

 README.md

Mavlink Interface

This is a Python interface between the Mavlink software used by the Sunfish's Drone Controller and a Python script, allowing for simple control of the drone with both simple movement control and advanced waypoint-based navigation. Additionally, it provides access to any sensors attached to the drone, allowing for both individual and aggregate sensor data retrieval.

Features

- Full Logging of some or all sensor data (configurable)
- All returning functions return either:
 - i. a single variable, or
 - ii. A JSON-formatted string
- [Multiple methods for handling sequential commands](#)
- All sensor info accessible
 - Per sensor
 - Per sensor module (eg. IMU, SENSOR_POD_1)
 - Together
- Easily switch between flight modes

Installation

1. Install python 3 (See instructions [here](#))
 - If pip3 was not installed along with python, install it now
2. Install pymavlink
 - `pip3 install pymavlink` Note: Use the `--user` flag on Windows
3. Install bluerobotics-ping
 - `pip3 install bluerobotics-ping` Note: Use the `--user` flag on Windows
4. Download this repository
5. Navigate a terminal or administrator CMD prompt to the folder containing `setup.py`
6. Run `python3 ./setup.py install`

Usage

To use this library, you must already have an underwater drone set up and working with qGroundControl. Once that is working, perform the following steps.

1. Import the library in your python script
 - `import mavlinkinterface`
2. Create an instance of the interface. All interactions with the drone will be completed through this interface. Note that you may provide a default execution mode (see [here](#) for details)
 - `MLI = mavlinkinterface.mavlinkInterface(execMode="queue")`
3. If you are diving for the first time in a given body of water, set the surface pressure.
 - `MLI.setSurfacePressure()`
4. If you are diving in a liquid that has a different density from fresh water or the last used medium (eg. salt water), set the density
 - `MLI.setFluidDensity(1027)`
5. Arm the Drone

- `MLI.arm()`

6. Proceed with script

7. Disarm the Drone

- `MLI.disarm()`

See [the examples](#) for more in-depth instructions.

Function List

For Full function list, see [here](#)

For Mission commands, see [here](#)

Common Parameters

execMode (string)

This sets the execution mode of the attached command. For information of the various execution modes, see [here](#)

Absolute (switch)

This argument causes movement commands to use absolute coordinates and directions, rather than coordinates and directions relative to the drone.

- When present, direction is relative to magnetic north, depth is relative to the surface, etc.
- When absent, direction coordinates, depth, and distances are all relative to the drone's current location and heading


Note: This argument is only relevant where a direction, depth, or coordinates are present

ChangeLog

Changelog is available [here](#).

Contributing

Contribution instructions available [here](#)

 docs\changelog.md

Changelog

All notable changes to this project will be documented in this file.

Current Release: [1.2.0]

Added

- Ability to change leak response actions
- Ability to altar recording interval
- Ability to disable certain sensors (in order to simulate failure)
- More effective message logging

Changed

- Small modifications to setup.py
- Fixed bug in Yaw function that was introduced in v1.1.0

[1.1.0]

Added

- Constant Manual Input mode
- More accurate yaw function
- Sitl Mode
- Scripts now wait for queue to end before returning

Changed

- Fixed lighting
- Tweaked dive safety threshold
- Tweaked dive acceptance threshold
- Fixed waitQueue not releasing semaphore

Removed

- wait function marked as deprecated

[1.0.0]

- Initial Release

 docs\executionModes.md

Execution Modes

There are multiple ways to define which execution mode to use for a file.

1. Set the default mode, and do not pass it with a function
 - `MLI = mavlinkinterface.mavlinkInterface(execMode="queue")`
 - `MLI.move(angle=0, time=3)`
2. Pass the mode with the function
 - `MLI.move(angle=0, time=3, execMode="queue")`
3. Change the default mode
 - `MLI.setDefaultExecutionMode(execMode="queue")`

Synchronous

In this mode, each movement command will return only once it is completed. An example is below:

```
# Assuming default execMode is Synchronous
MLI.move(angle=0, time=3)
MLI.move(angle=90, time=3)
MLI.move(angle=180, time=3)
MLI.move(angle=270, time=3)
print(MLI.getTemperature()) # getTemperature is not a movement command
```

In this example, the first 4 commands will be executed in series, following the completion of the prior command, then the print command will be executed

Output:

```
time=0: Move Command 1 started
time=3: Move Command 1 Finished
time=3: Move Command 2 started
time=6: Move Command 2 Finished
time=6: Move Command 3 started
time=9: Move Command 3 Finished
time=9: Move Command 4 started
time=12: Move Command 4 Finished
time=12: 26 # Temperature in degrees Celsius
```

