Nemesis

Generated by Doxygen 1.15.0

Chapter 1

Nemesis Flight Computer

Advanced flight computer software for high-power rocketry, featuring real-time telemetry, sensor fusion, and autonomous flight state management.

1.1 Table of Contents

1.2 Overview

Nemesis is a sophisticated flight computer designed for the **Aurora Rocket Team** competition rockets. Built on the ESP32 platform with PlatformIO, it provides:

- **Real-time sensor data acquisition** from IMUs, barometers, and GPS
- **Finite State Machine (FSM)** for autonomous flight phase management
- **Binary telemetry protocol** over LoRa for efficient ground station communication
- **Kalman filtering** for accurate altitude and velocity estimation
- **SD card logging** for post-flight analysis
- **FreeRTOS-based** concurrent task management

This system has been developed to meet the demands of high-altitude flights with real-time decision-making capabilities and robust data logging.

1.3 Features

1.3.1 Sensor Suite

- **Inertial Measurement**: BNO055 9-DOF IMU with sensor fusion, LIS3DHTR accelerometer
- **Barometric Pressure**: MS5611 high-precision barometers
- **Navigation**: u-blox GPS module with UBX protocol support

1.3.2 Flight Management

- **7-phase FSM**: Idle → Armed → Powered Flight → Coasting → Apogee → Descent → Landed
- **State-specific tasks**: Concurrent execution of sensor sampling, data logging, and telemetry
- **Transition detection**: Accelerometer-based launch detection, barometric apogee detection
- **Safety features**: Automated arming sequences, failsafe mechanisms

1.3.3 Telemetry & Communication

- **Binary protocol**: Custom telemetry protocol for packet fragmentation
- **LoRa radio**: Long-range communication (868 MHz)
- **Real-time metrics**: Altitude, velocity, acceleration, orientation, GPS position, battery status and logging
- **Ground station**: Dedicated receiver with OLED display and serial output

---

1.4 Hardware

1.4.1 Primary Flight Computer

- **MCU**: ESP32-based board (Arduino Nano ESP32 or similar)
- **Storage**: SD card module (SPI interface)
- **Radio**: SX1262 LoRa transceiver (RadioLib compatible)

1.4.2 Supported Sensors

Sensor	Interface	Purpose
BNO055	I2C	9-DOF IMU with built-in fusion
LIS3DHTR	I2C	High-G accelerometer
MS5611	I2C	Precision barometers
u-blox GPS	I2C/UART	Global positioning

1.4.3 Ground Station Hardware

- **Heltec WiFi LoRa 32 V3**: ESP32-S3 + SX1262 + OLED display
- **Power**: USB or battery (supports remote deployment)

1.4.4 Module Organization

```
lib/
+-- control/          # FSM, state machine, flight logic
|   +-- RocketFSM     # Main FSM implementation
|   +-- states/       # State actions and transitions
|   +-- tasks/        # FreeRTOS tasks per flight phase
+-- BNO055/           # IMU driver
+-- MS5611/           # Barometer driver
+-- GPS/              # GNSS driver
+-- ...
+-- telemetry/        # Binary protocol, packet management
+-- LoRa/             # Radio transmitter/receiver
+-- kalman/           # Kalman filter implementations
+-- logger/           # SD card logging utilities
+-- data/             # Data structures (TelemetryPacket, etc.)
```

1.4.5 Key Design Patterns

- **Dependency Injection**: Sensors passed as `shared_ptr` to FSM for testability
- **RAII**: Automatic resource management for SD files, I2C devices
- **Thread-safe logging**: Mutex-protected serial output via `Logger` namespace

---

1.5 Getting Started

1.5.1 Prerequisites

1. **Install PlatformIO**:

   ```
   # Via pip
   pip install platformio

   # Or install VSCode + PlatformIO IDE extension
   ```

2. **Clone the repository**:

   ```
   git clone https://github.com/AuroraRocketryTeam/Aurora_Rocketry_SW_24_25.git
   cd Aurora_Rocketry_SW_24_25
   ```

3. **Install dependencies**:

   ```
   pio pkg install
   ```

1.5.2 Configuration

Key settings are in `lib/global/src/`:

- **`config.h`**: Flight parameters, sensor calibration, thresholds
- **`pins.h`**: GPIO pin assignments for your hardware

Edit these files to match your specific hardware configuration.

1.6 Building & Flashing

1.6.1 Default Environment (Arduino Nano ESP32)

```
# Build the project
pio run

# Upload to board
pio run --target upload

# Open serial monitor (115200 baud)
pio device monitor
```

1.6.2 Custom Environment

Modify `platformio.ini` to add your board. Example for a custom ESP32 target:

```
[env:my_custom_board]
platform = espressif32
board = esp32dev
framework = arduino
monitor_speed = 115200
build_flags = ${env.build_flags}
lib_deps = ${env.lib_deps}
```

Then build with: `pio run -e my_custom_board`

1.6.3 Ground Station

The telemetry receiver is a separate Arduino sketch:

```
# Navigate to receiver folder
cd lib/LoRa/src/

# Open telemetry_lora_receiver.ino in Arduino IDE or:
pio ci --board=heltec_wifi_lora_32_V3 telemetry_lora_receiver.ino
```

1.7 Telemetry System

1.7.1 Binary Protocol

Telemetry uses a fixed-size packet structure for reliable LoRa transmission:

```
struct TelemetryPacket {
    uint32_t timestamp;          // Milliseconds since boot
    RocketState state;           // Current flight state
    float altitude;              // Meters ASL
    float vertical_velocity;     // m/s

---

## Building & Flashing

### Default Environment (Arduino Nano ESP32)
```

bash

1.8 Build the project

pio run

1.9 Upload to board

pio run –target upload

1.10 Open serial monitor (115200 baud)

pio device monitor

```
### Custom Environment

Modify `platformio.ini` to add your board. Example for a custom ESP32 target:
```

ini [env:my_custom_board] platform = espressif32 board = esp32dev framework = arduino monitor_speed = 115200 build_flags = ${env.build_flags} lib_deps = ${env.lib_deps}

```
Then build with: `pio run -e my_custom_board`

### Ground Station

The telemetry receiver is a separate Arduino sketch:
```

bash

1.11 Navigate to receiver folder

cd lib/LoRa/src/

1.12 Open telemetry_lora_receiver.ino in Arduino IDE or:

pio ci –board=heltec_wifi_lora_32_V3 telemetry_lora_receiver.ino

```
---

## Telemetry System

### Binary Protocol

Telemetry uses a fixed-size packet structure for reliable LoRa transmission:
```

cpp struct TelemetryPacket { uint32_t timestamp; // Milliseconds since boot RocketState state; // Current flight state float altitude; // Meters ASL float vertical_velocity; // m/s float acceleration[3]; // X, Y, Z in m/s^2 float angular_↩ velocity[3]; // Roll, pitch, yaw in deg/s float gps_lat, gps_lon; // Degrees uint8_t gps_fix; // Fix quality float battery↩ _voltage; // Volts // ... + CRC checksum };

```
### Packet Management
- **Fragmentation**: Large packets split into LoRa-compatible chunks
- **Reassembly**: Sequence numbers ensure correct reconstruction
- **CRC validation**: 16-bit CRC for error detection
- **Metrics tracking**: RSSI, SNR, packet loss, throughput

### Transmission Parameters
- **Frequency**: 868 MHz (Europe) / 915 MHz (US)
- **Spreading Factor**: SF7 (fast, shorter range) to SF12 (slow, max range)
- **Bandwidth**: 125 kHz
- **Coding Rate**: 4/7

Airtime for ~64-byte payload at SF7: **~50-80 ms**

---

## Flight States

| State | Entry Condition | Active Tasks | Exit Condition |
|-------|----------------|--------------|----------------|
| **IDLE** | Power-on | Status LED | Arming sequence |
| **ARMED** | User input | All sensors active | Accel > launch threshold |
| **POWERED_FLIGHT** | Launch detected | High-rate logging | Motor burnout (accel < threshold) |
| **COASTING** | Burnout | Altitude tracking | Apogee (velocity < 0) |
| **APOGEE** | Velocity negative | Deploy drogue chute | Altitude dropping |
| **DESCENT** | Post-apogee | GPS tracking | Altitude < 300m (deploy main) |
| **LANDED** | Near-zero velocity | Beeper, data flush | Manual reset |

Transition logic is in `lib/control/src/states/TransitionManager.cpp`.

---

## Development

### Code Style
- **C++17** standard (enforced by `-std=gnu++17`)
- **Doxygen comments** for all public APIs
- **Include guards** and `#pragma once` for headers

### Testing

Unit tests are in `test/`:
```

bash

1.13 Run all tests

pio test

1.14 Run specific test

pio test -f test_kalman_filter

```
Manual hardware tests are in `test/test_manual/`.

### Debugging

Enable verbose logging in `config.h`:
```

cpp #define LOG_LEVEL LOG_LEVEL_DEBUG

```
View logs via serial monitor:
```

bash pio device monitor –baud 115200

```
### Adding a New Sensor

1. Create driver in `lib/YourSensor/src/`
2. Implement `ISensor` interface (if applicable)
3. Add to `RocketFSM` initialization
4. Include in relevant state tasks
5. Update `TelemetryPacket` if needed

---

## Documentation

### Generate API Docs

The project includes a Doxygen configuration:
```
bash

1.15 Install Doxygen

sudo apt-get install doxygen

1.16 Generate HTML documentation

doxygen Doxyfile

1.17 Open in browser

xdg-open docs/html/index.html ```

1.17.0.1 Additional Resources

- **Flight State Patterns**: See `lib/control/docs/STATUS_PATTERNS.md`
- **Telemetry Migration Guide**: See `lib/telemetry/docs/TELEMETRY_MIGRATION.md`
- **Binary Protocol Spec**: See `lib/telemetry/docs/BINARY_TELEMETRY.md`

1.17.1 Contributing

We welcome contributions from the Aurora Rocket Team and the wider rocketry community!

1.17.1.1 Workflow

1. **Fork** the repository
2. **Create a feature branch**: `git checkout -b feature/amazing-feature`
3. **Commit changes**: `git commit -m 'Add amazing feature'`
4. **Push to branch**: `git push origin feature/amazing-feature`
5. **Open a Pull Request**

1.17.1.2 Guidelines

- Follow existing code style and naming conventions
- Add Doxygen comments for new public APIs
- Include unit tests for new algorithms
- Update documentation for user-facing changes
- Test on hardware before submitting (if possible)

1.17.1.3 Issue Reporting

Found a bug? Have a feature request? Open an issue with:

- Clear description of the problem/feature
- Steps to reproduce (for bugs)
- Expected vs. actual behavior
- Hardware/software versions

1.17.2 License

This project is licensed under the **MIT License** - see the LICENSE file for details.

1.17.2.1 Third-Party Libraries

- **RadioLib**: LGPL-3.0 (LoRa communication)
- **BME680 Library**: BSD (Bosch Sensortec)
- **SparkFun u-blox Library**: MIT
- **Eigen**: MPL2 (linear algebra)
- **TinyEKF**: LGPL (Kalman filtering)

1.17.3 Team

Aurora Rocket Team - Università di Bologna
For questions or collaboration opportunities, reach out via GitHub issues or the team's official channels.

1.17.4 Acknowledgments

- **Bosch Sensortec** for excellent sensor documentation
- **RadioLib community** for LoRa protocol support
- **PlatformIO team** for the best embedded development platform
- **FreeRTOS** for reliable real-time task scheduling

Back to Top

Chapter 2

Deprecated List

Struct StateTransition

Use TransitionManager and Transition struct instead

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

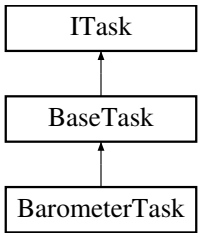
Here is a list of all documented files with brief descriptions:

Chapter 6

Class Documentation

6.1 BarometerTask Class Reference

Inheritance diagram for BarometerTask:



Public Member Functions

- **BarometerTask** (std::shared_ptr< SharedSensorData > sensorData, SemaphoreHandle_t sensorData↩ Mutex, std::shared_ptr< bool > isRising, std::shared_ptr< float > heightGainSpeed, std::shared_ptr< float > currentHeight)
- void taskFunction () override

Public Member Functions inherited from BaseTask

- **BaseTask** (const char *name)
- bool start (const TaskConfig &config) override

  Start the task with the given configuration.
- void stop () override

  Stop the task.
- bool isRunning () const override

  Check if the task is currently running.
- const char * getName () const override

  Get the name of the task.
- uint32_t getStackHighWaterMark () const override

  Get the stack high water mark for the task.

Additional Inherited Members

Protected Member Functions inherited from BaseTask

- virtual void **onTaskStart** ()
- virtual void **onTaskStop** ()

Protected Attributes inherited from BaseTask

- TaskHandle_t **taskHandle**
- TaskConfig **config**
- volatile bool **running**
- const char * **taskName**

6.1.1 Member Function Documentation

6.1.1.1 taskFunction()

```
void BarometerTask::taskFunction ()  [override], [virtual]
```

Implements BaseTask.

The documentation for this class was generated from the following files:

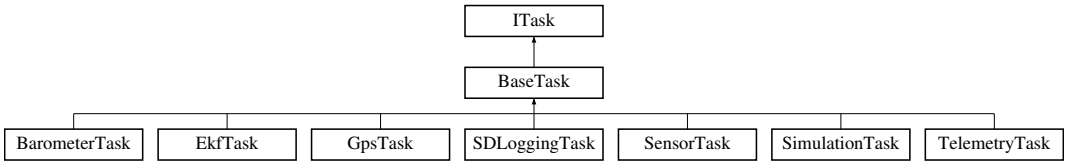
- lib/control/src/tasks/BarometerTask.hpp
- lib/control/src/tasks/BarometerTask.cpp

6.2 BaseTask Class Reference

Base class for all tasks in the system, providing common task management functionality.

`#include <BaseTask.hpp>`

Inheritance diagram for BaseTask:



Public Member Functions

- **BaseTask** (const char *name)
- bool start (const TaskConfig &config) override

  Start the task with the given configuration.
- void stop () override

  Stop the task.
- bool isRunning () const override

  Check if the task is currently running.
- const char * getName () const override

  Get the name of the task.
- uint32_t getStackHighWaterMark () const override

  Get the stack high water mark for the task.

Protected Member Functions

- virtual void taskFunction ()=0
- virtual void **onTaskStart** ()
- virtual void **onTaskStop** ()

Protected Attributes

- TaskHandle_t **taskHandle**
- TaskConfig **config**
- volatile bool **running**
- const char * **taskName**

6.2.1 Detailed Description

Base class for all tasks in the system, providing common task management functionality.
This abstract class implements the ITask interface and provides a foundation for creating FreeRTOS-based tasks with standardized lifecycle management, configuration, and monitoring. Derived classes must implement the pure virtual taskFunction() method to define their specific task behavior.
The class handles task creation, starting, stopping, and provides utilities for monitoring task status and stack usage. It uses FreeRTOS primitives for task management and ensures proper cleanup when tasks are destroyed.

Note

This class is abstract and cannot be instantiated directly.

Tasks are managed using FreeRTOS TaskHandle_t and follow FreeRTOS conventions.

6.2.2 Member Function Documentation

6.2.2.1 getName()

```
const char * BaseTask::getName () const  [inline], [override], [virtual]
```

Get the name of the task.

Returns

const char* Name of the task

Implements ITask.

6.2.2.2 getStackHighWaterMark()

```
uint32_t BaseTask::getStackHighWaterMark () const  [override], [virtual]
```

Get the stack high water mark for the task.

Returns

uint32_t High water mark in bytes

Implements ITask.

6.2.2.3 isRunning()

```
bool BaseTask::isRunning () const  [inline], [override], [virtual]
```

Check if the task is currently running.

Returns

true if running

false otherwise

Implements ITask.

6.2.2.4 start()

```
bool BaseTask::start (
            const TaskConfig & config)  [override], [virtual]
```

Start the task with the given configuration.

Parameters

config	TaskConfig: Configuration for the task

Returns

true if the task started successfully

false otherwise

Implements ITask.

6.2.2.5 stop()

```
void BaseTask::stop ()  [override], [virtual]
```

Stop the task.
Implements ITask.

6.2.2.6 taskFunction()

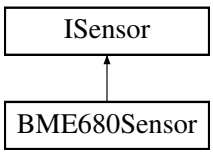
```
virtual void BaseTask::taskFunction ()  [protected], [pure virtual]
```

Implements ITask.
The documentation for this class was generated from the following files:

- lib/control/src/tasks/BaseTask.hpp
- lib/control/src/tasks/BaseTask.cpp

6.3 BME680Sensor Class Reference

A sensor class for the BME680 sensor, inheriting from the Sensor class template.
`#include <BME680Sensor.hpp>`
Inheritance diagram for BME680Sensor:



Public Member Functions

- **BME680Sensor** (uint8_t addr)
- bool init () override

  Initialize the sensor.
- std::optional< SensorData > getData () override

  Read and then get the sensor data.

Public Member Functions inherited from ISensor

- bool isInitialized () const

  Check if the sensor is initialized.

Additional Inherited Members

Protected Member Functions inherited from ISensor

- void **setInitialized** (bool initialized)

6.3.1 Detailed Description

A sensor class for the BME680 sensor, inheriting from the Sensor class template.
This class provides methods to initialize the sensor, read data synchronously and asynchronously, and retrieve the sensor data.

Note

The BME680 sensor is used for measuring temperature, humidity, pressure, and gas.

6.3.2 Member Function Documentation

6.3.2.1 getData()

```
std::optional< SensorData > BME680Sensor::getData ()  [override], [virtual]
```

Read and then get the sensor data.

Returns

Just read data.

Implements ISensor.

6.3.2.2 init()

```
bool BME680Sensor::init ()  [override], [virtual]
```

Initialize the sensor.

Returns

true if the sensor was initialized successfully

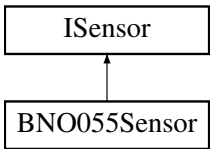
false if the sensor failed to initialize

Implements ISensor.
The documentation for this class was generated from the following files:

- lib/BME680/src/BME680Sensor.hpp
- lib/BME680/src/BME680Sensor.cpp

6.4 BNO055Sensor Class Reference

High-level driver for the Bosch BNO055 IMU using BNO055SensorInterface.
`#include <BNO055Sensor.hpp>`
Inheritance diagram for BNO055Sensor:



Classes

- struct CalibrationStatus

  Per-subsystem calibration completeness indicators. Fields typically range 0..3 where 3 indicates fully calibrated.

Public Member Functions

- **BNO055Sensor** ()

  Construct a new BNO055 sensor wrapper.
- bool init () override

  Initialize the BNO055 device and prepare it for measurements.
- bool calibrate ()

  Run the device calibration routine (blocking until complete if implemented).
- bool hardwareTest ()

  Execute a hardware self-test if supported by the interface.
- std::optional< SensorData > getData () override

  Read the latest available IMU data and package it as SensorData.
- CalibrationStatus getCalibration ()

  Get the current calibration status across subsystems.

Public Member Functions inherited from ISensor

- bool isInitialized () const

  Check if the sensor is initialized.

Additional Inherited Members

Protected Member Functions inherited from ISensor

- void **setInitialized** (bool initialized)

6.4.1 Detailed Description

High-level driver for the Bosch BNO055 IMU using BNO055SensorInterface.
Provides initialization, optional calibration helpers, hardware self-test and a consolidated data readout via ISensor::getData(). Typical SensorData keys include accelerometer, angular velocity (gyroscope), orientation and board temperature depending on interface configuration.

6.4.2 Member Function Documentation

6.4.2.1 calibrate()

```
bool BNO055Sensor::calibrate ()
```

Run the device calibration routine (blocking until complete if implemented).

Returns

true if calibration completed successfully, false otherwise.

6.4.2.2 getCalibration()

```
BNO055Sensor::CalibrationStatus BNO055Sensor::getCalibration ()
```

Get the current calibration status across subsystems.

Returns

CalibrationStatus with subsystem indicators.

6.4.2.3 getData()

```
std::optional< SensorData > BNO055Sensor::getData ()  [override], [virtual]
```

Read the latest available IMU data and package it as SensorData.

Returns

SensorData populated with the latest readings; std::nullopt on failure.

Implements ISensor.

6.4.2.4 hardwareTest()

```
bool BNO055Sensor::hardwareTest ()
```

Execute a hardware self-test if supported by the interface.

Returns

true if the test passes, false otherwise.

6.4.2.5 init()

```
bool BNO055Sensor::init ()  [override], [virtual]
```

Initialize the BNO055 device and prepare it for measurements.

Returns

true on success, false otherwise.

Implements ISensor.
The documentation for this class was generated from the following files:

- lib/BNO055/src/BNO055Sensor.hpp
- lib/BNO055/src/BNO055Sensor.cpp

6.5 BNO055SensorInterface Class Reference

Interface for BNO055 sensor operations, using low-level APIs from the BNO055_SensorAPI official library, instead of Adafruit_BNO055.

```
#include <BNO055SensorInterface.hpp>
```

Public Member Functions

- **BNO055SensorInterface** ()

  Default constructor for BNO055SensorInterface, the actual sensor structure will be initialized in init().
- bool init ()

  Initialize the BNO055 sensor.
- uint8_t check_calibration ()

  Check if all sensor components are calibrated.
- uint8_t check_calibration_accel ()

  Check accelerometer calibration status.
- uint8_t check_calibration_mag ()

  Check magnetometer calibration status.
- uint8_t check_calibration_gyro ()

  Check gyroscope calibration status.
- uint8_t check_calibration_sys ()

  Check system-wide calibration status.
- bool selftest_accel ()

  Perform accelerometer hardware self-test.
- bool selftest_mag ()

  Perform magnetometer hardware self-test.
- bool selftest_gyro ()

  Perform gyroscope hardware self-test.
- bool selftest_mcu ()

  Perform microcontroller unit hardware self-test.
- bool check_system_status ()

  Check if the system is running properly.

- bool check_system_error ()

  Check if there are any system errors.
- bool check_clock_status ()

  Check if the main clock is running properly.
- uint8_t get_system_error_code ()

  Get the specific system error code for diagnosis.
- uint8_t get_system_status_code ()

  Get the specific system status code for diagnosis.
- bool set_operation_mode (uint8_t mode)

  Set the sensor operation mode.
- bool get_operation_mode (uint8_t ∗mode)

  Get the current sensor operation mode.
- bool set_power_mode (uint8_t mode)

  Set the sensor power mode.
- bool get_power_mode (uint8_t ∗mode)

  Get the current sensor power mode.
- bool set_accel_power_mode (uint8_t mode)

  Set the accelerometer power mode.
- bool set_mag_power_mode (uint8_t mode)

  Set the magnetometer power mode.
- bool set_gyro_power_mode (uint8_t mode)

  Set the gyroscope power mode.
- std::vector< float > get_accel ()

  Get the current accelerometer data.
- std::vector< float > get_mag ()

  Get the current magnetometer data.
- std::vector< float > get_gyro_dps ()

  Get the current gyroscope data in degrees per second.
- std::vector< float > get_gyro_rps ()

  Get the current gyroscope data in radians per second.
- std::vector< float > get_euler_deg ()

  Get the current Euler angles in degrees.
- std::vector< float > get_euler_rad ()

  Get the current Euler angles in radians.
- std::vector< float > get_quaternion ()

  Get the current quaternion data.
- std::vector< float > get_linear_accel ()

  Get the current linear acceleration data.
- std::vector< float > get_gravity ()

  Get the current gravity vector data.
- float get_temperature ()

  Get the current temperature data.

6.5.1 Detailed Description

Interface for BNO055 sensor operations, using low-level APIs from the BNO055_SensorAPI official library, instead of Adafruit_BNO055.

6.5.2 Member Function Documentation

6.5.2.1 check_calibration()

`uint8_t BNO055SensorInterface::check_calibration ()`
Check if all sensor components are calibrated.

Returns

the minimum calibration value among all sensors (accel, mag, gyro, system), from 0 (not calibrated) to 3 (fully calibrated)

6.5.2.2 check_calibration_accel()

`uint8_t BNO055SensorInterface::check_calibration_accel ()`
Check accelerometer calibration status.

Returns

the calibration value of the accelerometer, from 0 (not calibrated) to 3 (fully calibrated)

6.5.2.3 check_calibration_gyro()

`uint8_t BNO055SensorInterface::check_calibration_gyro ()`
Check gyroscope calibration status.

Returns

the calibration value of the gyroscope, from 0 (not calibrated) to 3 (fully calibrated)

Note

Gyroscope calibration requires the sensor to remain stationary

6.5.2.4 check_calibration_mag()

`uint8_t BNO055SensorInterface::check_calibration_mag ()`
Check magnetometer calibration status.

Returns

the calibration value of the magnetometer, from 0 (not calibrated) to 3 (fully calibrated)

Note

Magnetometer calibration typically requires figure-8 movements !!! Look for how to start the actual calibration

6.5.2.5 check_calibration_sys()

`uint8_t BNO055SensorInterface::check_calibration_sys ()`
Check system-wide calibration status.

Returns

the calibration value of the system, from 0 (not calibrated) to 3 (fully calibrated)

6.5.2.6 check_clock_status()

`bool BNO055SensorInterface::check_clock_status ()`
Check if the main clock is running properly.

Returns

true if main clock is operational (status = 1), false otherwise

6.5.2.7 check_system_error()

`bool BNO055SensorInterface::check_system_error ()`
Check if there are any system errors.

Returns

true if no system errors detected (error code = 0), false if errors present

6.5.2.8 check_system_status()

`bool BNO055SensorInterface::check_system_status ()`
Check if the system is running properly.

Returns

true if system is in operational state (status 5 or 6), false otherwise

System status codes:

- 0: System idle, no operation being performed
- 1: System error
- 2: Initializing peripherals
- 3: System initialization
- 4: Executing self-test
- 5: Sensor fusion algorithm running (GOOD)
- 6: System running without fusion (GOOD)

6.5.2.9 get_accel()

`std::vector< float > BNO055SensorInterface::get_accel ()`
Get the current accelerometer data.

Returns

Accelerometer data structure containing x, y, z acceleration values

6.5.2.10 get_euler_deg()

`std::vector< float > BNO055SensorInterface::get_euler_deg ()`
Get the current Euler angles in degrees.

Returns

Euler angles structure containing heading, roll, pitch in degrees

6.5.2.11 get_euler_rad()

`std::vector< float > BNO055SensorInterface::get_euler_rad ()`
Get the current Euler angles in radians.

Returns

Euler angles structure containing heading, roll, pitch in radians

6.5.2.12 get_gravity()

```
std::vector< float > BNO055SensorInterface::get_gravity ()
```

Get the current gravity vector data.

Returns

Gravity vector data structure containing x, y, z acceleration values

6.5.2.13 get_gyro_dps()

```
std::vector< float > BNO055SensorInterface::get_gyro_dps ()
```

Get the current gyroscope data in degrees per second.

Returns

Gyroscope data structure containing x, y, z angular velocity values in degrees per second

6.5.2.14 get_gyro_rps()

```
std::vector< float > BNO055SensorInterface::get_gyro_rps ()
```

Get the current gyroscope data in radians per second.

Returns

Gyroscope data structure containing x, y, z angular velocity values in radians per second

6.5.2.15 get_linear_accel()

```
std::vector< float > BNO055SensorInterface::get_linear_accel ()
```

Get the current linear acceleration data.

Returns

Linear acceleration data structure containing x, y, z acceleration values

6.5.2.16 get_mag()

```
std::vector< float > BNO055SensorInterface::get_mag ()
```

Get the current magnetometer data.

Returns

Magnetometer data structure containing x, y, z magnetic field values

6.5.2.17 get_operation_mode()

```
bool BNO055SensorInterface::get_operation_mode (
            uint8_t * mode)
```

Get the current sensor operation mode.

Parameters

mode	Pointer to store the current operation mode

Returns

true if mode retrieved successfully, false otherwise

6.5.2.18 get_power_mode()

```
bool BNO055SensorInterface::get_power_mode (
            uint8_t * mode)
```

Get the current sensor power mode.

Parameters

mode	Pointer to store the current power mode

Returns

true if power mode retrieved successfully, false otherwise

6.5.2.19 get_quaternion()

```
std::vector< float > BNO055SensorInterface::get_quaternion ()
```

Get the current quaternion data.

Returns

Quaternion data structure containing x, y, z, w values

6.5.2.20 get_system_error_code()

```
uint8_t BNO055SensorInterface::get_system_error_code ()
```

Get the specific system error code for diagnosis.

Returns

System error code (0 = no error, >0 = specific error condition)

6.5.2.21 get_system_status_code()

```
uint8_t BNO055SensorInterface::get_system_status_code ()
```

Get the specific system status code for diagnosis.

Returns

System status code (see check_system_status() for code meanings)

Get the specific system status code for diagnosis

Returns

System status code (see check_system_status() for code meanings)

6.5.2.22 get_temperature()

```
float BNO055SensorInterface::get_temperature ()
```

Get the current temperature data.

Returns

Temperature in degrees Celsius

6.5.2.23 init()

```
bool BNO055SensorInterface::init ()
```

Initialize the BNO055 sensor.

Returns

true if initialization successful, false otherwise

6.5.2.24 selftest_accel()

```
bool BNO055SensorInterface::selftest_accel ()
```

Perform accelerometer hardware self-test.

Returns

true if accelerometer hardware test passes (result = 0x01), false if fails (result = 0x00)

6.5.2.25 selftest_gyro()

```
bool BNO055SensorInterface::selftest_gyro ()
```

Perform gyroscope hardware self-test.

Returns

true if gyroscope hardware test passes (result = 0x01), false if fails (result = 0x00)

6.5.2.26 selftest_mag()

```
bool BNO055SensorInterface::selftest_mag ()
```

Perform magnetometer hardware self-test.

Returns

true if magnetometer hardware test passes (result = 0x01), false if fails (result = 0x00)

6.5.2.27 selftest_mcu()

```
bool BNO055SensorInterface::selftest_mcu ()
```

Perform microcontroller unit hardware self-test.

Returns

true if MCU hardware test passes (result = 0x01), false if fails (result = 0x00)

Perform microcontroller unit hardware self-test

Returns

true if MCU hardware test passes (result = 0x01), false if fails (result = 0x00)

6.5.2.28 set_accel_power_mode()

```
bool BNO055SensorInterface::set_accel_power_mode (
            uint8_t mode)
```

Set the accelerometer power mode.

Parameters

mode	Accelerometer power mode (0x00-0x05)

Returns

true if power mode set successfully, false otherwise

6.5.2.29 set_gyro_power_mode()

```
bool BNO055SensorInterface::set_gyro_power_mode (
            uint8_t mode)
```

Set the gyroscope power mode.

Parameters

mode	Gyroscope power mode (0x00-0x04)

Returns

true if power mode set successfully, false otherwise

Set the gyroscope power mode

Parameters

mode	Gyroscope power mode (0x00-0x04)

Returns

true if power mode set successfully, false otherwise

6.5.2.30 set_mag_power_mode()

```
bool BNO055SensorInterface::set_mag_power_mode (
            uint8_t mode)
```

Set the magnetometer power mode.

Parameters

mode	Magnetometer power mode (0x00-0x03)

Returns

true if power mode set successfully, false otherwise

6.5.2.31 set_operation_mode()

```
bool BNO055SensorInterface::set_operation_mode (
            uint8_t mode)
```

Set the sensor operation mode.

