A User API

The user API offers a number of methods providing the user with cycle control, drone control, and drone state feedback.

The API uses Standard International Units –

- Time - seconds (s)
- Position - meters (m)
- Velocity - meters per second m/s

A.1 Custom Types

- **id:** Each drone can be represented by an ID object which includes a numeric ID and a name (as specified by the motion capture tag)
- **position_data:** This structure is used to return position data for a given drone to the user. It includes an *id* object, a time stamp, coordinates and yaw. It also provides a method called *isValid()* which specifies whether the data is still valid or if there is a more recent update.
- **velocity_data:** Identical to *position_data* except for velocity feedback
- **position_msg:** Represents the data forwarded to the drone server for a position command. Includes a coordinate, a boolean called *relative* which clarifies whether the x and y coordinates are relative, a boolean called

keepHeight if set to true, the server will treat the z coordinate as relative (if $0.0f$, will maintain height), a command duration and a yaw value.

- **velocity_msg:** Identical to *position_msg* except for velocity messages.
- **timings:** Includes various measures regarding the operating frequencies of different aspects of the multi-drone platform. Including –
  - Motion capture update rate
  - Desired drone server update rate
  - Actual drone server update rate
  - Time used to update drones
  - Wait time per frame (time remaining after completing all tasks)
- **drone_state:** This enum clarifies the five states any given drone can always be represented by, these states reflect the physical state, not necessarily the state the drone should be in (the implementation of the state variable is further elaborated in feature set 4)–
  - UNKNOWN
  - LANDED
  - HOVERING
  - MOVING
  - DELETED

A.2 Cycle control methods

- **initialise:** The initialise function creates a ROS node for the given script or program in order to operate within the ROS network. The function also establishes necessary topics to communicate user API requests, manage server feedback and to request specific drone feedback. Finally it establishes the asynchronous queue on which API requests are placed to ensure they are evaluated in the order they were requested and as soon as possible.

- **terminate:** As the name implies this terminates the user API connection, if any drones are still airborne it will send a land command to each of them and wait until all drones are in the 'LANDED' state. It will then dispose of the necessary ROS connections.
- **sleep_until_idle:** This function requires a drone ID and will wait until it has finished executing its current command before returning focus to the primary thread. This function is typically used following a command. This is used to ensure when the next command is added the drone has completed its previous command. If this function is not used between commands, the next command will begin execution immediately, this gives the user the option to interrupt the current command, or wait for its completion.
- **spin_until_rate:** This function is a singular version of the cycle provided in *sleep_until_idle* if the user would like to manage their own cycle this function will run one ROS iteration and return to the user script. The duration of the ROS loop is determined by the update rate (in Hz) specified when the user API is initialised.

A.3 Action methods

- **set_drone_velocity:** For the given drone *id* object, sets the desired velocity equal to the input values. This method requires an *id* object and a *velocity_msg* object as input parameters.
- **set_drone_position:** Sets the desired position for the given *id* object. This expects an *id* object and *position_msg* as inputs.
- **cmd_takeoff:** Sends a takeoff request to the specified drone. Requires an *id* object, a height, and a duration.
- **cmd_land:** Sends a land request to the specified drone. Requires an *id* object and a duration.

- **cmd_emergency:** Sends an emergency command to the specified drone, this will have immediate priority over all other commands and will kill the motors of the drone. Requires only an *id* object.
- **cmd_hover:** Sends a hover request to the drone pointed to by the *id* input. Requires an *id* object and a duration.
- **set_home:** This sets the home position for the drone, by default the drone's home position will be equal to its position when it is added to the drone server. This expects an *id* object and *position_msg* as inputs.
- **go_to_home:** Sends a request to send the drone to its home position. Given the height input is greater than 0 it will go to the home position and wait for the next command at the specified height. If the height input is less than 0, the drone will go to the home position and then land. Requires an *id* object, a height, and a duration.
- **set_drone_server_update_frequency:** This method sets the update frequency of the drone server (by default it is 100Hz). Requires a float specifying the requested operating frequency as an input.

A.4 Feedback methods

- **get_all_rigidbodies:** This function will return a vector of *id* objects each mapping to a specific drone currently in the platform. This is the easiest way to obtain the required *id* objects for scripts rather than generating them independently.
- **get_velocity:** This function returns a *velocity_data* object relating to the input *id* object.
- **get_position:** This returns a *position_data* object related to the input *id* object.

- **get_home** This method will return the home position of the specified drone in the form of a *position_data* object.
- **get_state** For the specified drone *id* object, this function will return its current state according to the defined state enum.
- **get_operating_frequencies:** A *timing* object is returned regarding the operating frequencies of the core components throughout the platform.