Queuing

In this asynchronous mode, a command returns directly after initiating. If, upon entering a movement command, there is not currently a movement command executing, this movement command will be immediately executed. If a movement command is currently executing, this movement command will be added to the command queue. The command queue will automatically execute the contained commands in the order added. An example is below.

```
# Assuming default execMode is queuing
```

```
MLI.move(angle=0, time=3)
MLI.move(angle=90, time=3)
MLI.move(angle=180, time=3)
MLI.move(angle=270, time=3)
print(MLI.getTemperature() # getTemperature is not a movement command
```

In this example, upon running this series of commands, the first command will be called, immediately start executing, and return. The second command will then be added to the queue, and then cease blocking, as will the third and fourth. Then, while the first move command is still executing, the temperature will be printed out. After the first command ends, the next three will execute in sequence.

Output:

```
time=0: Move Command 1 started
time=0: Move Command 2 Queued
time=0: Move Command 3 Queued
time=0: Move Command 4 Queued
time=0: 26 # Temperature in degrees Celsius

time=3: Move Command 1 Finished
time=3: Move Command 2 started

time=6: Move Command 2 Finished
time=6: Move Command 3 started

time=9: Move Command 3 Finished
time=9: Move Command 4 started

time=12: Move Command 4 Finished
```

Ignore

In this asynchronous mode, a command returns directly after initiating. If, upon entering a movement command, there is not currently a movement command executing, this movement command will be immediately executed. If a movement command is currently executing, this movement command will be ignored. An example is below.

```
# Assuming default execMode is ignore
move(angle=0, time=3)
move(angle=90, time=3)
move(angle=180, time=3)
move(angle=270, time=3)
print( getTemperature() ) # getTemperature is not a movement command
```

In this example, upon running this series of commands, the first command will be called, immediately start executing, and return. The Second, third, and fourth commands will be ignored, as a movement command is already executing. Then, while the first move command is still executing, the temperature will be printed out. The only movement to be taken will be the first.

Output:

```
time=0: Move Command 1 started
time=0: Move Command 2 ignored
time=0: Move Command 3 ignored
time=0: Move Command 4 ignored
time=0: 26 # Temperature in degrees Celsius

time=3: Move Command 1 Finished
```

Override

In this asynchronous mode, a command returns directly after initiating. If, upon entering a movement command, there is not currently a movement command executing, this movement command will be immediately executed. If a movement command is currently executing, the currently executing movement command will be immediately terminated, and this command will be executed. An example is below.

```
# Assuming default execMode is override
MLI.move(angle=0, time=3)
MLI.move(angle=90, time=3)
MLI.move(angle=180, time=3)
MLI.move(angle=270, time=3)
print(MLI.getTemperature()) # getTemperature is not a movement command
```

In this example, upon running this series of commands, the first command will be called, immediately start executing, and return. The second command will then be called, terminating the first and executing itself. The second and third commands will do likewise. Then, while the last movement command is still executing, the temperature will be printed out. The only movement command to be completed will be the last one, while the first three would be only partially executed (move only until the next command was able to terminate it).

Output:

```
time=0: Move Command 1 started
time=0: Move Command 1 Killed
time=0: Move Command 2 started
time=0: Move Command 2 Killed
time=0: Move Command 3 started
time=0: Move Command 3 Killed
time=0: Move Command 4 started
time=12: 26 # Temperature in degrees Celsius

time=3: Move Command 4 Finished
```

Interactions between modes

Below are some Examples:

Example 1: Queue + Synchronous

```
# Add Items to the queue
MLI.move(angle=0, time=10, execMode="queue")
MLI.move(angle=180, time=10, execMode="queue")
MLI.move(angle=0, time=10, execMode="queue")
for i in range(3):
    print("Print statement " + str(i))
    sleep(1)
MLI.move(angle=180, time=10, execMode="synchronous")
print("Last Line")
```

Output:

```
time=0: Move Command 1 started
time=0: Move Command 2 queued
time=0: Move Command 3 queued time=0: Print statement 1 time=1: Print statement 2 time=2: Print statement 3 time=3: Move
command 4 called, waiting to execute

time=10: Move Command 1 Finished
time=10: Move Command 2 started

time=20: Move Command 2 Finished
time=20: Move Command 3 started
```

```
time=30: Move Command 3 Finished  
time=20: Move Command 4 started  
  
time=30: Move Command 4 Finished  
time=30: Last Line
```

In this example, the first move command will be executed, and the next 2 put in the queue. The print loop will be started and will finish. When the 4th move command is called, it will wait until the CPU is free (until the entire queue has finished execution) to execute, blocking the entire time.

