
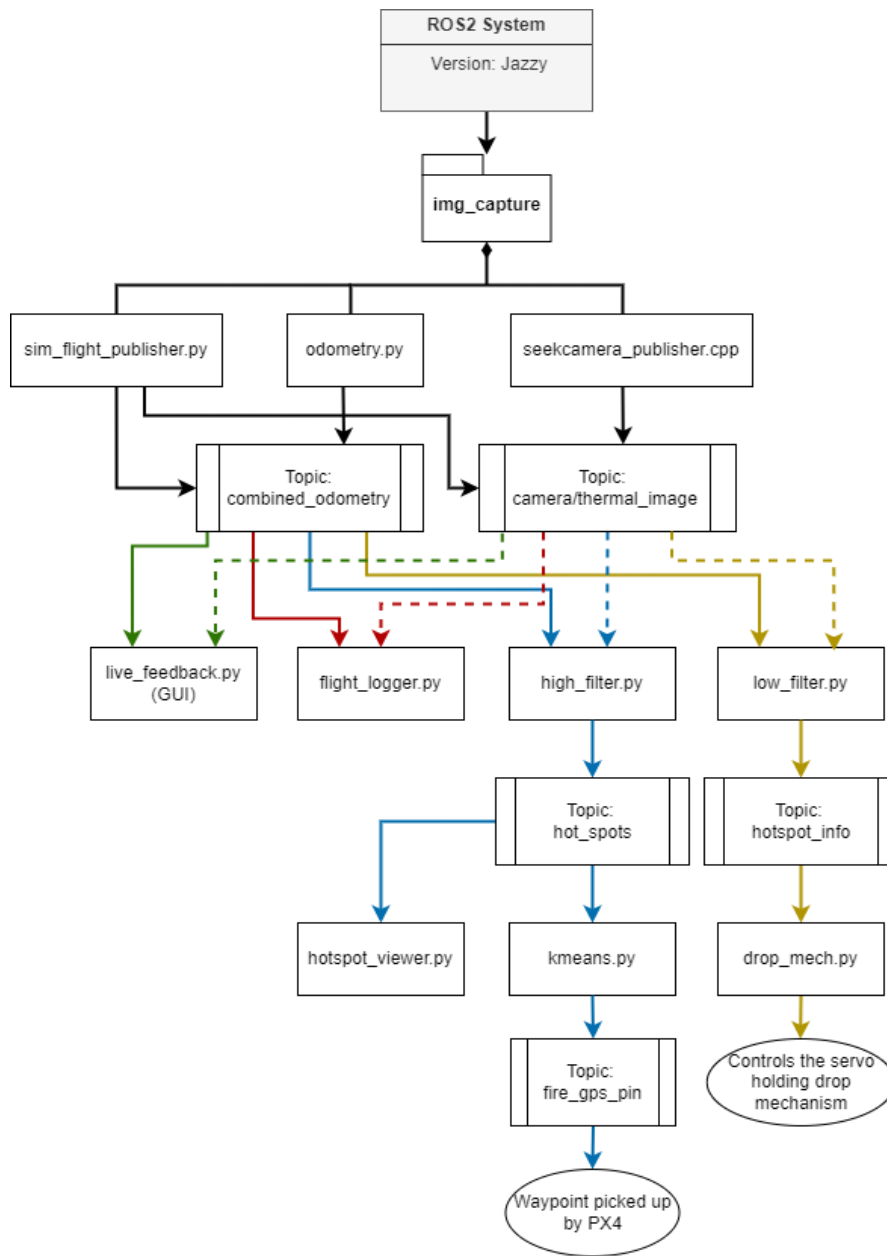


Artifact ID: ART-006	Artifact Title: High-altitude and Low-altitude Drone Code Overview		
Revision: 01	Revision Date: 2025-02-20		
Prepared by: Anthony Cardenas		Checked by: Isaac Davies	
Purpose: Explains the code flow for fire detection in the High-Altitude and Low-Altitude UAV, in relation to the Detection Code Block Diagram			

Revision History			
Revision	Revised by	Checked by	Date
01	Isaac Davies	Janie Linford	2025-02-20



High Altitude UAV Code Overview

Definitions:

Node - Code Block (shown with boxes)

Topic - Message Channel (shown with blocks with margins)

Publish – Send data to a topic

Subscribe – Collect data from a topic

Code Environment:

The high and low altitude UAVs are using Ros2 to separate tasks for Fire Detection. Ros2 is a common robotic code platform used for communication between code blocks that run concurrently. Our Code Block Diagram details the Ros2 node/topic communication. Our code is in python due to the range of libraries it provides, especially for image processing. The image input from the Seek Thermal Camera is done in C++, since the API connection is written in that language. It uses the "roscpp" library which allows us to use the C++ code in the Ros2 environment. The odometry input from the pix4 is done through the "px4_msgs" library, which is a public repository used to communicate with the pix4. All code is kept in a GitHub Repository called "XPRIZE-Snowflake".

Code Overview:

Node 1: seekcamera_publisher.cpp

Activates the Seek Thermal camera, and publishes the image data to the "camera/thermal_image" topic. This code uses the prebuilt API from the Seek Thermal company to communicate with the camera.

Node 2: odometry_publisher.py

Collects data from the "/fmu/out/vehicle_altitude" and "/fmu/out/vehicle_gps_position" topic that the pix4 publishes messages to. Then, it collects and combines the data into a single Json message that is sent to the "/combined_odometry" topic. This data includes latitude, longitude, altitude, pitch, roll, yaw, and a timestamp.

Node 3: high_alt_filter.py

Intakes images from the camera and odometry data from the "camera/thermal_image" and "/combined_odometry" topics, respectively. It starts by saving the first odometry data as a variable called "home_location". Once, it reaches an altitude above 30 meters, it matches the image and odometry data using their timestamps, and filters the images looking for the hottest spot. It returns the location of the hotspot relative to local GPS coordinates, and publishes the data to a topic called "hot_spots".

