Kestrel Robotics System Repository Overview

CMakeLists.txt and package.xml: Each package includes its own build (CMakeLists.txt) and dependency (package.xml) configurations to support modularity, inter-package communication, and ROS 2 integration.

1. Message Definitions: `src/kestrel_msgs`

Custom ROS 2 message, service, and action definitions used throughout the stack.

`msg/`

- `ObstacleGrid.msg`: 3D map showing where obstacles are in space.
- `ProximityAlert.msg`: Warning message when something gets too close.
- `ObjectTrack.msg`: Information about a single detected object.
- `TrackCentroid.msg`: Center points of grouped objects.
- `SensorReading.msg`: Data from one sensor (like distance or temperature).
- `SensorArray.msg`: Collection of data from multiple sensors.
- `CameraCommand.msg`: Instructions to move the camera up/down/left/right.
- `SystemHealth.msg`: Report on how well all systems are working.
- `FlightStatus.msg`: Current state of the drone (flying, landed, etc.).

`srv/`

- `SetFlightMode.srv`: Switch between different flying modes (auto, manual, etc.).
- `TriggerEmergencyStop.srv`: Immediately stop the drone in an emergency.
- `CalibrateCamera.srv`: Adjust camera settings for better image quality.
- `ResetSensors.srv`: Restart all sensors to fix any issues.
- `GetSystemStatus.srv`: Ask for a full report on system health.

`action/`

– `FollowTarget.action`: Make the drone track and follow a moving object.

– `NavigateToWaypoint.action`: Fly the drone to a specific GPS location.

2. Sensor Fusion Node: `src/kestrel_perception`

Reads 19 sensors and creates obstacle maps plus emergency alerts when things get too close.

`include/kestrel_perception/`

– `sensor_fusion_node.hpp`: Main sensor fusion interface.

– `i2c_driver.hpp`: I2C bus management.

– `uart_driver.hpp`: UART communication.

– `clustering_node.hpp`: DBSCAN clustering.

– `sensor_validator.hpp`: Data validation utilities.

– `spatial_grid.hpp`: Obstacle grid management.

`src/`

– `sensor_fusion_node.cpp`: Fuses 19 sensors into obstacle grid.

– `i2c_driver.cpp`: Multi-device I2C communication.

– `uart_driver.cpp`: UART sensor data handling.

– `clustering_node.cpp`: DBSCAN track clustering.

– `sensor_validator.cpp`: Outlier rejection and filtering.

– `spatial_grid.cpp`: 3D obstacle grid implementation.

`launch/`

– `perception_full.launch.py`: Starts all sensor and perception programs at once.

– `sensors_only.launch.py`: Starts just the sensor reading programs.

– `clustering_only.launch.py`: Starts just the object grouping program.

`config/`

- `sensors.yaml`: Settings for each sensor (addresses, calibration values).
- `dbscan_params.yaml`: Settings for how to group sensor readings together.
- `filter_params.yaml`: Settings for cleaning up noisy sensor data.
- `grid_params.yaml`: Settings for the 3D obstacle map (size, resolution).

`test/`

- `test_sensor_fusion.cpp`: Tests to make sure sensor combining works correctly.
- `test_clustering.cpp`: Tests to verify object grouping algorithms work.
- `test_i2c_driver.cpp`: Tests for sensor communication over I2C.

3. Path Planning: `src/kestrel_planning`

Calculates safe flight paths around obstacles using D* algorithm to avoid crashes.

`include/kestrel_planning/`

- `pathing_node.hpp`: Main program that figures out where the drone should fly.
- `dstar_planner.hpp`: Smart algorithm that finds the best route around obstacles.
- `collision_checker.hpp`: Constantly checks if the planned path is safe.
- `waypoint_manager.hpp`: Manages the sequence of GPS points the drone visits.

`src/`

- `pathing_node.cpp`: Main program that plans safe flight paths.
- `dstar_planner.cpp`: Algorithm that finds efficient routes while avoiding obstacles.
- `collision_checker.cpp`: Continuously verifies the drone won't hit anything.
- `waypoint_manager.cpp`: Creates and validates GPS waypoints for navigation.

`launch/`

- `planning.launch.py`: Starts the path planning program.

config/

- dstar_params.yaml: Settings for the pathfinding algorithm.
- safety_params.yaml: Safety distances and speed limits.
- planning_bounds.yaml: Defines where the drone is allowed to fly.

test/

- test_dstar.cpp: Tests to verify the pathfinding algorithm works correctly.
- test_collision_checker.cpp: Tests for the collision detection system.

4. ArduPilot Translation Node + Dynamic Camera Control: src/kestrel_control

Converts flight plans into drone commands and keeps camera pointed at the target.

include/kestrel_control/

- ardupilot_translator_node.hpp: Converts high-level commands into drone flight instructions.
- dynamic_camera_control_node.hpp: Controls the camera gimbal movement.
- frame_transformer.hpp: Converts between different coordinate systems.
- command_validator.hpp: Checks that flight commands are safe before executing.

src/

- ardupilot_translator_node.cpp: Translates waypoints into MAVLink commands for the autopilot.
- dynamic_camera_control_node.cpp: Controls servo motors that move the camera.
- frame_transformer.cpp: Converts GPS coordinates between different reference frames.
- command_validator.cpp: Ensures flight commands won't put the drone in danger.

python/kestrel_control/

- __init__.py: Makes this folder work as a Python package.
- servo_driver.py: Low-level control of servo motors using PWM signals.
- gpio_manager.py: Controls the input/output pins on the computer.
- pid_controller.py: Smooth control algorithm that reduces jittery movements.

`launch/`

– `control_stack.launch.py`: Starts all control system programs.

– `mavros_bridge.launch.py`: Starts just the autopilot communication program.

– `camera_control.launch.py`: Starts just the camera control program.

`config/`

– `ardupilot_interface.yaml`: Settings for communicating with the autopilot.

– `servo_params.yaml`: Servo motor limits and calibration values.

– `pid_gains.yaml`: Tuning parameters for smooth control.

– `frame_transforms.yaml`: Coordinate system conversion settings.

`test/`

– `test_frame_transformer.cpp`: Tests for coordinate system conversions.

– `test_command_validator.cpp`: Tests to verify command safety checking.

– `test_servo_driver.py`: Tests for servo motor control.

5. Base Station Communication + Emergency Protocols: `src/kestrel_communication`

Handles ground station commands and triggers emergency stops when things go wrong.

`src/`

– `base_station_node.cpp`: Main base station interface.

– `emergency_handler.cpp`: Emergency response logic.

– `telemetry_manager.cpp`: System telemetry aggregation.

– `heartbeat_monitor.cpp`: Connection monitoring and recovery.

`launch/`

– `communication.launch.py`: Full communication stack.

– `base_station_only.launch.py`: Base station node only.

– `emergency_monitor.launch.py`: Emergency monitoring only.

`config/`

- `emergency_triggers.yaml`: Emergency condition definitions.
- `telemetry_config.yaml`: Telemetry streaming configuration.
- `comms_params.yaml`: Communication parameters.
- `failsafe_actions.yaml`: Automated failsafe responses.

`test/`

- `test_emergency_handler.cpp`: Emergency logic tests.
- `test_base_station.cpp`: Base station tests.
- `test_gcs_cli.py`: CLI interface tests.