MISSION PLANNING REFERENCE

ARDUPILOT COMMAND IMPLEMENTATION GUIDE

GENERATED 2026 // MAVLINK HUD SYSTEMS

Mission Planning Reference

- DO_GRIPPER
- DO_AUTOTUNE_ENABLE
- DO_SET_RESUME_REPEAT_DIST
- DO_SPRAYER
- DO_SEND_SCRIPT_MESSAGE
- DO_AUX_FUNCTION
- DO_GUIDED_LIMITS
- DO_ENGINE_CONTROL
- DO_GIMBAL_MANAGER_PITCHYAW
- DO_WINCH

CAMERA

- SET_CAMERA_ZOOM
- SET_CAMERA_FOCUS
- SET_CAMERA_SOURCE
- IMAGE_START_CAPTURE (2000) / IMAGE_STOP_CAPTURE
- VIDEO_START_CAPTURE (2500) / VIDEO_STOP_CAPTURE

OTHER

- JUMP_TAG (600) / DO_JUMP_TAG

NAVIGATION

NAV_FENCE_RETURN_POINT (5000) / VERTEX (5001-5002) / CIRCLE (5003-5004) (ID 0)

Summary

The `NAV_FENCE` family of commands defines the geometry of ArduPilot's **Geofence** system directly within the mission list. This allows the vehicle to carry its own "Containment Logic" in non-volatile memory, ensuring that even if the ground station link is lost, the drone remains bound by the pre-approved spatial boundaries.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives these commands during a mission upload (specifically when using the MAVLink Fence Protocol).

Commands and Storage

- **5000: Return Point:** The "Safe Zone" where the drone will fly if a fence breach occurs. (Only 1 allowed).
- **5001: Polygon Inclusion:** The drone **must** stay inside the shape defined by these vertices.
- **5002: Polygon Exclusion:** The drone **must not** enter the shape.
- **5003: Circle Inclusion:** A circular keep-in zone.
- **5004: Circle Exclusion:** A circular keep-out zone.
- **Mechanism:** Stored as specialized location items in the `AP_Mission` buffer.

Execution (Engineer's View)

Containment Mathematics

The `AC_Fence` library implements **Point-in-Polygon (PIP)** algorithms to monitor the vehicle's position.

1. **Ray Casting:** For polygons (5001, 5002), the autopilot mathematically casts a ray from the vehicle's position. If the number of intersections with the polygon edges is odd, the vehicle is inside.
2. **Distance-Squared:** For circles (5003, 5004), the autopilot calculates the 2D distance:

$$D^2 = (Lat_v - Lat_c)^2 + (Lon_v - Lon_c)^2$$

   If $D > \text{Radius}$ for an Inclusion circle, a breach is triggered.
3. **The Floor and Ceiling:** While not defined by these commands, the Geofence system also monitors altitude via global parameters like `FENCE_ALT_MAX`.

Data Fields (MAVLink)

- `param1` **(Count/Radius):** Vertex count for polygons, or Radius (m) for circles.
- `x` **(Latitude):** Vertex coordinate.
- `y` **(Longitude):** Vertex coordinate.
- `z` **(Altitude):** Reserved.

Theory: The "Safe Pipe"

`NAV_FENCE` commands enable **Mission-Geometric Coupling**.

- **The Problem:** A drone is programmed to fly a path. But what if a sensor fails and the drone drifts?
- **The Solution:** By wrapping the `NAV_WAYPOINT` items with a `NAV_FENCE_POLYGON_VERTEX_INCLUSION` sequence, the operator defines a high-integrity "Pipe." The drone is physically unable to exit this pipe while the mission is active.

Practical Use Cases

1. **Sensitive Airspace Protection:**
   - *Scenario:* A drone is mapping near an airport runway.
   - *Action:* Use `NAV_FENCE_POLYGON_VERTEX_EXCLUSION` to draw a box around the runway. The drone will treat this as a "Solid Object" and trigger a failsafe if it approaches the boundary.
2. **Indoor/Hangar Safety:**
   - *Scenario:* Flying inside a warehouse.
   - *Action:* `NAV_FENCE_CIRCLE_INCLUSION` centered on the hangar to ensure the drone never crashes into the walls if it loses its local positioning fix.

Key Parameters

- `FENCE_TYPE`: Bitmask selection of which fences are active (Circle, Polygon, etc.).
- `FENCE_MARGIN`: The distance (meters) before the boundary where the drone will begin to brake.

Key Codebase Locations

- **libraries/AC_Fence/AC_Fence.cpp**: Ray-casting and distance math.
- **libraries/GCS_MAVLink/MissionItemProtocol_Fence.cpp**: Fence packet parsing.

NAV_WAYPOINT (ID 16)

Summary

The `NAV_WAYPOINT` command is the fundamental building block of autonomous missions. It defines a 3D coordinate (Latitude, Longitude, Altitude) that the vehicle must navigate to. Depending on the vehicle type and parameters, the vehicle may either stop at the waypoint or transition smoothly through it towards the next destination.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload (`MISSION_ITEM` or `MISSION_ITEM_INT`).

Mission Storage (AP_Mission)

When a `NAV_WAYPOINT` is stored in ArduPilot's EEPROM, the parameters are packed for efficiency:

- **Copter:** `param1` (Delay) is stored in the `p1` field.
- **Plane:** `param2` (Acceptance Radius) and `param3` (Pass-by Distance) are packed into the `p1` field.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command is triggered in `ModeAuto::do_nav_wp` (mode_auto.cpp).

1. **Path Generation:** The location is passed to the `AC_WPNav` library. ArduCopter uses **S-Curve** trajectory generation to ensure smooth acceleration and deceleration.
2. **Cornering Logic:**
   - If **Delay (p1)** is 0, the flight controller looks ahead to the *next* waypoint. It plans a path that rounds the corner, maintaining velocity.
   - If **Delay (p1)** is greater than 0, the vehicle is forced to stop exactly at the coordinate and wait for the specified time before proceeding.
3. **Yaw Control:** The vehicle will typically face the next waypoint unless a `CONDITION_YAW` or `ROI` command has overridden the heading logic.

ArduPlane Implementation

In Plane, the logic resides in `commands_logic.cpp`.

1. **L1 Controller:** Plane uses an L1 guidance algorithm which creates a lateral acceleration command to guide the aircraft onto the track between waypoints.
2. **Acceptance Radius (p2):** Defines a cylinder around the waypoint. As soon as the aircraft enters this cylinder, the waypoint is considered "complete." If set to 0, the global `WP_RADIUS` parameter is used.
3. **Pass-By (p3):** Allows the plane to start the turn early. If the aircraft comes within `p3` meters of the waypoint, it transitions to the next leg.

Data Fields (MAVLink)

- `param1` **(Delay):** Time to loiter at the waypoint in seconds (Copter only).
- `param2` **(Acceptance Radius):** Distance in meters at which the waypoint is considered reached (Plane only).
- `param3` **(Pass-by Distance):** For planes, the distance at which the aircraft should begin its turn towards the next waypoint.
- `param4` **(Yaw):** Desired yaw angle (Copter only).
- `x` **(Latitude):** Target latitude.
- `y` **(Longitude):** Target longitude.
- `z` **(Altitude):** Target altitude.

Practical Use Cases

1. **Survey Grid:**
   - *Scenario:* Mapping a field with a camera.
   - *Action:* A series of `NAV_WAYPOINT` commands are used to define the "lawnmower" pattern. `Delay` is set to 0 for continuous flight.
2. **Inspection Stop:**
   - *Scenario:* Inspecting a power line insulator.
   - *Action:* The drone flies to a `NAV_WAYPOINT` positioned near the insulator with a `Delay` of 5 seconds, allowing the camera to capture high-quality images while stationary.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:1547**: `do_nav_wp` implementation for Copter.
- **ArduPlane/commands_logic.cpp**: Plane mission command logic.
- **libraries/AP_Mission/AP_Mission.cpp:1065**: Mission storage packing for waypoints.

NAV_LOITER_UNLIM (ID 17)

Summary

The `NAV_LOITER_UNLIM` command instructs the vehicle to fly to a specified location and loiter (circle or hover) there indefinitely. This is a "blocking" command; the mission will not proceed to the next waypoint unless the pilot manually skips the item or changes the flight mode.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Packet Param 3 (Radius/Direction):**
    - Stored in the internal `p1` field.
    - **Radius:** The absolute value is stored as a 16-bit integer (meters).
    - **Direction:** The sign indicates direction (`-` = Counter-Clockwise, `+` = Clockwise).
- **Packet Param 4 (Yaw/XTrack):**
    - For Copter, `param4` represents the desired yaw (0 to 360 deg).
    - For Plane, `param4` > 0 enables "Tangent Exit" crosstracking.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_loiter_unlimited` (mode_auto.cpp).

1. **Target Acquisition:** The location is passed to the `AC_WPNav` library.
2. **Behavior:**
    - The copter flies to the target Lat/Lon/Alt.
    - It enters `WP_NAV` loiter mode, holding position against wind using the GPS/INS solution.
    - **Heading:** The vehicle will face the next waypoint (if one exists) or hold the current heading, unless overridden by a `CONDITION_YAW` command.

ArduPlane Implementation

In Plane, the logic resides in `commands_logic.cpp` and `verify_loiter_unlim`.

1. **Loiter Radius:** The plane uses the `param3` radius. If `param3` is 0, it defaults to the global `WP_LOITER_RAD` parameter.
2. **Orbit Direction:**
    - **Positive Radius:** Clockwise.
    - **Negative Radius:** Counter-Clockwise.
3. **L1 Controller:** The plane maintains the orbit using L1 guidance, adjusting bank angle to compensate for wind drift.

Data Fields (MAVLink)

- `param1`: Empty.

- `param2`: Empty.
- `param3` **(Radius):** Radius in meters. Positive = Clockwise, Negative = Counter-Clockwise. If 0, uses default.
- `param4` **(Yaw/XTrack):** Desired Yaw (deg) [Copter] or XTrack location [Plane].
- `x` **(Latitude):** Target latitude.
- `y` **(Longitude):** Target longitude.
- `z` **(Altitude):** Target altitude.

Theory: The L1 Loiter Logic (Plane)

For fixed-wing aircraft, loitering is not static; it is a dynamic energy management state.

- **Wind Compensation:** To maintain a perfect circle over the ground, the plane must vary its bank angle and ground speed.
- **Downwind:** Ground speed increases; bank angle must increase to increase centripetal force ($F_c = \frac{mv^2}{r}$).
- **Upwind:** Ground speed decreases; bank angle must decrease.

$$extBankAngle \propto \arctan\left(\frac{v_{ground}^2}{g \cdot r}\right)$$

Practical Use Cases

1. **Observation Point:**
   - *Scenario:* A surveillance drone needs to watch a specific intersection for an unknown duration.
   - *Action:* The operator inserts a `NAV_LOITER_UNLIM` command. The drone holds station until the operator commands it to "Resume Mission" or "Return to Launch".
2. **Holding Pattern:**
   - *Scenario:* Air traffic control (ATC) requires a delay before landing.
   - *Action:* The aircraft enters a holding pattern at a safe altitude.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp**: `do_loiter_unlimited` implementation.
- **libraries/AP_Mission/AP_Mission.cpp:1085**: Parameter packing for storage.

NAV_LOITER_TURNS (ID 18)

Summary

The `NAV_LOITER_TURNS` command instructs the vehicle to fly to a specified location and orbit it for a specific number of full 360-degree rotations. This is commonly used to ensure a camera has multiple opportunities to capture a target or to allow a vehicle to shed altitude/airspeed before a landing approach.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

ArduPilot uses complex bit-packing to store the number of turns and the radius within the limited EEPROM space.

- **Turns (Param 1):**
  - Stored in the low byte of `p1`.
  - Supports fractional turns (e.g., 1.5 turns) by multiplying by 256 and setting a "Fractional" bit in storage.
- **Radius (Param 3):**
  - Stored in the high byte of `p1`.
  - If the radius is > 255m, it is stored divided by 10, and a "Large Radius" bit is set.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, this command utilizes the `AC_CircleNav` library via `ModeAuto::do_circle` (mode_auto.cpp).

1. **Transition:** The copter first flies to the *edge* of the circle (`CIRCLE_MOVE_TO_EDGE`).
2. **Tracking:** Once on the perimeter, it begins counting the cumulative angle traveled.
3. **Completion:** The command completes when:

$$\frac{|\text{Total Angle Traveled}|}{2\pi} \geq \text{Target Turns}$$

4. **Yaw:** The vehicle can be configured to face the center, face the direction of travel, or hold a fixed heading via the `CIRCLE_YAW_BEHAVE` parameter.