Example 2: Queue + Override

```
MLI.move(0, 10, execMode="queue")  
MLI.move(180, 10, execMode="queue")  
MLI.move(90, 10, execMode="override")
```

Output:

```
time=0: Move Command 1 Started  
time=0: Move Command 2 Queued  
  
time=0: Move Command 3 Called  
time=0: Queue Cleared  
time=0: Move Command 1 Killed  
time=0: Move Command 3 Started  
  
time=10: Move Command 3 Finished
```

In this example, the first command immediately executes and the second is added to the queue. The third will (1): Clear the Queue, (2): stop the currently executing command, and (3) execute.

 docs\contribute.md

Contributing

If you want to add a feature, make a new Mavlink message available, or integrate a new sensor package, this guide will show you how to do that.

General Information

If you are unfamiliar with git and GitHub, check out [this guide](#).

If you are unfamiliar with contributing to open source projects, check out [this guide](#).

Bug Reports

To file a bug report, create a new Issue with the following information:

1. Steps to replicate the bug
2. All relevant logs

General guidelines

- Always perform logging using `from mavlinkinterface.logger import getLogger` . Logging levels are:
 - trace: for progress messages not useful to the user
 - debug: for information useful in debugging, but not important to user
 - rdata: When a function returns data, log the data with this before returning
 - info: for progress data useful to the user. This, and levels below this, are printed to the console when using the 'main' logger.
 - warn: For issues that are easily corrected without causing the program to stop.
 - error: for errors
- If a new function is made, use [type-hinting](#) for both parameters and return values.
- Always include a docstring at the beginning of functions
- Functions that cause the drone to take physical action must support all 4 execution modes
- All new functions must have a markdown file in the documentation folder, which must be linked in the appropriate section of [functions.md](#)
 - A short description of the function
 - All parameters of the function
 - The return value of the function
 - at least one example
 - If JSON is returned, an example output

Adding a New Mavlink Message

Not all mavlink messages are read by default due to the computation cost (and therefore power cost).


If a message is needed, but not read, add it to the list as follows:

In mavlinkInterface class `__init__` function in `main.py` , go to the 'Set Messages to be Read' section, add the full message name to the `self.readMessages` definition

Adding a new sensor

Because most attached sensors behave in different ways, most of this will be unique to your sensor, but there are several things that make it easier.

- If the sensor has its own python interface, consider creating a class for it (like how Sonar is implemented).
- All returned sensor data must be converted to the SI unit that makes the most sense
 - Exception: degrees are used for all angles, rather than the SI Unit of Radians

 docs\functions.md

Current Status

Completed functions

These functions perform exactly as described in the readme

Active

These functions cause the drone to take a physical action.

- [arm\(\)](#)
- [disarm\(\)](#)
- [setFlightMode\(flightMode, execMode <optional> \)](#)
- [move\(direction, time, throttle <optional>, absolute <optional>, execMode <optional> \)](#)
- [move3d\(throttleX, throttleY, throttleZ, time, execMode <optional> \)](#)
- [surface\(execMode <optional> \)](#)
- [setLights\(brightness, execMode <optional> \)](#)
- [gripperOpen\(time, execMode <optional> \)](#)
- [gripperClose\(time, execMode <optional> \)](#)
- DEPRECATED: [wait\(time, execMode <optional> \)](#)

Passive

These functions do not cause the drone to make any action other than reading a sensor

- [getDepth\(\)](#)
- [getGyroscopeData\(\)](#)
- [getAccelerometerData\(\)](#)
- [getBatteryData\(\)](#)
- [getHeading\(\)](#)
- [getMagnetometerData\(\)](#)
- [getPressureExternal\(\)](#)

Configuration

These functions modify configuration values, which persist on the device between missions

- [setSurfacePressure\(pressure <optional> \)](#)
- [setFluidDensity\(density <optional> \)](#)

Utility

These functions do not fit into any of the other categories.

- [disableSensor\(sensor, enable <optional> \)](#)
- [log\(message \)](#)
- [stopCurrentTask\(\)](#)
- [stopAllTasks\(\)](#)

- [waitQueue\(\)](#)

Mission Mode

Advanced functions relying on GPS fall under mission mode.

See [here](#) for more information on missions

Partially completed functions

These functions work as described in the documentation, but to a lesser grade of accuracy. Details on the failings of each one included. These are actively under development

- [dive\(depth, throttle <optional>, absolute <optional>, execMode <optional> \)](#)
 - Rotates to the depth and stops thrusting, but may pass the depth on momentum
- [setLeakAction\(action \)](#)
 - Currently the leak detection is implemented, but the return to base and custom script functions are not yet implemented.

New Functions

These functions were not originally intended, but were added since the last update.