Parameters

mode	Operation mode to set

Returns

true if mode set successfully, false otherwise

Available operation modes:

- BNO055_OPERATION_MODE_CONFIG (0x00): Configuration mode
- BNO055_OPERATION_MODE_ACCONLY (0x01): Accelerometer only
- BNO055_OPERATION_MODE_MAGONLY (0x02): Magnetometer only
- BNO055_OPERATION_MODE_GYRONLY (0x03): Gyroscope only
- BNO055_OPERATION_MODE_ACCMAG (0x04): Accelerometer + Magnetometer
- BNO055_OPERATION_MODE_ACCGYRO (0x05): Accelerometer + Gyroscope
- BNO055_OPERATION_MODE_MAGGYRO (0x06): Magnetometer + Gyroscope
- BNO055_OPERATION_MODE_AMG (0x07): All sensors, no fusion
- BNO055_OPERATION_MODE_IMUPLUS (0x08): IMU fusion (no magnetometer)
- BNO055_OPERATION_MODE_COMPASS (0x09): Compass mode
- BNO055_OPERATION_MODE_M4G (0x0A): M4G fusion mode
- BNO055_OPERATION_MODE_NDOF_FMC_OFF (0x0B): 9DOF without fast mag calibration
- BNO055_OPERATION_MODE_NDOF (0x0C): 9DOF fusion (recommended for rockets)

6.5.2.32 set_power_mode()

```
bool BNO055SensorInterface::set_power_mode (
            uint8_t mode)
```

Set the sensor power mode.

Parameters

mode	Power mode to set

Returns

true if power mode set successfully, false otherwise

Available power modes:

- BNO055_POWER_MODE_NORMAL (0x00): Normal operation
- BNO055_POWER_MODE_LOWPOWER (0x01): Low power mode
- BNO055_POWER_MODE_SUSPEND (0x02): Suspend mode

The documentation for this class was generated from the following files:

- lib/BNO055/src/BNO055SensorInterface.hpp
- lib/BNO055/src/BNO055SensorInterface.cpp

6.6 BuzzerController Class Reference

Public Member Functions

- **BuzzerController** (uint8_t buzzerPin)
- void **init** ()
- void **startPattern** (BuzzerPattern pattern, BuzzerTone baseTone=TONE_MID)
- void **stopPattern** ()
- void **playTone** (BuzzerTone frequency, uint16_t duration)
- void **playSequence** (const BuzzerSequence *sequence, uint8_t count, bool repeat=false)
- void **setOff** ()
- void **playToneFreq** (uint16_t frequency)
- void **stopTone** ()

The documentation for this class was generated from the following files:

- lib/StatusManager/src/BuzzerController.hpp
- lib/StatusManager/src/BuzzerController.cpp

6.7 BuzzerSequence Struct Reference

Public Attributes

- BuzzerTone **tone**
- uint16_t **duration**
- uint16_t **pause**

The documentation for this struct was generated from the following file:

- lib/StatusManager/src/BuzzerController.hpp

6.8 BNO055Sensor::CalibrationStatus Struct Reference

Per-subsystem calibration completeness indicators. Fields typically range 0..3 where 3 indicates fully calibrated.
`#include <BNO055Sensor.hpp>`

Public Attributes

- uint8_t sys
- uint8_t gyro
- uint8_t accel
- uint8_t mag

6.8.1 Detailed Description

Per-subsystem calibration completeness indicators. Fields typically range 0..3 where 3 indicates fully calibrated.

6.8.2 Member Data Documentation

6.8.2.1 accel

`uint8_t BNO055Sensor::CalibrationStatus::accel`
Accelerometer calibration level.

6.8.2.2 gyro

`uint8_t BNO055Sensor::CalibrationStatus::gyro`
Gyroscope calibration level.

6.8.2.3 mag

`uint8_t BNO055Sensor::CalibrationStatus::mag`
Magnetometer calibration level.

6.8.2.4 sys

`uint8_t BNO055Sensor::CalibrationStatus::sys`
System calibration level.
The documentation for this struct was generated from the following file:

- lib/BNO055/src/BNO055Sensor.hpp

6.9 CSVLogger Class Reference

Public Member Functions

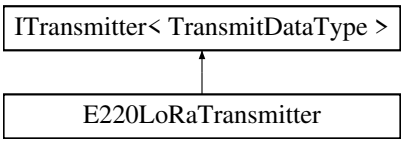
- **CSVLogger** (std::string fileName)
- bool **init** ()
- void **writeHeader** ()
- void **logSensorData** (const json &allData, unsigned long timestamp)

The documentation for this class was generated from the following file:

- lib/CSVlogger/src/CSVLogger.hpp

6.10 E220LoRaTransmitter Class Reference

Class for the Ebyte E220-900T22D LoRa module transmitter.
`#include <E220LoRaTransmitter.hpp>`
Inheritance diagram for E220LoRaTransmitter:



Public Member Functions

- E220LoRaTransmitter (HardwareSerial &serial, byte auxPin, byte m0Pin, byte m1Pin)
  Construct a new E220LoRaTransmitter object.
- E220LoRaTransmitter (HardwareSerial &serial, byte auxPin)
  Construct a new E220LoRaTransmitter object.
- E220LoRaTransmitter (HardwareSerial &serial)
  Construct a new E220LoRaTransmitter object.
- ResponseStatusContainer init () override
  Initialize the LoRa module with default configuration.
- ResponseStatusContainer init (Configuration config)
  Init the LoRa module with a given configuration.
- ResponseStatusContainer transmit (TransmitDataType data) override
  Transmit data over LoRa.
- ResponseStatusContainer configure (Configuration configuration)
  Set a new configuration for the LoRa module.
- ResponseStructContainer getConfiguration ()
  Get the configuration object.
- String getConfigurationString (Configuration configuration) const
  Get the configuration string (an utility function).

6.10.1 Detailed Description

Class for the Ebyte E220-900T22D LoRa module transmitter.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 E220LoRaTransmitter() [1/3]

```
E220LoRaTransmitter::E220LoRaTransmitter (
            HardwareSerial & serial,
            byte auxPin,
            byte m0Pin,
            byte m1Pin)  [inline]
```

Construct a new E220LoRaTransmitter object.

Parameters

serial	The serial port for the LoRa module
auxPin	AUX pin
m0Pin	M0 pin
m1Pin	M1 pin

6.10.2.2 E220LoRaTransmitter() [2/3]

```
E220LoRaTransmitter::E220LoRaTransmitter (
            HardwareSerial & serial,
            byte auxPin)  [inline]
```

Construct a new E220LoRaTransmitter object.

Parameters

serial	The serial port for the LoRa module
auxPin	AUX pin

6.10.2.3 E220LoRaTransmitter() [3/3]

```
E220LoRaTransmitter::E220LoRaTransmitter (
            HardwareSerial & serial)  [inline]
```

Construct a new E220LoRaTransmitter object.

Parameters

serial	The serial port for the LoRa module

6.10.3 Member Function Documentation

6.10.3.1 configure()

```
ResponseStatusContainer E220LoRaTransmitter::configure (
            Configuration configuration)
```

Set a new configuration for the LoRa module.

Parameters

configuration	The configuration to use

Returns

ResponseStatusContainer

6.10.3.2 getConfiguration()

```
ResponseStructContainer E220LoRaTransmitter::getConfiguration ()
```

Get the configuration object.

Returns

ResponseStructContainer

6.10.3.3 getConfigurationString()

```
String E220LoRaTransmitter::getConfigurationString (
            Configuration configuration) const
```

Get the configuration string (an utility function).

Parameters

configuration	The configuration to convert to a string

Returns

String

6.10.3.4 init() [1/2]

```
ResponseStatusContainer E220LoRaTransmitter::init ()  [override], [virtual]
```

Initialize the LoRa module with default configuration.

Returns

ResponseStatusContainer

Implements ITransmitter< TransmitDataType >.

6.10.3.5 init() [2/2]

```
ResponseStatusContainer E220LoRaTransmitter::init (
            Configuration config)
```

Init the LoRa module with a given configuration.

Parameters

config	The configuration to use (see LoRa_E220.h)

Returns

ResponseStatusContainer

6.10.3.6 transmit()

```
ResponseStatusContainer E220LoRaTransmitter::transmit (
            TransmitDataType data)  [override], [virtual]
```

Transmit data over LoRa.

Parameters

data	The data to transmit

Returns

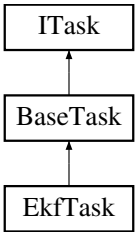
ResponseStatusContainer

Implements ITransmitter< TransmitDataType >.
The documentation for this class was generated from the following files:

- lib/LoRa/src/E220LoRaTransmitter.hpp
- lib/LoRa/src/E220LoRaTransmitter.cpp

6.11 EkfTask Class Reference

Inheritance diagram for EkfTask:



Public Member Functions

- **EkfTask** (std::shared_ptr< SharedSensorData > sensorData, SemaphoreHandle_t sensorDataMutex, std::shared_ptr< KalmanFilter1D > kalmanFilter)

Public Member Functions inherited from BaseTask

- **BaseTask** (const char *name)
- bool start (const TaskConfig &config) override

  Start the task with the given configuration.
- void stop () override

  Stop the task.
- bool isRunning () const override

  Check if the task is currently running.
- const char * getName () const override

  Get the name of the task.
- uint32_t getStackHighWaterMark () const override

  Get the stack high water mark for the task.

Protected Member Functions

- void taskFunction () override

Protected Member Functions inherited from BaseTask

- virtual void **onTaskStart** ()
- virtual void **onTaskStop** ()

Additional Inherited Members

Protected Attributes inherited from BaseTask

- TaskHandle_t **taskHandle**
- TaskConfig **config**
- volatile bool **running**
- const char * **taskName**

6.11.1 Member Function Documentation

6.11.1.1 taskFunction()

```
void EkfTask::taskFunction ()  [override], [protected], [virtual]
```

Implements BaseTask.

The documentation for this class was generated from the following files:

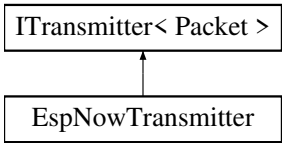
- lib/control/src/tasks/EkfTask.hpp
- lib/control/src/tasks/EkfTask.cpp

6.12 EspNowTransmitter Class Reference

ESP-NOW transmitter for sending Packet structures to a peer receiver.

```
#include <EspNowTransmitter.hpp>
```

Inheritance diagram for EspNowTransmitter:



Public Member Functions

- EspNowTransmitter (const uint8_t peerMacAddress[6], uint8_t wifiChannel=1)

  Construct a new ESP-NOW Transmitter.
- ~**EspNowTransmitter** ()

  Destroy the ESP-NOW Transmitter and cleanup resources.
- ResponseStatusContainer init () override

  Initialize the ESP-NOW transmitter.
- ResponseStatusContainer transmit (Packet packet) override

  Transmit a Packet via ESP-NOW.
- int8_t getLastRSSI () const

  Get the last recorded RSSI (signal strength).
- void getStats (uint32_t &sent, uint32_t &failed) const

  Get transmission statistics.

6.12.1 Detailed Description

ESP-NOW transmitter for sending Packet structures to a peer receiver.
This class implements the ITransmitter interface for Packet objects using ESP-NOW protocol. It handles peer registration, packet transmission, and delivery acknowledgment.
Thread-safe for concurrent transmissions.

6.12.2 Constructor & Destructor Documentation

6.12.2.1 EspNowTransmitter()

```
EspNowTransmitter::EspNowTransmitter (
            const uint8_t peerMacAddress[6],
            uint8_t wifiChannel = 1)
```

Construct a new ESP-NOW Transmitter.

Parameters

peerMacAddress	The MAC address of the peer receiver (6 bytes).
wifiChannel	The WiFi channel to use (1-13, default 1).

6.12.3 Member Function Documentation

6.12.3.1 getLastRSSI()

```
int8_t EspNowTransmitter::getLastRSSI () const  [inline]
```

Get the last recorded RSSI (signal strength).

Returns

int8_t RSSI in dBm, or 0 if no data received.

6.12.3.2 getStats()

```
void EspNowTransmitter::getStats (
            uint32_t & sent,
            uint32_t & failed) const
```

Get transmission statistics.

Parameters

sent	Output: number of successfully sent packets.
failed	Output: number of failed transmissions.

6.12.3.3 init()

```
ResponseStatusContainer EspNowTransmitter::init ()  [override], [virtual]
```

Initialize the ESP-NOW transmitter.
Sets up WiFi in STA mode, initializes ESP-NOW, and registers the peer.

Returns

ResponseStatusContainer Success (code 0) or error with description.

Implements ITransmitter< Packet >.

6.12.3.4 transmit()

```
ResponseStatusContainer EspNowTransmitter::transmit (
            Packet packet)  [override], [virtual]
```

Transmit a Packet via ESP-NOW.
Serializes the packet and sends it to the registered peer. Blocks until send completes or times out (default 1000ms).

Parameters

packet	The Packet to transmit.

Returns

ResponseStatusContainer Success (code 0) or error with description.

Implements ITransmitter< Packet >.
The documentation for this class was generated from the following files:

- lib/telemetry/src/EspNowTransmitter.hpp
- lib/telemetry/src/EspNowTransmitter.cpp

6.13 FSMEventData Struct Reference

Container for event data passed through the FSM event system.
`#include <FlightState.hpp>`

Public Member Functions

- FSMEventData (FSMEvent e, RocketState target=RocketState::INACTIVE, void *data=nullptr)

  Construct FSM event data.

Public Attributes

- FSMEvent **event**
- RocketState **targetState**
- void * **eventData**

6.13.1 Detailed Description

Container for event data passed through the FSM event system.
Encapsulates an event with optional target state (for force transitions) and arbitrary event data payload.

6.13.2 Constructor & Destructor Documentation

6.13.2.1 FSMEventData()

```
FSMEventData::FSMEventData (
            FSMEvent e,
            RocketState target = RocketState::INACTIVE,
            void * data = nullptr)  [inline]
```

Construct FSM event data.

Parameters

e	Event type
target	Target state for force transitions (default: INACTIVE)

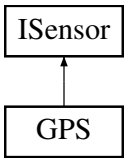
data	Optional event payload (default: nullptr)

The documentation for this struct was generated from the following file:

- lib/control/src/FlightState.hpp

6.14 GPS Class Reference

Driver for u-blox GNSS modules over I2C using the SparkFun library.
`#include <GPS.hpp>`
Inheritance diagram for GPS:



Public Member Functions

- **GPS** ()

  Construct a new GPS sensor object.
- bool init () override

  Initialize the GNSS receiver on the default I2C bus/address.
- std::optional< SensorData > getData () override

  Read current GNSS data and expose it as SensorData.

Public Member Functions inherited from ISensor

- bool isInitialized () const

  Check if the sensor is initialized.

Additional Inherited Members

Protected Member Functions inherited from ISensor

- void **setInitialized** (bool initialized)

6.14.1 Detailed Description

Driver for u-blox GNSS modules over I2C using the SparkFun library.
This class implements the ISensor interface and provides a minimal wrapper around SFE_UBLOX_GNSS to initialize the GNSS receiver and retrieve the most relevant navigation data (fix status, satellite count and, when valid, position and related metrics).
Data contract (keys in returned SensorData):

- "fix" (uint8_t): 0 no-fix, 2 2D-fix, 3 3D-fix
- "satellites" (uint8_t): number of satellites used in solution
- "latitude" (float, degrees) when fix >= 3
- "longitude" (float, degrees) when fix >= 3
- optional: "altitude" (float, meters), "speed" (float, km/h), "hdop" (float)

6.14.2 Member Function Documentation

6.14.2.1 getData()

`std::optional< SensorData > GPS::getData () [override], [virtual]`

Read current GNSS data and expose it as SensorData.
On success, always returns a SensorData with at least the fields "fix" and "satellites". When a 3D fix is available (fix >= 3), latitude and longitude (in decimal degrees) are included; additional fields like altitude (m), speed (km/h) and hdop may be provided depending on sensor configuration.

Returns

std::optional<SensorData> Populated data if the read succeeded; std::nullopt on communication failure.

Implements ISensor.

6.14.2.2 init()

`bool GPS::init () [override], [virtual]`

Initialize the GNSS receiver on the default I2C bus/address.
This will call Wire.begin() if needed and probe the module via SFE_UBLOX_GNSS::begin.

Returns

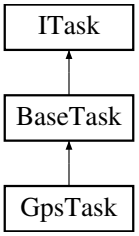
true on successful initialization, false otherwise.

Implements ISensor.
The documentation for this class was generated from the following files:

- lib/GPS/src/GPS.hpp
- lib/GPS/src/GPS.cpp

6.15 GpsTask Class Reference

Inheritance diagram for GpsTask:



Public Member Functions

- **GpsTask** (std::shared_ptr< SharedSensorData > sensorData, SemaphoreHandle_t sensorDataMutex, std::shared_ptr< ISensor > gps, std::shared_ptr< RocketLogger > rocketLogger, SemaphoreHandle_t loggerMutex)
- void **setGps** (std::shared_ptr< ISensor > gps)

Public Member Functions inherited from BaseTask

- **BaseTask** (const char *name)
- bool start (const TaskConfig &config) override

  Start the task with the given configuration.
- void stop () override

  Stop the task.
- bool isRunning () const override

Check if the task is currently running.

- const char * getName () const override

  Get the name of the task.

- uint32_t getStackHighWaterMark () const override

  Get the stack high water mark for the task.

Protected Member Functions

- void taskFunction () override
- void onTaskStart () override
- void onTaskStop () override

Additional Inherited Members

Protected Attributes inherited from BaseTask

- TaskHandle_t **taskHandle**
- TaskConfig **config**
- volatile bool **running**
- const char * **taskName**

6.15.1 Member Function Documentation

6.15.1.1 onTaskStart()

```
void GpsTask::onTaskStart () [override], [protected], [virtual]
```

Reimplemented from BaseTask.

6.15.1.2 onTaskStop()

```
void GpsTask::onTaskStop () [override], [protected], [virtual]
```

Reimplemented from BaseTask.

6.15.1.3 taskFunction()

```
void GpsTask::taskFunction () [override], [protected], [virtual]
```

Implements BaseTask.

The documentation for this class was generated from the following files:

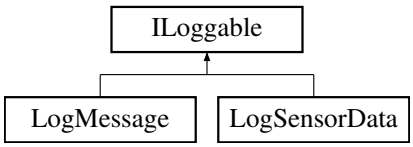
- lib/control/src/tasks/GpsTask.hpp
- lib/control/src/tasks/GpsTask.cpp

6.16 ILoggable Class Reference

Interface for objects that can be logged.

```
#include <ILoggable.hpp>
```

Inheritance diagram for ILoggable:



Public Member Functions

- virtual ~**ILoggable** ()=default

  Virtual destructor for proper cleanup.
- virtual std::string getSource () const

  Get the Source of the log.
- virtual json toJSON () const =0

  Get the JSON representation of the object.

Protected Attributes

- std::string **source**

6.16.1 Detailed Description

Interface for objects that can be logged.

6.16.2 Member Function Documentation

6.16.2.1 getSource()

```
virtual std::string ILoggable::getSource () const  [inline], [virtual]
```

Get the Source of the log.

Returns

A string representing the name of the source.

6.16.2.2 toJSON()

```
virtual json ILoggable::toJSON () const  [pure virtual]
```

Get the JSON representation of the object.

Returns

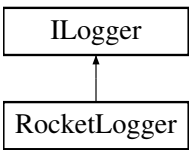
A json object.

Implemented in LogMessage, and LogSensorData.
The documentation for this class was generated from the following file:

- lib/data/src/ILoggable.hpp

6.17 ILogger Class Reference

Interface for loggers.
`#include <ILogger.hpp>`
Inheritance diagram for ILogger:



Public Member Functions

- int getLogCount () const

  Get the number of logged entries.
- virtual void logInfo (const std::string &message)=0

  Log an informational message.
- virtual void logWarning (const std::string &message)=0

  Log a warning message.
- virtual void logError (const std::string &message)=0

  Log an error message.
- virtual void logSensorData (const SensorData sensorData)=0

  Log sensor data.
- virtual void logSensorData (const std::string &sensorName, const SensorData sensorData)=0

  Log sensor data with a specific sensor name.
- virtual json getJSONAll () const =0

  Get all logged sensor data as a JSON list.
- virtual void clearData ()

  Clear all logged sensor data.
- virtual ~**ILogger** ()=default

  Virtual destructor to ensure proper cleanup in derived classes.

Protected Attributes

- std::vector< LogData > **logDataList**

6.17.1 Detailed Description

Interface for loggers.

6.17.2 Member Function Documentation

6.17.2.1 clearData()

```
virtual void ILogger::clearData ()  [inline], [virtual]
```

Clear all logged sensor data.
Reimplemented in RocketLogger.

6.17.2.2 getJSONAll()

```
virtual json ILogger::getJSONAll () const  [pure virtual]
```

Get all logged sensor data as a JSON list.

Returns

A json object.

Implemented in RocketLogger.

6.17.2.3 getLogCount()

```
int ILogger::getLogCount () const  [inline]
```

Get the number of logged entries.

Returns

int The number of logged entries.

6.17.2.4 logError()

```
virtual void ILogger::logError (
            const std::string & message)  [pure virtual]
```

Log an error message.

Parameters

message	The error message to log.

Implemented in RocketLogger.

6.17.2.5 logInfo()

```
virtual void ILogger::logInfo (
            const std::string & message)  [pure virtual]
```

Log an informational message.

Parameters

message	The informational message to log.

Implemented in RocketLogger.

6.17.2.6 logSensorData() [1/2]

```
virtual void ILogger::logSensorData (
            const SensorData sensorData)  [pure virtual]
```

Log sensor data.

Parameters

sensorData	The sensor data to log.

Implemented in RocketLogger.

6.17.2.7 logSensorData() [2/2]

```
virtual void ILogger::logSensorData (
            const std::string & sensorName,
            const SensorData sensorData)  [pure virtual]
```

Log sensor data with a specific sensor name.

Parameters

sensorName	The name of the sensor.
sensorData	The sensor data to log.

Implemented in RocketLogger.

6.17.2.8 logWarning()

```
virtual void ILogger::logWarning (
            const std::string & message)  [pure virtual]
```

Log a warning message.

Parameters

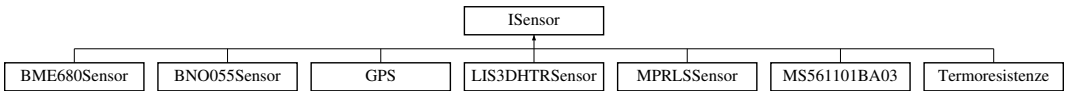
message	The warning message to log.

Implemented in RocketLogger.
The documentation for this class was generated from the following file:

- lib/logger/src/ILogger.hpp

6.18 ISensor Class Reference

Interface for sensors.
`#include <ISensor.hpp>`
Inheritance diagram for ISensor:



Public Member Functions

- virtual bool init ()=0

  Initialize the sensor.
- virtual std::optional< SensorData > getData ()=0

  Read and then get the sensor data.
- bool isInitialized () const

  Check if the sensor is initialized.

Protected Member Functions

- void **setInitialized** (bool initialized)

6.18.1 Detailed Description

Interface for sensors.

6.18.2 Member Function Documentation

6.18.2.1 getData()

`virtual std::optional< SensorData > ISensor::getData () [pure virtual]`
Read and then get the sensor data.

Returns

Just read data.

Implemented in BME680Sensor, BNO055Sensor, GPS, LIS3DHTRSensor, MPRLSSensor, MS561101BA03, and Termoresistenze.

6.18.2.2 init()

`virtual bool ISensor::init () [pure virtual]`
Initialize the sensor.

Returns

true if the sensor was initialized successfully

false if the sensor failed to initialize

Implemented in BME680Sensor, BNO055Sensor, GPS, LIS3DHTRSensor, MPRLSSensor, MS561101BA03, and Termoresistenze.

6.18.2.3 isInitialized()

```
bool ISensor::isInitialized () const  [inline]
```

Check if the sensor is initialized.

Returns

true if the sensor is initialized

false if the sensor is not initialized

The documentation for this class was generated from the following file:

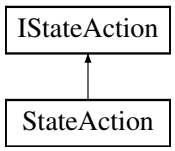
- lib/sensorCommon/src/ISensor.hpp

6.19 IStateAction Class Reference

Interface for state-specific actions and behaviors.

`#include <IStateAction.hpp>`

Inheritance diagram for IStateAction:



Public Member Functions

- virtual void onEntry ()
  Called when entering this state.
- virtual void onExit ()
  Called when exiting this state.
- virtual void onUpdate ()
  Called periodically while in this state.
- virtual RocketState getState () const =0
  Get the state this action handles.

6.19.1 Detailed Description

Interface for state-specific actions and behaviors.

This interface defines the contract for objects that handle state-specific logic including entry/exit actions and periodic updates during state execution.

6.19.2 Member Function Documentation

6.19.2.1 getState()

```
virtual RocketState IStateAction::getState () const  [pure virtual]
```

Get the state this action handles.

Returns

The RocketState this action is associated with

Implemented in StateAction.

6.19.2.2 onEntry()

```
virtual void IStateAction::onEntry ()  [inline], [virtual]
```

Called when entering this state.
Use this to:

- Initialize state-specific resources
- Start state-specific tasks
- Log state entry
- Set up hardware configurations

Reimplemented in StateAction.

6.19.2.3 onExit()

```
virtual void IStateAction::onExit ()  [inline], [virtual]
```

Called when exiting this state.
Use this to:

- Clean up state-specific resources
- Stop state-specific tasks
- Log state exit
- Save state data

Reimplemented in StateAction.

6.19.2.4 onUpdate()

```
virtual void IStateAction::onUpdate ()  [inline], [virtual]
```

Called periodically while in this state.
Use this for:

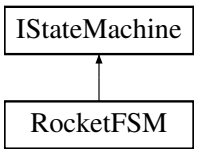
- Periodic state-specific processing
- Condition monitoring
- Data updates

The documentation for this class was generated from the following file:

- lib/control/src/states/IStateAction.hpp

6.20 IStateMachine Class Reference

Interface for a finite state machine managing rocket flight phases.
`#include <IStateMachine.hpp>`
Inheritance diagram for IStateMachine:



Public Member Functions

- virtual void init ()=0

  Initialize the state machine and its components.
- virtual void start ()=0

  Start the state machine and begin processing.
- virtual void stop ()=0

  Stop the state machine and clean up resources.
- virtual bool sendEvent (FSMEvent event, RocketState targetState=RocketState::INACTIVE, void *eventData=nullptr)=0

  Send an event to trigger state transitions.
- virtual RocketState getCurrentState ()=0

  Get the current state of the state machine.
- virtual FlightPhase getCurrentPhase ()=0

  Get the current high-level flight phase.
- virtual void forceTransition (RocketState newState)=0

  Force an immediate transition to the specified state.
- virtual bool isFinished ()=0

  Check if the state machine has completed its mission.

6.20.1 Detailed Description

Interface for a finite state machine managing rocket flight phases.
This interface defines the contract for a state machine that manages the complete lifecycle of a rocket flight, from initialization through recovery. Implementations should handle state transitions, event processing, and task management in a thread-safe manner suitable for real-time embedded systems.
State Machine Behavior:

- States represent distinct phases of rocket operation (INACTIVE, CALIBRATING, etc.)
- Events trigger state transitions when specific conditions are met
- Transitions can be automatic (time-based, sensor-based) or event-driven
- Each state may have associated tasks that run concurrently

Thread Safety:

- All methods must be thread-safe for use in multi-core environments
- State transitions should be atomic operations
- Event handling should be non-blocking where possible

See also

RocketState for available states

FSMEvent for triggerable events

FlightPhase for high-level flight phases

6.20.2 Member Function Documentation

6.20.2.1 forceTransition()

```
virtual void IStateMachine::forceTransition (
            RocketState newState)  [pure virtual]
```

Force an immediate transition to the specified state.
This bypasses normal transition rules and should be used with caution. Typical use cases:

- Emergency abort sequences
- Testing and debugging
- Manual override conditions
- Recovery from error states

Implementation should:

- Execute the transition immediately if possible
- Handle state cleanup (exit actions, task stopping)
- Start new state (entry actions, task starting)
- Log the forced transition for debugging
- Be thread-safe

Parameters

newState	The state to transition to immediately

Warning

Bypasses normal safety checks - use carefully

Note

Should be used sparingly, primarily for emergency/test scenarios

Implemented in RocketFSM.

6.20.2.2 getCurrentPhase()

```
virtual FlightPhase IStateMachine::getCurrentPhase ()  [pure virtual]
```

Get the current high-level flight phase.
Flight phases group related states for easier high-level logic:

- PRE_FLIGHT: INACTIVE, CALIBRATING, READY_FOR_LAUNCH
- FLIGHT: LAUNCH, ACCELERATED_FLIGHT, BALLISTIC_FLIGHT, APOGEE
- RECOVERY: STABILIZATION, DECELERATION, LANDING, RECOVERED

Implementation should map the current state to its corresponding phase.

Returns

Current FlightPhase based on current state

Note

Thread-safe, derived from getCurrentState()

Implemented in RocketFSM.

6.20.2.3 getCurrentState()

```
virtual RocketState IStateMachine::getCurrentState ()  [pure virtual]
```

Get the current state of the state machine.
Returns the current state atomically. Implementation should:

- Use appropriate locking to ensure atomic read
- Return immediately without blocking
- Be safe to call from any thread at any time

Returns

Current RocketState (INACTIVE, CALIBRATING, LAUNCH, etc.)

Note

Thread-safe getter

Should never block or fail

Implemented in RocketFSM.

6.20.2.4 init()

```
virtual void IStateMachine::init ()  [pure virtual]
```

Initialize the state machine and its components.
This method should:

- Set up internal data structures (event queues, mutexes, etc.)
- Initialize the state machine in the INACTIVE state
- Configure state actions and transition rules
- Prepare task management systems
- Allocate necessary memory resources

Should be called once before start() and must complete successfully before the state machine can be used.

Exceptions

May	throw/return error if initialization fails

Note

Must be called from main thread before starting tasks

Implemented in RocketFSM.

6.20.2.5 isFinished()

```
virtual bool IStateMachine::isFinished ()  [pure virtual]
```

Check if the state machine has completed its mission.
Returns true when the rocket has reached a terminal state (usually RECOVERED) and no further state transitions are expected. Implementation should:

- Return true for terminal states (RECOVERED, or error states)
- Return false for all active states
- Consider mission abort conditions

This can be used by higher-level systems to determine when mission operations are complete.

Returns

true if mission is complete, false if still active

Note

Useful for determining when to shut down systems

Implemented in RocketFSM.

6.20.2.6 sendEvent()

```
virtual bool IStateMachine::sendEvent (
            FSMEvent event,
            RocketState targetState = RocketState::INACTIVE,
            void * eventData = nullptr)  [pure virtual]
```

Send an event to trigger state transitions.
Events are the primary mechanism for triggering state changes. This method should:

- Queue the event for processing by the main FSM task
- Return immediately without blocking (asynchronous)
- Handle queue full conditions gracefully
- Support both targeted and general events

Event Processing:

- Events are processed by the main FSM task in FIFO order
- Each event is matched against current state transition rules
- Valid transitions execute immediately, invalid ones are ignored
- FORCE_TRANSITION events bypass normal transition rules

Parameters

event	The event type to send (START_CALIBRATION, LAUNCH_DETECTED, etc.)
targetState	For FORCE_TRANSITION events, the destination state
eventData	Optional data payload for the event (may be nullptr)

Returns

true if event was queued successfully, false if queue full or invalid

Note

Thread-safe, can be called from any task

Non-blocking operation

Implemented in RocketFSM.

6.20.2.7 start()

```
virtual void IStateMachine::start ()  [pure virtual]
```

Start the state machine and begin processing.
This method should:

- Create and start the main FSM task

- Begin event processing loop
- Start state-specific tasks based on current state
- Enable automatic transition checking
- Activate watchdog timers if used

The state machine will begin in INACTIVE state and should automatically transition to CALIBRATING if conditions are met.

Note

Should only be called after successful init()

May be called multiple times (should handle already-running case)

Implemented in RocketFSM.