ArduPlane Implementation

In Plane, the logic resides in `Plane::verify_loiter_turns`.

1. **L1 Orbit:** The plane enters an L1 loiter at the specified radius.
2. **Cumulative Counting:** The autopilot integrates the change in bearing to the center to track rotations.
3. **Exit Strategy:** Unlike Copter, Plane implements a **Secondary Heading Goal**. Once the turns are completed, the plane does not immediately exit. It continues orbiting until it is pointing toward the *next* waypoint, ensuring a smooth tangential exit.

Data Fields (MAVLink)

- `param1` **(Turns):** Number of full orbits to complete.
- `param2` **:** Empty.
- `param3` **(Radius):** Orbit radius in meters. Positive = Clockwise, Negative = Counter-Clockwise.
- `param4` **(XTrack):**
    - **0:** Cross-track from the center of the loiter.
    - **1:** Cross-track from the tangent exit location (Plane only).
- `x` **(Latitude):** Center of the orbit.
- `y` **(Longitude):** Center of the orbit.
- `z` **(Altitude):** Target altitude.

Theory: Angular Momentum vs. Ground Track

In the presence of wind, a constant airspeed orbit results in a "drifted" ground track. ArduPilot's L1 controller mathematically solves for the bank angle required to maintain a perfect circle over the ground.

- **Wind Speed (V_w):** Affects the ground speed (V_g).
- **Maximum Bank Angle:** Constrained by `ROLL_LIMIT_DEG`. If the wind is too high, the plane may be unable to maintain the radius and will "blow out" of the circle.

Practical Use Cases

1. **Aerial Cinematography:**
    - *Scenario:* A photographer wants a 360-degree reveal of a mountain peak.
    - *Action:* Use `NAV_LOITER_TURNS` with `Turns = 1` and `CIRCLE_YAW_BEHAVE = 1` (Face Center). The drone will orbit the peak while the camera stays locked on the target.
2. **Communications Relay:**
    - *Scenario:* A plane acts as a data link between a ground station and a distant rover.
    - *Action:* Loiter for 50 turns over the rover's location to provide persistent coverage.

Key Parameters

- `CIRCLE_RADIUS`: Default radius if mission radius is 0.
- `CIRCLE_RATE`: Maximum angular speed (deg/s) for the orbit.
- `WP_LOITER_RAD`: (Plane) Default loiter radius.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:2256**: `verify_circle` implementation.
- **ArduPlane/commands_logic.cpp:740**: `verify_loiter_turns` implementation.

NAV_LOITER_TIME (ID 19)

Summary

The `NAV_LOITER_TIME` command instructs the vehicle to fly to a location and loiter (hover or circle) for a specific duration in seconds. The timer begins only after the vehicle has reached the target coordinate.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Time (Param 1):** Stored in the internal `p1` field as a 16-bit integer (seconds).
- **Significance:** Since it uses all 16 bits for time, there is **no room** in the packed mission structure to store a custom radius for this specific command. It will always use the vehicle's default loiter radius.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_loiter_time` (mode_auto.cpp).

1. **Waypoint Entry:** The copter flies to the target Lat/Lon/Alt using the standard S-Curve path.
2. **Timer Start:** The `loiter_time` variable is initialized to 0. Once `reached_wp_destination()` returns true, the current system time is recorded.
3. **Completion:** The command completes when:

$$\text{millis}() - \text{startTime} \geq \text{Param1} \cdot 1000$$

ArduPlane Implementation

In Plane, the logic resides in `Plane::verify_loiter_time`.

1. **Target Radius:** Always uses the global `WP_LOITER_RAD` parameter.
2. **Timer Logic:** Similar to Copter, the timer starts only when `reached_loiter_target()` is true.
3. **Exit Heading:** Once the time expires, Plane transitions to a secondary goal: **verify_loiter_heading**. It will continue orbiting until it points toward the next mission item, ensuring it exits the circle on a tangent.

Data Fields (MAVLink)

- `param1` **(Time):** Duration to loiter in seconds.
- `param2` **:** Empty.
- `param3` **:** (Ignored by ArduPilot storage).
- `param4` **(Yaw):** Desired Yaw angle (deg).
- `x` **(Latitude):** Target location.
- `y` **(Longitude):** Target location.
- `z` **(Altitude):** Target altitude.

Theory: Time vs. Turns

While `NAV_LOITER_TURNS` is distance-dependent, `NAV_LOITER_TIME` is purely temporal.

- **Wind Impact:** In a high-wind scenario, a Plane might complete fewer turns in 60 seconds than in a no-wind scenario because it spends more time fighting the headwind.
- **Consistency:** For a Copter (multirotor), `LOITER_TIME` is the preferred method for creating pauses in a mission (e.g., waiting for a camera buffer to clear).

Practical Use Cases

1. **Timed Surveillance:**
    - *Scenario:* A security drone must orbit a gate for exactly 5 minutes every hour.
    - *Action:* Use `NAV_LOITER_TIME` with `Time = 300`.
2. **Sensor Warm-up:**
    - *Scenario:* A specialized gas sensor requires 30 seconds of stable airflow to calibrate.
    - *Action:* Insert a `NAV_LOITER_TIME` at the start of the survey grid to allow the sensor to stabilize.

Key Parameters

- `WP_LOITER_RAD`: (Plane) Controls the circle size.
- `LOITER_REPOSITION`: (Copter) Allows the pilot to "nudge" the loiter position with stick inputs without breaking the mission.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:1738**: `do_loiter_time` implementation.
- **ArduPlane/commands_logic.cpp:710**: `verify_loiter_time` implementation.

NAV_RETURN_TO_LAUNCH (ID 20)

Summary

The `NAV_RETURN_TO_LAUNCH` (RTL) command instructs the vehicle to return to the home location or a designated rally point. This command is the primary safety mechanism for terminating a mission and returning the aircraft to the operator.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command (Failsafe trigger).

Mission Storage (AP_Mission)

- **Packing:** This command contains no additional parameters (`param1` through `param7` are unused in missions). It simply serves as a state-change marker to trigger the vehicle's RTL mode.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_RTL()` (mode_auto.cpp).

1. **State Transition:** The vehicle switches from `AUTO` mode to `RTL` mode.
2. **Safety Logic:**
   - **Climb:** The vehicle first climbs to the `RTL_ALT` (or stays at current altitude if already higher, depending on `RTL_CLIMB_MIN`).
   - **Return:** It flies in a straight line towards Home.
   - **Descent:** Once over home, it hovers for the duration of `RTL_LOIT_TIME` before initiating the final land.

ArduPlane Implementation

In Plane, the logic resides in `Plane::verify_RTL` (commands_logic.cpp).

1. **Path Planning:** Plane identifies the "best" return location, which could be the Home point or the nearest Rally Point.
2. **Altitude Management:** The aircraft targets the `RTL_ALTITUDE` (Plane specific).
3. **Orbit:** Upon arrival, the plane enters a loiter (circle) at the return point. Unlike Copter, a standard Plane RTL does **not** automatically land unless the mission specifically contains a landing sequence or a landing failsafe is triggered.

Data Fields (MAVLink)

- `param1` to `param7`: Reserved / Unused.

Theory: The "Cone of Safety"

A critical concept in RTL logic is the **Return Cone**.

- **Problem:** If a drone is very far away, returning at a low altitude might hit terrain. If it's very close, climbing to a high "Return Altitude" is a waste of energy.
- **Solution:** Advanced ArduPilot configurations use a "Safe Return Path" or Rally Points to ensure the vehicle always has a clear line of sight to a safe recovery zone.

Practical Use Cases

1. **Mission Completion:**
   - *Scenario:* A mapping mission has finished its last photo transect.
   - *Action:* The mission list ends with `NAV_RETURN_TO_LAUNCH` to bring the drone back to the takeoff area for recovery.
2. **Radio Failsafe:**
   - *Scenario:* The control link between the GCS and the drone is severed.
   - *Action:* The internal failsafe logic injects a `NAV_RETURN_TO_LAUNCH` command (equivalent) to recover the asset automatically.

Key Parameters

- `RTL_ALT`: (Copter) Altitude to return at.
- `RTL_ALTITUDE`: (Plane) Altitude to return at.
- `RTL_RADIUS`: (Plane) Radius of the loiter circle over home.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:2184**: `verify_RTL` implementation.
- **ArduPlane/commands_logic.cpp:795**: `verify_RTL` implementation.

NAV_LAND (ID 21)

Summary

The `NAV_LAND` command initiates the final descent and landing sequence. This command transitions the vehicle from flight to a grounded state, typically concluding with the motors disarming once the autopilot detects a successful touchdown.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command (Guided Mode).

Mission Storage (AP_Mission)

- **Packet Param 1 (Abort Alt):**
  - **Plane:** Stored in the internal `p1` field as a 16-bit integer (meters). This defines the altitude to climb to if the landing is aborted.
- **Packet Param 4 (Yaw):**
  - **Plane:** Stored in the `loiter_ccw` field if `param4` is negative (used for deepstall direction).

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_land` (mode_auto.cpp).

1. **Vertical Descent:** The copter descends at the speed specified by `LAND_SPEED`.
2. **Touchdown Detection:** The flight controller monitors the throttle and vertical velocity. If the throttle is at minimum and the altitude is not changing for a specific duration, it sets the "Landed" flag.
3. **Disarm:** Once "Landed" is confirmed, the motors are automatically disarmed.
4. **Repositioning:** If `LAND_REPOSITION` is enabled, the pilot can "nudge" the copter horizontally during the descent to avoid obstacles.

ArduPlane Implementation

In Plane, the logic resides in the `AP_Landing` library and `Plane::do_land`.

1. **Flare Logic:** The plane follows a glideslope until it reaches the `LAND_FLARE_ALT`. At this point, it raises the nose and cuts the throttle to "flare" for a smooth touchdown.
2. **Abort Support:** If the pilot triggers an abort or the vehicle enters a go-around state, it climbs to the "Abort Altitude" specified in the command's `param1`.
3. **Deepstall (Advanced):** For airframes without wheels, Plane supports "Deepstall" landing, where the aircraft intentionally stalls its wing at high altitude to fall vertically onto a soft target (like a net or tall grass).

Data Fields (MAVLink)

- `param1` **(Abort Alt):** Altitude to climb to on abort (Plane only).
- `param2`: Empty.

- `param3`: Empty.
- `param4` **(Yaw):** Desired yaw heading (Copter only).
- `x` **(Latitude):** Target landing point.
- `y` **(Longitude):** Target landing point.
- `z` **(Altitude):** Target altitude (typically 0).

Theory: The Ground Effect

As a multicopter nears the ground (within ~0.5m), it enters **Ground Effect**. The downwash from the rotors creates a high-pressure cushion that increases lift efficiency.

- **The Hazard:** If the autopilot is not tuned correctly, this extra lift can cause the vehicle to "bounce" off the cushion, making it difficult to detect a true touchdown.
- **The Code:** ArduPilot's landing detector uses a robust statistical filter to distinguish between ground effect bounces and a firm touchdown.

Practical Use Cases

1. **Standard Mission End:**
    - *Scenario:* A mapping mission completes its grid and returns to the takeoff point.
    - *Action:* The last item is `NAV_LAND` at the Home coordinates.
2. **Emergency Forced Landing:**
    - *Scenario:* The battery reaches a critical level during a mission.
    - *Action:* The GCS or onboard script sends a `MAV_CMD_NAV_LAND` at the vehicle's current location to terminate the flight safely.

Key Parameters

- `LAND_SPEED`: Descent rate in cm/s during final landing.
- `LAND_ALT_LOW`: Altitude (cm) below which the land speed is reduced to the "slow" rate.
- `LAND_FLARE_ALT`: (Plane) Altitude to begin the flare.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:1651**: `do_land` implementation.
- **ArduPlane/commands_logic.cpp:402**: `do_land` implementation.

NAV_TAKEOFF (ID 22)

Summary

The `NAV_TAKEOFF` command instructs the vehicle to lift off from the ground and climb to a specified altitude. This is typically the first command in an autonomous mission.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command (Guided Mode).

Mission Storage (AP_Mission)

- **Packet Param 1 (Pitch/Ignored):**
    - **Plane:** Minimum pitch angle during takeoff (degrees).
    - **Copter:** Ignored (usually 0).
- **Packet Param 7 (Altitude):**
    - Target altitude in meters.
    - *Note:* The frame of reference depends on the mission command frame (Relative, Absolute, or Terrain).