After each update, these functions will be moved to the completed functions category.

- [getAltitude\(\)](#)
 - This takes advantage of the new sonar sensor
- [gps.getCoordinates\(\)](#)
 - This returns GPS Coordinates
- [mission commands](#)
 - This allows the user to plan and execute missions
- [getTemperature\(\)](#)
 - This returns the temperature as read by the external pressure sensor
- [setDefaultExecMode\(mode \)](#)
 - This allows for the changing of the default execution mode after initialization
 - Note that this requires the queue to be empty and no commands to be executing.
- [getPressureInternal\(\)](#)
 - Returns the internal pressure as a float

Modified Functions (complete)

These Functions were changed in a major way, which is explained below.


After each update, these functions will be moved to the completed functions category.

- [yaw\(degrees, absolute <optional>, execMode <optional> \)](#)
 - Now has 3 stages:
 - a. Until within 30 degrees of target, yaws at 50% power
 - b. Until within 5 degrees of target, yaws at 25% power
 - c. cancels rotational momentum by reversing thrust until rotation is stopped

Not Started functions

- [cameraTilt\(angle, speed <optional>, absolute <optional>, execMode <optional> \)](#)
- [cameraStartFeed\(\)](#)
- [cameraVideoStart\(time <optional>, resolution <optional> \)](#)

- [cameraVideoStop\(\)](#)
- [cameraPhoto\(resolution <optional>, zoom <optional>, \)](#)
- [getAllSensorData\(\)](#)
- [setLoggingLevel\(level \)](#)
- [setRecordingInterval\(sensor, interval \)](#)
- [getSonarMap\(\)](#)
- All GPS-Related functions, including:
 - [setGpsMode\(\)](#)
 - [getCoordinates\(\)](#)
 - [goToCoordinates\(\)](#)
 - [setHome\(\)](#)
 - [goToHome\(\)](#)

 docs\missions.md

Missions

In order to use advanced GPS-Related functionality, it is necessary to create and use missions.

Missions work differently from normal commands in that they must be created, planned, and uploaded. Only then can they be executed.

For more information on missions, refer to the links at the bottom of this page.

Mission Creation

```
# Assuming you have initiated a mavlinkInterface class "mli"
myMission = mavlinkinterface.mission(mli)
```

Mission Planning

By adding waypoints here, they will be executed in order once the mission is started.

```
# Add some waypoints to the mission, this does a small square
myMission.goToCoordinates(33.810561, -118.394265)
myMission.goToCoordinates(33.811006, -118.394265)
myMission.goToCoordinates(33.811006, -118.394749)
myMission.goToCoordinates(33.810561, -118.394749)
myMission.goToCoordinates(33.810561, -118.394265)
```

Valid Mission Commands

goToCoordinates(lat, lon)

This command causes the drone to move to a specific set of coordinates

setHome(lat <optional>, lon <optional>)

This command sets the drone's home location to a specific set of coordinates.
If no coordinates are given, uses the drone's current location

Uploading Mission

This uploads the mission to the drone. The SITL Simulator appears to retain missions between reboots, so actual drones may do that as well.

```
myMission.upload()
```

Running mission

This sends the signal to the drone to run the currently uploaded mission. Note that it cannot differentiate between uploaded missions, so if another mission is on the drone, that one will be run, sometimes failing.

If the Wait parameter is present, blocks until the mission is completed.

```
myMission.start(wait=True)
```

To see this in action, run the mission example with SITL.

Additional Documentation

- [Mavlink mission protocol](#)
- [Mission Planner documentation](#)

 docs\active\arm.md

arm(execMode <optional>)

This function enables the thrusters, allowing movement commands to work.

Parameters

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void.

Example

```
MLI.arm()  
# The propellers are now armed
```

Related Mavlink Commands

- MAV_CMD_COMPONENT_ARM_DISARM

 docs\utility\disableSensor.md

disableSensor(sensor, enable <optional>)

This function is used to disable certain sensors to simulate a sensor failure.

Parameters

sensor (str):

The sensor to disable Valid options are: pressure, gps, sonar

enable (bool, optional):

When true, the sensor will be enabled, rather than disabled.

Return Values

Returns void

Examples

```
MLI.disableSensor('gps')
```

```
# When calling gps-related functions, they will behave as if the drone's gps is not present
```


 docs\active\disarm.md

disarm(execMode <optional>)

This function disables the propellers. When the drone is disarmed, movement commands will be sent, but do nothing.

Parameters

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void

Examples

```
MLI.disarm()  
# The propellers are now disabled.  
# Due to the way the ArduSub works, movement commands will still be sent, but will not do anything.
```

Related Mavlink Commands

- MAV_CMD_COMPONENT_ARM_DISARM

 docs\active\dive.md

dive(depth, throttle <optional>, absolute <optional>, execMode <optional>)

This function can be used to change the depth of the drone, either descending or ascending by a certain depth, or moving to a specific depth.

