_APP, STOP_APP, REQUEST_STATUS)
- Receive acknowledgments and status updates
- Receive AI results as telemetry (summary data, small payloads)

**Gigabit Ethernet interface for high-bandwidth data**

- Implemented for high-bandwidth data transfer from the payload
- Simple TCP/UDP socket, lightweight file transfer, or DDS subscription[2]
  **Note**: Payload uses ROS2 DDS internally, but C&DH doesn't need ROS2 unless subscribing to payload topics for large data

## Hardware Abstraction Layer (TBD)

Create platform abstraction layer to enable easy migration to STM32 (e.g. cFS OSAL)

## Integration & Testing

- Unit tests for each module (pytest)
- Integration tests with EPS subsystem via CAN
- Integration tests with payload subsystem
- End-to-end scenarios: command injection → execution → telemetry collection
- Performance analysis
- Interface testing

## Technical Requirements

- **Operating System:** Ubuntu 22.04 on Raspberry Pi 4
- **Programming Languages:** Python 3.10+ (primary), C/C++ optional for performance-critical sections (investigate Cython implementation)
- **Communication Protocols:** CubeSat Space Protocol (CSP) over CAN bus
- **Libraries:** python-can, libcsp, psutil, PyYAML (**TBD**)
- **Version Control:** All code contributed to project GitHub repository

[2] https://github.com/jaxa/racs2_extended-dds

- **Testing:** pytest for unit tests, integration test scripts, CAN traffic analysis tools (**TBD**)
- **Documentation:** Code comments, README, architecture diagrams, cFS pattern mapping

## Deliverables

1. Working C&DH software running on Raspberry Pi 4
2. Complete module implementations (Mode Manager, Telecommand Handler, Telemetry Manager, Housekeeping, Time Services, Event System)
3. Integration with AI payload subsystem (CSP/CAN)
4. Configuration system (YAML files following cFS table pattern)
5. Hardware Abstraction Layer enabling future STM32 migration
6. Test results and performance analysis
7. Documentation: cFS patterns used, architecture diagrams, API reference
8. Source code in GitHub repository

## Technical Skills

- Python, some C/C++ or Cython
- Embedded systems concepts and Linux
- NASA cFS patterns (OpenSatKit)
- CAN bus and communication protocols
- Satellite architecture design and satellite operations

## Management

- Weekly coordination meetings ensure project tracking and knowledge sharing.
- All work is documented and shared via GitHub, following open-source guidelines and contributing to the open-source community.
- Master thesis can overlap with other theses and/or ongoing developments in the working group. Collaboration is encouraged but all topics have been designed to not introduce blocking points.

## Contact

- Simon Vellas: svellas@mail.ntua.gr
- Alexis Apostolakis: alexis.apostolakis@gmail.com
- Giorgos Athanasiou: georgios.athanasiou.ntua@gmail.com

## Appendix

### Future work

#### STM32 Port

- Port C&DH modules to STM32 Nucleo with FreeRTOS
- Implement HW abstraction layer for STM32 (CAN drivers, FreeRTOS task management)
- Convert Python code to C/C++ (Cython)
- Use FreeRTOS tasks instead of Python threads
- Maintain compatibility with CSP/CAN protocol
- Compare performance: RPi4 vs STM32
- Document porting process, challenges and lessons learned

## References

[1] H. Kato, D. Hirano, S. Mitani, T. Saito and S. Kawaguchi, "ROS and cFS System (RACS): Easing Space Robotic Development," 2021 IEEE Aerospace Conference (50100), Big Sky, MT, USA, 2021, pp. 1-8, doi: 10.1109/AERO50100.2021.9438288.

[2] Probe A B, Chambers S W, Oyake A, et al. *Space ROS: An Open-Source Framework for Space Robotics and Flight Software.* In: Proceedings of the 2023 AIAA Forum; AIAA 2023-2709.