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_takeoff`, which utilizes the `Mode::_TakeOff` class logic (takeoff.cpp).

1. **State Machine:**
    - The copter verifies it is armed and on the ground.
    - It spools up motors to the `THROTTLE_UNLIMITED` state.
2. **Climb Logic:**
    - The vertical position controller is fed a target altitude curve.
    - **Departure:** If the copter is still on the ground, it slews the throttle until it detects a climb (acceleration > threshold) or throttle saturation.
    - **No Navigation Zone:** If `WP_NAVALT_MIN` is set, the copter climbs straight up without horizontal navigation until it clears the "danger zone" (e.g., fence posts, people).
3. **Completion:** Takeoff is considered complete when the target altitude is reached (within a tolerance).

ArduPlane Implementation

In Plane, the logic resides in `Plane::do_takeoff` (commands_logic.cpp).

1. **Pitch Target:** The aircraft aims for the pitch angle specified in `param1`. If 0, it defaults to a safe minimum (typically 10-15 degrees depending on tuning).
2. **Heading Lock:** The plane locks its heading to the ground course projected towards the *next* waypoint. It uses the rudder (and potentially differential thrust) to fight crosswinds while on the ground.

3. **Throttle Management:** Throttle is clamped to maximum until the `TKOFF_THR_MIN_ACC` (acceleration) or `TKOFF_THR_MIN_SPD` (airspeed) thresholds are met, preventing premature motor spindown.

Data Fields (MAVLink)

- `param1` **(Pitch):** Minimum pitch angle in degrees (Plane only).
- `param2`: Empty.
- `param3`: Empty.
- `param4` **(Yaw):** Desired Yaw angle (deg). NaN to use current heading.
- `x` **(Latitude):** Target latitude (optional, used for initial heading alignment).
- `y` **(Longitude):** Target longitude (optional).
- `z` **(Altitude):** Target altitude in meters.

Theory: The "No-Nav" Zone

A critical concept in autonomous takeoff is the **"No-Nav" Zone** (controlled by `WP_NAVALT_MIN`).

- **Problem:** GPS positions drift. If a drone tries to navigate horizontally while still touching the ground, the landing gear might snag on grass or uneven terrain, causing a tip-over (Dynamic Rollover).
- **Solution:** The autopilot suppresses all horizontal position corrections (Roll/Pitch for Copter) until the vehicle has climbed to a safe altitude (e.g., 2 meters). It effectively shoots straight up like a rocket before engaging the navigation controller.

Practical Use Cases

1. **Hand Launch (Plane):**
    - *Scenario:* Launching a fixed-wing surveyor without a runway.
    - *Action:* The pilot shakes the plane to arm it (Shake-to-Wake). The `NAV_TAKEOFF` command manages the throttle delay and initial climb out, ensuring the prop doesn't spin up until the plane is thrown.
2. **Confined Area (Copter):**
    - *Scenario:* Taking off from a deep urban canyon or forest clearing.
    - *Action:* `WP_NAVALT_MIN` is set to 15m. The `NAV_TAKEOFF` command ensures the drone rises vertically above the treeline before attempting to fly to the first waypoint.

Key Codebase Locations

- **ArduCopter/takeoff.cpp**: Copter takeoff state machine.
- **ArduPlane/commands_logic.cpp:372**: Plane takeoff logic.
- **libraries/AP_Mission/AP_Mission.cpp**: Command unpacking.

NAV_LOITER_TO_ALT (ID 31)

Summary

The `NAV_LOITER_TO_ALT` command instructions the vehicle to fly to a location and loiter until a target altitude is reached. This is an essential "climb/descend" command for missions where altitude changes must be handled safely before continuing to the next waypoint.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Radius (Param 2):** Stored in the internal `p1` field as a 16-bit integer (meters).
- **Significance:** This command *does* support a custom radius in EEPROM storage, unlike `NAV_LOITER_TIME`.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_loiter_to_alt` (mode_auto.cpp).

1. **Vertical Convergence:** The copter holds the XY position while the vertical controller (Z) targets the altitude specified in `param7` (packet's `z` field).
2. **Completion:** The command completes only when:
   - `reached_destination_xy` is true (vehicle is over the coordinate).
   - `reached_alt` is true (vertical error is within tolerance).

ArduPlane Implementation

In Plane, the logic resides in `Plane::verify_loiter_to_alt`.

1. **Stuck Detection:** ArduPilot includes a safety feature for planes. If the plane has been loitering but is **unable to achieve the target altitude** (e.g., due to low power or extreme downdrafts), it will eventually time out and report "Loiter to alt was stuck" to the GCS, allowing the mission to continue rather than circling until the battery dies.
2. **Exit Strategy:** Once altitude is reached, the plane continues loitering until its heading aligns with the next waypoint (Tangent Exit).

Data Fields (MAVLink)

- `param1` **(Heading Required):** If 1, the aircraft will not leave the loiter until heading toward the next waypoint (Plane only).
- `param2` **(Radius):** Orbit radius in meters.
- `param3`: Empty.
- `param4` **(XTrack):** Exit tangent control (Plane).
- `x` **(Latitude):** Target location.
- `y` **(Longitude):** Target location.

- `z` **(Altitude):** Target altitude.

Theory: Spiral Climbs and Energy Management

For fixed-wing aircraft, `LOITER_TO_ALT` is safer than a direct waypoint climb.

- **Stall Prevention:** By climbing in a circle, the aircraft maintains a consistent airspeed and bank angle. A direct climb might result in a high pitch angle and potential stall if the autopilot attempts to climb too aggressively.
- **Thermal Hunting:** In glider modes (SOAR), `LOITER_TO_ALT` is used to stay within a rising thermal until a safe cruise altitude is reached.

Practical Use Cases

1. **Mountain Clearance:**
   - *Scenario:* A drone needs to cross a 2000m ridge from a takeoff point at 500m.
   - *Action:* Place a `NAV_LOITER_TO_ALT` at 2100m before the ridge waypoint. This ensures the drone circles and gains the required altitude *before* attempting the crossing.
2. **Safe Landing Approach:**
   - *Scenario:* A plane is returning at high altitude and needs to descend to 50m for a landing flare.
   - *Action:* Use `NAV_LOITER_TO_ALT` to spiral down to 50m over the runway threshold, preventing an overspeed approach.

Key Parameters

- `WP_LOITER_RAD`: Default radius.
- `ALT_HOLD_RTL`: (Plane) Minimum altitude for RTL/Loiter safety.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:2175**: `verify_loiter_to_alt` implementation.
- **ArduPlane/commands_logic.cpp:766**: `verify_loiter_to_alt` implementation.

NAV_SPLINE_WAYPOINT (ID 82)

Summary

The `NAV_SPLINE_WAYPOINT` command defines a 3D coordinate that the vehicle must pass through using a curved, "spline" trajectory. Unlike the standard linear waypoint, which produces sharp corners, the spline waypoint calculates a smooth path that considers the position of the previous and next waypoints to ensure the vehicle never has to come to a complete stop.

Status

Supported (ArduCopter Only)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Packet Param 1 (Delay):** Stored in the internal `p1` field as a 16-bit integer (seconds).
- **Packet Param 2-4:** Empty.
- **x, y, z:** Target Latitude, Longitude, and Altitude.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_spline_wp` (mode_auto.cpp).

1. **Catmull-Rom Spline:** ArduPilot uses a modified **Catmull-Rom** spline algorithm. To calculate the curve for the current segment (Segment B), the algorithm requires four points:
   - P_0: The waypoint *before* the previous one.
   - P_1: The previous waypoint (start of the curve).
   - P_2: The current spline waypoint (target).
   - P_3: The *next* mission item (used to determine the exit velocity vector).
2. **Velocity Control:** The spline ensures that the velocity vector at P_2 is tangent to the curve, allowing for high-speed passes without the "jerk" associated with linear cornering.
3. **Completion:** If `Delay (p1)` is 0, the waypoint is considered complete as soon as the vehicle passes through the coordinate, allowing it to transition immediately to the next leg. If `Delay > 0`, the vehicle will decelerate to a stop at the coordinate and wait before continuing.

Data Fields (MAVLink)

- `param1` **(Delay):** Hold time in seconds at the waypoint.
- `x` **(Latitude):** Target latitude.
- `y` **(Longitude):** Target longitude.
- `z` **(Altitude):** Target altitude.

Theory: The Hermite Spline Formulation

The path between waypoints P_1 and P_2 is defined by a cubic polynomial in terms of time $t \in [0, 1]$:

$$\mathbf{P}(t) = (2t^3 - 3t^2 + 1)\mathbf{P}_1 + (t^3 - 2t^2 + t)\mathbf{T}_1 + (-2t^3 + 3t^2)\mathbf{P}_2 + (t^3 - t^2)\mathbf{T}_2$$

Where:

- $\mathbf{P}_1, \mathbf{P}_2$ are the positions.
- $\mathbf{T}_1, \mathbf{T}_2$ are the tangent vectors (velocities) at those points, derived from the surrounding waypoints.

Practical Use Cases

1. **Cinematic Fly-bys:**
    - *Scenario:* A drone needs to fly a smooth arc around a building.
    - *Action:* Use a sequence of `NAV_SPLINE_WAYPOINT` items. The resulting path will be a fluid curve, preventing the "robotic" stop-and-turn behavior of standard waypoints.
2. **Obstacle Avoidance in Speed Runs:**
    - *Scenario:* A racing drone needs to navigate a series of gates at maximum speed.
    - *Action:* Spline waypoints allow the drone to maintain its kinetic energy by "rounding" the corners optimally.

Key Parameters

- `WPNAV_SPEED`: Maximum horizontal speed between waypoints.
- `WPNAV_ACCEL`: Maximum acceleration used to define the spline's "tightness."

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:1614**: Spline waypoint initialization.
- **libraries/AC_WPNav/AC_WPNav.cpp**: The core spline math implementation.

NAV_ALTITUDE_WAIT (ID 83)

Summary

The `NAV_ALTITUDE_WAIT` command is a specialized mission item designed for high-altitude balloon launches or gliders. It puts the vehicle into an "Idle" or "Wait" state until it reaches a target altitude or a specific vertical speed, allowing for autonomous activation after a balloon burst or a release from a mother-ship.

Status

Supported (ArduPlane Only)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Altitude (Param 1):** Target altitude (meters).
- **Descent Rate (Param 2):** Vertical speed (m/s) to trigger completion (e.g., detecting a burst).
- **Wiggle Time (Param 3):** Frequency (seconds) to move control surfaces to prevent freezing in extreme cold.
- **Packing:** Stored in the `altitude_wait` content struct.

Execution (Engineer's View)

High Altitude Logic

The command is managed in `Plane::do_altitude_wait` (commands_logic.cpp).

1. **Idle State:** The autopilot sets `auto_state.idle_mode = true`. In this state, servos are held at trim, and the motor is typically disabled.
2. **The "Wiggle":** At high altitudes ($> 15km$), temperatures drop below $-50^{\circ}C$. Lubricants in servos can thicken or freeze. ArduPilot uses `Param 3` to periodically cycle the servos, using internal friction to generate enough heat to keep the linkages moving.
3. **Burst Detection:** The autopilot monitors the vertical velocity (V_z). If V_z becomes more negative than `Param 2` (indicating the balloon has burst and the vehicle is falling), it completes the command immediately.
4. **Completion:** Once the target altitude is reached or a burst is detected, the mission advances to the next item (usually a `NAV_TAKEOFF` or `WAYPOINT`), and full flight control is restored.

Data Fields (MAVLink)

- `param1` **(Alt):** Target altitude (m).
- `param2` **(Descent):** Descent rate (m/s).
- `param3` **(Wiggle):** Time between wiggles (s).
- `param4` to `param7`: Unused.

Theory: Energy Conservation in Near-Space

In a balloon-launch mission, the vehicle is a passenger for 90\% of the ascent.

- **Thermal Management:** Passive electronics cooling is non-existent in the thin upper atmosphere. By idling the CPU and servos, the vehicle minimizes internal heat generation until the "Real" flight begins.
- **Pressure Dynamics:** The autopilot uses the EKF's altitude solution, which fuses barometric and GPS data. At very high altitudes, barometric pressure becomes non-linear; ArduPilot's `AP_Baro` library handles the transition to GPS-dominant altitude sensing safely.