6.20.2.8 stop()

```
virtual void IStateMachine::stop ()  [pure virtual]
```

Stop the state machine and clean up resources.
This method should:

- Signal all tasks to stop execution
- Wait for tasks to terminate gracefully
- Clean up FreeRTOS objects (tasks, queues, mutexes)
- Reset internal state to stopped condition
- Preserve current state for potential restart

Should be safe to call multiple times and should not block indefinitely. After calling stop(), the state machine can be restarted with start().

Note

Should handle case where FSM is already stopped

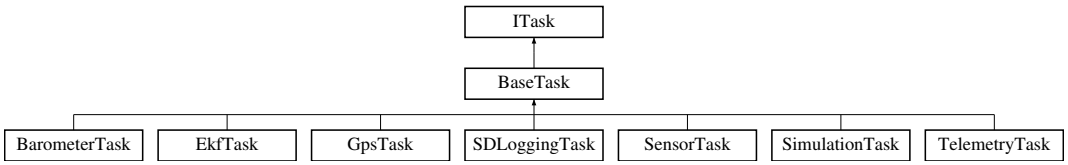
Should be safe to call from any thread

Implemented in RocketFSM.
The documentation for this class was generated from the following file:

- lib/control/src/IStateMachine.hpp

6.21 ITask Class Reference

Interface for defining tasks in the FSM.
`#include <ITask.hpp>`
Inheritance diagram for ITask:



Public Member Functions

- virtual bool start (const TaskConfig &config)=0

  Start the task with the given configuration.
- virtual void stop ()=0

  Stop the task.
- virtual bool isRunning () const =0

  Check if the task is currently running.
- virtual const char * getName () const =0

  Get the name of the task.
- virtual uint32_t getStackHighWaterMark () const =0

  Get the stack high water mark for the task.

Protected Member Functions

- virtual void **taskFunction** ()=0

6.21.1 Detailed Description

Interface for defining tasks in the FSM.

6.21.2 Member Function Documentation

6.21.2.1 getName()

```
virtual const char * ITask::getName () const  [pure virtual]
```

Get the name of the task.

Returns

const char* Name of the task

Implemented in BaseTask.

6.21.2.2 getStackHighWaterMark()

```
virtual uint32_t ITask::getStackHighWaterMark () const  [pure virtual]
```

Get the stack high water mark for the task.

Returns

uint32_t High water mark in bytes

Implemented in BaseTask.

6.21.2.3 isRunning()

```
virtual bool ITask::isRunning () const  [pure virtual]
```

Check if the task is currently running.

Returns

true if running

false otherwise

Implemented in BaseTask.

6.21.2.4 start()

```
virtual bool ITask::start (
            const TaskConfig & config)  [pure virtual]
```

Start the task with the given configuration.

Parameters

config	TaskConfig: Configuration for the task

Returns

true if the task started successfully

false otherwise

Implemented in BaseTask.

6.21.2.5 stop()

```
virtual void ITask::stop ()  [pure virtual]
```

Stop the task.
Implemented in BaseTask.
The documentation for this class was generated from the following file:

- lib/control/src/tasks/ITask.hpp

6.22 ITransitionCondition Class Reference

Interface for transition condition evaluation.
`#include <IStateAction.hpp>`

Public Member Functions

- virtual bool isConditionMet ()=0
  Evaluate whether the transition condition is met.
- virtual const char * getConditionName () const =0
  Get a human-readable name for this condition.

6.22.1 Detailed Description

Interface for transition condition evaluation.
This interface defines conditions that can trigger automatic state transitions. Implementations should evaluate specific conditions (time-based, sensor-based, etc.) and return whether the transition should occur.

6.22.2 Member Function Documentation

6.22.2.1 getConditionName()

```
virtual const char * ITransitionCondition::getConditionName () const  [pure virtual]
```

Get a human-readable name for this condition.
Used for logging and debugging purposes.

Returns

C-string describing this condition (e.g., "Calibration Timeout")

6.22.2.2 isConditionMet()

```
virtual bool ITransitionCondition::isConditionMet ()  [pure virtual]
```

Evaluate whether the transition condition is met.
This method should:

- Check relevant sensors, timers, or other data
- Return true if transition should occur
- Be lightweight and non-blocking
- Be thread-safe

Returns

true if condition is satisfied, false otherwise

The documentation for this class was generated from the following file:

- lib/control/src/states/IStateAction.hpp

6.23 ITransmitter< T > Class Template Reference

Interface for a transmitter.
`#include <ITransmitter.hpp>`

Public Member Functions

- virtual ResponseStatusContainer init ()=0
- virtual ResponseStatusContainer transmit (T data)=0

6.23.1 Detailed Description

template<typename T>
class ITransmitter< T >

Interface for a transmitter.

6.23.2 Member Function Documentation

6.23.2.1 init()

```
template<typename T>
virtual ResponseStatusContainer ITransmitter< T >::init ()  [pure virtual]
```

Implemented in E220LoRaTransmitter, EspNowTransmitter, and LoRaTransmitter.

6.23.2.2 transmit()

```
template<typename T>
virtual ResponseStatusContainer ITransmitter< T >::transmit (
            T data)  [pure virtual]
```

Implemented in E220LoRaTransmitter, EspNowTransmitter, and LoRaTransmitter.
The documentation for this class was generated from the following file:

- lib/telemetry/src/ITransmitter.hpp

6.24 KalmanFilter Class Reference

Extended Kalman Filter for attitude/kinematics estimation.
`#include <KalmanFilter.hpp>`

Public Member Functions

- KalmanFilter (Eigen::Vector3f gravity_value, Eigen::Vector3f magnometer_value)

  Construct the EKF with reference gravity and magnetometer values.
- std::vector< std::vector< float > > step (float dt, float omega[3], float accel[3])

  Advance the filter by one time step.
- float * state ()

  Access the internal state vector storage.

6.24.1 Detailed Description

Extended Kalman Filter for attitude/kinematics estimation.
State layout (EKF_N = 16):

- 3x position, 3x velocity, 4x quaternion (rotation), 3x accel bias, 3x gyro bias Measurement layout (EKF_M = 6):
- 3x linear acceleration, 3x orientation components (see implementation notes)

6.24.2 Constructor & Destructor Documentation

6.24.2.1 KalmanFilter()

```
KalmanFilter::KalmanFilter (
            Eigen::Vector3f gravity_value,
            Eigen::Vector3f magnometer_value)
```

Construct the EKF with reference gravity and magnetometer values.

Parameters

gravity_value	Gravity vector used for initial alignment/calibration.
magnometer_value	Magnetometer reference for yaw alignment.

6.24.3 Member Function Documentation

6.24.3.1 state()

```
float * KalmanFilter::state ()
```

Access the internal state vector storage.

Returns

Pointer to the first element of the EKF state array (size EKF_N).

6.24.3.2 step()

```
std::vector< std::vector< float > > KalmanFilter::step (
            float dt,
            float omega[3],
            float accel[3])
```

Advance the filter by one time step.

Parameters

dt	Time step in seconds.
omega	Angular rates [rad/s] as a triad {wx, wy, wz}.
accel	Linear accelerations [m/s^2] as a triad {ax, ay, az}.

Returns

A container with updated outputs (see source for structure semantics).

The documentation for this class was generated from the following files:

- lib/kalman/src/KalmanFilter.hpp
- lib/kalman/src/KalmanFilter.cpp

6.25 KalmanFilter1D Class Reference

One-dimensional Extended Kalman Filter for sensor fusion.
`#include <KalmanFilter1D.hpp>`

Public Member Functions

- KalmanFilter1D (Eigen::Vector3f gravity_value, Eigen::Vector3f magnometer_value)

  Construct a new KalmanFilter1D object and perform initial calibration.
- void step (float dt, float omega[3], float accel[3], float pressure, float gps)

  Perform one EKF prediction and update step.
- float * state ()

  Get the current EKF state vector.
- void **setParameters** (float p0, float v0, float qa, float a0, float g0)
- void **updateCovarianceMatrices** ()

6.25.1 Detailed Description

One-dimensional Extended Kalman Filter for sensor fusion.
This class implements a 1D EKF for fusing IMU (accelerometer, gyroscope) and barometer data. The state vector includes position, velocity, and orientation (as quaternion). The filter estimates the vertical position and velocity, as well as orientation, compensating for sensor biases and gravity.

6.25.2 Constructor & Destructor Documentation

6.25.2.1 KalmanFilter1D()

```
KalmanFilter1D::KalmanFilter1D (
            Eigen::Vector3f gravity_value,
            Eigen::Vector3f magnometer_value)
```

Construct a new KalmanFilter1D object and perform initial calibration.

Parameters

gravity_value	Initial gravity vector (from accelerometer).
magnometer_value	Initial magnetic field vector (from magnetometer).

6.25.3 Member Function Documentation

6.25.3.1 state()

```
float * KalmanFilter1D::state ()
```

Get the current EKF state vector.
Get the current state vector [altitude, z-vel, q_w, q_x, q_y, q_z].

Returns

Pointer to the state vector.

float* 6 values [altitude, z-vel, q_w, q_x, q_y, q_z]

6.25.3.2 step()

```
void KalmanFilter1D::step (
            float dt,
            float omega[3],
            float accel[3],
```

```
            float pressure,
            float gps)
```

Perform one EKF prediction and update step.

Parameters

dt	Time step in seconds.
omega	Gyroscope readings [rad/s].
accel	Accelerometer readings [m/s^2].
pressure	Barometer reading (altitude or pressure).

The documentation for this class was generated from the following files:

- lib/kalman/src/KalmanFilter1D.hpp
- lib/kalman/src/KalmanFilter1D.cpp

6.26 LEDController Class Reference

Public Member Functions

- **LEDController** (uint8_t redPin, uint8_t greenPin, uint8_t bluePin)
- void **addStatusLeds** (uint8_t *pins, uint8_t count)
- void **init** ()
- void **startPattern** (const LEDPattern &pattern)
- void **stopPattern** ()
- void **setColor** (LEDColor color)
- void **setOff** ()

Static Public Member Functions

- static void **getRGBValues** (LEDColor color, uint8_t &r, uint8_t &g, uint8_t &b)

The documentation for this class was generated from the following files:

- lib/StatusManager/src/LEDController.hpp
- lib/StatusManager/src/LEDController.cpp

6.27 LEDPattern Struct Reference

Public Attributes

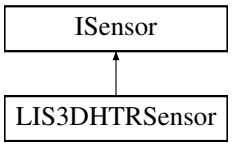
- LEDColor **color1**
- LEDColor **color2**
- uint16_t **duration**
- uint16_t **pause**
- uint8_t **times**
- uint8_t **auxLeds**

The documentation for this struct was generated from the following file:

- lib/StatusManager/src/LEDController.hpp

6.28 LIS3DHTRSensor Class Reference

Inheritance diagram for LIS3DHTRSensor:



Public Member Functions

- bool init () override

  Initialize the sensor.
- std::optional< SensorData > getData () override

  Read and then get the sensor data.

Public Member Functions inherited from ISensor

- bool isInitialized () const

  Check if the sensor is initialized.

Additional Inherited Members

Protected Member Functions inherited from ISensor

- void **setInitialized** (bool initialized)

6.28.1 Member Function Documentation

6.28.1.1 getData()

```
std::optional< SensorData > LIS3DHTRSensor::getData ()  [override], [virtual]
```

Read and then get the sensor data.

Returns

Just read data.

Implements ISensor.

6.28.1.2 init()

```
bool LIS3DHTRSensor::init ()  [override], [virtual]
```

Initialize the sensor.

Returns

true if the sensor was initialized successfully

false if the sensor failed to initialize

Implements ISensor.

The documentation for this class was generated from the following files:

- lib/LIS3DHTR/LIS3DHTRSensor.hpp
- lib/LIS3DHTR/LIS3DHTRSensor.cpp

6.29 LogData Class Reference

Class to store log data.

```
#include <LogData.hpp>
```

Public Member Functions

- LogData (const std::string &source, const ILoggable *data)

  Construct a new Log Data object.
- std::string getSource () const

  Get the Source of the log.
- const ILoggable * getData () const

  Get the Data pointer.
- json toJSON () const

  Return a JSON representation of the log data.

6.29.1 Detailed Description

Class to store log data.

6.29.2 Constructor & Destructor Documentation

6.29.2.1 LogData()

```
LogData::LogData (
            const std::string & source,
            const ILoggable * data)  [inline]
```

Construct a new Log Data object.

Parameters

source	Origin of the log.
data	Data to be logged.

6.29.3 Member Function Documentation

6.29.3.1 getData()

```
const ILoggable * LogData::getData () const  [inline]
```

Get the Data pointer.

Returns

A pointer to the ILoggable data.

6.29.3.2 getSource()

```
std::string LogData::getSource () const  [inline]
```

Get the Source of the log.

Returns

A string representing the name of the source.

6.29.3.3 toJSON()

```
json LogData::toJSON () const  [inline]
```

Return a JSON representation of the log data.

Returns

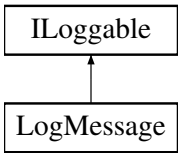
A json object.

The documentation for this class was generated from the following file:

- lib/logger/src/LogData.hpp

6.30 LogMessage Class Reference

A class to represent a log message.
`#include <LogMessage.hpp>`
Inheritance diagram for LogMessage:



Public Member Functions

- LogMessage (std::string source, std::string message)

  Construct a new Log Message object.
- std::string getMessage () const

  Get the message string.
- json toJSON () const override

  Get the JSON representation of the object.

Public Member Functions inherited from ILoggable

- virtual ~**ILoggable** ()=default

  Virtual destructor for proper cleanup.
- virtual std::string getSource () const

  Get the Source of the log.

Additional Inherited Members

Protected Attributes inherited from ILoggable

- std::string **source**

6.30.1 Detailed Description

A class to represent a log message.

6.30.2 Constructor & Destructor Documentation

6.30.2.1 LogMessage()

```
LogMessage::LogMessage (
            std::string source,
            std::string message)  [inline]
```

Construct a new Log Message object.

Parameters

source	The origin of the log.
message	The message to be logged.

6.30.3 Member Function Documentation

6.30.3.1 getMessage()

`std::string LogMessage::getMessage () const [inline]`
Get the message string.

Returns

A string representing the message.

6.30.3.2 toJSON()

`json LogMessage::toJSON () const [inline], [override], [virtual]`
Get the JSON representation of the object.

Returns

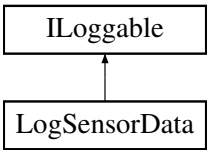
A json object.

Implements ILoggable.
The documentation for this class was generated from the following file:

- lib/data/src/LogMessage.hpp

6.31 LogSensorData Class Reference

A class to represent a log of sensor data.
`#include <LogSensorData.hpp>`
Inheritance diagram for LogSensorData:



Public Member Functions

- LogSensorData (std::string source, SensorData sensorData)

  Construct a new Log Sensor Data object.
- SensorData getSensorData () const

  Get the sensor data.
- json toJSON () const override

  Get the JSON representation of the object.

Public Member Functions inherited from ILoggable

- virtual ~**ILoggable** ()=default

  Virtual destructor for proper cleanup.
- virtual std::string getSource () const

  Get the Source of the log.

Additional Inherited Members

Protected Attributes inherited from ILoggable

- std::string **source**

6.31.1 Detailed Description

A class to represent a log of sensor data.

6.31.2 Constructor & Destructor Documentation

6.31.2.1 LogSensorData()

```
LogSensorData::LogSensorData (
            std::string source,
            SensorData sensorData)  [inline]
```

Construct a new Log Sensor Data object.

Parameters

source	The origin of the log.
sensorData	The sensor data to be logged.

6.31.3 Member Function Documentation

6.31.3.1 getSensorData()

```
SensorData LogSensorData::getSensorData () const  [inline]
```

Get the sensor data.

Returns

The sensor data.

6.31.3.2 toJSON()

```
json LogSensorData::toJSON () const  [inline], [override], [virtual]
```

Get the JSON representation of the object.

Returns

A json object.

Implements ILoggable.

The documentation for this class was generated from the following file:

- lib/data/src/LogSensorData.hpp

6.32 LoRaConfigurationDeserializer Class Reference

Class to deserialize LoRa configuration from JSON object or file.

`#include <LoRaConfigurationDeserializer.hpp>`

Public Member Functions

- **LoRaConfigurationDeserializer** (Configuration configuration, ILogger *logger)
- **LoRaConfigurationDeserializer** (ILogger *logger)
- bool isJsonValid (const nlohmann::json &json) const

  Check if JSON configuration is invalid. Required parameters are ADDH, ADDL, CHAN.
- bool deserializeConfiguration (const nlohmann::json &json)

  Deserialize configuration from JSON object.
- bool deserializeConfiguration (File &file)

  Deserialize configuration from JSON file.
- Configuration getConfiguration () const

  Get the Configuration object.

6.32.1 Detailed Description

Class to deserialize LoRa configuration from JSON object or file.

6.32.2 Member Function Documentation

6.32.2.1 deserializeConfiguration() [1/2]

```
bool LoRaConfigurationDeserializer::deserializeConfiguration (
            const nlohmann::json & json)  [inline]
```

Deserialize configuration from JSON object.

Parameters

json	A JSON object containing configuration for Ebyte E220-900T22D LoRa module.

Returns

true if configuration is deserialized successfully, false otherwise

6.32.2.2 deserializeConfiguration() [2/2]

```
bool LoRaConfigurationDeserializer::deserializeConfiguration (
            File & file)  [inline]
```

Deserialize configuration from JSON file.

Parameters

file	A file containing JSON configuration for Ebyte E220-900T22D LoRa module.

Returns

true if configuration is deserialized successfully, false otherwise

6.32.2.3 getConfiguration()

```
Configuration LoRaConfigurationDeserializer::getConfiguration () const  [inline]
```

Get the Configuration object.

Returns

Configuration

6.32.2.4 isJsonValid()

```
bool LoRaConfigurationDeserializer::isJsonValid (
            const nlohmann::json & json) const  [inline]
```

Check if JSON configuration is invalid. Required parameters are ADDH, ADDL, CHAN.

Parameters

json	A JSON object

Returns

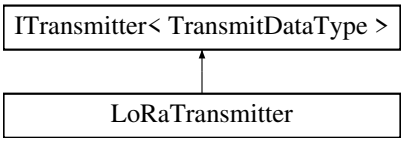
true if JSON object is valid, false if JSON object is invalid

The documentation for this class was generated from the following file:

- lib/LoRa/src/LoRaConfigurationDeserializer.hpp

6.33 LoRaTransmitter Class Reference

Transmitter based on RadioLib for SX1261 module.
`#include <SX1261LoRaTransmitter.hpp>`
Inheritance diagram for LoRaTransmitter:



Public Member Functions

- **LoRaTransmitter** ()

  Constructor.
- ResponseStatusContainer init () override

  Initializes the LoRa SX1261 module.
- ResponseStatusContainer transmit (TransmitDataType data) override

  Transmits data through LoRa.
- ResponseStatusContainer transmitCompact (const nlohmann::json &data)

  Transmits only essential data to reduce payload.
- void configure (float freq, int8_t pwr, float bw, uint8_t sf, uint8_t cr)

  Configures the LoRa parameters.
- uint16_t getPacketCount () const

  Gets the number of transmitted packets.
- void **printStats** ()

  Stampa le statistiche del modulo.

6.33.1 Detailed Description

Transmitter based on RadioLib for SX1261 module.
Manages initialization, basic configuration (frequency, power, BW, SF, CR) and transmission of payloads in various formats (char∗, String, std::string, JSON). Default pin mapping: NSS=19, DIO1=2, NRST=14, BUSY=4.

6.33.2 Member Function Documentation

6.33.2.1 configure()

```
void LoRaTransmitter::configure (
            float freq,
            int8_t pwr,
            float bw,
            uint8_t sf,
            uint8_t cr)  [inline]
```

Configures the LoRa parameters.

Parameters

freq	Frequency in MHz.
pwr	Power in dBm.
bw	Bandwidth in kHz.
sf	Spreading Factor (7..12).
cr	Coding Rate (4/CR).

6.33.2.2 getPacketCount()

```
uint16_t LoRaTransmitter::getPacketCount () const  [inline]
```

Gets the number of transmitted packets.

Returns

uint16_t Packet counter.

6.33.2.3 init()

```
ResponseStatusContainer LoRaTransmitter::init ()  [inline], [override], [virtual]
```

Initializes the LoRa SX1261 module.

Returns

ResponseStatusContainer with the initialization status.

Implements ITransmitter< TransmitDataType >.

6.33.2.4 transmit()

```
ResponseStatusContainer LoRaTransmitter::transmit (
            TransmitDataType data)  [inline], [override], [virtual]
```

Transmits data through LoRa.

Parameters

data	The data to transmit (variant type)

Returns

ResponseStatusContainer with the transmission status

Implements ITransmitter< TransmitDataType >.

6.33.2.5 transmitCompact()

```
ResponseStatusContainer LoRaTransmitter::transmitCompact (
            const nlohmann::json & data)  [inline]
```

Transmits only essential data to reduce payload.

Parameters

data	Complete JSON of the rocket logger

Returns

ResponseStatusContainer with the transmission status

The documentation for this class was generated from the following file:

- lib/LoRa/src/SX1261LoRaTransmitter.hpp

6.34 MedianFilter Class Reference

Public Member Functions

- **MedianFilter** (size_t windowSize)
- float **update** (float newValue)
- void **reset** ()
- bool **isReady** () const

The documentation for this class was generated from the following file:

- lib/control/src/tasks/BarometerTask.hpp

6.35 MovingAverageFilter Class Reference

Public Member Functions

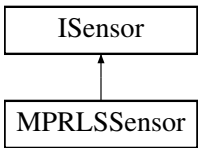
- **MovingAverageFilter** (size_t windowSize=10)
- float **update** (float newValue)
- void **reset** ()
- bool **isReady** () const

The documentation for this class was generated from the following file:

- lib/control/src/tasks/BarometerTask.hpp

6.36 MPRLSSensor Class Reference

Inheritance diagram for MPRLSSensor:



Public Member Functions

- bool init () override

  Initialize the sensor.
- std::optional< SensorData > getData () override

  Read and then get the sensor data.

Public Member Functions inherited from ISensor

- bool isInitialized () const

  Check if the sensor is initialized.

Additional Inherited Members

Protected Member Functions inherited from ISensor

- void **setInitialized** (bool initialized)

6.36.1 Member Function Documentation

6.36.1.1 getData()

`std::optional< SensorData > MPRLSSensor::getData () [override], [virtual]`
Read and then get the sensor data.

Returns

Just read data.

Implements ISensor.

6.36.1.2 init()

`bool MPRLSSensor::init () [override], [virtual]`
Initialize the sensor.

Returns

true if the sensor was initialized successfully

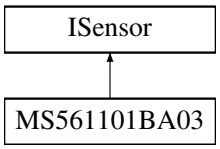
false if the sensor failed to initialize

Implements ISensor.
The documentation for this class was generated from the following files:

- lib/MPRLS/src/MPRLSSensor.hpp
- lib/MPRLS/src/MPRLSSensor.cpp

6.37 MS561101BA03 Class Reference

Inheritance diagram for MS561101BA03:



Public Member Functions

- **MS561101BA03** (uint8_t address=0x77)
- bool init () override

  Initialize the sensor.
- std::optional< SensorData > getData () override

  Read and then get the sensor data.

Public Member Functions inherited from ISensor

- bool isInitialized () const

  Check if the sensor is initialized.

Additional Inherited Members

Protected Member Functions inherited from ISensor

- void **setInitialized** (bool initialized)

6.37.1 Member Function Documentation

6.37.1.1 getData()

`std::optional< SensorData > MS561101BA03::getData () [override], [virtual]`
Read and then get the sensor data.

Returns

Just read data.

Implements ISensor.

6.37.1.2 init()

`bool MS561101BA03::init () [override], [virtual]`
Initialize the sensor.

Returns

true if the sensor was initialized successfully

false if the sensor failed to initialize

Implements ISensor.
The documentation for this class was generated from the following files:

- lib/MS561101BA03/src/MS561101BA03.hpp
- lib/MS561101BA03/src/MS561101BA03.cpp

6.38 Packet Struct Reference

A transmission packet with a header and a payload.
`#include <Packet.hpp>`

Public Member Functions

- void **calculateCRC** ()

  Calculate the CRC16 of the packet. This writes into `crc`.
- void printPacket ()

  Print a human-readable representation of the packet to Serial.

Public Attributes

- PacketHeader **header**
- PacketPayload **payload**
- uint16_t **crc**

6.38.1 Detailed Description

A transmission packet with a header and a payload.
The CRC field is placed after the payload. Packet::calculateCRC() computes the CRC over the bytes from the start of the struct up to (but excluding) this crc field.

6.38.2 Member Function Documentation

6.38.2.1 printPacket()

`void Packet::printPacket ()`
Print a human-readable representation of the packet to Serial.
The output is intended for debugging. Ensure `Serial.begin()` has been called by the sketch before calling this.
The documentation for this struct was generated from the following files:

- lib/protocol/src/Packet.hpp
- lib/protocol/src/Packet.cpp

6.39 PacketHeader Struct Reference

The packet header.
`#include <Packet.hpp>`

Public Attributes

- uint16_t **messageId** = 1
- uint8_t **totalChunks** = 0
- uint8_t **chunkIndex** = 0
- uint8_t **payloadSize** = 0
- uint8_t **flags** = 0
- uint8_t **protocolVersion** = 1

6.39.1 Detailed Description

The packet header.
Fields are chosen to be compact and sufficient for reassembly and control.

- messageId: 16-bit identifier for the logical message/transfer. Wraps on overflow.
- flags: 8-bit bitfield (bit0 = Start of Message, bit1 = End of Message, bit2 = ACK request, bits 3-7 reserved)
- totalChunks: total number of chunks in the message (0..255)
- chunkIndex: zero-based index of this chunk within the message (0..totalChunks-1)
- payloadSize: number of valid payload bytes in this packet (0..LORA_MAX_PAYLOAD_SIZE)
- protocolVersion: fixed protocol format version. Increment to indicate incompatible layout changes.

The documentation for this struct was generated from the following file:

- lib/protocol/src/Packet.hpp

6.40 PacketManager Class Reference

Static Public Member Functions

- static std::vector< uint8_t > serialize (const Packet &packet)
  Serializes a Packet object into a byte vector.
- static Packet deserialize (const uint8_t *data, size_t length)
  Deserializes a byte array into a Packet object.
- static std::vector< Packet > divideMessage (const uint8_t *message, size_t length)
  Divides a large message into multiple Packet objects.
- static std::vector< uint8_t > reassembleMessage (const std::vector< Packet > &packets)
  Reassembles a complete message from multiple Packet objects.

6.40.1 Member Function Documentation

6.40.1.1 deserialize()

```
Packet PacketManager::deserialize (
            const uint8_t * data,
            size_t length)  [static]
```

Deserializes a byte array into a Packet object.

Parameters

data	Pointer to the byte array.
length	Length of the byte array.

Returns

Packet The deserialized Packet object.

6.40.1.2 divideMessage()

```
std::vector< Packet > PacketManager::divideMessage (
            const uint8_t * message,
            size_t length)  [static]
```

Divides a large message into multiple Packet objects.

Parameters

message	Pointer to the message byte array.
length	Length of the message byte array.

Returns

std::vector<Packet> A vector of Packet objects representing the divided message.

6.40.1.3 reassembleMessage()

```
std::vector< uint8_t > PacketManager::reassembleMessage (
            const std::vector< Packet > & packets)  [static]
```

Reassembles a complete message from multiple Packet objects.

Parameters

packets	A vector of Packet objects to reassemble.

Returns

std::vector<uint8_t> The reassembled message as a byte vector.

6.40.1.4 serialize()

```
std::vector< uint8_t > PacketManager::serialize (
            const Packet & packet)  [static]
```

Serializes a Packet object into a byte vector.

Parameters

packet	The Packet object to serialize.

Returns

std::vector<uint8_t> The serialized byte vector.

The documentation for this class was generated from the following files:

- lib/protocol/src/PacketManager.hpp
- lib/protocol/src/PacketManager.cpp

6.41 PacketPayload Struct Reference

The payload of a packet. This is a fixed-size buffer equal to the maximum payload allowed by the header and constants. When a chunk is smaller than this size it should be padded deterministically (the code currently uses 0x01) so CRCs remain deterministic across sender/receiver.
`#include <Packet.hpp>`

Public Attributes

- uint8_t **data** [LORA_MAX_PAYLOAD_SIZE]

6.41.1 Detailed Description

The payload of a packet. This is a fixed-size buffer equal to the maximum payload allowed by the header and constants. When a chunk is smaller than this size it should be padded deterministically (the code currently uses 0x01) so CRCs remain deterministic across sender/receiver.
The documentation for this struct was generated from the following file:

- lib/protocol/src/Packet.hpp

6.42 PacketSerializer Class Reference

Static Public Member Functions

- static std::string **serialize** (const Packet &packet)

The documentation for this class was generated from the following file:

- lib/protocol/src/PacketSerializer.hpp

6.43 ResponseStatusContainer Class Reference

Class that contains the response status and the response message from methods that need to return a status and a message.
`#include <ResponseStatusContainer.hpp>`

Public Member Functions

- **ResponseStatusContainer** (int code, String description)
- int **getCode** () const
- String **getDescription** () const
- **ResponseStatusContainer** (int code, String description)
- int **getCode** () const
- String **getDescription** () const

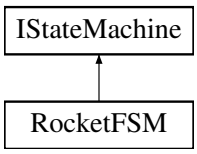
6.43.1 Detailed Description

Class that contains the response status and the response message from methods that need to return a status and a message.
The documentation for this class was generated from the following files:

- lib/telemetry/src/ResponseStatusContainer.hpp
- include/ResponseStatusContainer.hpp

6.44 RocketFSM Class Reference

Inheritance diagram for RocketFSM:



Public Member Functions

- **RocketFSM** (std::shared_ptr< ISensor > imu, std::shared_ptr< ISensor > barometer1, std::shared_ptr< ISensor > barometer2, std::shared_ptr< ISensor > accelerometer, std::shared_ptr< ISensor > gpsModule, std::shared_ptr< SD > sd, std::shared_ptr< RocketLogger > logger)
- void init () override

  Initialize the state machine and its components.
- void start () override

  Start the state machine and begin processing.
- void stop () override

  Stop the state machine and clean up resources.
- bool sendEvent (FSMEvent event, RocketState targetState=RocketState::INACTIVE, void *event↩ Data=nullptr) override

  Send an event to trigger state transitions.
- RocketState getCurrentState () override

  Get the current state of the state machine.
- FlightPhase getCurrentPhase () override

  Get the current high-level flight phase.
- void forceTransition (RocketState newState) override

  Force an immediate transition to the specified state.
- bool isFinished () override

  Check if the state machine has completed its mission.
- const char * **getStateString** (RocketState state) const

6.44.1 Member Function Documentation

6.44.1.1 forceTransition()

```
void RocketFSM::forceTransition (
            RocketState newState)  [override], [virtual]
```

Force an immediate transition to the specified state.
This bypasses normal transition rules and should be used with caution. Typical use cases:

- Emergency abort sequences
- Testing and debugging
- Manual override conditions
- Recovery from error states

Implementation should:

- Execute the transition immediately if possible
- Handle state cleanup (exit actions, task stopping)
- Start new state (entry actions, task starting)

- Log the forced transition for debugging
- Be thread-safe

Parameters

newState	The state to transition to immediately

Warning

Bypasses normal safety checks - use carefully

Note

Should be used sparingly, primarily for emergency/test scenarios

Implements IStateMachine.

6.44.1.2 getCurrentPhase()

`FlightPhase RocketFSM::getCurrentPhase () [override], [virtual]`

Get the current high-level flight phase.
Flight phases group related states for easier high-level logic:

- PRE_FLIGHT: INACTIVE, CALIBRATING, READY_FOR_LAUNCH
- FLIGHT: LAUNCH, ACCELERATED_FLIGHT, BALLISTIC_FLIGHT, APOGEE
- RECOVERY: STABILIZATION, DECELERATION, LANDING, RECOVERED

Implementation should map the current state to its corresponding phase.