Node 4: k-means.py

Intakes GPS locations from the "hot_spots" topic and saves them to an array of hot spot locations. It then parses through locations and clusters them with the closest hot spot locations, which is an algorithm called k-means. This allows the data collected from several images to be averaged, improving accuracy of the GPS coordinates for each fire. When a k-means cluster has more coordinates than a set threshold, set by "min_size", the GPS coordinate is sent to a topic called "fire_gps_pin". In the future, these locations will be sent to qgroundcontrol and will be saved in flight_logger.py.

Node 5: low_alt_filter.py

Intakes images from the camera and odometry data from the “camera/thermal_image” and “/combined_odometry” topics, respectively. It starts by saving the first odometry data as a variable called “home_location”. Once, it reaches an altitude above 30 meters, it matches the image and odometry data using their timestamps, and filters the images looking for the hottest spot. It returns the location of the hotspot relative to the UAV, and publishes the data to a topic called “/hotspot_info”.

Node 6: drop_mech.py

Intakes local odometry data and user commands from the topics “/hotspot_info” and “servo_command”, respectively. It uses that data to determine when to open the drop mechanism. In automatic mode, it opens the drop mechanism when the hotspot is within 5 meters. In manual mode, it opens the drop mechanism when the user sends a command to the “/servo_command” topic. The Pi uses the “python3-gpiozero” and “python3-rpi.gpio” libraries to control the servo. It can run by itself, no other nodes need to be run. The default settings are manual mode and the servo is set to close.

The following commands can be sent to the “/servo_command” topic:

Switching to automatic or manual mode:

```
ros2 topic pub /servo_command std_msgs/msg/String "{data: \"auto\"}"
```

```
ros2 topic pub /servo_command std_msgs/msg/String "{data: \"manual\"}"
```

Controlling the servo:

```
ros2 topic pub /servo_command std_msgs/msg/String "{data: \"close\"}"
```

```
ros2 topic pub /servo_command std_msgs/msg/String "{data: \"open\"}"
```

Node 7: flight_logger.py

Code that subscribes to the “camera/thermal_image” and “/combined_odometry” topics, and logs the data collected during runtime. It pairs data from the camera and pix4 using their timestamps. On exit, it saves all data to a file called “flight_logs/high_alt_images_{date}.npy”. Doesn’t need to be run for “high_alt_filter.py” to work. For the moment, high and low altitude share this logger since they perform the same function on different Raspberry Pis that have their own logs.

Node 8: sim_flight_publisher.py

Simulates a flight path and publishes to the “camera_themal/image” and “/combined_odometry” topics. It is used to test the “high_alt_filter.py” node. This should not be run with “seek_camera_publisher.cpp” and “odometry_publisher.py” nodes. This path is premade and copied into the pi in a file called “sim_data/video_matrix16.npy”.

Node 9: live_feedback.py

This node subscribes to the “camera/thermal_image” and “combined_odometry” topics. This node filters the images using the “OpenCV” library and pairs it with odometry data. It then shows them on a GUI, so the user can see live camera data and filtered data. This node often crashes with low network connections, so it is separated from the “high_alt_filter.py” node. This allows all other nodes to continue. This node needs to be run with putty, which can open a GUI from your computer monitor.

See “FullSetupDocumentation” for how to run “live_feedback.py” node.

Using the Code:

In order to run each node, the following commands need to be sent from the terminal:

1. Enter the Pi wirelessly:
 - a. Option 1: In the lab, with Wi-Fi

ssh username@10.2.118.167

- b. Option 2: In the field, without Wi-Fi

Create a hotspot from your computer or phone

On a windows computer go to: settings->network and internet->mobile hotspot

Find the new ip address of the Pi and enter using ssh:

ssh username@{ip_address}

2. Go to the ros2 workspace and build the package:

```
cd ros2_ws
```

```
colcon build --symlink-install --packages-select img_capture
```

```
source install/setup.bash
```

Note: If the pix4_msgs are creating an error, run:

```
colcon build --symlink-install --packages-select px4_msgs
```

3. Run the desired nodes:

Generic node activation:

```
ros2 run {package} {node}
```

- a. Option 1: For testing in the lab:

```
ros2 run img_capture sim_flight_publisher.py
```

```
ros2 run img_capture high_alt_filter.py
```

```
ros2 run img_capture k-means.py
```

```
ros2 run img_capture live_feedback.py (in Putty, see FullSetupDoc)
```

b. Option 2: For testing in the field:

```
ros2 run img_capture seekcamera_publisher
```

```
ros2 run img_capture odometry_publisher.py
```

```
ros2 run img_capture flight_logger.py
```

```
ros2 run img_capture live_feedback.py ( in Putty, see FullSetupDoc)
```

c. Option 3: Launch files

```
ros2 launch img_capture new_img_capture.launch.py [optional arguments]
```

```
ros2 launch img_capture new_img_capture.launch.py sim:=false log:=false high:=true
```

Here are the possible arguments:

```
sim:=false log:=false high:=true
```

sim – false: launches odometry_publisher.py and seekcamera_publisher

sim – true: launches sim_flight_publisher.py

log – true: launches flight_logger.py

log – false: doesn't launch flight_logger.py

high – true: launches high_alt_filter.py and kmeans.py

high – false: launches low_alt_filter.py and drop_mech.py

Print – log: only sends messages to the logger

Print – screen: prints messages directly on the terminal

Print – both: prints to screen and saves to the logger

4. Shutdown the scripts

```
pgrep -f img_capture | xargs kill -9
```

This command will stop running code and save the log from flight_logger.py