Practical Use Cases

1. **Weather Balloon Glider:**
   - *Scenario:* A glider is carried to 30,000m by a balloon.
   - *Action:* `NAV_ALTITUDE_WAIT (Alt: 30000, Descent: 5m/s, Wiggle: 60s)`. The drone stays "asleep" during the 2-hour ascent, wiggling every minute to stay limber, and wakes up the moment the balloon pops.
2. **Drop-Test:**
   - *Scenario:* Dropping a test airframe from a larger plane.
   - *Action:* Mission starts with `NAV_ALTITUDE_WAIT` to detect the rapid descent after release.

Key Parameters

- `TKOFF_THR_DELAY`: Often used after this command to delay motor start until clear of the balloon debris.

Key Codebase Locations

- **ArduPlane/commands_logic.cpp:90**: Mission command initialization.
- **ArduPlane/pullup.cpp**: Transition to active flight after wait.

NAV_VTOL_TAKEOFF (ID 84)

Summary

The `NAV_VTOL_TAKEOFF` command instructs a QuadPlane (or Tilt-Rotor) to lift off vertically using its multicopter motors and then transition to forward, fixed-wing flight. This command is distinct from a standard `NAV_TAKEOFF`, as it explicitly manages the high-energy state transition between hovering and cruising.

Status

Supported (ArduPlane / QuadPlane Only)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Packing:** ArduPilot stores the command in the standard location structure. The altitude is extracted from the `z` field of the MAVLink packet.

Execution (Engineer's View)

ArduPlane (QuadPlane) Implementation

In QuadPlane, the command triggers `QuadPlane::do_vtol_takeoff` (quadplane.cpp).

1. **Vertical Phase:**
    - The vehicle uses the vertical position controller (`pos_control`) to rise to the target altitude.
    - **Time-to-Altitude Estimate:** The autopilot calculates the expected duration of the climb using the following kinematic model:

$$t_{accel} = \frac{V_{max} - V_z}{a}$$

$$d_{accel} = V_z \cdot t_{accel} + \frac{1}{2} a \cdot t_{accel}^2$$

$$t_{total} = \max(t_{accel}, 0) + \max\left(\frac{d_{total} - d_{accel}}{V_{max}}, 0\right)$$

2. **Transition Phase:**
    - Once the vertical target is reached, the autopilot begins the **Transition to Fixed-Wing**.
    - It engages the forward motor (pusher) while maintaining multicopter attitude control.
    - Once the airspeed reaches the `ARSPD_FBW_MIN`, the multicopter motors are phased out, and the vehicle becomes a standard airplane.

Data Fields (MAVLink)

- `param1`: Empty.
- `param2`: Empty.
- `param3`: Empty.
- `param4` **(Yaw):** Target heading (degrees).
- `x` **(Latitude):** Target latitude.
- `y` **(Longitude):** Target longitude.
- `z` **(Altitude):** Target altitude (meters).

Theory: The "V-Alpha" Transition

The transition from vertical to horizontal flight is the most dangerous phase of flight for a VTOL aircraft.

- **Wing Loading:** During transition, lift is shared between the vertical rotors and the wing.
- **Pitch Control:** As forward speed increases, the elevators gain authority, but the multicopter pitch controller is still active. ArduPilot uses a "Blend" logic to ensure smooth control handover.
- **Stall Risk:** If the transition is too slow, the battery may deplete. If too fast, the aircraft may pitch up violently due to the increased airspeed over the wing.

Practical Use Cases

1. **Runway-Free Long Endurance:**
    - *Scenario:* A fixed-wing plane needs to map 100km of pipeline but must take off from a small jungle clearing.
    - *Action:* Use `NAV_VTOL_TAKEOFF` to rise 30m above the trees before accelerating into high-efficiency fixed-wing mode.
2. **Shipboard Launch:**
    - *Scenario:* Launching from a moving vessel.
    - *Action:* VTOL takeoff allows the aircraft to clear the deck and masts vertically before engaging the pusher motor to match the ship's speed and move away safely.

Key Parameters

- `Q_TAKEOFF_ALT`: Default altitude if mission altitude is 0.
- `Q_TRANSITION_MS`: Duration of the forward motor ramp-up.
- `Q_TILT_MAX`: (Tilt-Rotor only) Maximum tilt angle during transition.

Key Codebase Locations

- **ArduPlane/quadplane.cpp:3298**: VTOL takeoff initialization.
- **ArduPlane/quadplane.cpp:3390**: Transition logic handler.

NAV_VTOL_LAND (ID 85)

Summary

The `NAV_VTOL_LAND` command instructs a hybrid aircraft (QuadPlane) to transition from fixed-wing cruise to multicopter mode and perform a vertical descent to a precise location. This allows long-range fixed-wing assets to land in confined spaces without a runway.

Status

Supported (ArduPlane / QuadPlane Only)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a RTL/Failsafe action.

Mission Storage (AP_Mission)

- **Packet Param 1 (Options):** Stored in the internal `p1` field. This bitmask allows for advanced landing behaviors, such as "Descending Loiter" before the final drop.

Execution (Engineer's View)

ArduPlane (QuadPlane) Implementation

In QuadPlane, the command triggers a multi-stage state machine (quadplane.cpp).

1. **Back-Transition:**
    - The aircraft approaches the target waypoint in fixed-wing mode.
    - It cuts the forward motor and engages the VTOL motors (multicopter) as it decelerates below the stall speed.
2. **Approach:**
    - **Spiral Descent:** If configured, the aircraft loiters over the landing point while descending.
    - **Wind Alignment:** The autopilot attempts to align the vehicle's heading into the wind to minimize lateral drift during the hover phase.
3. **Final Descent:**
    - The vehicle uses the multicopter vertical controller to descend at `Q_LAND_SPEED`.
    - It uses the `landing_detect` logic to verify touchdown before disarming.

Data Fields (MAVLink)

- `param1` **(Options):** Bitmask for landing behavior.
- `param2`: Empty.
- `param3`: Empty.
- `param4` **(Yaw):** Desired heading for landing.
- `x` **(Latitude):** Target landing location.
- `y` **(Longitude):** Target landing location.
- `z` **(Altitude):** Final altitude (typically 0).

Theory: The Q-Approach

The "Q-Approach" is a sophisticated landing method designed for high-performance VTOLs.

- **Aerodynamic Transition:** Unlike multicopters, QuadPlanes have significant wing lift. If they descend too quickly while moving forward, the wing can generate asymmetric lift (if rolling), leading to instability.
- **Airbrake Effect:** ArduPilot can use the multicopter motors as "airbrakes" by spinning them up at low power while still in forward flight, creating high drag to slow the aircraft down for the final hover.

Practical Use Cases

1. **Precision Recovery:**
    - *Scenario:* A 3-meter wingspan drone returning to a 10m x 10m clearing.
    - *Action:* `NAV_VTOL_LAND` ensures the aircraft flies the long-distance return at high efficiency, then converts to a hovering drone for a pin-point landing.
2. **Autonomous Charging Dock:**
    - *Scenario:* A drone landing on a robotic charging platform.
    - *Action:* Use `NAV_VTOL_LAND` in conjunction with a Precision Landing sensor (IR/Camera) to land within centimeters of the charging pins.

Key Parameters

- `Q_TRANS_DECEL`: Deceleration rate (m/s/s) during back-transition.
- `Q_LAND_FINAL_ALT`: Altitude above ground at which the aircraft enters its final, slower descent phase.
- `Q_FWD_THR_USE`: Allows the pusher motor to assist with station-keeping in high winds during the vertical descent.

Key Codebase Locations

- **ArduPlane/quadplane.cpp**: Main VTOL landing handler.
- **libraries/AP_Landing/AP_Landing.cpp**: Shared landing library.

NAV_DELAY (ID 93)

Summary

The `NAV_DELAY` command pauses the vehicle's navigation progress for a specified duration or until a specific UTC time. Unlike `CONDITION_DELAY` (which is a mission-logic gate), `NAV_DELAY` is a navigation-level command that usually involves the vehicle hovering or loitering at its current location.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Duration (Param 1):** Seconds to delay (relative).
- **Time (Param 2-4):** Hour, Minute, and Second (UTC) for absolute time delay.
- **Packing:** Stored in the `nav_delay` content struct.

Execution (Engineer's View)

Delay Logic

Execution is handled by `ModeAuto::do_nav_delay` (mode_auto.cpp).

1. **Timer Modes:**
    - **Relative:** The timer starts the moment the vehicle reaches the waypoint.
    - **Absolute (UTC):** The mission will not proceed until the onboard GPS clock matches the requested UTC time. This requires a valid GPS lock and PPS (Pulse Per Second) synchronization.
2. **Vehicle State:** While delaying, the vehicle maintains its current coordinates.
    - **Copter:** Position Hold (Loiter).
    - **Plane:** Loiter (Orbit).
3. **Resumption:** Once `millis() - start_time > duration`, the mission state machine is released to the next item.

Data Fields (MAVLink)

- `param1` **(Delay):** Seconds.
- `param2` **(Hour):** UTC Hour [0-23].
- `param3` **(Min):** UTC Minute [0-59].
- `param4` **(Sec):** UTC Second [0-59].

Theory: Synchronized Swarms

The UTC delay is a foundational tool for **Multi-Vehicle Synchronization**.

- **The Problem:** GPS signals take different times to reach different drones. Onboard clocks can drift.
- **The Solution:** By using UTC time (derived from the atomic clocks on GPS satellites), multiple drones can be commanded to "Start Scan" at the exact same microsecond, regardless of when they took off

or how far they traveled.

Practical Use Cases

1. **Golden Hour Photography:**
   - *Scenario:* A drone needs to take a photo at exactly sunset.
   - *Action:* `NAV_WAYPOINT` → `NAV_DELAY (UTC Time of Sunset)`. The drone flies to the spot and orbits until the lighting is perfect.
2. **Coordinated Drop:**
   - *Scenario:* Two drones dropping a heavy net.
   - *Action:* Both missions use `NAV_DELAY` to synchronize the release at the same UTC second.

Key Parameters

- `GPS_TYPE`: Required for UTC time accuracy.
- `WP_LOITER_RAD`: (Plane) Orbit size during delay.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:723**: Mission command intake.
- **ArduCopter/mode_auto.cpp:2186**: Relative timer verification.

NAV_PAYLOAD_PLACE (ID 94)

Summary

The `NAV_PAYLOAD_PLACE` command (often called "Payload Delivery") instructions the vehicle to descend vertically until it detects that a winch-slung or fixed payload has touched the ground. Once touchdown is detected, the vehicle releases the payload (if a gripper is attached) and climbs back to its original altitude.

Status

Supported (ArduCopter and QuadPlane)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Packet Param 1 (Max Descent):** Stored in the internal `p1` field.
  - **Unit Conversion:** ArduPilot converts this value from meters to centimeters ($cm = m \cdot 100$) before storing it in EEPROM.
- **x, y, z:** The delivery coordinates.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_payload_place` (mode_auto.cpp).

1. **Approach:** The drone flies to the target Lat/Lon at the current mission altitude.
2. **Descent:** The drone begins a vertical descent at the speed defined by `PLDP_SPEED_DN`.
3. **Touchdown Detection:** The autopilot does not use simple altitude for detection. Instead, it monitors the **Thrust/Weight Ratio**.
   - As the payload touches the ground, the load on the motors decreases.
   - When the thrust required to maintain the descent rate drops below the threshold defined by `PLDP_THRESH`, the autopilot considers the payload "Placed".
4. **Release & Recovery:**
   - If a gripper or winch is configured, the autopilot triggers the release action.
   - The drone waits for `PLDP_DELAY` seconds to ensure a clean release.
   - The drone then climbs back to the initial altitude to continue the mission.

Data Fields (MAVLink)

- `param1` **(Max Descent):** The maximum distance (meters) the vehicle is allowed to descend. If the ground is not reached within this distance, the command is aborted to prevent a crash.
- `x` **(Latitude):** Target delivery location.
- `y` **(Longitude):** Target delivery location.
- `z` **(Altitude):** Current flight altitude.

Theory: The Load-Sensing Algorithm

Payload delivery in high-wind or turbulent environments is difficult because the vehicle's thrust is constantly fluctuating. ArduPilot uses a filtered approach to detect the "offloading" of weight:

$$\text{Placed} = \left(\frac{T_{current}}{T_{average\_descent}} \right) < \text{PLDP\_THRESH}$$

Where $T_{current}$ is the instantaneous thrust output and $T_{average\_descent}$ is the learned thrust required to descend at a steady rate with the payload.