This function depends on the [getPressureExternal\(\)](#) command.

Parameters

depth (float):

The distance to dive in meters.
Negative numbers indicate an increase in depth.

throttle (int, optional):

The percentage of vertical thrust to use.
Default is 50

absolute (boolean, optional):

When true, *depth* signifies the target depth, rather than the change in depth

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.
For details on how these modes work, see [Here](#)

Return Values

Returns void

If the given values would put the drone above the surface, throws a ValueError

Examples

```
MLI.dive(depth = -10)
# The drone descends by 10 meters or until it is obstructed

MLI.dive(depth = 9, throttle = 100)
# The drone ascends by 9 meters at 100 percent throttle or until it is obstructed

MLI.dive(depth = -5, absolute=True)
# The drone ascends or descends until it reaches a depth of 5 meters below the surface
```

```
MLI.dive(depth = 5, absolute=True)
```

```
# An ValueError is thrown, indicating that the drone cannot rise above the surface of the water
```

Related Mavlink Messages

- MANUAL_CONTROL

 docs\passive\getAccelerometerData.md

getAccelerometerData()

This function reads the IMU accelerometer and returns the output in units of meters per second squared (m/s^2)

Return Values

Returns a JSON-formatted string.

The output will data taken directly from the IMU.

Output will be in meters per second squared.

Example output (expanded)

```
{
  "X": 1.25,
  "Y": -4.3,
  "Z": -9.81
}
```

 docs\passive\getAltitude.md

getAltitude()

This function calculates the altitude based on the sonar sensor data.

Note: Confidence levels are not reliable at short or long range. see official sensor documentation [here](#)
Unless future software updates change this, this is not accurate enough to implement terrain following mode.

Note 2: Current sonar software supports only one attached Ping Sonar Sensor.

Return values

Returns a JSON-Formatted string.

Upon success, returns the altitude and the confidence

Upon failure, throws an exception based on the type of error

Examples

```
MLI.getAltitude()
```

```
{  
  "altitude": "5.25",  
  "confidence": "100"  
}
```

 docs\passive\getBatteryData.md

GetBatteryData()

This function reads battery statistics and calculates battery percentage.

Return Values

Returns a JSON-Formatted string.

Returns all available information about the battery.

Examples

Example output:

```
{
  "voltage":12.3,
  "currentDraw":1.2,
  "percentRemaining":72.5
}
```

Related Mavlink Enumerations

- MAV_BATTERY_TYPE
- MAV_BATTERY_FUNCTION
- MAV_BATTERY_CHARGE_STATE
- MAV_SMART_BATTERY_FAULT

Related Mavlink Functions

- BATTERY_STATUS
- SMART_BATTERY_INFO (Beta)
- SMART_BATTERY_STATUS (Beta)

 docs\passive\getDepth.md

getDepth()

This function calculates the depth based on the external pressure sensor data.

Return values

Returns a float Returns the depth of the drone as distance from the surface in meters

Examples

```
MLI.getDepth() # Assuming the drone is surfaced  
# returns 0
```

```
MLI.getDepth() # Assuming the drone is 5.4 meters below the surface  
# returns 5.4
```

 docs\passive\getGyroscopeData.md

getGyroscopeData()

This function reads the IMU gyroscope and returns the output in units of degrees per second

Return Values

Returns a JSON-formatted string.

The output will be the data taken directly from the gyroscope.

Output is in units of degrees per second (°/s)

Example output

```
{
  "Axis-1":90,
  "Axis-2":1.25,
  "Axis-3":-3.9
}
```


 docs\passive\getHeading.md

getHeading()

This function uses the IMU magnetometer to calculate the direction the drone is facing.

Return Values

Returns a float.

Returns the heading of the drone in degrees (as the smallest possible positive value).

Examples

```
MLI.getHeading() # assuming the drone is facing due north  
# returns 0
```

```
MLI.getHeading() # assuming the drone is facing due east  
# returns 90
```

 docs\passive\getMagnetometerData.md

getMagnetometerData()

This function reads the IMU magnetometer and returns the output in units of gauss.

Return Values

Returns a JSON-formatted string.

The output will data taken directly from the magnetometer.

Output is in units of Gauss

Example output

```
{
  "X": -123.45,
  "Y": 12.5,
  "Z": 50.1
}
```

 docs\passive\getPressureExternal.md

getPressureExternal()

This function reads the data from the external pressure sensor.

Return values

Returns a float.