Returns

Current FlightPhase based on current state

Note

Thread-safe, derived from getCurrentState()

Implements IStateMachine.

6.44.1.3 getCurrentState()

`RocketState RocketFSM::getCurrentState () [override], [virtual]`

Get the current state of the state machine.
Returns the current state atomically. Implementation should:

- Use appropriate locking to ensure atomic read
- Return immediately without blocking
- Be safe to call from any thread at any time

Returns

Current RocketState (INACTIVE, CALIBRATING, LAUNCH, etc.)

Note

Thread-safe getter

Should never block or fail

Implements IStateMachine.

6.44.1.4 init()

```
void RocketFSM::init ()  [override], [virtual]
```

Initialize the state machine and its components.
This method should:

- Set up internal data structures (event queues, mutexes, etc.)
- Initialize the state machine in the INACTIVE state
- Configure state actions and transition rules
- Prepare task management systems
- Allocate necessary memory resources

Should be called once before start() and must complete successfully before the state machine can be used.

Exceptions

May	throw/return error if initialization fails

Note

Must be called from main thread before starting tasks

Implements IStateMachine.

6.44.1.5 isFinished()

```
bool RocketFSM::isFinished ()  [override], [virtual]
```

Check if the state machine has completed its mission.
Returns true when the rocket has reached a terminal state (usually RECOVERED) and no further state transitions are expected. Implementation should:

- Return true for terminal states (RECOVERED, or error states)
- Return false for all active states
- Consider mission abort conditions

This can be used by higher-level systems to determine when mission operations are complete.

Returns

true if mission is complete, false if still active

Note

Useful for determining when to shut down systems

Implements IStateMachine.

6.44.1.6 sendEvent()

```
bool RocketFSM::sendEvent (
            FSMEvent event,
            RocketState targetState = RocketState::INACTIVE,
            void * eventData = nullptr)  [override], [virtual]
```

Send an event to trigger state transitions.
Events are the primary mechanism for triggering state changes. This method should:

- Queue the event for processing by the main FSM task

- Return immediately without blocking (asynchronous)
- Handle queue full conditions gracefully
- Support both targeted and general events

Event Processing:

- Events are processed by the main FSM task in FIFO order
- Each event is matched against current state transition rules
- Valid transitions execute immediately, invalid ones are ignored
- FORCE_TRANSITION events bypass normal transition rules

Parameters

event	The event type to send (START_CALIBRATION, LAUNCH_DETECTED, etc.)
targetState	For FORCE_TRANSITION events, the destination state
eventData	Optional data payload for the event (may be nullptr)

Returns

true if event was queued successfully, false if queue full or invalid

Note

Thread-safe, can be called from any task

Non-blocking operation

Implements IStateMachine.

6.44.1.7 start()

```
void RocketFSM::start () [override], [virtual]
```

Start the state machine and begin processing.
This method should:

- Create and start the main FSM task
- Begin event processing loop
- Start state-specific tasks based on current state
- Enable automatic transition checking
- Activate watchdog timers if used

The state machine will begin in INACTIVE state and should automatically transition to CALIBRATING if conditions are met.

Note

Should only be called after successful init()

May be called multiple times (should handle already-running case)

Implements IStateMachine.

6.44.1.8 stop()

`void RocketFSM::stop () [override], [virtual]`
Stop the state machine and clean up resources.
This method should:

- Signal all tasks to stop execution
- Wait for tasks to terminate gracefully
- Clean up FreeRTOS objects (tasks, queues, mutexes)
- Reset internal state to stopped condition
- Preserve current state for potential restart

Should be safe to call multiple times and should not block indefinitely. After calling stop(), the state machine can be restarted with start().

Note

Should handle case where FSM is already stopped

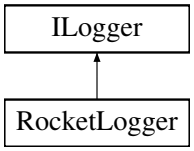
Should be safe to call from any thread

Implements IStateMachine.
The documentation for this class was generated from the following files:

- lib/control/src/RocketFSM.hpp
- lib/control/src/RocketFSM.cpp

6.45 RocketLogger Class Reference

Class to log messages and sensor data.
`#include <RocketLogger.hpp>`
Inheritance diagram for RocketLogger:



Public Member Functions

- ~**RocketLogger** ()
  Destructor to clean up dynamically allocated memory.
- void logInfo (const std::string &message) override
  Log an informational message.
- void logWarning (const std::string &message) override
  Log a warning message.
- void logError (const std::string &message) override
  Log an error message.
- void logSensorData (const SensorData sensorData) override
  Log sensor data.
- void logSensorData (const std::string &sensorName, const SensorData sensorData) override
  Log sensor data with a specific sensor name.
- json getJSONAll () const override
  Get all logged sensor data as a JSON list.
- void clearData () override
  Clear all logged sensor data.

Public Member Functions inherited from ILogger

- int getLogCount () const

  Get the number of logged entries.
- virtual ~**ILogger** ()=default

  Virtual destructor to ensure proper cleanup in derived classes.

Additional Inherited Members

Protected Attributes inherited from ILogger

- std::vector< LogData > **logDataList**

6.45.1 Detailed Description

Class to log messages and sensor data.

6.45.2 Member Function Documentation

6.45.2.1 clearData()

```
void RocketLogger::clearData ()  [override], [virtual]
```

Clear all logged sensor data.
Reimplemented from ILogger.

6.45.2.2 getJSONAll()

```
json RocketLogger::getJSONAll () const  [override], [virtual]
```

Get all logged sensor data as a JSON list.

Returns

A json object.

Implements ILogger.

6.45.2.3 logError()

```
void RocketLogger::logError (
            const std::string & message)  [override], [virtual]
```

Log an error message.

Parameters

message	The error message to log.

Implements ILogger.

6.45.2.4 logInfo()

```
void RocketLogger::logInfo (
            const std::string & message)  [override], [virtual]
```

Log an informational message.

Parameters

message	The informational message to log.

Implements ILogger.

6.45.2.5 logSensorData() [1/2]

```
void RocketLogger::logSensorData (
            const SensorData sensorData)  [override], [virtual]
```

Log sensor data.

Parameters

sensorData	The sensor data to log.

Implements ILogger.

6.45.2.6 logSensorData() [2/2]

```
void RocketLogger::logSensorData (
            const std::string & sensorName,
            const SensorData sensorData)  [override], [virtual]
```

Log sensor data with a specific sensor name.

Parameters

sensorName	The name of the sensor.
sensorData	The sensor data to log.

Implements ILogger.

6.45.2.7 logWarning()

```
void RocketLogger::logWarning (
            const std::string & message)  [override], [virtual]
```

Log a warning message.

Parameters

message	The warning message to log.

Implements ILogger.
The documentation for this class was generated from the following files:

- lib/rocket_logger/src/RocketLogger.hpp
- lib/rocket_logger/src/RocketLogger.cpp

6.46 SD Class Reference

Public Member Functions

- bool init ()

  A SD.begin() wrapper.
- bool openFile (std::string filename)

  A SdFile.open wrapper.
- bool closeFile ()

  A SDFile.close wrapper.
- bool writeFile (std::string filename, std::variant< std::string, String, char * > content)

  Writes content to a file.
- bool appendFile (std::string filename, std::variant< std::string, String, char * > content)

Appends content to a file.

- char * readFile (std::string filename)

  Reads the content of a file from start to end.

- bool clearSD ()

  Deletes all files from the SD card.

- bool fileExists (std::string filename)

  Checks if a file exists on the SD card.

- String readLine ()

  Reads a single line from the currently open file.

- SdFile * **getFile** ()

6.46.1 Member Function Documentation

6.46.1.1 appendFile()

```
bool SD::appendFile (
            std::string filename,
            std::variant< std::string, String, char * > content)
```

Appends content to a file.

Parameters

filename	the file to append to
content	the content to append

Returns

true if the content is appended, false otherwise

6.46.1.2 clearSD()

```
bool SD::clearSD ()
```

Deletes all files from the SD card.

Returns

true if the SD card is cleared, false otherwise.

6.46.1.3 closeFile()

```
bool SD::closeFile ()
```

A SDFile.close wrapper.

Returns

true if the file is closed, false otherwise.

6.46.1.4 fileExists()

```
bool SD::fileExists (
            std::string filename)
```

Checks if a file exists on the SD card.

Parameters

filename	the file to check for existence

Returns

true if the file exists, false otherwise

6.46.1.5 init()

```
bool SD::init ()
```

A SD.begin() wrapper.

Returns

true if the SD card is initialized, false otherwise

6.46.1.6 openFile()

```
bool SD::openFile (
            std::string filename)
```

A SdFile.open wrapper.

Parameters

filename	

Returns

true if the file is opened, false otherwise

6.46.1.7 readFile()

```
char * SD::readFile (
            std::string filename)
```

Reads the content of a file from start to end.

Note

You need to free the memory after using the content.

Returns

char* the content of the file.

6.46.1.8 readLine()

```
String SD::readLine ()
```

Reads a single line from the currently open file.

Returns

String containing the next line, or empty String if EOF or error

6.46.1.9 writeFile()

```
bool SD::writeFile (
            std::string filename,
            std::variant< std::string, String, char * > content)
```

Writes content to a file.

Parameters

filename	the file to write to
content	the content to write

Returns

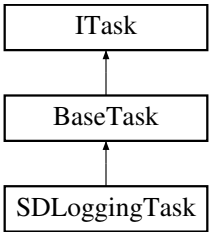
true if the file is written, false otherwise

The documentation for this class was generated from the following files:

- lib/SD/src/SD-master.hpp
- lib/SD/src/SD-master.cpp

6.47 SDLoggingTask Class Reference

Inheritance diagram for SDLoggingTask:



Public Member Functions

- **SDLoggingTask** (std::shared_ptr< RocketLogger > rocketLogger=nullptr, SemaphoreHandle_t loggerMutex=nullptr, std::shared_ptr< SD > sdCard=nullptr)

Public Member Functions inherited from BaseTask

- **BaseTask** (const char *name)
- bool start (const TaskConfig &config) override

  Start the task with the given configuration.
- void stop () override

  Stop the task.
- bool isRunning () const override

  Check if the task is currently running.
- const char * getName () const override

  Get the name of the task.
- uint32_t getStackHighWaterMark () const override

  Get the stack high water mark for the task.

Protected Member Functions

- void taskFunction () override

Protected Member Functions inherited from BaseTask

- virtual void **onTaskStart** ()
- virtual void **onTaskStop** ()

Additional Inherited Members

Protected Attributes inherited from BaseTask

- TaskHandle_t **taskHandle**
- TaskConfig **config**
- volatile bool **running**
- const char * **taskName**

6.47.1 Member Function Documentation

6.47.1.1 taskFunction()

```
void SDLoggingTask::taskFunction ()  [override], [protected], [virtual]
```

Implements BaseTask.

The documentation for this class was generated from the following files:

- lib/control/src/tasks/SDLoggingTask.hpp
- lib/control/src/tasks/SDLoggingTask.cpp

6.48 SensorData Class Reference

Class to store sensor data.

`#include <SensorData.hpp>`

Public Member Functions

- SensorData (const std::string &sensorName)

  Construct a new Sensor Data object.
- SensorData & operator= (const SensorData &other)

  Copy assignment operator.
- void setData (const std::string &key, const SensorDataVariant &value)

  Set the Data object.
- std::optional< SensorDataVariant > getData (const std::string &key) const

  Function to retrieve single data, now using std::optional to handle missing keys.
- std::map< std::string, SensorDataVariant > getDataMap () const

  Get the Data Map object.
- std::string getSensorName () const

  Get the Sensor Name object.

Protected Attributes

- const std::string **sensorName**
- std::map< std::string, SensorDataVariant > **dataMap**

6.48.1 Detailed Description

Class to store sensor data.

6.48.2 Constructor & Destructor Documentation

6.48.2.1 SensorData()

```
SensorData::SensorData (
            const std::string & sensorName)  [inline]
```

Construct a new Sensor Data object.

Parameters

sensorName	Name of the sensor.

6.48.3 Member Function Documentation

6.48.3.1 getData()

```
std::optional< SensorDataVariant > SensorData::getData (
            const std::string & key) const  [inline]
```

Function to retrieve single data, now using std::optional to handle missing keys.

Parameters

key	The key to retrieve the data.

Returns

The data if the key is found, otherwise std::nullopt.

6.48.3.2 getDataMap()

```
std::map< std::string, SensorDataVariant > SensorData::getDataMap () const  [inline]
```

Get the Data Map object.

Returns

The data map.

6.48.3.3 getSensorName()

```
std::string SensorData::getSensorName () const  [inline]
```

Get the Sensor Name object.

Returns

A string representing the name of the sensor.

6.48.3.4 operator=()

```
SensorData & SensorData::operator= (
            const SensorData & other)  [inline]
```

Copy assignment operator.

Parameters

other	The SensorData object to copy from.

Returns

Reference to this object.

6.48.3.5 setData()

```
void SensorData::setData (
            const std::string & key,
            const SensorDataVariant & value)  [inline]
```

Set the Data object.

Parameters

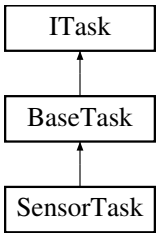
key	The key to store the data.
value	The value to store.

The documentation for this class was generated from the following file:

- lib/sensorCommon/src/SensorData.hpp

6.49 SensorTask Class Reference

Inheritance diagram for SensorTask:



Public Member Functions

- **SensorTask** (std::shared_ptr< SharedSensorData > sensorData, SemaphoreHandle_t mutex, std::shared_ptr< ISensor > imu=nullptr, std::shared_ptr< ISensor > barometer1=nullptr, std::shared_ptr< ISensor > barometer2=nullptr, std::shared_ptr< RocketLogger > rocketLogger=nullptr, SemaphoreHandle_t loggerMutex=nullptr)
- void **setSensors** (std::shared_ptr< ISensor > imu, std::shared_ptr< ISensor > barometer1, std::shared_ptr< ISensor > barometer2)

Public Member Functions inherited from BaseTask

- **BaseTask** (const char *name)
- bool start (const TaskConfig &config) override

  Start the task with the given configuration.
- void stop () override

  Stop the task.
- bool isRunning () const override

  Check if the task is currently running.
- const char * getName () const override

  Get the name of the task.
- uint32_t getStackHighWaterMark () const override

  Get the stack high water mark for the task.

Protected Member Functions

- void taskFunction () override
- void onTaskStart () override
- void onTaskStop () override

Additional Inherited Members

Protected Attributes inherited from BaseTask

- TaskHandle_t **taskHandle**
- TaskConfig **config**
- volatile bool **running**
- const char * **taskName**

6.49.1 Member Function Documentation

6.49.1.1 onTaskStart()

```
void SensorTask::onTaskStart () [override], [protected], [virtual]
```

Reimplemented from BaseTask.

6.49.1.2 onTaskStop()

```
void SensorTask::onTaskStop () [override], [protected], [virtual]
```

Reimplemented from BaseTask.

6.49.1.3 taskFunction()

```
void SensorTask::taskFunction () [override], [protected], [virtual]
```

Implements BaseTask.

The documentation for this class was generated from the following files:

- lib/control/src/tasks/SensorTask.hpp
- lib/control/src/tasks/SensorTask.cpp

6.50 SharedFilteredData Struct Reference

Public Attributes

- float **altitude**
- float **verticalVelocity**
- float **orientation** [4]
- uint32_t **timestamp**
- bool **dataValid**

The documentation for this struct was generated from the following file:

- lib/control/src/SharedData.hpp

6.51 SharedSensorData Struct Reference

Public Attributes

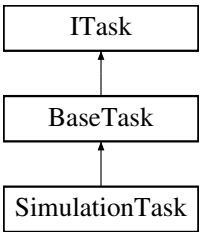
- class SensorData **imuData**
- class SensorData **baroData1**
- class SensorData **baroData2**
- class SensorData **gpsData**
- uint32_t **timestamp**
- bool **dataValid**

The documentation for this struct was generated from the following file:

- lib/control/src/SharedData.hpp

6.52 SimulationTask Class Reference

Inheritance diagram for SimulationTask:



Public Member Functions

- **SimulationTask** (const std::string &csvFilePath, std::shared_ptr< SharedSensorData > sensorData, SemaphoreHandle_t mutex, std::shared_ptr< RocketLogger > rocketLogger, SemaphoreHandle_t loggerMutex)
- void onTaskStart () override
- void onTaskStop () override
- void taskFunction () override
- void **reset** ()

Public Member Functions inherited from BaseTask

- **BaseTask** (const char *name)
- bool start (const TaskConfig &config) override

  Start the task with the given configuration.
- void stop () override

  Stop the task.
- bool isRunning () const override

  Check if the task is currently running.
- const char * getName () const override

  Get the name of the task.
- uint32_t getStackHighWaterMark () const override

  Get the stack high water mark for the task.

Additional Inherited Members

Protected Attributes inherited from BaseTask

- TaskHandle_t **taskHandle**
- TaskConfig **config**
- volatile bool **running**
- const char * **taskName**

6.52.1 Member Function Documentation

6.52.1.1 onTaskStart()

```
void SimulationTask::onTaskStart ()  [override], [virtual]
```

Reimplemented from BaseTask.

6.52.1.2 onTaskStop()

```
void SimulationTask::onTaskStop ()  [override], [virtual]
```

Reimplemented from BaseTask.

6.52.1.3 taskFunction()

```
void SimulationTask::taskFunction ()  [override], [virtual]
```

Implements BaseTask.

The documentation for this class was generated from the following files:

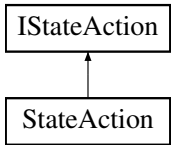
- lib/control/src/tasks/SimulationTask.hpp
- lib/control/src/tasks/SimulationTask.cpp

6.53 StateAction Class Reference

Concrete implementation of IStateAction that manages rocket state actions and tasks.

`#include <StateAction.hpp>`

Inheritance diagram for StateAction:



Public Member Functions

- **StateAction** (RocketState s)
- StateAction & setEntryAction (const std::function< void()> &action)

  Set the entry action callback.
- StateAction & setExitAction (const std::function< void()> &action)

  Set the exit action callback.
- StateAction & addTask (const TaskConfig &config)

  Add a task configuration associated with this state.
- void onEntry () override

  Called when entering this state.
- void onExit () override

  Called when exiting this state.
- RocketState getState () const override

  Get the state this action handles.
- const std::vector< TaskConfig > & getTaskConfigs () const

  Get the configured task configurations.

Public Member Functions inherited from IStateAction

- virtual void onUpdate ()

  Called periodically while in this state.

6.53.1 Detailed Description

Concrete implementation of IStateAction that manages rocket state actions and tasks.

The StateAction class provides a fluent interface for configuring state-specific behaviors including entry/exit actions and associated tasks. It supports method chaining for convenient configuration.

This class encapsulates:

- A specific rocket state
- Optional entry and exit actions (callbacks)
- A collection of task configurations associated with the state

The class uses std::function for flexible callback assignment and provides const-correct access to internal data.

6.53.2 Member Function Documentation

6.53.2.1 addTask()

```
StateAction & StateAction::addTask (
            const TaskConfig & config)  [inline]
```

Add a task configuration associated with this state.

Parameters

config	The TaskConfig to add

Returns

Reference to this StateAction for method chaining

6.53.2.2 getState()

```
RocketState StateAction::getState () const  [inline], [override], [virtual]
```

Get the state this action handles.

Returns

The RocketState this action is associated with

Implements IStateAction.

6.53.2.3 getTaskConfigs()

```
const std::vector< TaskConfig > & StateAction::getTaskConfigs () const  [inline]
```

Get the configured task configurations.

Returns

A const reference to the vector of TaskConfig

6.53.2.4 onEntry()

```
void StateAction::onEntry ()  [inline], [override], [virtual]
```

Called when entering this state.
Reimplemented from IStateAction.

6.53.2.5 onExit()

```
void StateAction::onExit ()  [inline], [override], [virtual]
```

Called when exiting this state.
Reimplemented from IStateAction.

6.53.2.6 setEntryAction()

```
StateAction & StateAction::setEntryAction (
            const std::function< void()> & action)  [inline]
```

Set the entry action callback.

Parameters

action	The function to call on state entry

Returns

Reference to this StateAction for method chaining

6.53.2.7 setExitAction()

```
StateAction & StateAction::setExitAction (
            const std::function< void()> & action)  [inline]
```

Set the exit action callback.

Parameters

action	The function to call on state exit

Returns

Reference to this StateAction for method chaining

The documentation for this class was generated from the following file:

- lib/control/src/states/StateAction.hpp

6.54 StateTransition Struct Reference

Represents a simple state transition rule (legacy).
`#include <FlightState.hpp>`

Public Member Functions

- **StateTransition** (RocketState from, RocketState to, FSMEvent event)

Public Attributes

- RocketState **fromState**
- RocketState **toState**
- FSMEvent **triggerEvent**

6.54.1 Detailed Description

Represents a simple state transition rule (legacy).

Deprecated Use TransitionManager and Transition struct instead

See also

TransitionManager

Transition

The documentation for this struct was generated from the following file:

- lib/control/src/FlightState.hpp

6.55 StatusManager Class Reference

Public Member Functions

- **StatusManager** (LEDController &ledController, BuzzerController &buzzerController)
- void **init** ()

- void **setSystemCode** (SystemCode code)
- SystemCode **getCurrentCode** () const
- const StatusPattern & **getPattern** (SystemCode code) const
- uint32_t **getPatternDuration** (SystemCode code) const
- void **stopAllPatterns** ()
- void **playBlockingPattern** (SystemCode code, uint32_t minDuration=0)

The documentation for this class was generated from the following files:

- lib/StatusManager/src/StatusManager.hpp
- lib/StatusManager/src/StatusManager.cpp

6.56 StatusPattern Struct Reference

Public Attributes

- LEDPattern **ledPattern**
- BuzzerPattern **buzzerPattern**
- BuzzerTone **buzzerTone**
- bool **buzzerSync**

The documentation for this struct was generated from the following file:

- lib/StatusManager/src/StatusManager.hpp

6.57 TaskConfig Struct Reference

Public Member Functions

- TaskConfig (TaskType t=TaskType::SENSOR, const char *n="Task", uint32_t stack=2048, TaskPriority prio=TaskPriority::TASK_MEDIUM, TaskCore core=TaskCore::ANY_CORE, bool run=true)

  Construct a new Task Config object.

Public Attributes

- TaskType **type**
- const char * **name**
- uint32_t **stackSize**
- TaskPriority **priority**
- TaskCore **coreId**
- bool **shouldRun**

6.57.1 Constructor & Destructor Documentation

6.57.1.1 TaskConfig()

```
TaskConfig::TaskConfig (
            TaskType t = TaskType::SENSOR,
            const char * n = "Task",
            uint32_t stack = 2048,
            TaskPriority prio = TaskPriority::TASK_MEDIUM,
            TaskCore core = TaskCore::ANY_CORE,
            bool run = true)  [inline]
```

Construct a new Task Config object.

Parameters

t	TaskType: The type of task

See also

enum class TaskType

Parameters

n	const char∗: The name of the task
stack	uint32_t: The stack size for the task
prio	TaskPriority: The priority of the task

See also

enum class TaskPriority

Parameters

core	TaskCore: The core ID for the task

See also

enum class TaskCore

Parameters

run	bool: Flag indicating if the task should run

The documentation for this struct was generated from the following file:

- lib/control/src/tasks/TaskConfig.hpp

6.58 TaskManager Class Reference

Public Member Functions

- **TaskManager** (std::shared_ptr< SharedSensorData > sensorData, std::shared_ptr< ISensor > imu, std::shared_ptr< ISensor > barometer1, std::shared_ptr< ISensor > barometer2, std::shared_ptr< ISensor > gps, SemaphoreHandle_t sensorMutex, std::shared_ptr< SD > sd, std::shared_ptr< RocketLogger > rocketLogger, SemaphoreHandle_t loggerMutex, std::shared_ptr< bool > isRising, std::shared_ptr< float > heightGainSpeed, std::shared_ptr< float > currentHeight)
- void **initializeTasks** ()
- bool **startTask** (TaskType type, const TaskConfig &config)
- void **stopTask** (TaskType type)
- void **stopAllTasks** ()
- int **getRunningTaskCount** ()
- bool **isTaskRunning** (TaskType type) const
- uint32_t **getTaskStackUsage** (TaskType type) const
- void **printTaskStatus** () const

The documentation for this class was generated from the following files:

- lib/control/src/tasks/TaskManager.hpp
- lib/control/src/tasks/TaskManager.cpp

6.59 TelemetryPacket Struct Reference

Binary telemetry packet structure for efficient transmission.
`#include <TelemetryTask.hpp>`

Public Attributes

- uint32_t **timestamp**

  Milliseconds since boot.
- bool **dataValid**

  True if sensor data was successfully collected.
- struct {
  float **accel_x**
  float **accel_y**
  float **accel_z**
  Accelerometer (m/s^2).
  float **gyro_x**
  float **gyro_y**
  float **gyro_z**
  Gyroscope (rad/s).
  } **imu**

- struct {
  float **pressure**
  Pressure (hPa).
  float **temperature**
  Temperature (°C).
  } **baro1**

- struct {
  float **pressure**
  Pressure (hPa).
  float **temperature**
  Temperature (°C).
  } **baro2**

- struct {
  float **latitude**
  Latitude (degrees).
  float **longitude**
  Longitude (degrees).
  float **altitude**
  GPS altitude (meters).
  } **gps**

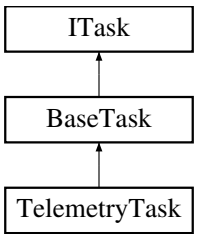
6.59.1 Detailed Description

Binary telemetry packet structure for efficient transmission.
This structure is tightly packed (no padding) for efficient transmission over ESP-NOW. Total size is approximately 64 bytes.
All multi-byte values are little-endian (ESP32 native).
The documentation for this struct was generated from the following file:

- lib/control/src/tasks/TelemetryTask.hpp

6.60 TelemetryTask Class Reference

Task that periodically collects sensor data and transmits it via ESP-NOW.
`#include <TelemetryTask.hpp>`
Inheritance diagram for TelemetryTask:



Public Member Functions

- TelemetryTask (std::shared_ptr< SharedSensorData > sensorData, SemaphoreHandle_t mutex, std::shared_ptr< EspNowTransmitter > espNowTransmitter, uint32_t intervalMs=1000)

  Construct a new Telemetry Task.
- void getStats (uint32_t &messages, uint32_t &packets, uint32_t &errors) const

  Get transmission statistics.

Public Member Functions inherited from BaseTask

- **BaseTask** (const char *name)
- bool start (const TaskConfig &config) override

  Start the task with the given configuration.
- void stop () override

  Stop the task.
- bool isRunning () const override

  Check if the task is currently running.
- const char * getName () const override

  Get the name of the task.
- uint32_t getStackHighWaterMark () const override

  Get the stack high water mark for the task.

Protected Member Functions

- void taskFunction () override
- void onTaskStart () override
- void onTaskStop () override

Additional Inherited Members

Protected Attributes inherited from BaseTask

- TaskHandle_t **taskHandle**
- TaskConfig **config**
- volatile bool **running**
- const char * **taskName**

6.60.1 Detailed Description

Task that periodically collects sensor data and transmits it via ESP-NOW.
This task reads from SharedSensorData, serializes it to binary format, divides it into Packet chunks, and transmits them using EspNowTransmitter.
Transmission rate is configurable via constructor.

6.60.2 Constructor & Destructor Documentation

6.60.2.1 TelemetryTask()

```
TelemetryTask::TelemetryTask (
            std::shared_ptr< SharedSensorData > sensorData,
            SemaphoreHandle_t mutex,
            std::shared_ptr< EspNowTransmitter > espNowTransmitter,
            uint32_t intervalMs = 1000)
```

Construct a new Telemetry Task.

Parameters

sensorData	Shared sensor data to read from.
mutex	Mutex protecting sensor data access.
espNowTransmitter	ESP-NOW transmitter instance.
intervalMs	Interval between transmissions in milliseconds (default 1000ms = 1Hz).

6.60.3 Member Function Documentation

6.60.3.1 getStats()

```
void TelemetryTask::getStats (
            uint32_t & messages,
            uint32_t & packets,
            uint32_t & errors) const
```

Get transmission statistics.

Parameters

messages	Output: number of messages created.
packets	Output: number of packets sent.
errors	Output: number of transmission errors.

6.60.3.2 onTaskStart()

```
void TelemetryTask::onTaskStart ()  [override], [protected], [virtual]
```

Reimplemented from BaseTask.

6.60.3.3 onTaskStop()

```
void TelemetryTask::onTaskStop ()  [override], [protected], [virtual]
```

Reimplemented from BaseTask.

6.60.3.4 taskFunction()

```
void TelemetryTask::taskFunction ()  [override], [protected], [virtual]
```

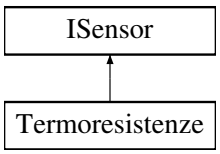
Implements BaseTask.

The documentation for this class was generated from the following files:

- lib/control/src/tasks/TelemetryTask.hpp
- lib/control/src/tasks/TelemetryTask.cpp

6.61 Termoresistenze Class Reference

Inheritance diagram for Termoresistenze:



Public Member Functions

- **Termoresistenze** (int pin=THERMISTOR_PIN, double seriesResistor=SERIES_RESISTOR, double nominalResistance=NOMINAL_RESISTANCE, double nominalTemperature=NOMINAL_TEMPERATURE, double bCoefficient=B_COEFFICIENT)
- bool init () override

  Initialize the sensor.
- std::optional< SensorData > getData () override

  Read and then get the sensor data.

Public Member Functions inherited from ISensor

- bool isInitialized () const

  Check if the sensor is initialized.

Additional Inherited Members

Protected Member Functions inherited from ISensor

- void **setInitialized** (bool initialized)

6.61.1 Member Function Documentation

6.61.1.1 getData()

```
std::optional< SensorData > Termoresistenze::getData ()  [override], [virtual]
```

Read and then get the sensor data.

Returns

Just read data.

Implements ISensor.

6.61.1.2 init()

```
bool Termoresistenze::init ()  [override], [virtual]
```

Initialize the sensor.

Returns

true if the sensor was initialized successfully

false if the sensor failed to initialize

Implements ISensor.
The documentation for this class was generated from the following files:

- lib/Termoresistenze/Termoresistenze.hpp
- lib/Termoresistenze/Termoresistenze.cpp

6.62 Transition Struct Reference

Represents a state transition rule in the finite state machine.
`#include <TransitionManager.hpp>`

Public Member Functions

- Transition (RocketState from, RocketState to, FSMEvent event, std::shared_ptr< ITransitionCondition > cond=nullptr)

  Constructor for Transition.

Public Attributes

- RocketState **fromState**
- RocketState **toState**
- FSMEvent **triggerEvent**
- std::shared_ptr< ITransitionCondition > **condition**

6.62.1 Detailed Description

Represents a state transition rule in the finite state machine.
A Transition defines how the state machine can move from one state to another. It encapsulates the source state, destination state, triggering event, and optional conditions that must be met for the transition to occur.
Transition Types:

- Event-driven: Triggered by explicit FSMEvent (e.g., LAUNCH_DETECTED)
- Automatic: Triggered by conditions (e.g., timeout, sensor thresholds)
- Hybrid: Event-driven with additional condition checks

6.62.2 Constructor & Destructor Documentation

6.62.2.1 Transition()

```
Transition::Transition (
            RocketState from,
            RocketState to,
            FSMEvent event,
            std::shared_ptr< ITransitionCondition > cond = nullptr)  [inline]
```

Constructor for Transition.