Practical Use Cases

1. **Automated Logistics:**
   - *Scenario:* Delivering a medical package to a remote village.
   - *Action:* The drone uses `NAV_PAYLOAD_PLACE`. It descends until the box touches the ground, opens the gripper, and climbs away without needing the pilot to see the ground or manage the descent manually.
2. **Scientific Sensor Deployment:**
   - *Scenario:* Placing a seismometer on a steep, inaccessible slope.
   - *Action:* The `PLDP_THRESH` logic ensures the sensor is firmly on the ground before release, even if the terrain altitude is not perfectly known.

Key Parameters

- `PLDP_THRESH`: The fraction of hover thrust that indicates a touchdown (Default 0.9).
- `PLDP_RNG_MAX`: Uses a downward-facing rangefinder to prevent "false touchdowns" if the drone is too high.
- `PLDP_SPEED_DN`: The vertical velocity during the delivery phase.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:2024**: Copter delivery state machine.
- **ArduPlane/quadplane.cpp**: QuadPlane implementation.

NAV_RALLY_POINT (ID 5100)

Summary

The `NAV_RALLY_POINT` command defines an alternative "Safe Harbor" location for the vehicle. Unlike the Home position (which is usually where the drone took off), Rally Points are used as backup landing or loiter sites that the autopilot can choose from during a failsafe or RTL event based on proximity.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives these commands during a mission upload (specifically when using the MAVLink Rally Protocol).

Mission Storage (AP_Mission)

- **Packing:** Rally points are stored in a dedicated region of the EEPROM, separate from the main waypoint mission, allowing the drone to maintain its safety anchors even if the main mission is cleared.
- **Quantity:** ArduPilot typically supports up to 20-50 individual rally points depending on hardware.

Execution (Engineer's View)

Intelligent RTL

When an RTL (Return to Launch) is triggered:

1. **Comparison:** The autopilot calls `calc_best_rally_or_home_location()` (AP_Rally.cpp).
2. **Distance Heuristic:** It calculates the 2D distance to Home and *all* loaded Rally Points.
3. **Selection:** It chooses the **Geographically Closest** point.
4. **Action:** The vehicle flies to the chosen point and enters a loiter (Plane) or hover (Copter).

Data Fields (MAVLink)

- `x` **(Latitude):** Rally location.
- `y` **(Longitude):** Rally location.
- `z` **(Altitude):** Rally location.

Theory: Distributed Recovery

Standard recovery is centralized (Home). Rally Points enable **Distributed Recovery**.

- **The Hazard:** In long-distance missions (Linear Corridors), the vehicle may be 20km from Home when a battery failsafe occurs. It might not have enough energy to return 20km.
- **The Solution:** By placing Rally Points every 2km along the path, the autopilot ensures it never has to fly more than 1km to find a safe "Lifeboat."

Practical Use Cases

1. **Cross-Country FPV:**

    - *Scenario:* A plane is flying between two mountain peaks.
    - *Action:* Rally points are placed at known-clear meadows. If the radio link fails, the plane flies to the nearest meadow rather than trying to fly back over the peaks.
2. **Redundant Landing Pads:**
    - *Scenario:* A drone landing at a busy facility.
    - *Action:* Multiple rally points represent different landing pads. The drone chooses the one it can reach most efficiently.

Key Parameters

- `RALLY_LIMIT_KM`: The maximum distance the autopilot will look for a rally point.
- `RALLY_INCL_HOME`: Defines if Home should be considered in the proximity calculation.

Key Codebase Locations

- **libraries/AP_Rally/AP_Rally.cpp**: Proximity comparison logic.
- **libraries/GCS_MAVLink/MissionItemProtocol_Rally.cpp**: Communication protocol for rally points.

NAV_SCRIPT_TIME (ID 42702)

Summary

The `NAV_SCRIPT_TIME` command is a powerful hybrid command that combines mission flow control with **Lua Scripting**. Unlike the "Do" version (`DO_SEND_SCRIPT_MESSAGE`), `NAV_SCRIPT_TIME` is a **Blocking Navigation Command**. The mission will not advance to the next item until the Lua script explicitly signals completion or a timeout is reached.

Status

Supported (All Vehicles with Lua Scripting enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Command ID (Param 1):** A user-defined ID the script uses to know what to do.
- **Timeout (Param 2):** Max duration in seconds before the mission auto-advances.
- **Arg 1 & 2 (Param 3-4):** Floats passed to the script.
- **Arg 3 & 4 (x, y):** Integers passed to the script.
- **Packing:** Stored in the `nav_script_time` content struct.

Execution (Engineer's View)

Handshake Logic

The command creates a tight loop between the Mission Engine and the Lua Virtual Machine (AP_Scripting.cpp).

1. **Trigger:** The mission state machine sets the `nav_scripting.done` flag to `false` and records the start time.
2. **Script Detection:** The Lua script polls `mission:get_nav_script_time_id()`.
3. **Operation:** The script performs its task (e.g., complex pattern flying or sensor analysis).
4. **Completion Signal:** The script calls `mission:nav_script_time_done(id)`.
5. **Watchdog:** If the script fails to call the "done" method before `Param 2` seconds pass, the mission engine logs a warning and proceeds to the next item regardless.

Data Fields (MAVLink)

- `param1` **(ID):** Custom command ID.
- `param2` **(Timeout):** Seconds.
- `param3` **(Arg1):** Float.
- `param4` **(Arg2):** Float.
- `x` **(Arg3):** Int.
- `y` **(Arg4):** Int.

Theory: The Co-Processor Model

`NAV_SCRIPT_TIME` implements an **Asynchronous Co-Processor** model for mission logic.

- **Decoupling:** The "Heavy Lifting" (AI, Image Processing, Path Re-planning) happens in the Lua VM, which is sandboxed and memory-managed.
- **Deterministic Safety:** The main C++ flight code remains simple and stable. If the script crashes, the mission watchdog (Timeout) ensures the drone doesn't loiter until it crashes.

Practical Use Cases

1. **AI Search Pattern:**
    - *Scenario:* A drone finds a target but needs to perform a "spiral-out" search to find more.
    - *Action:* `NAV_SCRIPT_TIME (ID: 101, Timeout: 60s)`. The Lua script takes control of the position setpoints, flies the spiral, and signals "Done" once the search is complete.
2. **External Hardware Sync:**
    - *Scenario:* A drone landing on a robotic dock that takes 15 seconds to open.
    - *Action:* The script communicates with the dock, waits for the "Open" signal, and then releases the mission to continue to `NAV_LAND`.

Key Parameters

- `SCR_ENABLE`: Enables the scripting engine.
- `SCR_HEAP_SIZE`: Important for complex scripts to avoid OOM errors.

Key Codebase Locations

- **libraries/AP_Scripting/AP_Scripting.cpp**: Implementation of the "Done" signal and binding.
- **ArduCopter/mode_auto.cpp:2311**: Verification logic.

NAV_ATTITUDE_TIME (ID 42703)

Summary

The `NAV_ATTITUDE_TIME` command instructs the vehicle to maintain a specific attitude (Roll, Pitch, and Yaw) and a constant climb/descent rate for a set duration. This is an advanced "Open Loop" navigation command used for airframe testing, specialized physics research, or high-speed dashes where GPS-based position hold is not required.

Status

Supported (ArduCopter)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Time (Param 1):** Duration in seconds.
- **Roll (Param 2):** Degrees.
- **Pitch (Param 3):** Degrees.
- **Yaw (Param 4):** Degrees.
- **Climb Rate (x):** Meters per second.
- **Packing:** Stored in the `nav_attitude_time` content struct.

Execution (Engineer's View)

Direct Attitude Control

The command bypasses the standard "Waypoint-to-Waypoint" navigation and interacts directly with the **Attitude Controller** (mode_auto.cpp:739).

1. **Targeting:** The autopilot feeds the requested Roll, Pitch, and Yaw angles to the `AC_AttitudeControl` library.
2. **Vertical Control:** The vertical position controller maintains the climb rate specified in the `x` field (Param 5).
3. **Horizontal Behavior:** The drone **drifts with the wind**. Because no Lat/Lon target is provided, the drone behaves as if it were in a manual mode (like AltHold or Stabilize) but with a robotic pilot holding the sticks at fixed angles.
4. **Completion:** The mission advances once the system time exceeds `Param 1` seconds from the start of the command.

Data Fields (MAVLink)

- `param1` **(Time):** s.
- `param2` **(Roll):** deg.
- `param3` **(Pitch):** deg.
- `param4` **(Yaw):** deg.
- `x` **(Climb Rate):** m/s.

Theory: Flight Dynamics Research

`NAV_ATTITUDE_TIME` is the primary tool for **Empirical Aerodynamic Modeling**.

- **The Problem:** GPS navigation masks the "True" physics of the drone by constantly correcting for error.
- **The Solution:** By flying at a fixed pitch and roll (e.g., 5 degrees pitch forward) for 10 seconds, engineers can measure the resulting airspeed to calculate the airframe's **Drag Coefficient** (C_d) and **Thrust Curve**.
- **Safety:** It is critical to ensure enough "Clear Air" is available, as the vehicle will move in the direction of the lean without regard for geofences or waypoints during the execution.

Practical Use Cases

1. **High-Speed Dash:**
    - *Scenario:* A drone needs to cross a field as fast as possible.
    - *Action:* `NAV_ATTITUDE_TIME (Pitch: -45, Time: 5s)`. The drone tilts aggressively and accelerates until the timer pops, ignoring GPS braking logic.
2. **Structural Vibration Testing:**
    - *Scenario:* Identifying resonant frequencies at specific bank angles.
    - *Action:* A script cycles through various `NAV_ATTITUDE_TIME` commands while recording IMU data.

Key Parameters

- `ANGLE_MAX`: Still limits the absolute maximum tilt to prevent tip-over.
- `ATC_INPUT_TC`: Determines how quickly the drone snaps to the requested attitude.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:739**: Command intake.
- **ArduCopter/mode_auto.cpp:2327**: Timer verification.

CONDITION-COMMANDS

CONDITION_DELAY (ID 112)

Summary

The `CONDITION_DELAY` command pauses the mission state machine for a specific number of seconds. Unlike a `NAV_DELAY` or a loiter with time, which are navigation-level commands, `CONDITION_DELAY` is a flow-control item that prevents the *next* mission item from starting until the timer expires.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Packet Param 1 (Time):** Stored in the internal `condition_value` field (seconds).
- **Mechanism:** When the mission reaches this item, it records the current system time in `condition_start`.

Execution (Engineer's View)

State Machine Logic

ArduPilot's mission engine treats "Condition" commands as gates.

1. **Verification:** Every loop, the `verify_wait_delay()` function is called (mode_auto.cpp).
2. **Comparison:** The function checks if the elapsed time since the command started exceeds the target delay.

$$\text{current\_time} - \text{condition\_start} > \text{condition\_value} \cdot 1000$$

3. **Completion:** Once the condition is met, the gate opens, and the mission state machine increments to the next command ID.

Vehicle Behavior during Delay

It is critical to understand that `CONDITION_DELAY` **does not stop the vehicle's movement** if it was already performing a navigation task that supports concurrent execution (though typically conditions are placed *between* nav points).

- **Copter:** If placed after a `NAV_WAYPOINT`, the drone will hover at that waypoint while the delay runs.
- **Plane:** The aircraft will continue its previous navigation state (e.g., loitering) while the condition timer counts down.

Data Fields (MAVLink)

- `param1` **(Time):** Delay time in seconds.
- `param2` to `param7`: Unused.

Theory: Synchronous vs. Asynchronous Delays

In computer science, a delay can be blocking or non-blocking.

- **MAVLink HUD Context:** `CONDITION_DELAY` is a non-blocking delay for the flight controller (it still stabilizes the airframe), but a blocking delay for the **Mission Script**.
- **Race Conditions:** If a GCS sends a "Set Current Waypoint" command while a `CONDITION_DELAY` is active, the timer is typically reset or abandoned as the mission engine jumps to the new index.

Practical Use Cases

1. **Camera Boot-up:**
    - *Scenario:* A high-end mapping camera takes 10 seconds to initialize after being powered on via a relay.
    - *Action:* `DO_SET_RELAY` → `CONDITION_DELAY (10s)` → `NAV_WAYPOINT`.
2. **Safety Buffer:**
    - *Scenario:* Ensuring a drone has come to a complete, stable hover before dropping a payload.
    - *Action:* `NAV_WAYPOINT` → `CONDITION_DELAY (3s)` → `DO_GRIPPER`.

Key Parameters

- `MIS_OPTIONS`: Can affect how the mission engine handles pauses and restarts.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:2192**: `verify_wait_delay` implementation.
- **ArduPlane/commands_logic.cpp**: Plane condition handling.