Returns the external pressure on the drone in Pascals.

Examples

```
MLI.getPressureExternal() # assuming external pressure of 155.32 pa  
# returns 155.32
```

 docs\passive\getPressureInternal.md

getPressureInternal()

This function reads the data from the internal pressure sensor.

Return values

Returns a float.

Returns the internal pressure on the drone in Pascals

Examples

```
MLI.getPressureInternal() # Assuming internal pressure of 101.32 pa  
# returns 101.32
```

 docs\passive\getTemperature.md

GetTemperature()

reads the data from the external temperature sensor (by default, the external pressure sensor)

Return Values

Returns a float.

Returns the temperature in degrees Celsius

Examples

```
MLI.getTemperature() # assuming water temperature of 26.3  
# returns 26.3
```

 docs\passive\gps.getCoordinates.md

gps.getCoordinates()

This function gets the coordinates from the GPS unit.
Coordinates will only be returned if a GPS Lock is present.

Return Values

Returns a string.
If a gps lock is present, returns the current coordinates (lat/lon only)
If no lock was present, throws a `ConnectionError`.
If any other error occurs, throws a relevant exception.

example output (expanded)

```
{
  "lat": 33.810313,
  "lon": -118.393867
}
```

Examples

```
try:
    coordinates = MLI.gps.getCoordinates() # Get Coordinates
    coordinateDict = json.loads(coords)    # Convert JSON to dict
    print('latitude: ' + coordinateDict['lat'])
    print('longitude: ' + coordinateDict['lon'])
except ConnectionError: # Catch exception thrown if lock is not present
    print('GPS does not have lock')
```

 docs\active\gripperClose.md

gripperClose(time, execMode <optional>)

Power the gripper arm closed for *time* seconds.

Closing the gripper from a fully open position is 1.75 sec

Parameters

time (float):

The number of seconds to send the "close" signal to the gripper arm This has a resolution of 0.25 sec.

Closing the gripper from a fully open position is 1.75 sec

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void

Examples

```
MLI.gripperClose(0.5)
```

```
# The grabber arm closes for 1/2 second
```

Related Mavlink Messages

- MANUAL_CONTROL

 docs\active\gripperOpen.md

gripperOpen(time, execMode <optional>)

Power the gripper arm open for *time* seconds

Opening the gripper from a fully closed position is 1.75 sec

Parameters

time (float):

The number of seconds to send the "close" signal to the gripper arm

This has a resolution of 0.25 sec.

Fully opening the gripper is 1.75 sec

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void

Examples

```
MLI.gripperOpen(0.5)
```

```
# The grabber arm powers open for 1/2 second
```

```
MLI.gripperOpen(1.75)
```

```
# The grabber arm powers open for 1 3/4 seconds
```

Related Mavlink Messages

- MANUAL_CONTROL

 docs\active\lights.md

setLights(brightness, execMode <optional>)

This function sets brightness of the lights.

Parameters

brightness (integer):

An integer from 0 to 100, representing the percent brightness of the lights.
This will be rounded to the nearest allowed lighting level.

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.
For details on how these modes work, see [Here](#)

Return Values

Returns void

Examples


```
MLI.setLights(0)
# Turns the lights off

MLI.setLights(65)
# sets the lights to 65% brightness

MLI.setLights(brightness=100)
# sets the lights to full brightness
```

Related Mavlink Messages

- MANUAL_CONTROL

 docs\utility\log.md

log(message)

This function is used to write a string to the program log.

Parameters

message (str):

The string to write to the log

Return Values

Returns void

Examples

```
MLI.log('something important is about to happen')  
# the given string will be visible (with timestamp) in the program log
```

 docs\active\move.md

move(direction, time, throttle <optional>, absolute <optional>, execMode <optional>)

This function moves the drone across the X/Y plane in a specified direction for a specified time.

Parameters

direction (integer):

An integer indicating the direction in degrees the drone will be moving.

time (float):

A real number representing the time in seconds between activation and deactivation of the propellers

throttle (integer, optional):

An integer from 1 to 100 representing the percentage of propeller power to use.
Defaults to 50

absolute (boolean, optional):

When true, *direction* is relative to magnetic north
When false or absent, *direction* is relative to the current heading of the drone

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.
For details on how these modes work, see [Here](#)

Return Values

Returns void

Examples

```
MLI.move(direction = 0, time = 15, throttle = 75)
# Moves the drone straight forward at 75% power for 15 seconds
```

```
MLI.move(direction = -15, time = 0.5, absolute=True)
# moves in the direction of 15 degrees to the right of magnetic north at 100% power for half a second
```

Related Mavlink Messages

- MANUAL_CONTROL

 docs\active\move3d.md

move3d(throttleX, throttleY, throttleZ, time, execMode <optional>)

This function moves the drone in 3 dimensions in a given direction for a specified period of time.