Parameters

from	Source state
to	Destination state
event	Triggering event
cond	Optional condition (nullptr if none)

The documentation for this struct was generated from the following file:

- lib/control/src/states/TransitionManager.hpp

6.63 TransitionManager Class Reference

Manages state transitions and transition rules for the rocket state machine.
`#include <TransitionManager.hpp>`

Public Member Functions

- void addTransition (const Transition &transition)

  Adds a new transition to the manager Transitions are stored in the order they are added. When multiple transitions could apply, the first matching transition in the list will be used.
- std::optional< RocketState > findTransition (RocketState currentState, FSMEvent event)

  Find a valid transition based on current state and event.
- bool checkAutomaticTransitions (RocketState currentState, unsigned long stateStartTime)

  Checks for automatic transitions based on elapsed time.

6.63.1 Detailed Description

Manages state transitions and transition rules for the rocket state machine.
The TransitionManager is responsible for:

- Storing and organizing all possible state transitions
- Finding valid transitions based on current state and events
- Evaluating automatic transition conditions
- Providing a centralized location for transition logic

This class separates transition logic from the main state machine implementation, making it easier to modify transition rules without affecting other components.

Note

Transitions are evaluated in the order they were added

See also

Transition for individual transition rules

ITransitionCondition for automatic transition conditions

6.63.2 Member Function Documentation

6.63.2.1 addTransition()

```
void TransitionManager::addTransition (
            const Transition & transition)
```

Adds a new transition to the manager Transitions are stored in the order they are added. When multiple transitions could apply, the first matching transition in the list will be used.

Parameters

transition	The Transition object to add

Note

Duplicated transitions are allowed but may lead to ambiguous and unexpected behavior

6.63.2.2 checkAutomaticTransitions()

```
bool TransitionManager::checkAutomaticTransitions (
            RocketState currentState,
            unsigned long stateStartTime)
```

Checks for automatic transitions based on elapsed time.
Evaluates all transitions from the current state that have conditions defined, regardless of their trigger event. This enables automatic transitions based on time, sensor data, or other criteria.
Typical automatic conditions are:
Timeouts, Sensor thresholds (e.g. velocity or altitude for apogee detection), System status (e.g. battery level, calibration status)

Parameters

currentState	the current state of the FSM
stateStartTime	the time when the current state was entered

Returns

true if an automatic condition was met and should be processed, false otherwise.

6.63.2.3 findTransition()

```
std::optional< RocketState > TransitionManager::findTransition (
            RocketState currentState,
            FSMEvent event)
```

Find a valid transition based on current state and event.
Searched through all defined transitions to find one that matches:
Current state == fromState; Event == triggerEvent; Condition (if any) is met

Parameters

currentState	the current state of the FSM
event	the event that was received

Returns

std::optional<RocketState> The next state if a valid transition is found, otherwise std::nullopt

Note

If multiple transitions match, the first one found is returned

The documentation for this class was generated from the following files:

- lib/control/src/states/TransitionManager.hpp
- lib/control/src/states/TransitionManager.cpp

Chapter 7

File Documentation

7.1 BME680Sensor.hpp

```
00001 #pragma once
00019 #include <Adafruit_BME680.h>
00020 #include <ISensor.hpp>
00021 #include <pins.h>
00022 #include <config.h>
00023
00024 class BME680Sensor : public ISensor
00025 {
00026 public:
00027     BME680Sensor(uint8_t addr);
00028     bool init() override;
00029     std::optional<SensorData> getData() override;
00030
00031 private:
00032     Adafruit_BME680 bme;
00033     uint8_t addr;
00034 };
```

7.2 BNO055Sensor.hpp

```
00001 #pragma once
00002
00003 #include <BNO055SensorInterface.hpp>
00004 #include <ISensor.hpp>
00005
00015 class BNO055Sensor : public ISensor
00016 {
00017 public:
00019     BNO055Sensor();
00020
00025     bool init() override;
00026
00031     bool calibrate();
00032
00037     bool hardwareTest();
00038
00043     std::optional<SensorData> getData() override;
00044
00050     struct CalibrationStatus {
00051         uint8_t sys;
00052         uint8_t gyro;
00053         uint8_t accel;
00054         uint8_t mag;
00055     };
00056
00061     CalibrationStatus getCalibration();
00062
00063 private:
00065     BNO055SensorInterface bno_interface;
00066 };
00067
```

7.3 BNO055SensorInterface.hpp

```
00001 #ifndef BNO055_SENSOR_INTERFACE_HPP
```

```
00002 #define BNO055_SENSOR_INTERFACE_HPP
00003
00004 extern "C" {
00005   #include "bno055.h"
00006 }
00007 #include <config.h>
00008 #include <Wire.h>
00009 #include <Arduino.h>
00010
00011 // I2C communication functions are now private static members of BNO055SensorInterface
00012
00013 // !!! Errors given by the BNO APIs should be managed, maybe with a Publisher-Subscriber pattern?
00014
00018
00019 class BNO055SensorInterface
00020 {
00021 private:
00022     bno055_t bno;
00023
00024     // Private static I2C bus functions
00025     static s8 BNO055_I2C_bus_write(u8 dev_addr, u8 reg_addr, u8 *reg_data, u8 cnt);
00026     static s8 BNO055_I2C_bus_read(u8 dev_addr, u8 reg_addr, u8 *reg_data, u8 cnt);
00027     static void BNO055_delay_msek(u32 msek);
00028
00029 public:
00033     BNO055SensorInterface();
00034
00039     bool init();
00040
00045     uint8_t check_calibration();
00046
00047     // ========== INDIVIDUAL CALIBRATION FUNCTIONS ==========
00052     uint8_t check_calibration_accel();
00053
00059     uint8_t check_calibration_mag();
00060
00066     uint8_t check_calibration_gyro();
00067
00072     uint8_t check_calibration_sys();
00073
00074     // ========== SELF-TEST FUNCTIONS ==========
00079     bool selftest_accel();
00080
00085     bool selftest_mag();
00086
00091     bool selftest_gyro();
00092
00101     bool selftest_mcu();
00102
00103     // ========== SYSTEM STATUS AND ERROR CHECKING ==========
00116     bool check_system_status();
00117
00122     bool check_system_error();
00123
00128     bool check_clock_status();
00129
00134     uint8_t get_system_error_code();
00135
00144     uint8_t get_system_status_code();
00145
00146     // ========== OPERATION AND POWER MODE MANAGEMENT ==========
00166     bool set_operation_mode(uint8_t mode);
00167
00173     bool get_operation_mode(uint8_t* mode);
00174
00184     bool set_power_mode(uint8_t mode);
00185
00191     bool get_power_mode(uint8_t* mode);
00192
00198     bool set_accel_power_mode(uint8_t mode);
00199
00205     bool set_mag_power_mode(uint8_t mode);
00206
00217     bool set_gyro_power_mode(uint8_t mode);
00218
00219     // ========== SENSOR DATA READING FUNCTIONS ==========
00220
00225     std::vector<float> get_accel();
00226
00231     std::vector<float> get_mag();
00232
00237     std::vector<float> get_gyro_dps();
00238
00243     std::vector<float> get_gyro_rps();
00244
00249     std::vector<float> get_euler_deg();
00250
```

```
00255     std::vector<float> get_euler_rad();
00256
00261     std::vector<float> get_quaternion();
00262
00267     std::vector<float> get_linear_accel();
00268
00273     std::vector<float> get_gravity();
00274
00279     float get_temperature();
00280 };
00281 #endif
```

7.4 FlightState.hpp

```
00001 #pragma once
00002
00003 #include <functional>
00004 #include <freertos/FreeRTOS.h>
00005 #include <freertos/task.h>
00006 #include <freertos/queue.h>
00007 #include <freertos/semphr.h>
00008
00016 enum class RocketState
00017 {
00018     // Initial state
00019     INACTIVE, // System startup, no active operations
00020
00021     // Pre-flight phase
00022     CALIBRATING,      // Sensor calibration and system checks
00023     READY_FOR_LAUNCH, // Armed and waiting for launch detection
00024
00025     // Flight phase
00026     LAUNCH,             // Launch sequence initiated
00027     ACCELERATED_FLIGHT, // Active propulsion phase
00028     BALLISTIC_FLIGHT,   // Coasting phase after motor burnout
00029     APOGEE,             // Highest point reached
00030
00031     // Recovery phase
00032     STABILIZATION, // Drogue chute deployment and stabilization
00033     DECELERATION,  // Main chute deployment phase
00034     LANDING,       // Final descent and landing
00035     RECOVERED      // Mission complete, rocket recovered
00036 };
00037
00044 enum class FlightPhase
00045 {
00046     PRE_FLIGHT, // INACTIVE, CALIBRATING, READY_FOR_LAUNCH
00047     FLIGHT,     // LAUNCH, ACCELERATED_FLIGHT, BALLISTIC_FLIGHT, APOGEE
00048     RECOVERY    // STABILIZATION, DECELERATION, LANDING, RECOVERED
00049 };
00050
00058 enum class FSMEvent
00059 {
00060     NONE = 0, // No event (default/invalid)
00061
00062     // Pre-flight events
00063     START_CALIBRATION,    // Begin sensor calibration process
00064     CALIBRATION_COMPLETE, // Calibration finished successfully
00065
00066     // Launch events
00067     LAUNCH_DETECTED, // Launch sequence initiated
00068     LIFTOFF_STARTED, // Physical liftoff detected
00069
00070     // Flight events
00071     ACCELERATION_COMPLETE, // Motor burnout detected
00072     APOGEE_REACHED,        // Maximum altitude reached
00073
00074     // Recovery events
00075     DROGUE_READY,           // Drogue chute ready for deployment
00076     STABILIZATION_COMPLETE, // Vehicle stabilized after drogue deployment
00077     DECELERATION_COMPLETE,  // Main chute phase complete
00078     LANDING_COMPLETE,       // Touchdown detected
00079
00080     // System events
00081     FORCE_TRANSITION, // Manual override transition
00082     EMERGENCY_ABORT   // Emergency abort sequence
00083 };
00084
00091 struct FSMEventData
00092 {
00093     FSMEvent event;          // The event type
00094     RocketState targetState; // Target state for FORCE_TRANSITION events
00095     void *eventData;         // Optional event payload data
00096
```

```
00104     FSMEventData(FSMEvent e, RocketState target = RocketState::INACTIVE, void *data = nullptr)
00105         : event(e), targetState(target), eventData(data) {}
00106 };
00107
00115 struct StateTransition
00116 {
00117     RocketState fromState; // Source state
00118     RocketState toState;   // Destination state
00119     FSMEvent triggerEvent; // Triggering event
00120
00121     StateTransition(RocketState from, RocketState to, FSMEvent event)
00122         : fromState(from), toState(to), triggerEvent(event) {}
00123 };
00124
00125 // Forward declaration of TaskConfig (defined in ITask.hpp)
00126 struct TaskConfig;
```

7.5 IStateMachine.hpp

```
00001 #pragma once
00002
00003 #include "FlightState.hpp"
00004 #include <functional>
00005
00029 class IStateMachine
00030 {
00031 public:
00032     virtual ~IStateMachine() = default;
00033
00050     virtual void init() = 0;
00051
00068     virtual void start() = 0;
00069
00086     virtual void stop() = 0;
00087
00112     virtual bool sendEvent(FSMEvent event, RocketState targetState = RocketState::INACTIVE, void
      *eventData = nullptr) = 0;
00113
00126     virtual RocketState getCurrentState() = 0;
00127
00141     virtual FlightPhase getCurrentPhase() = 0;
00142
00164     virtual void forceTransition(RocketState newState) = 0;
00165
00181     virtual bool isFinished() = 0;
00182 };
```

7.6 Logger.hpp

```
00001 #pragma once
00002
00003 #include <Arduino.h>
00004 #include <freertos/FreeRTOS.h>
00005 #include <freertos/semphr.h>
00006
00007 namespace Logger
00008 {
00009     // Log levels for structured logging
00010     enum class LogLevel
00011     {
00012         ERROR,
00013         WARNING,
00014         INFO,
00015         DEBUG,
00016         TRACE
00017     };
00018
00020     void init();
00021
00027     void log(LogLevel level, const char *tag, const char *format, ...);
00028
00031     void debugMemory(const char *location);
00032
00035     SemaphoreHandle_t getSerialMutex();
00036 }
00037
00038 #define LOG_ERROR(tag, format, ...) Logger::log(Logger::LogLevel::ERROR, tag, format, ##__VA_ARGS__)
00039 #define LOG_WARNING(tag, format, ...) Logger::log(Logger::LogLevel::WARNING, tag, format,
      ##__VA_ARGS__)
00040 #define LOG_INFO(tag, format, ...) Logger::log(Logger::LogLevel::INFO, tag, format, ##__VA_ARGS__)
00041 #define LOG_DEBUG(tag, format, ...) Logger::log(Logger::LogLevel::DEBUG, tag, format, ##__VA_ARGS__)
```

```
00042 #define LOG_TRACE(tag, format, ...) Logger::log(Logger::LogLevel::TRACE, tag, format, ##__VA_ARGS__)
```

7.7 RocketFSM.hpp

```
00001 #pragma once
00002
00003 #include "IStateMachine.hpp"
00004 #include "tasks/TaskManager.hpp"
00005 #include "states/StateAction.hpp"
00006 #include "states/TransitionManager.hpp"
00007 #include <SharedData.hpp>
00008 #include <Logger.hpp>
00009 #include "RocketLogger.hpp"
00010 #include "config.h"
00011 #include <SD-master.hpp>
00012 #include <memory>
00013 #include <map>
00014
00015 class RocketFSM : public IStateMachine
00016 {
00017 private:
00018     // Core FSM components
00019     std::unique_ptr<TaskManager> taskManager;
00020     std::unique_ptr<TransitionManager> transitionManager;
00021     std::map<RocketState, std::unique_ptr<StateAction» stateActions;
00022
00023     // FreeRTOS components
00024     TaskHandle_t fsmTaskHandle;
00025     QueueHandle_t eventQueue;
00026     SemaphoreHandle_t stateMutex;
00027
00028     // State management
00029     RocketState currentState;
00030     RocketState previousState;
00031     unsigned long stateStartTime;
00032     volatile bool isRunning;
00033     volatile bool isTransitioning;
00034
00035     // Shared data
00036     std::shared_ptr<SharedSensorData> sharedData;
00037     std::shared_ptr<RocketLogger> logger;
00038     SemaphoreHandle_t sensorDataMutex;
00039     SemaphoreHandle_t loggerMutex;
00040     std::shared_ptr<ISensor> bno055;
00041     std::shared_ptr<ISensor> baro1;
00042     std::shared_ptr<ISensor> baro2;
00043     std::shared_ptr<ISensor> accl;
00044     std::shared_ptr<ISensor> gps;
00045     // Rising Flag
00046     std::shared_ptr<bool> isRising=std::make_shared<bool>(true);
00047     std::shared_ptr<float> heightGainSpeed=std::make_shared<float>(0.0f);
00048     std::shared_ptr<float> currentHeight=std::make_shared<float>(0.0f);
00049
00050     std::shared_ptr<SD> sd;
00051
00052     // Important timers and tresholds
00053     const unsigned long LAUNCH_TO_BALLISTIC_THRESHOLD = 7000; //5300;
00054     const unsigned long LAUNCH_TO_APOGEE_THRESHOLD = 27000; //24850 + 500 = 25350
00055     unsigned long launchDetectionTime = 0;
00056
00057
00058 public:
00059     RocketFSM(std::shared_ptr<ISensor> imu,
00060               std::shared_ptr<ISensor> barometer1,
00061               std::shared_ptr<ISensor> barometer2,
00062               std::shared_ptr<ISensor> accelerometer,
00063               std::shared_ptr<ISensor> gpsModule,
00064               std::shared_ptr<SD> sd,
00065               std::shared_ptr<RocketLogger> logger
00066             );
00067     ~RocketFSM();
00068
00069     // IStateMachine interface
00070     void init() override;
00071     void start() override;
00072     void stop() override;
00073     bool sendEvent(FSMEvent event, RocketState targetState = RocketState::INACTIVE, void *eventData =
      nullptr) override;
00074     RocketState getCurrentState() override;
00075     FlightPhase getCurrentPhase() override;
00076     void forceTransition(RocketState newState) override;
00077     bool isFinished() override;
00078
00079     // Utility methods
```

```
00080     const char* getStateString(RocketState state) const;
00081
00082 private:
00083     void setupStateActions();
00084     void setupTransitions();
00085     void transitionTo(RocketState newState);
00086     void processEvent(const FSMEventData &eventData);
00087     void checkTransitions();
00088
00089     // FreeRTOS task function
00090     static void fsmTaskWrapper(void *parameter);
00091     void fsmTask();
00092 };
```

7.8 SharedData.hpp

```
00001 #pragma once
00002
00003 #include <SensorData.hpp>
00004 #include <freertos/FreeRTOS.h>
00005
00006 struct SharedSensorData
00007 {
00008     class SensorData imuData;
00009     class SensorData baroData1;
00010     class SensorData baroData2;
00011     class SensorData gpsData;
00012     uint32_t timestamp;
00013     bool dataValid;
00014
00015     SharedSensorData() : imuData("bno055"), baroData1("baro1"), baroData2("baro2"), gpsData("gps"),
      timestamp(0), dataValid(false) {}
00016 };
00017
00018 struct SharedFilteredData
00019 {
00020     float altitude;
00021     float verticalVelocity;
00022     float orientation[4]; // Quaternion [w, x, y, z]
00023     uint32_t timestamp;
00024     bool dataValid;
00025 };
```

7.9 IStateAction.hpp

```
00001 #pragma once
00002
00003 #include "FlightState.hpp"
00004 #include <functional>
00005
00012 class IStateAction
00013 {
00014 public:
00015     virtual ~IStateAction() = default;
00016
00026     virtual void onEntry() {}
00027
00037     virtual void onExit() {}
00038
00047     virtual void onUpdate() {}
00048
00053     virtual RocketState getState() const = 0;
00054 };
00055
00063 class ITransitionCondition
00064 {
00065 public:
00066     virtual ~ITransitionCondition() = default;
00067
00079     virtual bool isConditionMet() = 0;
00080
00088     virtual const char *getConditionName() const = 0;
00089 };
```

7.10 StateAction.hpp

```
00001 #pragma once
00002
```

```
00003 #include "IStateAction.hpp"
00004 #include <memory>
00005 #include <vector>
00006
00022 class StateAction : public IStateAction
00023 {
00024 private:
00025     RocketState state;
00026     std::function<void()> entryAction;
00027     std::function<void()> exitAction;
00028     std::vector<TaskConfig> taskConfigs;
00029
00030 public:
00031     StateAction(RocketState s) : state(s) {}
00032
00038     StateAction &setEntryAction(const std::function<void()> &action)
00039     {
00040         tone(BUZZER_PIN, 100, 10);
00041         entryAction = action;
00042         return *this;
00043     }
00044
00050     StateAction &setExitAction(const std::function<void()> &action)
00051     {
00052         exitAction = action;
00053         return *this;
00054     }
00055
00061     StateAction &addTask(const TaskConfig &config)
00062     {
00063         taskConfigs.push_back(config);
00064         return *this;
00065     }
00066
00070     void onEntry() override
00071     {
00072         if (entryAction)
00073             entryAction();
00074     }
00075
00079     void onExit() override
00080     {
00081         if (exitAction)
00082             exitAction();
00083     }
00084
00089     RocketState getState() const override { return state; }
00090
00095     const std::vector<TaskConfig> &getTaskConfigs() const { return taskConfigs; }
00096 };
```

7.11 TransitionManager.hpp

```
00001 #pragma once
00002
00003 #include "IStateAction.hpp"
00004 #include "FlightState.hpp"
00005 #include <vector>
00006 #include <memory>
00019 struct Transition
00020 {
00021     RocketState fromState;                          // Source state for this transition
00022     RocketState toState;                            // Destination state for this transition
00023     FSMEvent triggerEvent;                          // Event that triggers this transition
00024     std::shared_ptr<ITransitionCondition> condition; // Optional condition for this transition to
      occur (nullptr if none)
00025
00034     Transition(RocketState from, RocketState to, FSMEvent event, std::shared_ptr<ITransitionCondition>
      cond = nullptr)
00035         : fromState(from), toState(to), triggerEvent(event), condition(cond) {}
00036 };
00037
00054 class TransitionManager
00055 {
00056 private:
00057     std::vector<Transition> transitions; // Collection of all defined transitions
00058
00059 public:
00067     void addTransition(const Transition &transition);
00083     std::optional<RocketState> findTransition(RocketState currentState, FSMEvent event);
00084
00099     bool checkAutomaticTransitions(RocketState currentState, unsigned long stateStartTime);
00100 };
```

7.12 BarometerTask.hpp

```
00001 #pragma once
00002 #include "BaseTask.hpp"
00003
00004 #include <MS561101BA03.hpp>
00005 #include <Logger.hpp>
00006 #include <SharedData.hpp>
00007 #include <config.h>
00008 #include <vector>
00009 #include <algorithm>
00010
00011 // If not commented, the Baro1 is used, otherwise Baro2
00012 #define BARO_1
00013
00014 // Simple moving average filter for noise reduction
00015 class MovingAverageFilter {
00016 public:
00017     MovingAverageFilter(size_t windowSize = 10) : windowSize(windowSize) {
00018         buffer.reserve(windowSize);
00019     }
00020
00021     float update(float newValue) {
00022         buffer.push_back(newValue);
00023         if (buffer.size() > windowSize) {
00024             buffer.erase(buffer.begin());
00025         }
00026
00027         float sum = 0.0f;
00028         for (float val : buffer) {
00029             sum += val;
00030         }
00031         return sum / buffer.size();
00032     }
00033
00034     void reset() {
00035         buffer.clear();
00036     }
00037
00038     bool isReady() const {
00039         return buffer.size() >= windowSize;
00040     }
00041
00042 private:
00043     size_t windowSize;
00044     std::vector<float> buffer;
00045 };
00046
00047 // Median filter for spike rejection (better for outliers than moving average)
00048 class MedianFilter {
00049 public:
00050     MedianFilter(size_t windowSize) : windowSize(windowSize) {
00051         buffer.reserve(windowSize);
00052     }
00053
00054     float update(float newValue) {
00055         buffer.push_back(newValue);
00056         if (buffer.size() > windowSize) {
00057             buffer.erase(buffer.begin());
00058         }
00059
00060         // Create sorted copy to find median
00061         std::vector<float> sorted = buffer;
00062         std::sort(sorted.begin(), sorted.end());
00063
00064         size_t mid = sorted.size() / 2;
00065         if (sorted.size() % 2 == 0) {
00066             return (sorted[mid - 1] + sorted[mid]) / 2.0f;
00067         } else {
00068             return sorted[mid];
00069         }
00070     }
00071
00072     void reset() {
00073         buffer.clear();
00074     }
00075
00076     bool isReady() const {
00077         return buffer.size() >= windowSize;
00078     }
00079
00080 private:
00081     size_t windowSize;
00082     std::vector<float> buffer;
00083 };
00084
```

```
00085 class BarometerTask : public BaseTask
00086 {
00087 public:
00088     BarometerTask(std::shared_ptr<SharedSensorData> sensorData,
00089                   SemaphoreHandle_t sensorDataMutex,
00090                   std::shared_ptr<bool> isRising,
00091                   std::shared_ptr<float> heightGainSpeed,
00092                   std::shared_ptr<float> currentHeight)
00093         : BaseTask("BarometerTask"),
00094           sensorData(sensorData),
00095           dataMutex(sensorDataMutex),
00096           isRising(isRising),
00097           heightGainSpeed(heightGainSpeed),
00098           currentHeight(currentHeight)
00099     {}
00100
00101     void taskFunction() override;
00102
00103     ~BarometerTask() override
00104     {
00105         stop();
00106     }
00107
00108 private:
00109     std::shared_ptr<SharedSensorData> sensorData;
00110     SemaphoreHandle_t dataMutex;
00111     std::shared_ptr<bool> isRising;
00112     std::shared_ptr<float> heightGainSpeed;
00113     std::shared_ptr<float> currentHeight;
00114
00115     // Maximum altitude reached for easy access in BarometerTask
00116     static float max_altitude_read;
00117
00118     // Noise reduction: Median filters reject spikes better than moving average
00119     // Window size from config.h - tune BAROMETER_FILTER_WINDOW for your needs
00120     MedianFilter pressureFilter{BAROMETER_FILTER_WINDOW};
00121
00122     // Buffer for tendency filtering, used in isStillRising()
00123     std::vector<float> pressureTrendBuffer;
00124     size_t trendBufferSize = APOGEE_DETECTION_WINDOW_SIZE;
00125     size_t mainDeploymentAltitude = 450; // Altitude for main deployment in meters
00126
00127     // Called in update to add new values to the filter and remove old ones
00128     void addPressureTrendValue(float value) {
00129         if (pressureTrendBuffer.size() >= trendBufferSize) {
00130             pressureTrendBuffer.erase(pressureTrendBuffer.begin());
00131         }
00132
00133         pressureTrendBuffer.push_back(value);
00134     }
00135
00136     // If max altitude reached is needed to be retrived
00137     float getMaxAltitudeReached() { return max_altitude_read; }
00138
00139 };
```

7.13 BaseTask.hpp

```
00001 #pragma once
00002
00003 #include "ITask.hpp"
00004 #include "esp_task_wdt.h"
00005
00021 class BaseTask : public ITask
00022 {
00023 protected:
00024     TaskHandle_t taskHandle;
00025     TaskConfig config;
00026     volatile bool running;
00027     const char *taskName;
00028
00029 public:
00030     BaseTask(const char *name);
00031     virtual ~BaseTask();
00032
00033     bool start(const TaskConfig &config) override;
00034     void stop() override;
00035     bool isRunning() const override { return running; }
00036     const char *getName() const override { return taskName; }
00037     uint32_t getStackHighWaterMark() const override;
00038
00039 protected:
00040     virtual void taskFunction() = 0;
00041     virtual void onTaskStart() {}
```

```
00042     virtual void onTaskStop() {}
00043
00044 private:
00045     static void taskWrapper(void *parameter);
00046     void internalTaskFunction();
00047 };
```

7.14 EkfTask.hpp

```
00001 #pragma once
00002
00003 #include "BaseTask.hpp"
00004 #include "SharedData.hpp"
00005 #include "KalmanFilter1D.hpp"
00006
00007 // NEED TO BE PROPERLY INITIALIZED !!!
00008 class EkfTask : public BaseTask {
00009 public:
00010     EkfTask(std::shared_ptr<SharedSensorData> sensorData,
00011         SemaphoreHandle_t sensorDataMutex,
00012         std::shared_ptr<KalmanFilter1D> kalmanFilter) :
00013         BaseTask("EkfTask"),
00014         sensorData(sensorData),
00015         sensorDataMutex(sensorDataMutex),
00016         kalmanFilter(kalmanFilter) {}
00017     ~EkfTask() override;
00018
00019 protected:
00020     void taskFunction() override;
00021
00022 private:
00023     std::shared_ptr<SharedSensorData> sensorData;
00024     SemaphoreHandle_t sensorDataMutex;
00025     std::shared_ptr<KalmanFilter1D> kalmanFilter;
00026
00027     uint32_t lastTimestamp = 0;
00028 };
```

7.15 GpsTask.hpp

```
00001 #pragma once
00002
00003 #include "BaseTask.hpp"
00004 #include "SharedData.hpp"
00005 #include <cstring>
00006 #include <freertos/FreeRTOS.h>
00007 #include <freertos/task.h>
00008 #include <GPS.hpp>
00009 #include "Logger.hpp"
00010 #include <RocketLogger.hpp>
00011
00012 class GpsTask : public BaseTask
00013 {
00014 public:
00015     GpsTask(std::shared_ptr<SharedSensorData> sensorData,
00016             SemaphoreHandle_t sensorDataMutex,
00017             std::shared_ptr<ISensor> gps,
00018             std::shared_ptr<RocketLogger> rocketLogger,
00019             SemaphoreHandle_t loggerMutex
00020         )
00021         : BaseTask("GpsTask"),
00022           sensorData(sensorData),
00023           dataMutex(sensorDataMutex),
00024           gps(gps ? gps.get() : nullptr),
00025           rocketLogger(rocketLogger),
00026           loggerMutex(loggerMutex)
00027     {
00028         LOG_INFO("GpsTask", "Initialized with GPS: %s", gps ? "OK" : "NULL");
00029     }
00030
00031     ~GpsTask() override
00032     {
00033         stop();
00034     }
00035
00036     void setGps(std::shared_ptr<ISensor> gps)
00037     {
00038         this->gps = gps.get();
00039         LOG_INFO("GpsTask", "Updated GPS: %s", gps ? "OK" : "NULL");
00040     }
00041
```

```
00042 protected:
00043     void taskFunction() override;
00044     void onTaskStart() override;
00045     void onTaskStop() override;
00046 private:
00047     std::shared_ptr<SharedSensorData> sensorData;
00048     SemaphoreHandle_t dataMutex;
00049     ISensor *gps;
00050
00051     std::shared_ptr<RocketLogger> rocketLogger;
00052     SemaphoreHandle_t loggerMutex;
00053 };
```

7.16 ITask.hpp

```
00001 #pragma once
00002
00003 #include <freertos/FreeRTOS.h>
00004 #include <freertos/task.h>
00005 #include <Arduino.h>
00006 #include "TaskConfig.hpp"
00007
00012 class ITask
00013 {
00014 public:
00015     virtual ~ITask() = default;
00016
00024     virtual bool start(const TaskConfig &config) = 0;
00029     virtual void stop() = 0;
00036     virtual bool isRunning() const = 0;
00042     virtual const char *getName() const = 0;
00048     virtual uint32_t getStackHighWaterMark() const = 0;
00049
00050 protected:
00051     virtual void taskFunction() = 0;
00052 };
```

7.17 SDLoggingTask.hpp

```
00001 #pragma once
00002
00003 #include <cstring>
00004 #include <freertos/FreeRTOS.h>
00005 #include <freertos/task.h>
00006 #include "BaseTask.hpp"
00007 #include "RocketLogger.hpp"
00008 #include <config.h>
00009 #include <SD-master.hpp>
00010 #include <Logger.hpp>
00011
00012 class SDLoggingTask : public BaseTask {
00013 public:
00014     SDLoggingTask(std::shared_ptr<RocketLogger> rocketLogger = nullptr,
00015                   SemaphoreHandle_t loggerMutex = nullptr,
00016                   std::shared_ptr<SD> sdCard = nullptr);
00017
00018         ~SDLoggingTask() override;
00019
00020 protected:
00021     void taskFunction() override;
00022
00023 private:
00024     std::shared_ptr<RocketLogger> rocketLogger;
00025     SemaphoreHandle_t loggerMutex;
00026
00027     std::shared_ptr<SD> sdCard;
00028     bool sdInitialized = false;
00029     int file_counter = 0;
00030
00031 };
```

7.18 SensorTask.hpp

```
00001 #pragma once
00002
00003 #include "BaseTask.hpp"
00004 #include "SharedData.hpp"
00005 #include "Logger.hpp"
```

```
00006 #include "RocketLogger.hpp"
00007 #include <ISensor.hpp>
00008 #include <memory>
00009
00010 class SensorTask : public BaseTask
00011 {
00012 private:
00013     std::shared_ptr<SharedSensorData> sensorData;
00014     SemaphoreHandle_t dataMutex;
00015
00016     std::shared_ptr<RocketLogger> rocketLogger;
00017     SemaphoreHandle_t loggerMutex;
00018
00019     // Use shared pointers for sensors
00020     ISensor *bno055;
00021     ISensor *baro1;
00022     ISensor *baro2;
00023
00024 public:
00025     SensorTask(std::shared_ptr<SharedSensorData> sensorData,
00026                SemaphoreHandle_t mutex,
00027                std::shared_ptr<ISensor> imu = nullptr,
00028                std::shared_ptr<ISensor> barometer1 = nullptr,
00029                std::shared_ptr<ISensor> barometer2 = nullptr,
00030                std::shared_ptr<RocketLogger> rocketLogger = nullptr,
00031                SemaphoreHandle_t loggerMutex = nullptr);
00032
00033     void setSensors(std::shared_ptr<ISensor> imu,
00034                     std::shared_ptr<ISensor> barometer1,
00035                     std::shared_ptr<ISensor> barometer2);
00036
00037 protected:
00038     void taskFunction() override;
00039     void onTaskStart() override;
00040     void onTaskStop() override;
00041 };
```