CONDITION_DISTANCE (ID 114)

Summary

The `CONDITION_DISTANCE` command creates a proximity-based gate in the mission. It prevents the next mission item from starting until the vehicle is within a specific distance (meters) of the *next* navigation waypoint.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Packet Param 1 (Distance):** Stored in the internal `condition_value` field (meters).

Execution (Engineer's View)

Verification Logic

The mission engine checks the proximity in every loop cycle using `verify_within_distance()` (mode_auto.cpp).

1. **Distance Calculation:** The flight controller calculates the 2D (horizontal) distance between the current GPS coordinate and the coordinate of the `NAV_WAYPOINT` immediately following the condition.
2. **Comparison:**

$$\text{dist\_to\_wp} < \text{condition\_value}$$

3. **Completion:** As soon as the vehicle crosses the "distance threshold," the condition returns true, and the next command (which is usually a "Do" command like `DO_DIGICAM_CONTROL`) is executed.

Data Fields (MAVLink)

- `param1` **(Distance):** Distance in meters.
- `param2` to `param7`: Unused.

Theory: The "Inner Circle"

`CONDITION_DISTANCE` allows for spatial triggering that is more precise than simple waypoint arrival.

- **Waypoint Acceptance:** Standard waypoints use a radius (e.g., `WP_RADIUS`) to determine when to turn.
- **Condition Precision:** By using `CONDITION_DISTANCE`, you can trigger an action (like a laser fire or camera snap) at a much tighter distance (e.g., 1 meter) than the navigation waypoint's acceptance radius (e.g., 5 meters).

Practical Use Cases

1. **Precise Photo Capture:**
   - *Scenario:* A surveyor needs a photo taken exactly as the drone passes over a marker.
   - *Action:* `NAV_WAYPOINT (Marker)` → `CONDITION_DISTANCE (0.5m)` → `DO_DIGICAM_CONTROL`.
2. **Obstacle Proximity Actions:**
   - *Scenario:* A drone needs to turn on high-intensity landing lights as it approaches a narrow dock.
   - *Action:* `NAV_WAYPOINT (Dock)` → `CONDITION_DISTANCE (10m)` → `[DO_SET_SERVO](/mission-planning/do-commands.html#DO_SET_SERVO) (Lights ON)`.

Key Parameters

- `WPNAV_RADIUS`: (Copter) The horizontal radius used for waypoint completion; often used in conjunction with `CONDITION_DISTANCE`.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:2201**: `verify_within_distance` implementation.
- **ArduPlane/commands_logic.cpp**: Plane condition handler.

`CONDITION_YAW` (ID 115)

Summary

The `CONDITION_YAW` command forces the vehicle to rotate to a specific heading. This command is a "blocking" mission item; the mission will not advance to the next item until the aircraft has successfully achieved the target yaw angle within a small tolerance.

Status

Supported (ArduCopter and QuadPlane Only)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Angle (Param 1):** Target angle in degrees.
- **Rate (Param 2):** Rotation speed in deg/s.
- **Direction (Param 3):** -1 for Counter-Clockwise, 1 for Clockwise.
- **Relative (Param 4):** 0 for Absolute (North-up), 1 for Relative (Offset from current heading).

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_yaw` (mode_auto.cpp).

1. **Request:** The command is passed to the `auto_yaw` controller, which sets a `FIXED` yaw target.

2. **Tracking:** The autopilot calculates the shortest path (or the path requested by the direction parameter) to the new heading.

3. **Completion:** The verification function `ModeAuto::verify_yaw()` checks if the current heading matches the target:

$$|\text{Yaw}_{current} - \text{Yaw}_{target}| < 2^\circ$$

4. **Priority:** Note that if a subsequent `NAV_WAYPOINT` command has a different yaw requirement (like "Face Next Waypoint"), it will override the `CONDITION_YAW` once the condition is met and the mission advances.

Data Fields (MAVLink)

- `param1` **(Angle):** Target angle in degrees [0-360].
- `param2` **(Rate):** Desired turn rate in deg/s.
- `param3` **(Direction):** -1: CCW, 1: CW.
- `param4` **(Relative):** 0: Absolute, 1: Relative.
- `param5` to `param7`: Unused.

Theory: The Singularity of North

Yaw management in ArduPilot must account for the 360/0 degree wrap-around.

- **Angular Error:** The autopilot uses the `wrap_180()` function to calculate the smallest error.
- **Magnetic vs. True:** Mission angles are typically relative to **Magnetic North** unless the system has been configured with a high-precision GPS (Moving Baseline) or a specific declination offset.

Practical Use Cases

1. **Fixed-Camera Alignment:**
    - *Scenario:* A drone is inspecting a solar panel array. The camera is fixed (no gimbal).
    - *Action:* `CONDITION_YAW` is used to point the entire drone airframe at the panels before the next survey leg.
2. **Antenna Tracking:**
    - *Scenario:* A drone needs to point a high-gain directional antenna back at the GCS for a data burst.
    - *Action:* `CONDITION_YAW` targets the bearing to the Home point.

Key Parameters

- `ATC_SLEW_YAW`: Limits the maximum rate of change for yaw to prevent mechanical stress on the frame.
- `WP_YAW_BEHAVE`: Determines how the drone handles yaw during navigation *between* `CONDITION_YAW` commands.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:1928**: `do_yaw` implementation.
- **ArduCopter/mode_auto.cpp:2211**: `verify_yaw` completion check.

DO_JUMP (ID 177)

Summary

The `DO_JUMP` command provides procedural flow control within a mission. It allows the autopilot to jump back (or forward) to a specific mission item number and repeat that sequence a set number of times. This is the primary mechanism for creating loops (e.g., repeating a survey grid or a loiter-and-check sequence).

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Target (Param 1):** The sequence number of the mission item to jump to.
- **Repeat Count (Param 2):** The number of times to perform the jump.
- **Mechanism:** ArduPilot maintains an internal `jump_tracking` array (AP_Mission.cpp) to persist the state of each jump command across reboots or mode changes.

Execution (Engineer's View)

Logic Flow

When the mission engine encounters a `DO_JUMP`:

1. **Check Counter:** It retrieves the current `num_times_run` for this specific command index from the `jump_tracking` RAM.
2. **Comparison:**
   - If `num_times_run` < `Repeat Count`, it increments the counter and sets the current mission index to the **Target**.
   - If `num_times_run` == `Repeat Count`, it ignores the jump and proceeds to the *next* mission item (breaking the loop).
3. **Unlimited Loops:** If the `Repeat Count` is set to a high value (like 255 or 65535, depending on the GCS), ArduPilot can be configured to loop indefinitely.

Jump Tracking Constraints

- **Max Jumps:** ArduPilot typically supports tracking up to 15-20 individual `DO_JUMP` commands in a single mission (defined by `AP_MISSION_MAX_NUM_DO_JUMP_COMMANDS`).
- **Nested Loops:** While technically possible, nesting jumps can lead to complex state behavior. ArduPilot's tracker is indexed by the *index of the jump command itself*, which prevents collisions between different loops.

Data Fields (MAVLink)

- `param1` **(Item #):** Mission sequence number to jump to.
- `param2` **(Repeat):** Total number of times to perform the jump.
- `param3` to `param7`: Unused.

Theory: Finite State Machines and Halting

A mission is essentially a **Linear Finite State Machine**. `DO_JUMP` introduces **Cycles** into the graph.

- **Determinism:** Because ArduPilot tracks the repeat count in non-volatile-ready RAM, the mission remains deterministic. If the vehicle loses power and reboots, the mission can resume and "remember" how many loops it has already completed.
- **The Halting Problem:** Infinite loops are dangerous in autonomous flight. Always ensure a `DO_JUMP` has a finite repeat count unless the vehicle is in a monitored "holding pattern" state.

Practical Use Cases

1. **Survey Re-runs:**
    - *Scenario:* A drone is scanning for a lost hiker. The search grid needs to be flown 3 times to ensure coverage.
    - *Action:* `WP 1` ... `WP 10` → `DO_JUMP (Target: 1, Repeat: 3)`.
2. **Delayed Entry:**
    - *Scenario:* A plane must loiter at a waypoint until a specific time, but ArduPilot's clock isn't synchronized yet.
    - *Action:* Use a `DO_JUMP` to a loiter point with a small repeat count to "wait" for GCS synchronization.

Key Parameters

- `MIS_RESTART`: Controls whether the jump counters are reset when the mission is restarted.

Key Codebase Locations

- **libraries/AP_Mission/AP_Mission.cpp:2311**: `increment_jump_times_run` implementation.
- **libraries/AP_Mission/AP_Mission.h**: Definition of the jump tracking structure.

DO_CHANGE_SPEED (ID 178)

Summary

The `DO_CHANGE_SPEED` command allows the autopilot to dynamically adjust the vehicle's speed and throttle limits during a mission. This is essential for missions that require high-speed transit between work areas but slow, precise flight during data collection.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command (Guided Mode).

Mission Storage (AP_Mission)

- **Speed Type (Param 1):** 0 = Airspeed, 1 = Ground Speed.
- **Target Speed (Param 2):** Target speed in m/s.
- **Throttle (Param 3):** Desired throttle percentage (0-100).
- **Mechanism:** Stored in the internal `speed` content struct.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command triggers `ModeAuto::do_change_speed` (mode_auto.cpp).

1. **WPNav Update:** The command updates the `AC_WPNav` library's internal speed variables.
2. **Overrides:**
   - **Horizontal:** Sets `set_speed_xy` (cm/s).
   - **Vertical:** If the type is set to climb or descent (ArduPilot extension), it updates `set_speed_up` or `set_speed_down`.
3. **Persistence:** The new speed remains in effect until the end of the mission or until another `DO_CHANGE_SPEED` command is encountered.

ArduPlane Implementation

In Plane, the command updates the navigation controller's target velocity.

1. **Airspeed vs. Groundspeed:**
   - If **Airspeed** is selected: The plane adjusts pitch and throttle to maintain the target airspeed (IAS/EAS).
   - If **Groundspeed** is selected: The plane uses its "Min Groundspeed" logic (`MIN_GNDSPD_CM`) to ensure it maintains progress against headwinds.
2. **Safety Limits:** The autopilot will always clamp the requested speed between `ARSPD_FBW_MIN` and `ARSPD_FBW_MAX`.

Data Fields (MAVLink)

- `param1` **(Type):** 0:Airspeed, 1:Groundspeed, 2:Climb speed, 3:Descent speed.
- `param2` **(Speed):** Target speed in m/s.

- `param3` **(Throttle):** Throttle setpoint (0-100). -1 to ignore.
- `param4` **(Relative):** 0: Absolute speed, 1: Offset from default.

Theory: Groundspeed vs. Airspeed

Understanding the difference is critical for safety:

- **Fixed-Wing:** Airspeed is what keeps you in the air (lift). Changing airspeed affects your stall margin. `DO_CHANGE_SPEED` in a plane is often used to fly slowly for photography or fast for "dash" legs.
- **Multicopter:** Multicopters primarily care about Groundspeed. Airspeed is only relevant in high-wind scenarios where "Lean Angle" limits (e.g., `ANGLE_MAX`) might prevent the drone from achieving the requested groundspeed.

Practical Use Cases

1. **Long Range Transit:**
   - *Scenario:* A delivery drone needs to fly 10km to a destination.
   - *Action:* `TAKEOFF` → `DO_CHANGE_SPEED (25 m/s)` → `WAYPOINT (Destination)`.
2. **Precision Mapping:**
   - *Scenario:* A high-resolution camera requires a slow ground speed to prevent motion blur.
   - *Action:* `DO_CHANGE_SPEED (5 m/s)` → `WAYPOINT (Start of Grid)`.

Key Parameters

- `WPNAV_SPEED`: (Copter) Default mission speed.
- `TRIM_ARSPD_CM`: (Plane) Default cruise airspeed.
- `ARSPD_FBW_MIN`: Absolute minimum airspeed safety limit.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:1936**: Copter implementation.
- **ArduPlane/commands_logic.cpp**: Plane implementation.

DO_SET_HOME (ID 179)

Summary

The `DO_SET_HOME` command redefines the vehicle's "Home" position. The Home position is used as the reference point for RTL (Return to Launch), altitude-above-home calculations, and distance-from-home failsafes.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command (Guided/Command Mode).

Mission Storage (AP_Mission)

- **Use Current (Param 1):**
  - 1: Use the vehicle's current location as home.
  - 0: Use the Lat/Lon/Alt provided in the command's x/y/z fields.
- **Packing:** Stored in the standard location struct.