Parameters

throttleX (integer):

An integer from -100 to 100 indicating the percent throttle to use in the X direction

throttleY (integer):

An integer from -100 to 100 indicating the percent throttle to use in the Y direction

throttleZ (integer):

An integer from -100 to 100 indicating the percent throttle to use in the Z direction

time (float):

An real number representing the time in seconds between activation and deactivation of the propellers

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void


Examples

```
MLI.move3d(throttleX=100, throttleY=100, throttleZ=0, time = 15)
# Moves the drone forward and to the right at 100% power for 15 seconds
```

```
MLI.move3d(0, 0, 100, 10)
# Thrust upward for 10 seconds
```

Related Mavlink Messages

- MANUAL_CONTROL

 docs\configuration\setDefaultExecMode.md

setDefaultExecMode(mode)

This function sets the execution mode to use when the common parameter `execMode` is not given.

This will only work if the queue is empty, and no commands are currently executing.

The possible modes are as follows:

Synchronous mode:

Commands will not return a value or allow the entry of another command until completed or interrupted

Queue mode:

If a movement command is currently executing and a new move command is initiated, the new move command will be placed in an execution queue.

Override mode:

If a movement command is currently executing and a new move command is initiated, the currently executing movement command will be halted and discarded, and the new command will be executed.

Ignore mode:

If a movement command is currently executing and a new move command is initiated, the new move command will be ignored and discarded.

For more information, see [here](#),

Parameters

Mode (enum):

The queuing mode to use by default. Possible values are:

synchronous
queue
override
ignore

Return Values

Returns void.

If given an invalid mode, raises a `ValueError`.

If there is an item in the queue or a currently executing command, raises a `ResourceWarning`

Example

```
MLI.setDefaultQueueMode( 'queue' )
```

 docs\active\setFlightMode.md

setFlightMode(flightMode, execMode <optional>)

This function sets the drone's flight mode to the given value. Valid flight modes are listed below.

Flight Modes

MANUAL

Manual mode passes the pilot inputs directly to the motors, with no stabilization. ArduSub always boots in Manual mode.

STABILIZE

Stabilize mode is like Manual mode, with heading and attitude stabilization.

ALT_HOLD

Depth Hold is like Stabilize mode with the addition of depth stabilization when the pilot throttle input is zero. A depth sensor is required to use depth hold mode.

Position Enabled Modes

These modes require an under water positioning system. A GPS antenna will not work under water.

POS_HOLD

Position Hold mode will stabilize the vehicle's absolute position, attitude, and heading when the pilot control inputs are neutral. The vehicle can be maneuvered and repositioned by the pilot.

AUTO

Auto mode executes the mission stored on the autopilot autonomously. Pilot control inputs are ignored in most cases. The vehicle may be disarmed, or the mode can be changed to abort the mission.

CIRCLE

Circle mode navigates in circles with the front of the vehicle facing the center point.

GUIDED

Guided mode allows the vehicle's target position to be set dynamically by a ground control station or companion computer. This allows 'Click to Navigate Here' interactions with a map.

ACRO

Acro (Acrobatic) mode performs angular rate stabilization.

For official documentation, see [here](#)

Parameters

mode (string):

The flight mode to use

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void

Example


```
MLI.setFlightMode('ALT_HOLD')  
# sets the drone to depth hold mode
```

Related Mavlink Enumerations

- MAV_MODE
- MAV_MODE_DECODE_POSITION

Related Mavlink Commands

- MAV_CMD_DO_SET_MODE

 docs\configuration\setFluidDensity.md

setFluidDensity(density)

This function sets the fluid density (used in depth calculations) to the given value.

Note: changes made by this command persist between dives.

Parameters

density (int):

The density in kg/m^3 to set as default.

Fresh water is 1000 Salt water is 1020-1030, depending on salinity

Chlorinated pool water is 1000


Return Values

Returns void

Examples

```
MLI.setFluidDensity(1025)
```

```
# Sets the fluidDensity value to 1025  $\text{kg/m}^3$ 
```

 docs\configuration\setLeakAction.md

setLeakAction(action)

This function sets the action to be taken on encountering a leak.

Parameters

action (string)

There are several default options as well as a custom option (all listed below)

nothing - No action, other than warning the user and noting the leak in the log, will be taken surface - This causes the drone to surface and cease other actions upon detecting a leak

Not yet Implemented: home - This causes the drone to surface, wait for gps signal, and return to the designated home point, ignoring other non-override commands. If no gps is present, or drone is unable to get a GPS lock, just surfaces

Not yet Implemented: <Path to python file> - This will execute the function customLeakAction from the given file. A template to use for that file is below.