7.19 SimulationTask.hpp

```
00001 #pragma once
00002
00003
00004 #include "SD-master.hpp"
00005 #include "BaseTask.hpp"
00006 #include "SharedData.hpp"
00007 #include "Logger.hpp"
00008 #include "RocketLogger.hpp"
00009 #include <ISensor.hpp>
00010 #include <memory>
00011 #include <vector>
00012 #include <string>
00013 #include <chrono>
00014
00015 // If this variable is not commented, the format for old simulation of mission analysis (August 2025),

00016 // is expected, otherwise the one obtained for Euroc 2025 is expected
00017 //#define OLD_DATA
00018
00019 class SimulationTask : public BaseTask {
00020 private:
00021     bool started = false;
00022
00023     // Shared static variables for simulation state
00024     static SD sdManager;
00025     static std::string csvFilePath;
00026     static uint32_t filePosition;
00027     static bool fileInitialized;
00028     // Used to calculate the elapsed time from the first task start
00029     static unsigned long startTime;
00030     // The first time it'll skip the first line (header)
00031     static bool firstTime;
00032
00033     std::shared_ptr<SharedSensorData> sensorData;
00034     SemaphoreHandle_t dataMutex;
00035     std::shared_ptr<RocketLogger> rocketLogger;
00036     SemaphoreHandle_t loggerMutex;
00037
00038 public:
00039     SimulationTask(
00040         const std::string& csvFilePath,
00041         std::shared_ptr<SharedSensorData> sensorData,
00042         SemaphoreHandle_t mutex,
00043         std::shared_ptr<RocketLogger> rocketLogger,
00044         SemaphoreHandle_t loggerMutex);
```

```
00045     ~SimulationTask();
00046     void onTaskStart() override;
00047     void onTaskStop() override;
00048     void taskFunction() override;
00049     void reset();
00050
00051 };
```

7.20 TaskConfig.hpp

```
00001 #pragma once
00002 #include <freertos/FreeRTOS.h>
00003 #include <freertos/task.h>
00004 #include <Arduino.h>
00005
00006 enum class TaskType
00007 {
00008     SENSOR,
00009     SIMULATION,
00010     EKF,
00011     RECOVERY,
00012     DATA_COLLECTION,
00013     SD_LOGGING,
00014     TELEMETRY,
00015     GPS,
00016     BAROMETER,
00017     LOGGING
00018 };
00019
00020 enum class TaskPriority
00021 {
00022     TASK_LOW = 1,
00023     TASK_MEDIUM = 2,
00024     TASK_HIGH = 3,
00025     TASK_CRITICAL = 4,
00026     TASK_REAL_TIME = 5
00027 };
00028
00029 enum class TaskCore
00030 {
00031     CORE_0 = 0, // Critical tasks
00032     CORE_1 = 1, // Non-critical tasks
00033     ANY_CORE = tskNO_AFFINITY
00034 };
00035
00036 struct TaskConfig
00037 {
00038     TaskType type;
00039     const char *name;
00040     uint32_t stackSize;
00041     TaskPriority priority;
00042     TaskCore coreId;
00043     bool shouldRun;
00044
00055     TaskConfig(TaskType t = TaskType::SENSOR, const char *n = "Task", uint32_t stack = 2048,
00056                TaskPriority prio = TaskPriority::TASK_MEDIUM,
00057                TaskCore core = TaskCore::ANY_CORE, bool run = true)
00058         : type(t), name(n), stackSize(stack), priority(prio), coreId(core), shouldRun(run) {}
00059 };
```

7.21 TaskManager.hpp

```
00001 #pragma once
00002
00003 #include "ITask.hpp"
00004 #include "SharedData.hpp"
00005 #include <memory>
00006 #include <map>
00007 #include <string>
00008 #include "TaskConfig.hpp"
00009 #include "Logger.hpp"
00010 #include "SD-master.hpp"
00011
00012 #include "SensorTask.hpp"
00013 #include "SDLoggingTask.hpp"
00014 #include "EkfTask.hpp"
00015 #include "GpsTask.hpp"
00016 #include "SimulationTask.hpp"
00017 #include "TelemetryTask.hpp"
00018 #include "BarometerTask.hpp"
00019 #include <EspNowTransmitter.hpp>
```

```
00020
00021 //#define SIMULATION_DATA // Comment this out to use real sensors
00022
00023 class TaskManager {
00024 private:
00025     std::map<TaskType, std::unique_ptr<ITask» tasks;
00026     std::shared_ptr<SharedSensorData> sensorData;
00027     std::shared_ptr<ISensor> bno055;
00028     std::shared_ptr<ISensor> baro1;
00029     std::shared_ptr<ISensor> baro2;
00030     std::shared_ptr<ISensor> gps;
00031     std::shared_ptr<RocketLogger> rocketLogger;
00032     SemaphoreHandle_t sensorDataMutex;
00033     SemaphoreHandle_t loggerMutex;
00034
00035     std::shared_ptr<SD> sd;
00036
00037     // Telemetry
00038     std::shared_ptr<EspNowTransmitter> espNowTransmitter;
00039
00040     // Flight state
00041     std::shared_ptr<bool> isRising;
00042     std::shared_ptr<float> heightGainSpeed;
00043     std::shared_ptr<float> currentHeight;
00044
00045 public:
00046     TaskManager(std::shared_ptr<SharedSensorData> sensorData,
00047             std::shared_ptr<ISensor> imu,
00048             std::shared_ptr<ISensor> barometer1,
00049             std::shared_ptr<ISensor> barometer2,
00050             std::shared_ptr<ISensor> gps,
00051             SemaphoreHandle_t sensorMutex,
00052             std::shared_ptr<SD> sd,
00053             std::shared_ptr<RocketLogger> rocketLogger,
00054             SemaphoreHandle_t loggerMutex,
00055             std::shared_ptr<bool> isRising,
00056             std::shared_ptr<float> heightGainSpeed,
00057             std::shared_ptr<float> currentHeight);
00058
00059     ~TaskManager();
00060
00061     void initializeTasks();
00062     bool startTask(TaskType type, const TaskConfig& config);
00063     void stopTask(TaskType type);
00064     void stopAllTasks();
00065     int getRunningTaskCount();
00066
00067     bool isTaskRunning(TaskType type) const;
00068     uint32_t getTaskStackUsage(TaskType type) const;
00069
00070     void printTaskStatus() const;
00071 };
```

7.22 TelemetryTask.hpp

```
00001 #pragma once
00002
00003 #include "BaseTask.hpp"
00004 #include "SharedData.hpp"
00005 #include "EspNowTransmitter.hpp"
00006 #include "Logger.hpp"
00007 #include <Packet.hpp>
00008 #include <PacketManager.hpp>
00009 #include <memory>
00010 #include <cstdint>
00011
00020 #pragma pack(push, 1)
00021 struct TelemetryPacket
00022 {
00023     uint32_t timestamp;
00024     bool dataValid;
00025
00026     struct
00027     {
00028         float accel_x, accel_y, accel_z;
00029         float gyro_x, gyro_y, gyro_z;
00030     } imu;
00031
00032     struct
00033     {
00034         float pressure;
00035         float temperature;
00036     } baro1;
00037
```

```
00038     struct
00039     {
00040         float pressure;
00041         float temperature;
00042     } baro2;
00043
00044     struct
00045     {
00046         float latitude;
00047         float longitude;
00048         float altitude;
00049     } gps;
00050 };
00051 #pragma pack(pop)
00052
00061 class TelemetryTask : public BaseTask
00062 {
00063 private:
00064     std::shared_ptr<SharedSensorData> sensorData;
00065     SemaphoreHandle_t dataMutex;
00066     std::shared_ptr<EspNowTransmitter> transmitter;
00067
00068     uint32_t transmitIntervalMs;
00069     uint32_t lastTransmitTime;
00070
00071     // Statistics
00072     uint32_t messagesCreated;
00073     uint32_t packetsSent;
00074     uint32_t transmitErrors;
00075
00076 public:
00085     TelemetryTask(std::shared_ptr<SharedSensorData> sensorData,
00086                   SemaphoreHandle_t mutex,
00087                   std::shared_ptr<EspNowTransmitter> espNowTransmitter,
00088                   uint32_t intervalMs = 1000);
00089
00097     void getStats(uint32_t &messages, uint32_t &packets, uint32_t &errors) const;
00098
00099 protected:
00100     void taskFunction() override;
00101     void onTaskStart() override;
00102     void onTaskStop() override;
00103
00104 private:
00111     bool collectSensorData(TelemetryPacket &packet);
00112
00119     bool transmitMessage(const std::vector<uint8_t> &message);
00120 };
```

7.23 CSVLogger.hpp

```
00001 #ifndef CSV_LOGGER_HPP
00002 #define CSV_LOGGER_HPP
00003
00004 #include <Arduino.h>
00005 #include <nlohmann/json.hpp>
00006 #include <SD-master.hpp> // La tua classe SD personalizzata
00007
00008 using json = nlohmann::json;
00009
00010 class CSVLogger {
00011 private:
00012     SD sdCard;
00013     bool headerWritten = false;
00014     std::string filename;
00015
00016 public:
00017     CSVLogger(std::string fileName) : filename(fileName) {}
00018
00019     bool init() {
00020         if (!sdCard.init()) {
00021             Serial.println("Errore inizializzazione SD per CSV Logger");
00022             return false;
00023         }
00024         return true;
00025     }
00026
00027     void writeHeader() {
00028         if (headerWritten) return;
00029
00030         std::string header = "timestamp,"
00031                              "bno_temp,bno_acc_x,bno_acc_y,bno_acc_z,bno_acc_mag,"
00032                              "bno_gyro_x,bno_gyro_y,bno_gyro_z,"
00033                              "bno_mag_x,bno_mag_y,bno_mag_z,"
```

```
00034                           "bno_grav_x,bno_grav_y,bno_grav_z,"
00035                           "bno_lin_acc_x,bno_lin_acc_y,bno_lin_acc_z,"
00036                           "bno_orient_x,bno_orient_y,bno_orient_z,"
00037                           "bno_quat_w,bno_quat_x,bno_quat_y,bno_quat_z,"
00038                           "bno_accel_cal,bno_gyro_cal,bno_mag_cal,bno_sys_cal,"
00039                           "lis_acc_x,lis_acc_y,lis_acc_z,lis_acc_mag,"
00040                           "bar1_pres,bar1_temp,"
00041                           "bar2_pres,bar2_temp,"
00042                           "main_actuator_state,drogue_actuator_state,buzzer_state,"
00043                           "voltage_adc,voltage_v,voltage_perc,"
00044                           "gps_available\n";
00045
00046       if (sdCard.writeFile(filename, header)) {
00047           headerWritten = true;
00048           Serial.println("Header CSV scritto");
00049       } else {
00050           Serial.println("Errore scrittura header CSV");
00051       }
00052       sdCard.closeFile();
00053   }
00054
00055   void logSensorData(const json& allData, unsigned long timestamp) {
00056       if (!headerWritten) {
00057           writeHeader();
00058       }
00059
00060       // Struttura per memorizzare i dati dei sensori
00061       struct SensorData {
00062           // BNO055
00063           float bno_temp = 0;
00064           float bno_acc_x = 0, bno_acc_y = 0, bno_acc_z = 0, bno_acc_mag = 0;
00065           float bno_gyro_x = 0, bno_gyro_y = 0, bno_gyro_z = 0;
00066           float bno_mag_x = 0, bno_mag_y = 0, bno_mag_z = 0;
00067           float bno_grav_x = 0, bno_grav_y = 0, bno_grav_z = 0;
00068           float bno_lin_acc_x = 0, bno_lin_acc_y = 0, bno_lin_acc_z = 0;
00069           float bno_orient_x = 0, bno_orient_y = 0, bno_orient_z = 0;
00070           float bno_quat_w = 0, bno_quat_x = 0, bno_quat_y = 0, bno_quat_z = 0;
00071           int bno_accel_cal = 0, bno_gyro_cal = 0, bno_mag_cal = 0, bno_sys_cal = 0;
00072
00073           // LIS3DHTR
00074           float lis_acc_x = 0, lis_acc_y = 0, lis_acc_z = 0, lis_acc_mag = 0;
00075
00076           // Barometri
00077           float bar1_pres = 0, bar1_temp = 0;
00078           float bar2_pres = 0, bar2_temp = 0;
00079
00080           // Attuatori
00081           std::string main_actuator = "OFF";
00082           std::string drogue_actuator = "OFF";
00083           std::string buzzer = "OFF";
00084
00085           // Voltaggio
00086           int voltage_adc = 0;
00087           float voltage_v = 0;
00088           std::string voltage_perc = "0%";
00089
00090           // GPS
00091           bool gps_available = false;
00092       } data;
00093
00094       // Parsing dei dati JSON dal RocketLogger
00095       if (allData.is_array()) {
00096           for (const auto& sensor : allData) {
00097               if (sensor.contains("content") && sensor["content"].contains("source")) {
00098                   std::string source = sensor["content"]["source"];
00099
00100                   if (source == "BNO055") {
00101                       const auto& sensorData = sensor["content"]["sensorData"];
00102
00103                       if (sensorData.contains("board_temperature")) {
00104                           data.bno_temp = sensorData["board_temperature"];
00105                       }
00106
00107                       if (sensorData.contains("accelerometer")) {
00108                           const auto& acc = sensorData["accelerometer"];
00109                           if (acc.contains("x")) data.bno_acc_x = acc["x"];
00110                           if (acc.contains("y")) data.bno_acc_y = acc["y"];
00111                           if (acc.contains("z")) data.bno_acc_z = acc["z"];
00112                           if (acc.contains("magnitude")) data.bno_acc_mag = acc["magnitude"];
00113                       }
00114
00115                       if (sensorData.contains("angular_velocity")) {
00116                           const auto& gyro = sensorData["angular_velocity"];
00117                           if (gyro.contains("x")) data.bno_gyro_x = gyro["x"];
00118                           if (gyro.contains("y")) data.bno_gyro_y = gyro["y"];
00119                           if (gyro.contains("z")) data.bno_gyro_z = gyro["z"];
00120                       }
```

```
00121
00122                         if (sensorData.contains("magnetometer")) {
00123                             const auto& mag = sensorData["magnetometer"];
00124                             if (mag.contains("x")) data.bno_mag_x = mag["x"];
00125                             if (mag.contains("y")) data.bno_mag_y = mag["y"];
00126                             if (mag.contains("z")) data.bno_mag_z = mag["z"];
00127                         }
00128
00129                         if (sensorData.contains("gravity")) {
00130                             const auto& grav = sensorData["gravity"];
00131                             if (grav.contains("x")) data.bno_grav_x = grav["x"];
00132                             if (grav.contains("y")) data.bno_grav_y = grav["y"];
00133                             if (grav.contains("z")) data.bno_grav_z = grav["z"];
00134                         }
00135
00136                         if (sensorData.contains("linear_acceleration")) {
00137                             const auto& linAcc = sensorData["linear_acceleration"];
00138                             if (linAcc.contains("x")) data.bno_lin_acc_x = linAcc["x"];
00139                             if (linAcc.contains("y")) data.bno_lin_acc_y = linAcc["y"];
00140                             if (linAcc.contains("z")) data.bno_lin_acc_z = linAcc["z"];
00141                         }
00142
00143                         if (sensorData.contains("orientation")) {
00144                             const auto& orient = sensorData["orientation"];
00145                             if (orient.contains("x")) data.bno_orient_x = orient["x"];
00146                             if (orient.contains("y")) data.bno_orient_y = orient["y"];
00147                             if (orient.contains("z")) data.bno_orient_z = orient["z"];
00148                         }
00149
00150                         if (sensorData.contains("quaternion")) {
00151                             const auto& quat = sensorData["quaternion"];
00152                             if (quat.contains("w")) data.bno_quat_w = quat["w"];
00153                             if (quat.contains("x")) data.bno_quat_x = quat["x"];
00154                             if (quat.contains("y")) data.bno_quat_y = quat["y"];
00155                             if (quat.contains("z")) data.bno_quat_z = quat["z"];
00156                         }
00157
00158                         if (sensorData.contains("accel_calibration")) {
00159                             data.bno_accel_cal = sensorData["accel_calibration"];
00160                         }
00161                         if (sensorData.contains("gyro_calibration")) {
00162                             data.bno_gyro_cal = sensorData["gyro_calibration"];
00163                         }
00164                         if (sensorData.contains("mag_calibration")) {
00165                             data.bno_mag_cal = sensorData["mag_calibration"];
00166                         }
00167                         if (sensorData.contains("system_calibration")) {
00168                             data.bno_sys_cal = sensorData["system_calibration"];
00169                         }
00170                     }
00171
00172                     else if (source == "LIS3DHTR") {
00173                         const auto& sensorData = sensor["content"]["sensorData"];
00174                         if (sensorData.contains("accel_x")) data.lis_acc_x = sensorData["accel_x"];
00175                         if (sensorData.contains("accel_y")) data.lis_acc_y = sensorData["accel_y"];
00176                         if (sensorData.contains("accel_z")) data.lis_acc_z = sensorData["accel_z"];
00177                         if (sensorData.contains("accel_magnitude")) data.lis_acc_mag =
      sensorData["accel_magnitude"];
00178                     }
00179
00180                     else if (source == "BAR1") {
00181                         const auto& sensorData = sensor["content"]["sensorData"];
00182                         if (sensorData.contains("pressure")) data.bar1_pres = sensorData["pressure"];
00183                         if (sensorData.contains("temperature")) data.bar1_temp =
      sensorData["temperature"];
00184                     }
00185
00186                     else if (source == "BAR2") {
00187                         const auto& sensorData = sensor["content"]["sensorData"];
00188                         if (sensorData.contains("pressure")) data.bar2_pres = sensorData["pressure"];
00189                         if (sensorData.contains("temperature")) data.bar2_temp =
      sensorData["temperature"];
00190                     }
00191
00192                     else if (source == "MainActuators") {
00193                         if (sensor["content"]["sensorData"].contains("State")) {
00194                             data.main_actuator = sensor["content"]["sensorData"]["State"];
00195                         }
00196                     }
00197
00198                     else if (source == "DrogueActuators") {
00199                         if (sensor["content"]["sensorData"].contains("State")) {
00200                             data.drogue_actuator = sensor["content"]["sensorData"]["State"];
00201                         }
00202                     }
00203
00204                     else if (source == "Buzzer") {
```

```
00205                         if (sensor["content"]["sensorData"].contains("State")) {
00206                             data.buzzer = sensor["content"]["sensorData"]["State"];
00207                         }
00208                     }
00209 
00210                     else if (source == "Voltage") {
00211                         const auto& sensorData = sensor["content"]["sensorData"];
00212                         if (sensorData.contains("ADC_Value")) data.voltage_adc =
    sensorData["ADC_Value"];
00213                         if (sensorData.contains("Voltage")) data.voltage_v = sensorData["Voltage"];
00214                         if (sensorData.contains("Percentage")) data.voltage_perc =
    sensorData["Percentage"];
00215                     }
00216 
00217                     else if (source == "GPS") {
00218                         // Se troviamo dati GPS, significa che sono disponibili
00219                         data.gps_available = true;
00220                     }
00221                 }
00222 
00223                 // Gestione errori GPS
00224                 else if (sensor.contains("type") && sensor["type"] == "ERROR" &&
00225                          sensor.contains("content") && sensor["content"].contains("source") &&
00226                          sensor["content"]["source"] == "RocketLogger" &&
00227                          sensor["content"].contains("message")) {
00228                     std::string message = sensor["content"]["message"];
00229                     if (message.find("GPS") != std::string::npos) {
00230                         data.gps_available = false;
00231                     }
00232                 }
00233             }
00234         }
00235 
00236         // Crea la riga CSV
00237         std::string csvLine = std::to_string(timestamp) + "," +
00238                               std::to_string(data.bno_temp) + "," +
00239                               std::to_string(data.bno_acc_x) + "," +
00240                               std::to_string(data.bno_acc_y) + "," +
00241                               std::to_string(data.bno_acc_z) + "," +
00242                               std::to_string(data.bno_acc_mag) + "," +
00243                               std::to_string(data.bno_gyro_x) + "," +
00244                               std::to_string(data.bno_gyro_y) + "," +
00245                               std::to_string(data.bno_gyro_z) + "," +
00246                               std::to_string(data.bno_mag_x) + "," +
00247                               std::to_string(data.bno_mag_y) + "," +
00248                               std::to_string(data.bno_mag_z) + "," +
00249                               std::to_string(data.bno_grav_x) + "," +
00250                               std::to_string(data.bno_grav_y) + "," +
00251                               std::to_string(data.bno_grav_z) + "," +
00252                               std::to_string(data.bno_lin_acc_x) + "," +
00253                               std::to_string(data.bno_lin_acc_y) + "," +
00254                               std::to_string(data.bno_lin_acc_z) + "," +
00255                               std::to_string(data.bno_orient_x) + "," +
00256                               std::to_string(data.bno_orient_y) + "," +
00257                               std::to_string(data.bno_orient_z) + "," +
00258                               std::to_string(data.bno_quat_w) + "," +
00259                               std::to_string(data.bno_quat_x) + "," +
00260                               std::to_string(data.bno_quat_y) + "," +
00261                               std::to_string(data.bno_quat_z) + "," +
00262                               std::to_string(data.bno_accel_cal) + "," +
00263                               std::to_string(data.bno_gyro_cal) + "," +
00264                               std::to_string(data.bno_mag_cal) + "," +
00265                               std::to_string(data.bno_sys_cal) + "," +
00266                               std::to_string(data.lis_acc_x) + "," +
00267                               std::to_string(data.lis_acc_y) + "," +
00268                               std::to_string(data.lis_acc_z) + "," +
00269                               std::to_string(data.lis_acc_mag) + "," +
00270                               std::to_string(data.bar1_pres) + "," +
00271                               std::to_string(data.bar1_temp) + "," +
00272                               std::to_string(data.bar2_pres) + "," +
00273                               std::to_string(data.bar2_temp) + "," +
00274                               data.main_actuator + "," +
00275                               data.drogue_actuator + "," +
00276                               data.buzzer + "," +
00277                               std::to_string(data.voltage_adc) + "," +
00278                               std::to_string(data.voltage_v) + "," +
00279                               data.voltage_perc + "," +
00280                               (data.gps_available ? "1" : "0") + "\n";
00281 
00282         // Scrivi su SD
00283         if (!sdCard.appendFile(filename, csvLine)) {
00284             Serial.println("Errore scrittura CSV");
00285         }
00286         sdCard.closeFile();
00287     }
00288 };
00289 
```

```
00290 #endif // CSV_LOGGER_HPP
```

7.24 ILoggable.hpp

```
00001 #pragma once
00002
00003 #include <nlohmann/json.hpp>
00004
00005 using json = nlohmann::json;
00006
00011 class ILoggable {
00012 protected:
00013     // Origin of the log
00014     std::string source;
00015 public:
00019     virtual ~ILoggable() = default;
00020
00026     virtual std::string getSource() const { return this->source; };
00027
00033     virtual json toJSON() const = 0;
00034 };
00035
00036
```

7.25 LogMessage.hpp

```
00001 #pragma once
00002
00003 #include <string>
00004 #include <nlohmann/json.hpp>
00005 #include <SensorData.hpp>
00006 #include "ILoggable.hpp"
00007
00008 using json = nlohmann::json;
00009
00014 class LogMessage : public ILoggable {
00015 private:
00016     // Message to be logged
00017     std::string message;
00018
00019 public:
00026     LogMessage(std::string source, std::string message) {
00027         this->source = source;
00028         this->message = message;
00029     }
00030
00036     std::string getMessage() const {
00037         return message;
00038     }
00039
00045     json toJSON() const override {
00046         json j;
00047         j["source"] = source;
00048         j["message"] = message;
00049         return j;
00050     }
00051 };
00052
00053
```

7.26 LogSensorData.hpp

```
00001 #pragma once
00002
00003 #include <string>
00004 #include <nlohmann/json.hpp>
00005 #include <SensorData.hpp>
00006 #include "ILoggable.hpp"
00007
00012 class LogSensorData : public ILoggable
00013 {
00014 private:
00015     // The sensor data to be logged
00016     SensorData sensorData;
00017
00018 public:
00025     LogSensorData(std::string source, SensorData sensorData)
00026         : sensorData(sensorData)
```

```
00027     {
00028         this->source = source;
00029     }
00030
00036     SensorData getSensorData() const
00037     {
00038         return sensorData;
00039     }
00040
00046     json toJSON() const override
00047     {
00048         json j;
00049
00050         // Add source to JSON
00051         j["source"] = source;
00052
00053         // Add sensor data to JSON
00054         json sensorDataJson;
00055         auto dataMap = sensorData.getDataMap();
00056
00057         for (const auto &[key, value] : dataMap)
00058         {
00059             // Use std::visit to handle each variant type
00060             std::visit([&sensorDataJson, &key](const auto &arg)
00061                        { sensorDataJson[key] = arg; }, value);
00062         }
00063         j["sensorData"] = sensorDataJson;
00064
00065         return j;
00066     }
00067 };
00068
```

7.27 config.h

```
00001 #pragma once
00002
00003 #include <cstdint>
00004
00005 #define __DEBUG__
00006
00007 #define SUFFICIENT_SENSOR_CALIBRATION 2
00008 #define SENSOR_LOOKUP_MAX_ATTEMPTS 5
00009 #define SENSOR_LOOKUP_TIMEOUT 1000
00010
00011 #define MS56_I2C_ADDR_1 0x77
00012 #define MS56_I2C_ADDR_2 0x76
00013 #define BNO055_I2C_ADDR 0x28
00014
00015 /* Legacy sensors */
00016 // #define BME680_I2C_ADDR_1 0x77
00017 // #define BME680_I2C_ADDR_2 0x76
00018 // #define MPRLS_I2C_ADDR 0x18
00019 // #define I2C_MULTIPLEXER_ADDRESS 0x70
00020 // #define I2C_MULTIPLEXER_MPRLS1 1
00021 // #define I2C_MULTIPLEXER_MPRLS2 0
00022
00023 /* LoRa configuration */
00024 #define E220_22       // E220-900T22D
00025 #define FREQUENCY_868 // 868 MHz
00026
00027 #define LORA_SERIAL 1 // Serial port for LoRa (1 = Serial1, 2 = Serial2)
00028
00029 /* Legacy Transmitter antenna address */
00030 #define LORA_ADDH 0x00
00031 #define LORA_ADDL 0x03
00032 /* Legacy Receiver antenna address */
00033 #define LORA_RECEIVER_ADDH 0x00
00034 #define LORA_RECEIVER_ADDL 0x05
00035
00036 #define LORA_CHANNEL 23 // Legacy Communication channel
00037
00038 /* Uncomment the following line to enable RSSI */
00039 #define ENABLE_RSSI
00040
00041 #define SERIAL_BAUD_RATE 115200 // Serial baud rate
00042
00043 // #define TRANSMITTER_CONFIG_MODE_ENABLE // Enable configuration mode for the transmitter (needs to
      be tested on new hardware)
00044
00045 // Number of log entries to batch before writing to SD card
00046 #define BATCH_SIZE 30
00047
00048 // Maximum log entries before forcing a clear (emergency protection)
```

```
00049 #define MAX_LOG_ENTRIES 20
00050
00051 /* Flight parameters configuration */
00052 #define LIFTOFF_ACCELERATION_THRESHOLD GRAVITY * 2.0f // Threshold for the detection of liftoff when
      relative_acceleration is > 2G in any direction (relative_acceleration = acceleration - gravity)
00053 #define LIFTOFF_TIMEOUT_MS 1000    // Threshold for the detection of liftoff
00054 #define DROGUE_APOGEE_TIMEOUT 2000 // Threshold for opening the drogue parachute after apogee is
      detected
00055 #define MAIN_ALTITUDE_THRESHOLD 450.0f // Altitude threshold for the deployment of the main parachute
      (in meters)
00056 #define TOUCHDOWN_VELOCITY_THRESHOLD 2.0f // Vertical velocity threshold for touchdown detection (in
      m/s)
00057
00058 /* ESP-NOW Telemetry Configuration */
00059 // MAC address of the peer receiver (ESP32 that will relay to LoRa)
00060 // Replace with your actual receiver MAC address
00061 #define ESPNOW_PEER_MAC { 0x34, 0xCD, 0xB0, 0x3C, 0x54, 0xB4 }
00062 #define ESPNOW_CHANNEL 1 // WiFi channel (1-13)
00063 #define TELEMETRY_INTERVAL_MS 500
00064 #define TOUCHDOWN_ALTITUDE_THRESHOLD 15.0f // Altitude threshold for touchdown detection (in meters)
00065
00066 // In the case of the model velocity is extremely wrong, the access to the parachute is denied, and
      let to the CatsVega, to avoid the opening of it during ascension
00067 #define MIN_ACCECTABLE_VELOCITY -1000
00068 #define MAX_ACCECTABLE_VELOCITY 1000
00069
00070 // Kalman Constants
00071 #define NUM_CALIBRATION_SAMPLES 200
00072 #define STD_THRESHOLD 0.1f
00073 #define SEA_LEVEL 63.0f  // Your launch site altitude above sea level (meters)
00074
00075 // Sea level pressure reference for barometric altitude calculation
00076 // IMPORTANT: This must match your local atmospheric conditions!
00077 // Option 1: Check local weather station for "sea level pressure" (QNH in aviation)
00078 // Option 2: Calculate from known altitude: If at SEA_LEVEL meters reading P hPa,
00079 //           then SEA_LEVEL_PRESSURE_HPA = P / (1 - 0.0065 * SEA_LEVEL / 288.15)^5.255
00080 // Example: At 63m reading 1010.28 hPa -> sea level pressure = 1017.5 hPa
00081 #define SEA_LEVEL_PRESSURE_HPA 1017.5f  // Local sea level pressure in hPa
00082
00083 #define H_BIAS_PRESSURE_SENSOR 2.0f
00084 #define GPS_BIAS 3.0f
00085
00086 // Barometer noise filtering
00087 // BAROMETER_FILTER_WINDOW: Size of median filter window for pressure/altitude smoothing
00088 // Smaller = faster response but more noise (1 = no filtering)
00089 // Larger = smoother but more lag (recommended: 3-7)
00090 // At 10Hz sampling: window=5 adds 50ms lag
00091 #define BAROMETER_FILTER_WINDOW 9
00092 #define APOGEE_DETECTION_WINDOW_SIZE 15
00093
00094 #define STATE_INDEX_ALTITUDE 0
00095 #define STATE_INDEX_VELOCITY 1
00096 #define STATE_INDEX_QUAT_W 2
00097 #define STATE_INDEX_QUAT_X 3
00098 #define STATE_INDEX_QUAT_Y 4
00099 #define STATE_INDEX_QUAT_Z 5
00100
00101 // GPS Configuration
00102 #define GPS_FIX_TIMEOUT_MS 180000      // 3 minutes
00103 #define GPS_MIN_FIX 3                  // Minimum fix type for valid GPS data (2 = 2D fix, 3 = 3D fix)
00104 #define GPS_FIX_LOOKUP_INTERVAL_MS 500 // Interval between GPS fix status checks during initialization
00105
00106 // IMU Calibration
00107 #define IMU_MINIMUM_CALIBRATION 3 // Minimum calibration level for IMU sensors (0-3)
00108
00109 #define GRAVITY 9.80665f
00110
00111
00112 // Telemetry configuration
00113 constexpr uint8_t RECEIVER_MAC_ADDRESS[] = { 0x34, 0xCD, 0xB0, 0x3D, 0x97, 0xFC };  // MAC dell'ESP32
      ricevente: // 34:CD:B0:3D:97:FC
```

7.28 lib/global/src/pins.h File Reference

PIN definition.