Execution (Engineer's View)

Logic

The command calls `AP_Vehicle::set_home()` (or vehicle-specific wrappers like `ModeAuto::do_set_home`).

1. **Safety Check:** ArduPilot generally allows setting Home while disarmed. If the vehicle is armed, setting home is typically restricted to the current location to prevent the vehicle from performing an RTL to a dangerously distant or unreachable coordinate.
2. **Coordinate Update:** The EKF (Extended Kalman Filter) uses the new Home as the origin for its local NE (North-East) coordinate system if the "Absolute Altitude" is updated.
3. **GCS Notification:** Upon successful update, the vehicle broadcasts a new `HOME_POSITION` (242) MAVLink message to all connected Ground Control Stations.

Data Fields (MAVLink)

- `param1` **(Flag):** 1: Current Location, 0: Specified Location.
- `x` **(Latitude):** New home latitude.
- `y` **(Longitude):** New home longitude.
- `z` **(Altitude):** New home altitude.

Theory: The EKF Origin vs. Home

It is important to distinguish between the **EKF Origin** and the **Home Position**.

- **EKF Origin:** The absolute GPS coordinate where the Kalman Filter initialized. It never changes during a flight.
- **Home Position:** A user-defined coordinate used for navigation. It can be changed multiple times.

- **Math:** Internal navigation is done in meters relative to the EKF Origin. When you "Set Home," ArduPilot calculates the meter-offset from the Origin and stores that as the new Home reference.

Practical Use Cases

1. **Mobile Landing Platform:**
    - *Scenario:* A drone takes off from a moving boat.
    - *Action:* As the boat moves, the GCS periodically sends `DO_SET_HOME (Current Location: 1)` to ensure the drone's RTL point stays near the boat.
2. **Long Corridor Missions:**
    - *Scenario:* A drone is inspecting 50km of power lines.
    - *Action:* The mission can include `DO_SET_HOME` commands at safe landing clearings along the route, so if a failsafe occurs, the drone flies to the *nearest* safe point rather than all the way back to the start.

Key Parameters

- `FS_GCS_ENABL`: GCS failsafe.
- `RTL_ALT`: Altitude used when returning to the (potentially new) home.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:1954**: `do_set_home` implementation.
- **libraries/AP_Vehicle/AP_Vehicle.cpp**: Core Home management logic.

DO_SET_RELAY (181) / DO_REPEAT_RELAY (ID 182)

Summary

The `DO_SET_RELAY` and `DO_REPEAT_RELAY` commands provide simple digital control over the vehicle's GPIO pins. Relays are typically used to trigger non-MAVLink hardware like power switches, smoke generators, water pumps, or simple camera shutters.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command (Guided Mode).

Mission Storage (AP_Mission)

- **Relay Number (Param 1):** The index of the relay to control (typically 0-5).
- **State / Repeat (Param 2):**
  - **181:** 0: Off, 1: On.
  - **182:** The number of times to toggle the relay.
- **Cycle Time (Param 3):** (182 only) The duration of each on/off cycle in seconds.
- **Packing:** Stored in the `relay` or `repeat_relay` content struct within `AP_Mission`.

Execution (Engineer's View)

Logic Implementation

The commands utilize the `AP_ServoRelayEvents` library (AP_ServoRelayEvents.cpp).

1. **Pin Mapping:** ArduPilot maps "Relay 0" to a physical GPIO pin on the flight controller via the `RELAY_PIN` parameter.
2. **Toggle Logic (Repeat):**
   - The `delay_ms` is calculated as `Param3 * 500` (half of the cycle time).
   - The relay state is flipped every `delay_ms` until the repeat count is exhausted.
3. **Conflict Handling:** If a new relay command is received for a pin currently running a "Repeat" sequence, the existing sequence is immediately cancelled in favor of the new state.

Data Fields (MAVLink)

DO_SET_RELAY (181)

- `param1` **(Index):** Relay instance number.
- `param2` **(State):** 0: OFF, 1: ON.

DO_REPEAT_RELAY (182)

- `param1` **(Index):** Relay instance number.
- `param2` **(Count):** Number of toggle cycles.
- `param3` **(Time):** Cycle time in seconds.

Theory: PWM vs. GPIO

Most flight controller pins are configured for PWM (Pulse Width Modulation) by default to drive servos.

- **Relay Transformation:** When a pin is assigned to a Relay, the autopilot reconfigures the hardware timer/DMA for that pin into **Digital I/O Mode**.
- **Voltage:** The output is typically 3.3V (standard microcontrollers) or 5V (level-shifted boards), which is used to trigger a transistor or a physical mechanical relay module.

Practical Use Cases

1. **Water Sprayer (Agri-Drone):**
    - *Scenario:* A crop-dusting drone needs to spray only while over a specific field.
    - *Action:* `WAYPOINT (Start of Field)` → `DO_SET_RELAY (On)` ... `WAYPOINT (End of Field)` → `DO_SET_RELAY (Off)`.
2. **Emergency Strobe:**
    - *Scenario:* A drone needs to flash its high-intensity lights during an RTL.
    - *Action:* Mission starts with `DO_REPEAT_RELAY (Count: 999, Time: 1s)` to create a persistent blink.

Key Parameters

- `RELAY_PIN`: Defines which physical pin is assigned to which relay instance.
- `RELAY_DEFAULT`: The state (On/Off) the relay enters upon autopilot boot.

Key Codebase Locations

- **libraries/AP_ServoRelayEvents/AP_ServoRelayEvents.cpp**: Main execution handler.
- **libraries/AP_Relay/AP_Relay.cpp**: Low-level GPIO driver.

DO_SET_SERVO (183) / DO_REPEAT_SERVO (ID 184)

Summary

The `DO_SET_SERVO` and `DO_REPEAT_SERVO` commands allow the mission script to directly override the PWM (Pulse Width Modulation) output of specific flight controller pins. This is used to control auxiliary hardware like mechanical grippers, gimbal pitches (non-stabilized), or robotic arms.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Channel (Param 1):** The Servo/Output number to control (typically 1-16).
- **PWM Value / Repeat (Param 2):**
  - **183:** The target PWM value (typically 1000 to 2000 microseconds).
  - **184:** The target PWM value for the "pulsed" state.
- **Repeat Count (Param 3):** (184 only) The number of times to pulse.
- **Cycle Time (Param 4):** (184 only) The duration of each pulse cycle in seconds.
- **Packing:** Stored in the `servo` or `repeat_servo` content struct.

Execution (Engineer's View)

Handling Logic

Execution is managed by `AP_ServoRelayEvents::do_set_servo` (AP_ServoRelayEvents.cpp).

1. **Safety Check:** ArduPilot checks the "Servo Function" (`SERVOX_FUNCTION`) of the requested pin.
   - If the pin is assigned to a critical flight function (e.g., `Motor 1` or `Aileron`), the mission command is **rejected** and an info message is sent to the GCS: "ServoRelayEvent: Channel \%d is already in use".
   - The pin must be set to `0` (Disabled), `1` (RCPassThru), or another non-critical auxiliary function.
2. **Output:** The autopilot writes the target PWM value directly to the hardware timer registry for that pin.
3. **Repeat Pattern (184):** The autopilot toggles between the target **PWM Value** and the pin's **Trim Value** at the requested frequency.

Data Fields (MAVLink)

DO_SET_SERVO (183)

- `param1` **(Index):** Output channel number.
- `param2` **(PWM):** PWM value [1000-2000].

DO_REPEAT_SERVO (184)

- `param1` **(Index):** Output channel number.

- `param2` **(PWM):** PWM value for the pulse.
- `param3` **(Count):** Number of cycles.
- `param4` **(Time):** Cycle time (seconds).

Theory: The Duty Cycle

Servos are controlled by the width of a pulse sent at a specific frequency (typically 50Hz).

- **Pulse Width (au):** 1000aus is typically "Full Left/Closed," 1500aus is "Center," and 2000aus is "Full Right/Open."
- **Timing:** ArduPilot's main loop handles the logic, but the actual high-precision pulse generation is offloaded to the SoC's hardware timers (DMA/PWM) to ensure jitter-free control.

Practical Use Cases

1. **Cargo Hook:**
   - *Scenario:* A drone needs to drop a package at a specific waypoint.
   - *Action:* `DO_SET_SERVO (Channel: 9, PWM: 2000)` triggers the hook to open.
2. **Bait Dropper (Fishing Drone):**
   - *Scenario:* Releasing a fishing line into the surf.
   - *Action:* `DO_REPEAT_SERVO (Channel: 10, PWM: 2000, Count: 2, Time: 1s)` ensures the release mechanism triggers twice to prevent a snag.

Key Parameters

- `SERVOX_FUNCTION`: Must be configured as `Disabled` or `Scripting` for mission commands to work on that channel.
- `SERVOX_MIN/MAX`: Bounds for the PWM output.

Key Codebase Locations

- **libraries/AP_ServoRelayEvents/AP_ServoRelayEvents.cpp:30**: Command verification and safety logic.
- **libraries/SRV_Channel/SRV_Channels.cpp**: Low-level servo output manager.

DO_RETURN_PATH_START (ID 188)

Summary

The `DO_RETURN_PATH_START` command defines the beginning of a safe return segment in a mission. This is an advanced "failsafe-aware" mission marker that allows a vehicle to rejoin a pre-defined safe corridor if an RTL is triggered, rather than flying a direct (and potentially dangerous) line back to Home.

Status

Supported (ArduPlane / All Vehicles using Mission re-entry)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Segment Boundary:** This command marks the start of the safe segment. The segment ends at the next encountered `DO_LAND_START` or mission end.
- **Coordinates:** Optional. If provided, they define the geometric start of the segment.

Execution (Engineer's View)

Corridor Logic

When a vehicle using missions for return-to-launch triggers a failsafe:

1. **Search:** The autopilot calls `get_return_path_start()` (AP_Mission.cpp).
2. **Closest re-entry:** It identifies the segment between `DO_RETURN_PATH_START` and `DO_LAND_START`.
3. **Orthogonal Projection:** The autopilot calculates the closest point on that line segment to the vehicle's current position.
4. **Action:** The vehicle flies to that "Re-entry Point" and then follows the mission forward towards the landing sequence.

Data Fields (MAVLink)

- `param1` to `param4`: Unused.
- `x` **(Latitude):** Optional coordinate.
- `y` **(Longitude):** Optional coordinate.
- `z` **(Altitude):** Optional coordinate.

Theory: Corridor Missions

Standard RTL logic is "Point A to Home." **Corridor Logic** is "Point A to the nearest safe pipe, then through the pipe to Home."

- **Geofence Integration:** If a mission is flown through a narrow canyon or a corridor bounded by Inclusion Fences, a direct RTL would immediately trigger a fence breach.
- **Geometric Join:** ArduPilot mathematically treats the mission between these markers as a vector. The vehicle joins the vector tangentially to ensure it never exits the "safe zone" during the recovery.

Practical Use Cases

1. **Canyon Mapping:**
    - *Scenario:* A drone is mapping a deep canyon. It is flying at 50m, but the canyon walls rise to 200m on either side.
    - *Action:* Place `DO_RETURN_PATH_START` at the canyon entrance. If the drone loses link deep in the canyon, it will not attempt to climb and fly over the walls; it will fly back along the canyon floor until it exits the marked segment.
2. **Urban BVLOS:**
    - *Scenario:* Flying along a pre-approved corridor between skyscrapers.
    - *Action:* The markers define the safe air-road. Recovery always follows the approved path.

Key Parameters

- `MIS_OPTIONS`: Must be configured to allow Mission-based RTL.

Key Codebase Locations

- **libraries/AP_Mission/AP_Mission.cpp:2467**: `get_return_path_start` logic.

DO_LAND_START (ID 189)

Summary

The `DO_LAND_START` command acts as a marker or "entry point" for the landing sequence in a mission. It is not an active flight command itself, but rather a metadata tag that tells the autopilot where the landing-specific mission items begin.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command (to trigger an immediate mission landing).

Mission Storage (AP_Mission)

- **Latitude/Longitude/Altitude:** Optional. If provided, they help the autopilot find the *closest* landing sequence if multiple exist in a single mission.
- **Packing:** Stored as a standard location-based mission item.