Return Data

Returns void

Examples

```
try:
    MLI.setLeakAction( '~/myFile.py')
except:
    print("There was an issue with the file")
    MLI.setLeakAction('surface')
```

~/myFile.py

```
from mavlinkinterface.logger import getLogger
# User imports go here

# Do NOT alter this line \
def customLeakAction(mli) -> None:
    log = getLogger('leakAction')
    log.trace('Entered custom leak action function')
    # User code starts here

    # Example, delete before inserting your code
    mli.surface()

    try:
        if mli.gpsEnabled:
            c = mli.gps.getCoordinates()
            log.warn('surfaced at ' + c + ' upon detecting leak')
    except ConnectionError:
        log.error('Failed to get GPS')

    # User code ends here
    log.trace('Custom leak action completed')
```


 docs\configuration\setSurfacePressure.md

setSurfacePressure(pressure <optional>)

This function sets the surface pressure (used in depth calculations) to the given value.

Note: changes made by this command persist between dives.

Parameters

pressure (int):

The pressure in pascals to set as default.

If no value is given, uses the current external pressure of the drone

Sea Level is 101325

Return Values

Returns void

Examples

```
MLI.setSurfacePressure()    # While on surface  
# Sets the surfacePressure value to the current exterior pressure
```

```
MLI.setSurfacePressure(101000)  
# Sets the surfacePressure value to 101000 pascals
```

 docs\utility\stopAllTasks.md

stopAllTasks()

This function clears the queue and kills the currently executing task.

Return Values

Returns void

Examples

```
MLI.move(direction=0, time=1000, execMode='queue')
MLI.move(direction=180, time=1000, execMode='queue')
MLI.stopAllTasks()
# The drone will stop moving
```

 docs\utility\stopCurrentTask.md

stopCurrentTask()

This function kills the currently executing task.

Return Values

Returns void

Examples

```
MLI.move(direction=0, time=1000, execMode='queue')    # task A
MLI.move(direction=180, time=1000, execMode='queue')  # task B
MLI.stopCurrentTask()
# The drone stop executing task a, and continue executing the queue (executing task B)
```

 docs\active\surface.md

surface(execMode <optional>)

This function brings the drone to the surface at full throttle.

Parameters

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void.

Example

```
MLI.surface()  
# The drone ascends to the surface
```

Related Mavlink Messages

- MANUAL_CONTROL

 docs\active\wait.md

DEPRECATED: wait(time, execMode <optional>)

This function is functionally identical to sleep(), except it is able to be used with queue modes

Parameters

time (float):

The amount of time to wait.

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void.

Example

```
# This is an example for which a wait is useful.  
for i in range(3)  
    MLI.dive(depth=-1, execMode='queue')  
    MLI.wait(3, execMode='queue')
```

 docs\utility\waitQueue.md

waitQueue()

This function blocks until:

1. The queue has completed and the current task has ended
2. a Keyboard Interrupt (Ctrl+C) has been received

Return Values

Returns void

Examples

```
MLI.move(direction=0, time=10, execMode='queue')
MLI.move(direction=180, time=10, execMode='queue')
MLI.waitQueue()
# the waitQueue function will block for 20 seconds, returning once the last function has been completed.
```

 docs\active\yaw.md

yaw(degrees, absolute <optional>, execMode <optional>)

This command rotates the drone to face a certain direction.

Note: This command will not rotate the drone by more than 180 degrees in either direction.
See examples below for how this is implemented.

Parameters

degrees (integer)

An integer for how many degrees to rotate. If a number greater than 360 is given, it will be reduced to the lowest equivalent angle.

absolute (boolean, optional):

When true, *degrees* is relative to magnetic north

When false or absent, *degrees* is relative to the current heading of the drone

execMode (string, optional):

The execution mode to use for this command. Possible execution modes are:

1. Synchronous
2. Queue
3. Ignore
4. Override

If not given, defaults to the execution mode given on class initiation.

For details on how these modes work, see [Here](#)

Return Values

Returns void.

Examples

```
MLI.yaw(degrees = 15)
```

```
# Result: The drone yaws to the right by 15 degrees
```

```
MLI.yaw(degrees = 270)
```

```
# Result: The drone yaws to the left by 90 degrees
```

```
MLI.yaw(degrees = -410)
```

```
# Result: The drone yaws to the left by 50 degrees
```

```
MLI.yaw(degrees = -15, absolute)
```

```
# Result: The drone yaws the shortest distance to face 15 degrees to the left of magnetic north
```

Related Mavlink Messages

- MANUAL_CONTROL