Macros

- #define **I2C_SDA** 11

  I2C STD PIN.
- #define **I2C_SCL** 12

- #define **I2C_MASTER_NUM** I2C_NUM_0
- #define **I2C_MASTER_FREQ_HZ** 100000
- #define **I2C_MASTER_TX_BUF_DISABLE** 0
- #define **I2C_MASTER_RX_BUF_DISABLE** 0
- #define **SD_CLK** D13
- #define **SD_SO** D12
- #define **SD_SI** D11
- #define **SD_CS** D10
- #define **SD_DET** -1
- #define **DROGUE_ACTUATOR_PIN** D0
- #define **MAIN_ACTUATOR_PIN** D1
- #define **BUZZER_PIN** D2
- #define **LED_RED_PIN** A2
- #define **LED_GREEN_PIN** A3
- #define **LED_BLUE_PIN** A1
- #define **GPS_LED** D6
- #define **ARMING_PIN** D6

7.28.1 Detailed Description

PIN definition.

Author

luca.pulga@studio.unibo.it

Version

0.1

Date

2024-10-02

Copyright

7.29 pins.h

Go to the documentation of this file.

```
00001
00011 #pragma once
00012
00017 // Pin I2C.
00018 #define I2C_SDA 11
00019 #define I2C_SCL 12
00020 #define I2C_MASTER_NUM I2C_NUM_0
00021 #define I2C_MASTER_FREQ_HZ 100000
00022 #define I2C_MASTER_TX_BUF_DISABLE 0
00023 #define I2C_MASTER_RX_BUF_DISABLE 0
00024
00025 // SD Card pins.
00026 #define SD_CLK D13
00027 #define SD_SO D12
00028 #define SD_SI D11
00029 #define SD_CS D10
00030 #define SD_DET -1
00031
00032 // Actuators
00033 #define DROGUE_ACTUATOR_PIN D0
00034 #define MAIN_ACTUATOR_PIN D1
00035 #define BUZZER_PIN D2
00036
00037 // LED
00038 #define LED_RED_PIN A2
```

```
00039 #define LED_GREEN_PIN A3
00040 #define LED_BLUE_PIN A1
00041
00042 #define GPS_LED D6
00043
00044 #define ARMING_PIN D6
00045
```

7.30 lib/global/src/TelemetryFields.h File Reference

Standardized field names for telemetry JSON messages.

Macros

- #define **TELEM_FIELD_TIMESTAMP** "timestamp"
- #define **TELEM_FIELD_DATA_VALID** "dataValid"
- #define **TELEM_FIELD_IMU** "imu"
- #define **TELEM_FIELD_BARO1** "baro1"
- #define **TELEM_FIELD_BARO2** "baro2"
- #define **TELEM_FIELD_GPS** "gps"
- #define **TELEM_FIELD_IMU_ACCEL** "accel"
- #define **TELEM_FIELD_IMU_GYRO** "gyro"
- #define **TELEM_FIELD_IMU_MAG** "mag"
- #define **TELEM_FIELD_VEC_X** "x"
- #define **TELEM_FIELD_VEC_Y** "y"
- #define **TELEM_FIELD_VEC_Z** "z"
- #define **TELEM_FIELD_BARO_PRESSURE** "pressure"
- #define **TELEM_FIELD_BARO_TEMP** "temperature"
- #define **TELEM_FIELD_BARO_ALTITUDE** "altitude"
- #define **TELEM_FIELD_GPS_LAT** "lat"
- #define **TELEM_FIELD_GPS_LON** "lon"
- #define **TELEM_FIELD_GPS_ALT** "alt"
- #define **TELEM_FIELD_GPS_SPEED** "speed"
- #define **TELEM_FIELD_GPS_HEADING** "heading"
- #define **TELEM_FIELD_GPS_SATS** "sats"
- #define **TELEM_FIELD_EKF** "ekf"
- #define **TELEM_FIELD_EKF_ALT** "altitude"
- #define **TELEM_FIELD_EKF_VEL** "velocity"
- #define **TELEM_FIELD_EKF_QUAT** "quaternion"
- #define **TELEM_FIELD_STATE** "state"
- #define **TELEM_FIELD_PHASE** "phase"

7.30.1 Detailed Description

Standardized field names for telemetry JSON messages.
These macros ensure consistency between sender (TelemetryTask) and receiver implementations. Changing a field name only requires updating this file.

7.31 TelemetryFields.h

Go to the documentation of this file.

```
00001 #pragma once
00002
00011
00012 // Root level fields
00013 #define TELEM_FIELD_TIMESTAMP       "timestamp"
00014 #define TELEM_FIELD_DATA_VALID      "dataValid"
00015 #define TELEM_FIELD_IMU             "imu"
00016 #define TELEM_FIELD_BARO1           "baro1"
00017 #define TELEM_FIELD_BARO2           "baro2"
```

```
00018 #define TELEM_FIELD_GPS             "gps"
00019
00020 // IMU sub-fields
00021 #define TELEM_FIELD_IMU_ACCEL       "accel"
00022 #define TELEM_FIELD_IMU_GYRO        "gyro"
00023 #define TELEM_FIELD_IMU_MAG         "mag"
00024
00025 // Vector components (accel, gyro, mag)
00026 #define TELEM_FIELD_VEC_X           "x"
00027 #define TELEM_FIELD_VEC_Y           "y"
00028 #define TELEM_FIELD_VEC_Z           "z"
00029
00030 // Barometer sub-fields
00031 #define TELEM_FIELD_BARO_PRESSURE   "pressure"
00032 #define TELEM_FIELD_BARO_TEMP       "temperature"
00033 #define TELEM_FIELD_BARO_ALTITUDE   "altitude"
00034
00035 // GPS sub-fields
00036 #define TELEM_FIELD_GPS_LAT         "lat"
00037 #define TELEM_FIELD_GPS_LON         "lon"
00038 #define TELEM_FIELD_GPS_ALT         "alt"
00039 #define TELEM_FIELD_GPS_SPEED       "speed"
00040 #define TELEM_FIELD_GPS_HEADING     "heading"
00041 #define TELEM_FIELD_GPS_SATS        "sats"
00042
00043 // Optional: EKF/Filter fields (if you want to add filtered data)
00044 #define TELEM_FIELD_EKF             "ekf"
00045 #define TELEM_FIELD_EKF_ALT         "altitude"
00046 #define TELEM_FIELD_EKF_VEL         "velocity"
00047 #define TELEM_FIELD_EKF_QUAT        "quaternion"
00048
00049 // Flight state (optional - can be added later)
00050 #define TELEM_FIELD_STATE           "state"
00051 #define TELEM_FIELD_PHASE           "phase"
```

7.32 GPS.hpp

```
00001 #pragma once
00002
00003 #include <ISensor.hpp>
00004 #include <Wire.h>
00005 #include <SparkFun_u-blox_GNSS_Arduino_Library.h>
00006 #include <config.h>
00007
00024 class GPS : public ISensor
00025 {
00026 public:
00030     GPS();
00031
00040     bool init() override;
00041
00054     std::optional<SensorData> getData() override;
00055
00056 private:
00060     SFE_UBLOX_GNSS myGNSS;
00061 };
```

7.33 KalmanFilter.hpp

```
00001 #ifndef KALMANFILTER_HPP
00002 #define KALMANFILTER_HPP
00003
00004 // Macros needed for a conflict between similar macro variables names of Arduino.h and Eigen.h
00005 #ifdef B1
00006 #undef B1
00007 #endif
00008 #ifdef B2
00009 #undef B2
00010 #endif
00011 #ifdef B3
00012 #undef B3
00013 #endif
00014 #ifdef B0
00015 #undef B0
00016 #endif
00017
00018 #define EKF_N 16 // Size of state space [3-positions, 3-velocities, 3-accelerations, 4-quaternion_rot]

00019 #define EKF_M 6 // Size of observation (measurement) space [3-positions, 3-accelerations,
      4-quaternion_rot]
00020
```

```
00021 #include <tinyekf.h>
00022 #include <cmath>
00023 #include <math.h>
00024 #include <stdlib.h>
00025 #include <random>
00026 #include <ArduinoEigen.h>
00027 #include "esp_task_wdt.h"
00028
00038 class KalmanFilter {
00039 public:
00045     KalmanFilter(Eigen::Vector3f gravity_value, Eigen::Vector3f magnometer_value);
00046
00054     std::vector<std::vector<float» step(float dt, float omega[3], float accel[3]);
00055
00060     float* state();
00061
00062 private:
00063     ekf_t ekf;
00064
00065     const float P0 = 1e-4;
00066     const float V0 = 1e-4;
00067     const float q_a = 1e-8;
00068     const float b_a = 1e-8;
00069     const float b_g = 1e-8;
00070
00071     // R matrix (measurements), obtain this value from a comparison between a model and the real data.
00072     const float A0 = 1e-3;
00073     const float G0 = 1e-5;
00074
00075     // Process noise covariance
00076     float Q_diag[EKF_N] = {
00077         P0, P0, P0,
00078         V0, V0, V0,
00079         q_a, q_a, q_a, q_a,
00080         b_a, b_a, b_a,
00081         b_g, b_g, b_g
00082     };
00083
00084     // Measurement noise covariance
00085     float R[EKF_M * EKF_M] = {
00086         A0, 0, 0, 0, 0, 0,
00087         0, A0, 0, 0, 0, 0,
00088         0, 0, A0, 0, 0, 0,
00089         0, 0, 0, G0, 0, 0,
00090         0, 0, 0, 0, G0, 0,
00091         0, 0, 0, 0, 0, G0
00092
00093     };
00094
00095     // Initially, the acceleration is constantly zero, so it won't change
00096     float H[EKF_M*EKF_N] = {
00097         0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
00098         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
00099         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
00100         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
00101         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
00102         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1
00103     };
00104
00105     // Measurement Jacobian with input/output relations
00106     float F[EKF_N*EKF_N];\
00107
00108     // Gravity vector in ENU coordinates
00109     const Eigen::Vector3f gravity{0, 0, -9.81};
00110
00124     std::tuple<Eigen::Quaternionf, Eigen::Vector3f, Eigen::Vector3f> calibration(Eigen::Vector3f
      gravity_value, Eigen::Vector3f magnetometer_value);
00125
00126     Eigen::Vector3f rotateToBody(const Eigen::Quaternionf& q, const Eigen::Vector3f& vec_world);
00127
00128     // Compute H_q^{(a)} numerically
00136     Eigen::Matrix<float, 3, 4> computeHqAccelJacobian(const Eigen::Quaternionf& q_nominal, const
      Eigen::Vector3f& accel_world, float epsilon = 1e-5);
00137
00141     void run_model(float dt, float fx[EKF_N], float hx[EKF_M], float omega_x, float omega_y, float
      omega_z, float accel_z[3]);
00142
00146     void computeJacobianF_tinyEKF(float dt, float omega_x, float omega_y, float omega_z, float
      accel_z[3], float F_out[EKF_N * EKF_N]);
00147 };
00148
00149 #endif // KALMANFILTER_HPP
```

7.34 KalmanFilter1D.hpp

```
00001 #ifndef KALMANFILTER1D_HPP
00002 #define KALMANFILTER1D_HPP
00003
00004 // Macros needed for a conflict between similar macro variables names of Arduino.h and Eigen.h
00005 #ifdef B1
00006 #undef B1
00007 #endif
00008 #ifdef B2
00009 #undef B2
00010 #endif
00011 #ifdef B3
00012 #undef B3
00013 #endif
00014 #ifdef B0
00015 #undef B0
00016 #endif
00017
00018 #define EKF_N 6 // Size of state space [1-position, 1-velocity, 4-quaternion_rot]
00019 #define EKF_M 8 // Size of measurement space [3-accelerations, 3-angular velocities, 1-GPS,
      1-pressure]
00020
00021 #include <tinyekf.h>
00022 #include <cmath>
00023 #include <math.h>
00024 #include <stdlib.h>
00025 #include <random>
00026 #include <vector>
00027 #include <ArduinoEigen.h>
00028 #include "esp_task_wdt.h"
00029 #include "config.h"
00030
00039 class KalmanFilter1D {
00040 public:
00047     KalmanFilter1D(Eigen::Vector3f gravity_value, Eigen::Vector3f magnometer_value);
00048
00057     void step(float dt, float omega[3], float accel[3], float pressure, float gps);
00058
00064     float* state();
00065
00066     void setParameters(float p0, float v0, float qa, float a0, float g0) {
00067         P0 = p0; V0 = v0; q_a = qa; A0 = a0; G0 = g0;
00068         updateCovarianceMatrices();
00069     }
00070
00071     void updateCovarianceMatrices() {
00072         // Update Q matrix (process noise)
00073         Q[0] = P0;     // Q[0,0]
00074         Q[7] = V0;     // Q[1,1]
00075         Q[14] = q_a;   // Q[2,2]
00076         Q[21] = q_a;   // Q[3,3]
00077         Q[28] = q_a;   // Q[4,4]
00078         Q[35] = q_a;   // Q[5,5]
00079
00080         // Update R matrix (measurement noise) - initialize with zeros first
00081         for (int i = 0; i < EKF_M*EKF_M; i++) R[i] = 0.0f;
00082
00083         // Set diagonal elements
00084         R[0] = A0;    // accel_x variance
00085         R[9] = A0;    // accel_y variance
00086         R[18] = A0;   // accel_z variance
00087         R[27] = G0;   // gyro_x variance
00088         R[36] = G0;   // gyro_y variance
00089         R[45] = G0;   // gyro_z variance
00090         R[54] = 1.0f; // pressure variance (you may want to make this configurable)
00091         R[63] = GPS_Z0; // GPS variance
00092     }
00093
00094 private:
00095         // TinyEKF structure for the Extended Kalman Filter
00096     ekf_t ekf;
00097
00098     // Make these non-const so they can be modified
00099     float P0 = 1e-6f;      // position (m^2)
00100     float V0 = 1e-7f;      // velocity (m^2/s^2)
00101     float q_a = 1e-8f;     // quaternion
00102
00103     // How much we trust the Sensors
00104     float A0 = 1e-6f;        // accelerometer (m/s^2)
00105     float G0 = 5e-7f;        // gyroscope (rad/s)
00106     float GPS_Z0 = 3.0f;     // GPS altitude (m^2)
00107
00108     // Bias of the sensors (initialized in the constructor functino with calibration data)
00109     Eigen::Vector3f bias_a; // Bias vector for accelerometer
00110     Eigen::Vector3f bias_g; // Bias vector for gyroscope
```

```
00111     Eigen::Vector3f gravity; // Gravity vector in ENU coordinates
00112
00113     // Process noise covariance
00114     float Q[EKF_N*EKF_N] = {
00115         P0, 0, 0, 0, 0, 0,
00116         0, V0, 0, 0, 0, 0,
00117         0, 0, q_a, 0, 0, 0,
00118         0, 0, 0, q_a, 0, 0,
00119         0, 0, 0, 0, q_a, 0,
00120         0, 0, 0, 0, 0, q_a
00121     };
00122
00123     // Measurement noise covariance
00124     float R[EKF_M*EKF_M];
00125
00126     // Initially, the acceleration is constantly zero, so it won't change
00127     float H[EKF_M*EKF_N] = {
00128         0, 0, 0, 0, 0, 0,
00129         0, 0, 0, 0, 0, 0,
00130         0, 0, 0, 0, 0, 0,
00131         0, 0, 0, 0, 0, 0,
00132         0, 0, 0, 0, 0, 0,
00133         0, 0, 0, 0, 0, 0,
00134         1, 0, 0, 0, 0, 0,
00135         1, 0, 0, 0, 0, 0
00136     };
00137
00138     // Measurement Jacobian with input/output relations
00139     float F[EKF_N*EKF_N];
00140
00148     std::tuple<Eigen::Quaternionf, Eigen::Vector3f, Eigen::Vector3f> calibration(Eigen::Vector3f
      gravity_value, Eigen::Vector3f magnetometer_value);
00149
00161     void run_model(float dt, float fx[EKF_N], float hx[EKF_M], float omega_z[3], float accel_z[3],
      float pressure, float z_gps);
00162
00172     void computeJacobianF_tinyEKF(float dt, float omega_z[3], float accel_z[3], float
      h_pressure_sensor, float z_gps);
00173
00174     /*
00175         SUPPORT FUNCTIONS
00176     */
00177
00184     float estimateBaroVar(float v);
00185
00198     float pressureToAltitude(
00199         float pressure,
00200         float seaLevelPressurePa = 101325.0,
00201         float T0 = 288.15,
00202         float L = 0.0065,
00203         float g0 = 9.80665,
00204         float R = 8.31447,
00205         float M = 0.0289644);
00206
00214     Eigen::Vector3f rotateToBody(const Eigen::Quaternionf& q, const Eigen::Vector3f& vec_world);
00215 };
00216
00217 #endif // KALMANFILTER1D_HPP
```

7.35 LIS3DHTRSensor.hpp

```
00001 #pragma once
00002 #include <ISensor.hpp>
00003 #include <LIS3DHTR.h>
00004 #include <Wire.h>
00005 #include <config.h>
00006
00007 class LIS3DHTRSensor : public ISensor
00008 {
00009 public:
00010     LIS3DHTRSensor();
00011     bool init() override;
00012     std::optional<SensorData> getData() override;
00013
00014 private:
00015     LIS3DHTR<TwoWire> lis;
00016 };
```

7.36 ILogger.hpp

```
00001 // LoggerInterface.h
```

```
00002 #pragma once
00003
00004 #include <string>
00005 #include <nlohmann/json.hpp>
00006 #include "LogData.hpp"
00007 #include <SensorData.hpp>
00008
00009 using json = nlohmann::json;
00010
00015 class ILogger {
00016 protected:
00017     // Vector to store logged sensor data
00018     std::vector<LogData> logDataList;
00019 public:
00025     int getLogCount() const {
00026         return logDataList.size();
00027     }
00028
00034     virtual void logInfo(const std::string& message) = 0;
00035
00041     virtual void logWarning(const std::string& message) = 0;
00042
00048     virtual void logError(const std::string& message) = 0;
00049
00055     virtual void logSensorData(const SensorData sensorData) = 0;
00056
00063     virtual void logSensorData(const std::string& sensorName, const SensorData sensorData) = 0;
00064
00070     virtual json getJSONAll() const = 0;
00071
00076     virtual void clearData() {
00077         logDataList.clear();
00078     }
00079
00083     virtual ~ILogger() = default;
00084 };
00085
00086
```

7.37 LogData.hpp

```
00001 #pragma once
00002
00003 #include <string>
00004 #include <nlohmann/json.hpp>
00005 #include <SensorData.hpp>
00006 #include <ILoggable.hpp>
00007
00008 using json = nlohmann::json;
00009
00014 class LogData {
00015 private:
00016     // Origin of the log
00017     std::string source;
00018
00019     // Data to be logged
00020     const ILoggable* data;
00021
00022 public:
00029     LogData(const std::string& source, const ILoggable* data) : source(source), data(data) {}
00030
00036     std::string getSource() const { return this->source; }
00037
00043     const ILoggable* getData() const { return this->data; }
00044
00050     json toJSON() const {
00051         json j;
00052         j["type"] = this->getSource();
00053         j["content"] = this->data->toJSON();
00054         return j;
00055     }
00056 };
00057
00058
```

7.38 E220LoRaTransmitter.hpp

```
00001 #pragma once
00002 #include <ITransmitter.hpp>
00003 #include <Packet.hpp>
00004 #include <config.h>
```

```
00005 #include <variant>
00006 #include <LoRa_E220.h>
00007 #include <HardwareSerial.h>
00008 #include <cstdint>
00009
00010 using TransmitDataType = std::variant<char *, String, std::string, nlohmann::json>;
00011
00016 class E220LoRaTransmitter : public ITransmitter<TransmitDataType>
00017 {
00018 public:
00027 #ifdef TRANSMITTER_CONFIG_MODE_ENABLE
00028     E220LoRaTransmitter(HardwareSerial &serial, byte auxPin, byte m0Pin, byte m1Pin) :
      transmitter(&serial, auxPin, m0Pin, m1Pin, UART_BPS_RATE_9600) {};
00029 #else
00030     E220LoRaTransmitter(HardwareSerial &serial, byte auxPin, byte m0Pin, byte m1Pin) :
      transmitter(&serial, auxPin, m0Pin, m1Pin, UART_BPS_RATE_115200) {};
00031 #endif
00038 #ifdef TRANSMITTER_CONFIG_MODE_ENABLE
00039     E220LoRaTransmitter(HardwareSerial &serial, byte auxPin) : transmitter(&serial, auxPin, -1, -1,
      UART_BPS_RATE_9600) {};
00040 #else
00041     E220LoRaTransmitter(HardwareSerial &serial, byte auxPin) : transmitter(&serial, auxPin, -1, -1,
      UART_BPS_RATE_115200) {};
00042 #endif
00048 #ifdef TRANSMITTER_CONFIG_MODE_ENABLE
00049     E220LoRaTransmitter(HardwareSerial &serial) : transmitter(&serial, -1, -1, -1, UART_BPS_RATE_9600)
      {};
00050 #else
00051     E220LoRaTransmitter(HardwareSerial &serial) : transmitter(&serial, -1, -1, -1,
      UART_BPS_RATE_115200) {};
00052 #endif
00058     ResponseStatusContainer init() override;
00059
00066     ResponseStatusContainer init(Configuration config);
00067
00074     ResponseStatusContainer transmit(TransmitDataType data) override;
00075
00082     ResponseStatusContainer configure(Configuration configuration);
00083
00089     ResponseStructContainer getConfiguration();
00090
00097     String getConfigurationString(Configuration configuration) const;
00098
00099 private:
00100     LoRa_E220 transmitter;
00101     //* NOTE: These methods might be moved to a utility class if they are needed elsewhere.
00102
00108     UART_BPS_RATE getBpsRate() const;
00114     UART_BPS_TYPE getBpsType() const;
00115
00124     template <typename T>
00125     T getBpsValue(const std::map<int, T> &baudRateMap, T defaultValue) const;
00126     uint16_t packetNumber = 1;
00127 };
```

7.39 LoRaConfigurationDeserializer.hpp

```
00001 #pragma once
00002
00003 #include <nlohmann/json.hpp>
00004 #include <LoRa_E220.h>
00005 #include "utils/logger/ILogger.hpp"
00006 #include <FS.h> // Include the appropriate header for the File class
00007
00012 class LoRaConfigurationDeserializer
00013 {
00014 public:
00015     LoRaConfigurationDeserializer(Configuration configuration, ILogger *logger) :
      configuration(configuration), logger(logger) {};
00016     LoRaConfigurationDeserializer(ILogger *logger) : logger(logger)
00017     {
00018         // Set default configuration
00019         configuration.ADDL = 0x02;
00020         configuration.ADDH = 0x00;
00021
00022         configuration.CHAN = 23;
00023
00024         configuration.SPEED.uartBaudRate = UART_BPS_9600;
00025         configuration.SPEED.airDataRate = AIR_DATA_RATE_010_24;
00026         configuration.SPEED.uartParity = MODE_00_8N1;
00027
00028         configuration.OPTION.subPacketSetting = SPS_200_00;
00029         configuration.OPTION.RSSIAmbientNoise = RSSI_AMBIENT_NOISE_DISABLED;
00030         configuration.OPTION.transmissionPower = POWER_22;
```

```
00031
00032         configuration.TRANSMISSION_MODE.enableRSSI = RSSI_ENABLED;
00033         configuration.TRANSMISSION_MODE.fixedTransmission = FT_FIXED_TRANSMISSION;
00034         configuration.TRANSMISSION_MODE.enableLBT = LBT_DISABLED;
00035         configuration.TRANSMISSION_MODE.WORPeriod = WOR_2000_011;
00036     };
00037
00045     bool isJsonValid(const nlohmann::json &json) const
00046     {
00047         return !(json.empty() || json.is_null() || !json.is_object() || json.size() == 0 ||
00048                  json["ADDH"].is_null() || json["ADDL"].is_null() || json["CHAN"].is_null() ||);
00049     }
00050
00057     bool deserializeConfiguration(const nlohmann::json &json)
00058     {
00059         if (!isJsonValid(json))
00060         {
00061             logger->logError("[E220 Module] Failed to deserialize configuration from JSON object. JSON
      object is empty or invalid.");
00062             return false;
00063         }
00064
00065         configuration.COMMAND = json["COMMAND"];
00066         configuration.STARTING_ADDRESS = json["STARTING_ADDRESS"];
00067         configuration.LENGHT = json["LENGHT"];
00068         configuration.ADDH = json["ADDH"];
00069         configuration.ADDL = json["ADDL"];
00070         configuration.CHAN = json["CHAN"];
00071
00072         configuration.SPEED.airDataRate = json["SPEED"]["airDataRate"];
00073         configuration.SPEED.uartParity = json["SPEED"]["uartParity"];
00074         configuration.SPEED.uartBaudRate = json["SPEED"]["uartBaudRate"];
00075
00076         configuration.OPTION.transmissionPower = json["OPTION"]["transmissionPower"];
00077         configuration.OPTION.reserved = json["OPTION"]["reserved"];
00078         configuration.OPTION.RSSIAmbientNoise = json["OPTION"]["RSSIAmbientNoise"];
00079         configuration.OPTION.subPacketSetting = json["OPTION"]["subPacketSetting"];
00080
00081         configuration.TRANSMISSION_MODE.WORPeriod = json["TRANSMISSION_MODE"]["WORPeriod"];
00082         configuration.TRANSMISSION_MODE.reserved2 = json["TRANSMISSION_MODE"]["reserved2"];
00083         configuration.TRANSMISSION_MODE.enableLBT = json["TRANSMISSION_MODE"]["enableLBT"];
00084         configuration.TRANSMISSION_MODE.reserved = json["TRANSMISSION_MODE"]["reserved"];
00085         configuration.TRANSMISSION_MODE.fixedTransmission =
      json["TRANSMISSION_MODE"]["fixedTransmission"];
00086         configuration.TRANSMISSION_MODE.enableRSSI = json["TRANSMISSION_MODE"]["enableRSSI"];
00087
00088         configuration.CRYPT.CRYPT_H = json["CRYPT"]["CRYPT_H"];
00089         configuration.CRYPT.CRYPT_L = json["CRYPT"]["CRYPT_L"];
00090
00091         return true;
00092     }
00099     bool deserializeConfiguration(File &file)
00100     {
00101         if (!file)
00102         {
00103             return false;
00104         }
00105
00106         size_t size = file.size();
00107         if (size == 0)
00108         {
00109             String error = "[E220 Module] JSON file is empty: " + String(file.name());
00110             logger->logError(error.c_str());
00111             return false;
00112         }
00113
00114         // Read the file content into a string
00115         std::ostringstream contentStream;
00116         contentStream << file.readString().c_str();
00117         std::string content = contentStream.str();
00118
00119         nlohmann::json json;
00120         try
00121         {
00122             json = nlohmann::json::parse(content);
00123         }
00124         catch (nlohmann::json::parse_error &e)
00125         {
00126             String error = "[E220 Module] Failed to parse JSON file: " + String(file.name()) + ".
      Error: " + e.what();
00127             logger->logError(error.c_str());
00128             return false;
00129         }
00130         file.close();
00131         return deserializeConfiguration(json);
00132     }
00133
```

```
00139     Configuration getConfiguration() const
00140     {
00141         return configuration;
00142     }
00143
00144 private:
00145     Configuration configuration;
00146     ILogger *logger;
00147 };
00148
```

7.40 SX1261LoRaTransmitter.hpp

```
00001 #pragma once
00002
00003 #include <RadioLib.h>
00004 #include <nlohmann/json.hpp>
00005 #include <variant>
00006 #include <string>
00007 #include <Arduino.h>
00008 #include <ITransmitter.hpp>
00009 #include <ResponseStatusContainer.hpp>
00010
00011 using TransmitDataType = std::variant<char *, String, std::string, nlohmann::json>;
00012
00021 class LoRaTransmitter : public ITransmitter<TransmitDataType>
00022 {
00023 private:
00024     // Pin configuration per SX1261
00025     static constexpr int LORA_NSS_PIN = 19;     // Chip Select
00026     static constexpr int LORA_DIO1_PIN = 2;  // DIO1 (interrupt)
00027     static constexpr int LORA_NRST_PIN = 14;  // Reset
00028     static constexpr int LORA_BUSY_PIN = 4;   // Busy
00029
00030     SX1261 radio;
00031     bool initialized;
00032     uint16_t packetCounter;
00033
00034     // Configurazione LoRa (modificabile se necessario)
00035     float frequency = 868.0;        // MHz (Europa)
00036     int8_t power = 14;              // dBm
00037     float bandwidth = 125.0;        // kHz
00038     uint8_t spreadingFactor = 7;    // SF7-SF12
00039     uint8_t codingRate = 5;         // 4/5
00040     uint8_t syncWord = 0x12;        // Private network
00041
00047     std::string dataToString(const TransmitDataType& data) {
00048         std::string result;
00049
00050         std::visit([&result](const auto& value) {
00051             using T = std::decay_t<decltype(value)>;
00052
00053             if constexpr (std::is_same_v<T, char*>) {
00054                 result = std::string(value);
00055             }
00056             else if constexpr (std::is_same_v<T, String>) {
00057                 result = std::string(value.c_str());
00058             }
00059             else if constexpr (std::is_same_v<T, std::string>) {
00060                 result = value;
00061             }
00062             else if constexpr (std::is_same_v<T, nlohmann::json>) {
00063                 result = value.dump(); // Serializza JSON in stringa
00064             }
00065         }, data);
00066
00067         return result;
00068     }
00069
00070 public:
00074     LoRaTransmitter()
00075         : radio(new Module(LORA_NSS_PIN, LORA_DIO1_PIN, LORA_NRST_PIN, LORA_BUSY_PIN))
00076         , initialized(false)
00077         , packetCounter(0)
00078     {
00079     }
00080
00085     ResponseStatusContainer init() override {
00086         Serial.print("Initializing SX1261... ");
00087
00088         // Initialize the module
00089         int state = radio.begin();
00090         if (state != RADIOLIB_ERR_NONE) {
00091             Serial.print("failed, code: ");
```

```
00092             Serial.println(state);
00093             return ResponseStatusContainer(state, String("Errore inizializzazione SX1261, codice: ") +
      String(state));
00094         }
00095
00096         if (radio.setFrequency(frequency) != RADIOLIB_ERR_NONE) {
00097             return ResponseStatusContainer(01, "Errore impostazione frequenza");
00098         }
00099         Serial.println("Frequenza: " + String(frequency) + " MHz");
00100
00101         if (radio.setOutputPower(power) != RADIOLIB_ERR_NONE) {
00102             return ResponseStatusContainer(02, "Errore impostazione potenza");
00103         }
00104         Serial.println("Potenza: " + String(power) + " dBm");
00105
00106         if (radio.setBandwidth(bandwidth) != RADIOLIB_ERR_NONE) {
00107             return ResponseStatusContainer(03, "Errore impostazione bandwidth");
00108         }
00109         Serial.println("Bandwidth: " + String(bandwidth) + " kHz");
00110
00111         if (radio.setSpreadingFactor(spreadingFactor) != RADIOLIB_ERR_NONE) {
00112             return ResponseStatusContainer(04, "Errore impostazione spreading factor");
00113         }
00114         Serial.println("Spreading Factor: SF" + String(spreadingFactor));
00115
00116         if (radio.setCodingRate(codingRate) != RADIOLIB_ERR_NONE) {
00117             return ResponseStatusContainer(05, "Errore impostazione coding rate");
00118         }
00119         Serial.println("Coding Rate: 4/" + String(codingRate));
00120
00121         if (radio.setSyncWord(syncWord) != RADIOLIB_ERR_NONE) {
00122             return ResponseStatusContainer(06, "Errore impostazione sync word");
00123         }
00124         Serial.println("Sync Word: 0x" + String(syncWord, HEX));
00125
00126         initialized = true;
00127         Serial.println("LoRa SX1261 inizializzato correttamente!");
00128
00129         return ResponseStatusContainer(00, "SX1261 inizializzato correttamente");
00130     }
00131
00137     ResponseStatusContainer transmit(TransmitDataType data) override {
00138         if (!initialized) {
00139             return ResponseStatusContainer(99, "LoRa not initialized");
00140         }
00141
00142         // Converts the data to a string
00143         std::string dataStr = dataToString(data);
00144
00145         // Crea un pacchetto con header per identificazione
00146         nlohmann::json packet = {
00147             {"id", packetCounter++},
00148             {"timestamp", millis()},
00149             {"payload", dataStr}
00150         };
00151
00152         std::string packetStr = packet.dump();
00153
00154         Serial.print("Lora transmission (");
00155         Serial.print(packetStr.length());
00156         Serial.print(" bytes): ");
00157
00158         // Mostra solo i primi 100 caratteri per evitare spam
00159         if (packetStr.length() > 100) {
00160             Serial.print(packetStr.substr(0, 100).c_str());
00161             Serial.println("...");
00162         } else {
00163             Serial.println(packetStr.c_str());
00164         }
00165
00166         // Trasmetti
00167         int state = radio.transmit(packetStr.c_str());
00168
00169         if (state == RADIOLIB_ERR_NONE) {
00170             Serial.println("Lora transmission completed!");
00171             return ResponseStatusContainer(state, String("Transmission completed, ID: ") +
      String(packetCounter - 1));
00172         } else {
00173             Serial.print("Lora transmission error, code: ");
00174             Serial.println(state);
00175             return ResponseStatusContainer(state, String("Transmission error, code: ") +
      String(state));
00176         }
00177     }
00178
00184     ResponseStatusContainer transmitCompact(const nlohmann::json& data) {
00185         if (!initialized) {
```

```
00186             return ResponseStatusContainer(99, "LoRa not initialized");
00187         }
00188 
00189         // Estrae solo i dati critici per ridurre la dimensione del pacchetto
00190         nlohmann::json compactData = {
00191             {"id", packetCounter++},
00192             {"t", millis()}, // timestamp abbreviato
00193         };
00194 
00195         // Cerca i dati BNO055 (IMU principale)
00196         for (const auto& sensor : data) {
00197             if (sensor.contains("content") && sensor["content"].contains("source")) {
00198                 std::string source = sensor["content"]["source"];
00199 
00200                 if (source == "BNO055") {
00201                     const auto& sensorData = sensor["content"]["sensorData"];
00202                     if (sensorData.contains("accelerometer")) {
00203                         compactData["acc"] = {
00204                             sensorData["accelerometer"]["x"],
00205                             sensorData["accelerometer"]["y"],
00206                             sensorData["accelerometer"]["z"]
00207                         };
00208                     }
00209                     if (sensorData.contains("angular_velocity")) {
00210                         compactData["gyr"] = {
00211                             sensorData["angular_velocity"]["x"],
00212                             sensorData["angular_velocity"]["y"],
00213                             sensorData["angular_velocity"]["z"]
00214                         };
00215                     }
00216                     if (sensorData.contains("board_temperature")) {
00217                         compactData["temp"] = sensorData["board_temperature"];
00218                     }
00219                 }
00220                 else if (source == "BAR1") {
00221                     const auto& sensorData = sensor["content"]["sensorData"];
00222                     if (sensorData.contains("pressure")) {
00223                         compactData["pres"] = sensorData["pressure"];
00224                     }
00225                 }
00226                 else if (source == "Voltage") {
00227                     const auto& sensorData = sensor["content"]["sensorData"];
00228                     if (sensorData.contains("Voltage")) {
00229                         compactData["bat"] = sensorData["Voltage"];
00230                     }
00231                 }
00232                 else if (source == "MainActuators") {
00233                     const auto& sensorData = sensor["content"]["sensorData"];
00234                     if (sensorData.contains("State")) {
00235                         compactData["main"] = (sensorData["State"] == "ON") ? 1 : 0;
00236                     }
00237                 }
00238                 else if (source == "DrogueActuators") {
00239                     const auto& sensorData = sensor["content"]["sensorData"];
00240                     if (sensorData.contains("State")) {
00241                         compactData["drogue"] = (sensorData["State"] == "ON") ? 1 : 0;
00242                     }
00243                 }
00244             }
00245         }
00246 
00247         std::string compactStr = compactData.dump();
00248 
00249         Serial.print("Compact transmission (");
00250         Serial.print(compactStr.length());
00251         Serial.print(" bytes): ");
00252         Serial.println(compactStr.c_str());
00253 
00254         // Trasmetti
00255         int state = radio.transmit(compactStr.c_str());
00256 
00257         if (state == RADIOLIB_ERR_NONE) {
00258             Serial.println("Compact transmission completed!");
00259             return ResponseStatusContainer(state, "Compact transmission completed");
00260         } else {
00261             Serial.print("Compact transmission error, code: ");
00262             Serial.println(state);
00263             return ResponseStatusContainer(state, String("Compact transmission error, code: ") +
      String(state));
00264         }
00265     }
00266 
00275     void configure(float freq, int8_t pwr, float bw, uint8_t sf, uint8_t cr) {
00276         frequency = freq;
00277         power = pwr;
00278         bandwidth = bw;
00279         spreadingFactor = sf;
```

```
00280         codingRate = cr;
00281     }
00282
00287     uint16_t getPacketCount() const {
00288         return packetCounter;
00289     }
00290
00294     void printStats() {
00295         if (!initialized) {
00296             Serial.println("LoRa non inizializzato");
00297             return;
00298         }
00299         Serial.println("=== LoRa SX1261 Statistics ===");
00300     }
00301 };
```

7.41 MPRLSSensor.hpp

```
00001 #pragma once
00002 #include <ISensor.hpp>
00003 #include <Adafruit_MPRLS.h>
00004
00005 class MPRLSSensor : public ISensor
00006 {
00007 public:
00008     MPRLSSensor();
00009     bool init() override;
00010     std::optional<SensorData> getData() override;
00011
00012 private:
00013     Adafruit_MPRLS mprls;
00014     float pressure;
00015 };
```

7.42 MS561101BA03.hpp

```
00001 #pragma once
00002 #include <ISensor.hpp>
00003 #include <Wire.h>
00004
00005 // MS5611 Commands
00006 #define MS5611_CMD_RESET        0x1E
00007 #define MS5611_CMD_CONV_D1_256  0x40
00008 #define MS5611_CMD_CONV_D1_512  0x42
00009 #define MS5611_CMD_CONV_D1_1024 0x44
00010 #define MS5611_CMD_CONV_D1_2048 0x46
00011 #define MS5611_CMD_CONV_D1_4096 0x48
00012 #define MS5611_CMD_CONV_D2_256  0x50
00013 #define MS5611_CMD_CONV_D2_512  0x52
00014 #define MS5611_CMD_CONV_D2_1024 0x54
00015 #define MS5611_CMD_CONV_D2_2048 0x56
00016 #define MS5611_CMD_CONV_D2_4096 0x58
00017 #define MS5611_CMD_ADC_READ     0x00
00018 #define MS5611_CMD_PROM_READ    0xA0
00019
00020 class MS561101BA03 : public ISensor
00021 {
00022 public:
00023     MS561101BA03(uint8_t address = 0x77);
00024     bool init() override;
00025     std::optional<SensorData> getData() override;
00026
00027 private:
00028     uint8_t _address;
00029     uint16_t _calibrationData[8];
00030
00031     bool readCalibrationData();
00032     void reset();
00033     uint32_t readADC();
00034     uint16_t readPROM(uint8_t address);
00035     void writeCommand(uint8_t command);
00036     uint32_t readRawPressure();
00037     uint32_t readRawTemperature();
00038     void calculatePressureAndTemperature(uint32_t D1, uint32_t D2, float& pressure, float&
      temperature);
00039
00040     float _pressure;
00041     float _temperature;
00042 };
```

7.43 Packet.hpp

```
00001 #pragma once
00002
00003 #include <cstddef>
00004 #include <cstdint>
00005
00006 // Fixed packet size for transmission (optimized for 64-byte TelemetryPacket)
00007 // 70 bytes = Header(7) + Payload(57) + CRC(2) + Padding(4) = 3x faster than 250 bytes
00008 constexpr size_t FIXED_PACKET_SIZE = 70;
00009 // Maximum raw packet size we assume for transmit buffers (including header and CRC)
00010 constexpr size_t MAX_PACKET_SIZE = FIXED_PACKET_SIZE;
00011 constexpr size_t RESERVED_BYTES = 0;
00012
00013 /* 2 bytes for CRC */
00014 constexpr size_t CRC_SIZE = sizeof(uint16_t);
00015 /* HEADER size + PAYLOAD size + crc size */
00016 constexpr size_t MAX_TX_PACKET_SIZE = MAX_PACKET_SIZE - RESERVED_BYTES;
00017
00018 // NOTE: All multi-byte integer fields are encoded in little-endian order on the
00019 // wire. This matches the common endianness of most microcontrollers used with
00020 // PlatformIO/Arduino. If you change the target architecture, document any
00021 // endianness differences here.
00022
00023 // The CRC used by Packet::calculateCRC() below uses a 16-bit polynomial with
00024 // initial value 0xFFFF and polynomial 0xA001 (the reflected representation of
00025 // the CRC-16/ARC / CRC-16-IBM polynomial). The implementation deliberately
00026 // computes the CRC over the packed Packet structure up to (but excluding) the
00027 // `crc` field using offsetof(Packet, crc). If you change fields or their
00028 // ordering, update HEADER_SIZE and be aware this changes CRC coverage.
00029
00030 #pragma pack(push, 1) // Avoid padding inside the packet structures
00031
00043 struct PacketHeader
00044 {
00045     uint16_t messageId = 1;      // 2 bytes - keep at start for natural alignment
00046     uint8_t totalChunks = 0;     // 1 byte - message-level info
00047     uint8_t chunkIndex = 0;      // 1 byte - chunk-level info (pairs logically with totalChunks)
00048     uint8_t payloadSize = 0;     // 1 byte - this packet's payload size
00049     uint8_t flags = 0;           // 1 byte - control bits
00050     uint8_t protocolVersion = 1; // 1 byte - least likely to change during processing
00051 };
00052
00053 constexpr size_t HEADER_SIZE = sizeof(PacketHeader);
00054 constexpr size_t LORA_MAX_PAYLOAD_SIZE =
00055     MAX_TX_PACKET_SIZE - HEADER_SIZE - CRC_SIZE - 4;
00056
00063 struct PacketPayload
00064 {
00065     uint8_t data[LORA_MAX_PAYLOAD_SIZE];
00066 };
00067
00075 struct Packet
00076 {
00077     PacketHeader header;
00078     PacketPayload payload;
00079     uint16_t crc;
00080
00084     void calculateCRC();
00091     void printPacket();
00092 };
00093 #pragma pack(pop) // Restore default alignment
```

7.44 PacketManager.hpp

```
00001 #pragma once
00002
00003 #include <vector>
00004 #include <cstdint>
00005 #include <Packet.hpp>
00006
00007 class PacketManager
00008 {
00009 public:
00016     static std::vector<uint8_t> serialize(const Packet &packet);
00017
00024     static Packet deserialize(const uint8_t *data, size_t length);
00025
00033     static std::vector<Packet> divideMessage(const uint8_t *message, size_t length);
00040     static std::vector<uint8_t> reassembleMessage(const std::vector<Packet> &packets);
00041 };
```

7.45 PacketSerializer.hpp

```
00001 class PacketSerializer {
00002 public:
00003     static std::string serialize(const Packet& packet);
00004 };
```

7.46 RocketLogger.hpp

```
00001 #pragma once
00002
00003 #include <iostream>
00004 #include <string>
00005 #include <vector>
00006 #include <nlohmann/json.hpp>
00007 #include <ILogger.hpp>
00008 #include <LogMessage.hpp>
00009 #include <LogSensorData.hpp>
00010 #include "Logger.hpp"
00011
00012 using json = nlohmann::json;
00013
00018 class RocketLogger : public ILogger {
00019 public:
00023     ~RocketLogger();
00024
00030     void logInfo(const std::string& message) override;
00031
00037     void logWarning(const std::string& message) override;
00038
00044     void logError(const std::string& message) override;
00045
00051     void logSensorData(const SensorData sensorData) override;
00052
00059     void logSensorData(const std::string& sensorName, const SensorData sensorData) override;
00060
00066     json getJSONAll() const override;
00067
00072     void clearData() override;
00073 };
```

7.47 SD-master.hpp

```
00001 #pragma once
00002
00003 #include <variant>
00004 #include <Arduino.h>
00005 #include "SdFat.h"
00006 #include <pins.h>
00007 #include <string>
00008 #include <Logger.hpp>
00009
00010 class SD
00011 {
00012 private:
00013     SdFat SD;
00014     SdFile *file;
00015     bool fileInitialized = false;
00016
00017 public:
00018     bool init();
00019     bool openFile(std::string filename);
00020     bool closeFile();
00021     bool writeFile(std::string filename, std::variant<std::string, String, char *> content);  // se
      true file trovato e scritto, se false file non trovato
00022     bool appendFile(std::string filename, std::variant<std::string, String, char *> content); // se
      true contenuto aggiunto al file, se false errore
00023     char *readFile(std::string filename);                                                     //
      stampa il contenuto del file. ritorna true se tutto ok senno no
00024     bool clearSD();                                                                           //
      cancella tutto il contenuto dell'sd
00025     bool fileExists(std::string filename);                                                    //
      ritorna true se il file esiste, false se non esiste
00026     String readLine();
00027     SdFile* getFile() { return file; }                                                        //
      ritorna il puntatore al file aperto
00028 };
```

7.48 ISensor.hpp

```
00001 #pragma once
00002
00003 #include <SensorData.hpp>
00004 #include <optional>
00005
00010 class ISensor
00011 {
00012 public:
00019     virtual bool init() = 0;
00020
00026     virtual std::optional<SensorData> getData() = 0;
00027
00034     bool isInitialized() const
00035     {
00036         return initialized;
00037     }
00038 protected:
00039     void setInitialized(bool initialized)
00040     {
00041         this->initialized = initialized;
00042     }
00043
00044 private:
00045     bool initialized = false;
00046 };
00047
```

7.49 SensorData.hpp

```
00001 #pragma once
00002
00003 #include <string>
00004 #include <nlohmann/json.hpp>
00005 #include <map>
00006 #include <variant>
00007 #include <optional>
00008
00009 using json = nlohmann::json;
00010 using SensorDataVariant = std::variant<
00011     uint8_t,
00012     int,
00013     unsigned int,
00014     float,
00015     double,
00016     std::string,
00017     std::vector<float>,
00018     std::vector<double>,
00019     std::map<std::string, double>,
00020     std::map<std::string, float>
00021 >;
00022
00027 class SensorData
00028 {
00029 protected:
00030     // Name of the sensor
00031     const std::string sensorName;
00032     // Map to store key-value pairs where values can be of different types
00033     std::map<std::string, SensorDataVariant> dataMap;
00034
00035 public:
00041     SensorData(const std::string &sensorName) : sensorName(sensorName) {}
00042
00049     SensorData& operator=(const SensorData& other)
00050     {
00051         if (this != &other)
00052         {
00053             dataMap = other.dataMap;
00054         }
00055         return *this;
00056     }
00063     void setData(const std::string &key, const SensorDataVariant &value)
00064     {
00065         dataMap[key] = value;
00066     }
00067
00074     std::optional<SensorDataVariant> getData(const std::string &key) const
00075     {
00076         auto it = dataMap.find(key);
00077         if (it != dataMap.end())
00078         {
00079             return it->second;
```

```
00080         }
00081
00082         // Return std::nullopt if key is not found
00083         return std::nullopt;
00084     }
00085
00091     std::map<std::string, SensorDataVariant> getDataMap() const
00092     {
00093         return dataMap;
00094     }
00095
00101     std::string getSensorName() const
00102     {
00103         return sensorName;
00104     }
00105 };
00106
00107
```

7.50 BuzzerController.hpp

```
00001 #pragma once
00002 #include <Arduino.h>
00003 #include <freertos/FreeRTOS.h>
00004 #include <freertos/task.h>
00005
00006 // Buzzer tones (frequencies in Hz)
00007 enum BuzzerTone
00008 {
00009     TONE_OFF = 0,
00010     TONE_LOW = 220,     // Low A
00011     TONE_MID = 440,     // Middle A
00012     TONE_HIGH = 880,    // High A
00013     TONE_ALERT = 1760, // Very high A
00014     TONE_ERROR = 330,  // E - creates dissonance for error sounds
00015     TONE_SUCCESS = 587 // D - pleasant for success sounds
00016 };
00017
00018 // Buzzer patterns
00019 enum BuzzerPattern
00020 {
00021     BUZZER_OFF,
00022     BUZZER_CONTINUOUS,  // Constant tone
00023     BUZZER_SHORT_BEEP,  // Single short beep
00024     BUZZER_DOUBLE_BEEP, // Two short beeps
00025     BUZZER_TRIPLE_BEEP, // Three short beeps
00026     BUZZER_SOS,         // SOS pattern (... --- ...)
00027     BUZZER_WARNING,     // Alternating high/low tone
00028     BUZZER_COUNTDOWN,   // Descending tones
00029     BUZZER_SUCCESS,     // Rising tone sequence
00030     BUZZER_ERROR        // Error pattern
00031 };
00032
00033 // Buzzer Pattern Definition
00034 struct BuzzerSequence
00035 {
00036     BuzzerTone tone;   // Frequency
00037     uint16_t duration; // Duration in ms
00038     uint16_t pause;    // Pause after the tone
00039 };
00040
00041 class BuzzerController
00042 {
00043 private:
00044     // Buzzer pin
00045     uint8_t buzzerPin;
00046
00047     // Current active pattern
00048     TaskHandle_t buzzerTaskHandle;
00049
00050     // Internal task context structure
00051     struct TaskContext
00052     {
00053         BuzzerController *instance;
00054         BuzzerPattern pattern;
00055         BuzzerTone baseTone; // Base tone for the pattern
00056     };
00057
00058     // Static task function for FreeRTOS
00059     static void buzzerTaskFunction(void *param);
00060
00061     // Internal pattern implementation
00062     void playPatternInternal(BuzzerPattern pattern, BuzzerTone baseTone = TONE_MID);
00063
```

```
00064 public:
00065     // Constructor & Destructor
00066     BuzzerController(uint8_t buzzerPin);
00067     ~BuzzerController();
00068
00069     // Initialize pins
00070     void init();
00071
00072     // Control buzzer
00073     void startPattern(BuzzerPattern pattern, BuzzerTone baseTone = TONE_MID);
00074     void stopPattern();
00075
00076     // Simple direct controls
00077     void playTone(BuzzerTone frequency, uint16_t duration);
00078     void playSequence(const BuzzerSequence *sequence, uint8_t count, bool repeat = false);
00079     void setOff();
00080
00081     // Play a single tone
00082     void playToneFreq(uint16_t frequency);
00083     void stopTone();
00084 };
```

7.51 LEDController.hpp

```
00001 #pragma once
00002 #include <Arduino.h>
00003 #include <freertos/FreeRTOS.h>
00004 #include <freertos/task.h>
00005
00006 // Available colors
00007 enum LEDColor
00008 {
00009     ART_LED_OFF,
00010     ART_LED_RED,
00011     ART_LED_GREEN,
00012     ART_LED_BLUE,
00013     ART_LED_YELLOW,
00014     ART_LED_CYAN,
00015     ART_LED_MAGENTA,
00016     ART_LED_WHITE,
00017     ART_LED_ORANGE // Simulated with PWM values
00018 };
00019
00020 // LED pattern definition
00021 struct LEDPattern
00022 {
00023     LEDColor color1;
00024     LEDColor color2;   // For alternating colors
00025     uint16_t duration; // ms on time
00026     uint16_t pause;    // ms pause time
00027     uint8_t times;     // Number of blinks (0 = continuous)
00028     uint8_t auxLeds;   // Bitmask for additional LEDs
00029 };
00030
00031 class LEDController
00032 {
00033 private:
00034     // LED pins
00035     uint8_t redPin;
00036     uint8_t greenPin;
00037     uint8_t bluePin;
00038
00039     // Additional status LEDs
00040     uint8_t statusLedCount;
00041     uint8_t *statusLedPins;
00042
00043     // Current active pattern
00044     TaskHandle_t ledTaskHandle;
00045
00046     // Internal task context structure
00047     struct TaskContext
00048     {
00049         LEDController *instance;
00050         LEDPattern pattern;
00051     };
00052
00053     // RGB LED control
00054     void setRGB(uint8_t r, uint8_t g, uint8_t b);
00055
00056     // Status LEDs control
00057     void setStatusLeds(uint8_t mask);
00058
00059     // LED pattern display helper
00060     void showOutputLED(LEDColor color, uint16_t duration_ms, uint16_t pause_ms, uint8_t times);
```

```
00061
00062     // Static task function for FreeRTOS
00063     static void ledTaskFunction(void *param);
00064
00065 public:
00066     // Constructor & Destructor
00067     LEDController(uint8_t redPin, uint8_t greenPin, uint8_t bluePin);
00068     ~LEDController();
00069
00070     // Add additional status LEDs
00071     void addStatusLeds(uint8_t *pins, uint8_t count);
00072
00073     // Initialize pins
00074     void init();
00075
00076     // Control LED pattern
00077     void startPattern(const LEDPattern &pattern);
00078     void stopPattern();
00079
00080     // Simple direct controls
00081     void setColor(LEDColor color);
00082     void setOff();
00083
00084     // Get RGB values for a color
00085     static void getRGBValues(LEDColor color, uint8_t &r, uint8_t &g, uint8_t &b);
00086 };
```

7.52 StatusManager.hpp

```
00001 #pragma once
00002 #include "LEDController.hpp"
00003 #include "BuzzerController.hpp"
00004 #include <map>
00005
00006 // System status codes
00007 enum SystemCode
00008 {
00009     SYSTEM_OK = 0,
00010     PRE_FLIGHT_MODE,
00011     CALIBRATING,
00012     WAITING_INPUT,
00013     FLIGHT_MODE,
00014     FSM_STARTED,
00015
00016     IMU_FAIL = 20,
00017     IMU_DATA_INVALID,
00018     BARO1_FAIL,
00019     BARO1_DATA_INVALID,
00020     BARO2_FAIL,
00021     BARO2_DATA_INVALID,
00022     GPS_NO_SIGNAL,
00023     GPS_DATA_INVALID,
00024
00025     SD_MOUNT_FAIL = 40,
00026     SD_WRITE_FAIL,
00027     SD_READ_FAIL,
00028
00029     LORA_INIT_FAIL = 50,
00030     LORA_CONFIG_FAIL,
00031     LORA_TX_FAIL,
00032
00033     MEMORY_ERROR = 80,
00034     TASK_FAIL,
00035     MUTEX_ERROR,
00036
00037     POWER_LOW_WARNING = 90,
00038     UNKNOWN_ERROR = 99,
00039
00040     // Additional flight status codes
00041     LAUNCH_READY = 100,
00042     LAUNCH_DETECTED,
00043     APOGEE_DETECTED,
00044     DROGUE_DEPLOY,
00045     MAIN_DEPLOY,
00046     LANDING_DETECTED,
00047     RECOVERY_COMPLETE,
00048
00049     // Test mode status codes
00050     TEST_MENU = 200,
00051     TEST_POWER,
00052     TEST_SENSORS,
00053     TEST_ACTUATORS,
00054     TEST_SD,
00055     TEST_TELEMETRY,
```

```
00056     TEST_ALL,
00057     TEST_SUCCESS,
00058     TEST_FAILURE
00059 };
00060
00061 // Combined pattern definition
00062 struct StatusPattern
00063 {
00064     LEDPattern ledPattern;
00065     BuzzerPattern buzzerPattern;
00066     BuzzerTone buzzerTone;
00067     bool buzzerSync; // Whether to sync buzzer with LED
00068 };
00069
00070 class StatusManager
00071 {
00072 private:
00073     // Controllers
00074     LEDController &ledController;
00075     BuzzerController &buzzerController;
00076
00077     // Current status
00078     SystemCode currentCode;
00079
00080     // Pattern mapping
00081     std::map<SystemCode, StatusPattern> codeToPattern;
00082
00083     // Task handles for special pattern handling
00084     TaskHandle_t statusTaskHandle;
00085
00086     // Internal task context
00087     struct TaskContext
00088     {
00089         StatusManager *instance;
00090         SystemCode code;
00091         StatusPattern pattern;
00092     };
00093
00094     // Static task function for special patterns
00095     static void statusTaskFunction(void *param);
00096
00097 public:
00098     // Constructor
00099     StatusManager(LEDController &ledController, BuzzerController &buzzerController);
00100
00101     // Initialize the pattern map
00102     void init();
00103
00104     // Set system status
00105     void setSystemCode(SystemCode code);
00106
00107     // Get current system code
00108     SystemCode getCurrentCode() const { return currentCode; }
00109
00110     // Get pattern for a system code
00111     const StatusPattern &getPattern(SystemCode code) const;
00112
00113     // Calculate duration of a pattern
00114     uint32_t getPatternDuration(SystemCode code) const;
00115
00116     // Stop any running patterns
00117     void stopAllPatterns();
00118
00119     // Play a blocking pattern (waits for completion)
00120     void playBlockingPattern(SystemCode code, uint32_t minDuration = 0);
00121 };
```

7.53 EspNowTransmitter.hpp

```
00001 #pragma once
00002
00003 #include <ITransmitter.hpp>
00004 #include <Packet.hpp>
00005 #include <esp_now.h>
00006 #include <WiFi.h>
00007 #include <vector>
00008 #include <freertos/FreeRTOS.h>
00009 #include <freertos/semphr.h>
00010
00019 class EspNowTransmitter : public ITransmitter<Packet>
00020 {
00021 public:
00028     EspNowTransmitter(const uint8_t peerMacAddress[6], uint8_t wifiChannel = 1);
00029
```

```
00033     ~EspNowTransmitter();
00034
00042     ResponseStatusContainer init() override;
00043
00053     ResponseStatusContainer transmit(Packet packet) override;
00054
00060     int8_t getLastRSSI() const { return lastRSSI; }
00061
00068     void getStats(uint32_t &sent, uint32_t &failed) const;
00069
00070 private:
00071     uint8_t peerMac[6];
00072     uint8_t channel;
00073     bool initialized;
00074
00075     // Statistics
00076     uint32_t packetsSent;
00077     uint32_t packetsFailed;
00078     int8_t lastRSSI;
00079
00080     // Synchronization
00081     SemaphoreHandle_t sendMutex;
00082     SemaphoreHandle_t sendCompleteSemaphore;
00083     volatile bool sendSuccess;
00084
00085     // ESP-NOW callbacks (must be static)
00086     static void onDataSent(const uint8_t *mac_addr, esp_now_send_status_t status);
00087
00088     // Instance pointer for callback access
00089     static EspNowTransmitter* instance;
00090
00096     bool addPeer();
00097 };
```

7.54 ITransmitter.hpp

```
00001 #pragma once
00002
00003 #include <nlohmann/json.hpp>
00004 #include <variant>
00005 #include <string>
00006 #include <Arduino.h>
00007 #include "ResponseStatusContainer.hpp"
00008
00013 template <typename T>
00014 class ITransmitter
00015 {
00016     public:
00017         virtual ResponseStatusContainer init() = 0;
00018         virtual ResponseStatusContainer transmit(T data) = 0;
00019 };
00020
```

7.55 ResponseStatusContainer.hpp

```
00001 #pragma once
00002
00003 // Simple response container used across transmitters. Placed in top-level
00004 // include/ so it's available during per-library compilation.
00005 class ResponseStatusContainer
00006 {
00007   private:
00008     int code;
00009     String description;
00010   public:
00011     ResponseStatusContainer(int code, String description) : code(code), description(description) {};
00012     int getCode() const { return code; }
00013     String getDescription() const { return description; }
00014 };
```

7.56 ResponseStatusContainer.hpp

```
00001 #pragma once
00007 class ResponseStatusContainer
00008 {
00009     private:
00010         int code;
00011         String description;
```

```
00012     public:
00013         ResponseStatusContainer(int code, String description) : code(code), description(description)
    {};
00014         int getCode() const { return code; }
00015         String getDescription() const { return description; }
00016 };
```

7.57 Termoresistenze.hpp

```
00001 #pragma once
00002 #include <Arduino.h>
00003 #include <ISensor.hpp>
00004 #include <config.h>
00005
00006 #define THERMISTOR_PIN A0
00007 #define SERIES_RESISTOR 1000
00008 #define NOMINAL_RESISTANCE 10000
00009 #define NOMINAL_TEMPERATURE 25.0
00010 #define B_COEFFICIENT 3500
00011
00012 class Termoresistenze : public ISensor
00013 {
00014 public:
00015     Termoresistenze(int pin = THERMISTOR_PIN,
00016                     double seriesResistor = SERIES_RESISTOR,
00017                     double nominalResistance = NOMINAL_RESISTANCE,
00018                     double nominalTemperature = NOMINAL_TEMPERATURE,
00019                     double bCoefficient = B_COEFFICIENT);
00020     bool init() override;
00021     std::optional<SensorData> getData() override;
00022
00023 private:
00024     int thermistorPin;
00025     double seriesResistor;
00026     double nominalResistance;
00027     double nominalTemperature;
00028     double bCoefficient;
00029
00030     double calculateTemperature(int adcValue);
00031 };
```

7.58 Nemesis.hpp

```
00001 // #pragma once
00002
00003 // #include <Arduino.h>
00004 // #include <BNO055Sensor.hpp>
00005 // #include <MPRLSSensor.hpp>
00006 // #include <LIS3DHTRSensor.hpp>
00007 // #include <MS561101BA03.hpp>
00008 // #include <GPS.hpp>
00009 // #include <Termoresistenze.hpp>
00010 // #include <config.h>
00011
00012
00013 // // State machine for actuator sequencing
00014 // enum class ActuatorState {
00015 //     MAIN_ON,
00016 //     MAIN_PAUSE,
00017 //     DROGUE_ON,
00018 //     DROGUE_PAUSE
00019 // };
00020
00021 // class Nemesis {
00022 // public:
00023 //     Nemesis();
00024 //     void init();
00025 //     void run();
00026
00027 //     BNO055Sensor bno;
00028 //     //MPRLSSensor mprls;
00029 //     LIS3DHTRSensor lis3dh;
00030 //     MS561101BA03 ms56_1;
00031 //     MS561101BA03 ms56_2;
00032 //     GPS gps;
00033 //     //Termoresistenze termoresistenze;
00034
00035 //     RocketLogger rocketLogger;
00036
00037 // private:
00038 //     // Timing variables for actuators and buzzer
```

```
00039 //     unsigned long actuatorsTimer;
00040 //     unsigned long actuatorsDuration;
00041 //     unsigned long actuatorsDelay;
00042 //     bool toggleStateMainAct;
00043 //     bool toggleStateDrogueAct;
00044 //     ActuatorState currentActuatorState;
00045
00046 //     unsigned long buzzerTimer;
00047 //     unsigned long buzzerDuration;
00048 //     unsigned long buzzerInterval;
00049 //     bool buzzerPlaying;
00050
00051 //     void readSensors();
00052 //     void controlActuators();
00053 //     void controlBuzzer();
00054 //     void readBattery();
00055 //     void logData();
00056 // };
```

Index