Execution (Engineer's View)

Autopilot Logic

ArduPilot uses `DO_LAND_START` in several critical scenarios:

1. **Mission Landing Trigger:** When the vehicle enters RTL or a Battery Failsafe, and the mission contains a `DO_LAND_START` marker, the autopilot will skip all intermediate waypoints and jump directly to the item *following* the marker to begin a structured approach and landing.
2. **Closest Sequence Discovery:** If a mission contains multiple landing sequences (e.g., for different wind directions), ArduPilot's `get_landing_sequence_start()` function (AP_Mission.cpp) will calculate which `DO_LAND_START` is geographically closest to the vehicle's current position and use that sequence.
3. **Arming Check:** ArduPilot can be configured to require a `DO_LAND_START` in the mission as a pre-arm safety check for certain high-value autonomous operations.

Data Fields (MAVLink)

- `param1` to `param4`: Unused.
- `x` **(Latitude):** Optional coordinate for proximity matching.
- `y` **(Longitude):** Optional coordinate for proximity matching.
- `z` **(Altitude):** Optional coordinate for proximity matching.

Theory: The Non-Destructive Jump

Standard missions are sequential. Failsafes usually involve returning home (`RTL`). `DO_LAND_START` enables a **Mission-Aware Failsafe**.

- **The Problem:** RTL is often a direct line that might cross restricted airspace or terrain.

- **The Solution:** By marking a landing sequence, the pilot defines a known-safe approach path (e.g., spiral down → align with runway → land). `DO_LAND_START` ensures the autopilot uses this "known-good" procedure instead of a "naive" RTL.

Practical Use Cases

1. **Runway Alignment (Plane):**
   - *Scenario:* A plane needs to land on a narrow runway.
   - *Action:* `DO_LAND_START` → `WAYPOINT (Approach)` → `WAYPOINT (Final)` → `NAV_LAND`. If a failsafe occurs, the plane jumps to "Approach" instead of flying directly to the runway.
2. **Emergency Rally Landing (Copter):**
   - *Scenario:* Mapping a large forest.
   - *Action:* Include multiple `DO_LAND_START` markers at various clearings. The drone will choose the closest one during a low-battery event.

Key Parameters

- `MIS_OPTIONS`: Controls how the mission engine handles the landing sequence.

Key Codebase Locations

- **libraries/AP_Mission/AP_Mission.cpp:2395**: `get_landing_sequence_start` implementation.
- **ArduPlane/commands_logic.cpp**: Plane-specific landing sequence handling.

DO_GO_AROUND (ID 191)

Summary

The `DO_GO_AROUND` command is an safety override used during an autonomous landing. It instructs the vehicle to immediately abort the landing sequence, climb to a safe altitude, and typically enter a loiter or wait for further instructions.

Status

Supported (ArduPlane and QuadPlane Only)

Directionality

- **RX (Receive):** The vehicle receives this command as a direct override from the GCS or as an automatic response to a landing sensor failure.

Mission Storage (AP_Mission)

- **Packet Param 1 (Altitude):** Altitude to climb to (meters).
- **Mechanism:** Stored as an immediate execution marker in the landing state machine.

Execution (Engineer's View)

ArduPlane Implementation

When a Go-Around is triggered:

1. **State Reset:** The landing state machine (Flare/Touchdown detection) is immediately terminated.
2. **Climb-out:** The plane applies full throttle and targets the "Abort Altitude" specified in the command (or the last takeoff altitude).
3. **Navigation:** The plane typically returns to the `DO_LAND_START` waypoint or the waypoint immediately preceding the landing sequence to re-attempt the approach.

QuadPlane Implementation

For hybrid VTOLs, the Go-Around behavior depends on the current stage:

- **Fixed-Wing Approach:** Performs a standard fixed-wing abort climb.
- **VTOL Descent:** The vehicle transitions to a multicopter climb, rising vertically to a safe height before either loitering or transitioning back to fixed-wing flight.

Data Fields (MAVLink)

- `param1` **(Altitude):** Target altitude for the abort climb.
- `param2` to `param7`: Unused.

Theory: Energy vs. Safety

A Go-Around is a trade-off between **Impact Risk** and **Stall Risk**.

- **The Hazard:** During landing, a plane is at its lowest energy state (low speed, low altitude).
- **The Reaction:** ArduPilot prioritized **Airspeed first**. It will prioritize gaining speed over gaining altitude to ensure the aircraft remains controllable during the high-stress transition away from the

ground.

Practical Use Cases

1. **Runway Incursion:**
    - *Scenario:* A vehicle or person enters the runway while a drone is on final approach.
    - *Action:* The GCS operator clicks "Abort Landing," sending a `MAV_CMD_DO_GO_AROUND`. The drone climbs away and circles until the runway is clear.
2. **Lidar/Rangefinder Glitch:**
    - *Scenario:* The landing Lidar fails 5 meters above the ground.
    - *Action:* ArduPilot's internal safety logic triggers an automatic Go-Around to prevent a blind landing.

Key Parameters

- `TECS_LAND_SINK`: Controls the sink rate that might trigger an automatic abort if exceeded.
- `LAND_ABORT_THR`: (Plane) Throttle level used during the abort climb.

Key Codebase Locations

- **ArduPlane/commands_logic.cpp**: Landing logic handler.
- **libraries/AP_Landing/AP_Landing.cpp**: Shared landing abort state machine.

DO_PAUSE_CONTINUE (ID 193)

Summary

The DO_PAUSE_CONTINUE command provides a mechanism to suspend or resume the mission's horizontal movement without changing the flight mode. It is primarily used to "Pause" the vehicle at its current location during a mission.

Status

Supported (ArduCopter and Rover)

Directionality

- **RX (Receive):** The vehicle receives this command as a direct override from the GCS.

Mission Storage (AP_Mission)

- **Packing:** This command is typically sent as an immediate command (COMMAND_LONG), not stored in a mission. If encountered in a mission, ArduPilot treats it as a state change.

Execution (Engineer's View)

ArduCopter Implementation

In Copter, the command calls `ModeAuto::pause()` or `ModeAuto::resume()` (mode_auto.cpp).

1. **WPNav Interaction:** The autopilot tells the `AC_WPNav` library to freeze the current trajectory.
2. **Position Hold:** The drone enters a "GPS Position Hold" at its current coordinates.
3. **Vertical Behavior:** The drone maintains its current mission altitude.
4. **Resumption:** When `Continue (1)` is sent, the `WPNav` resumes following the path from where it was paused.

Data Fields (MAVLink)

- `param1` **(State):** 0: Pause, 1: Continue.
- `param2` to `param7`: Unused.

Theory: Suspending the Vector

Most flight modes are "State-based." `AUTO` mode is "Vector-based."

- **The Pause:** `DO_PAUSE_CONTINUE` zero-scales the mission's velocity vector while keeping the PID loops active.
- **Benefit:** This is safer than switching to `LOITER` and back to `AUTO`, as the mission state machine (sequence number, jump counters, timers) remains exactly where it was.

Practical Use Cases

1. **Bird Incursion:**
   - *Scenario:* A pilot sees a hawk approaching the drone during a mapping grid.
   - *Action:* The pilot clicks "Pause." The drone stops moving immediately. Once the bird leaves, the pilot clicks "Continue."

2. **Visual Inspection:**
    - *Scenario:* During an autonomous tower inspection, the inspector needs more time to look at a specific bracket.
    - *Action:* Send `Pause`, inspect, then `Continue`.

Key Parameters

- `WPNAV_ACCEL`: Determines how quickly the drone stops when paused.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:2339**: Implementation of pause/resume methods.

DO_SET_REVERSE (ID 194)

Summary

The `DO_SET_REVERSE` command tells the autopilot to change its primary direction of travel. This is used by Rovers and Boats to backup, and by Planes equipped with reversible ESCs to perform steep descents or reverse-thrust landings.

Status

Supported (ArduPlane, Rover, and Boat)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Reverse (Param 1):**
  - 0: Forward.
  - 1: Reverse.
- **Packing:** Stored in the internal `p1` field.

Execution (Engineer's View)

ArduPlane Implementation

In Plane, the command enables the use of **Negative Thrust** (reverse_thrust.cpp).

1. **ESC Configuration:** Requires a bi-directional ESC (DShot or specialized PWM).
2. **Navigation:** The autopilot maintains its orientation but reverses the throttle output.
3. **Use Case:** Typically used for "Beta Range" aerodynamic braking during a steep approach to clear obstacles on short runways.

Rover/Sub/Boat Implementation

In Rover, this command is used to navigate "Tail First."

1. **Ackermann Logic:** For cars, the steering logic is inverted when in reverse.
2. **Pivot Turns:** Skid-steer rovers use this to back out of dead ends.

Data Fields (MAVLink)

- `param1` **(State):** 1: Reverse, 0: Forward.
- `param2` to `param7`: Unused.

Theory: Thrust Vectoring and Polarity

In standard propulsion, thrust is a scalar value $T \in [0, 1]$. `DO_SET_REVERSE` redefines the propulsion model as a signed value $T \in [-1, 1]$.

- **The Transition:** Transitioning from forward to reverse requires the motor to come to a complete stop (to prevent back-EMF spikes) before spinning in the opposite direction. ArduPilot's motor library manages this "Zero-Crossing" safely.

Practical Use Cases

1. **Boat Docking:**
   - *Scenario:* An autonomous boat needs to back into a slip.
   - *Action:* `WAYPOINT (Front of Slip)` → `DO_SET_REVERSE (1)` → `WAYPOINT (Back of Slip)`.
2. **STOL (Short Takeoff and Landing):**
   - *Scenario:* A plane landing on a very short mountain strip.
   - *Action:* After `NAV_LAND` flare, the mission triggers `DO_SET_REVERSE (1)` to use the prop as an airbrake.

Key Parameters

- `USE_REV_THRUST`: (Plane) Global enable for reverse thrust logic.
- `SERVOX_REVERSED`: (Rover) Hardware-level motor direction.

Key Codebase Locations

- **ArduPlane/reverse_thrust.cpp**: Plane-specific reverse logic.
- **libraries/AP_Motors/AP_Motors_Class.cpp**: Low-level bidirectional motor control.

DO_SET_ROI (201) / DO_SET_ROI_LOCATION (195) / DO_SET_ROI_NONE (ID 197)

Summary

The "Region of Interest" (ROI) commands instruction the vehicle to point its camera (and potentially its entire airframe) at a specific coordinate or object. This is a critical command for surveillance, cinematography, and inspection missions where the sensor's target is independent of the vehicle's flight path.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command (Guided Mode).

Mission Storage (AP_Mission)

- **Mode (Param 1):**
  - 0: No ROI (Cancels current ROI).
  - 1: ROI at next waypoint.
  - 2: ROI at specific mission item.
  - 3: ROI at fixed Lat/Lon location.
- **Location (x, y, z):** The GPS coordinates of the target (used for 195 and Mode 3).
- **Mechanism:** Stored as a standard location item or a mode-switch item.

Execution (Engineer's View)

Gimbal and Airframe Integration

ArduPilot handles ROI via the `AP_Mount` and `AutoYaw` libraries (mode_auto.cpp).

1. **3D Geometry:** The autopilot calculates the vector between the vehicle's current 3D position (GPS/INS) and the target 3D coordinate.
2. **Point of Aim:**
   - **Gimbal:** If a stabilized gimbal is present, ArduPilot sends Pitch and Yaw angles to the mount controller.
   - **Airframe:** If no gimbal is present, or if the gimbal lacks a pan (yaw) axis, the multicopter will rotate its entire body (Yaw) to face the target while keeping the camera centered.
3. **Persistence:** An ROI command is "sticky." Once set, the vehicle will continue to track that coordinate even as it flies through subsequent waypoints, until a `DO_SET_ROI_NONE` command is encountered or a new ROI is defined.

Data Fields (MAVLink)

- `param1` **(Mode):** Selection between location, waypoint, or cancel.
- `param2` **(Item #):** Mission index if mode=2.
- `x` **(Latitude):** Target latitude.
- `y` **(Longitude):** Target longitude.
- `z` **(Altitude):** Target altitude.

Theory: Trigonometry of Tracking

To point the camera, the autopilot solves for the Azimuth (ψ) and Elevation ($theta$) angles relative to North:

$$\psi = \text{atan2}(\Delta\text{East}, \Delta\text{North})$$

$$\theta = \text{atan2}(\Delta\text{Alt}, \sqrt{\Delta\text{North}^2 + \Delta\text{East}^2})$$

The EKF solution provides high-frequency updates to these angles to ensure the camera remains locked even during aggressive vehicle maneuvering.

Practical Use Cases

1. **POI (Point of Interest) Orbit:**
   - *Scenario:* Filming a lighthouse from a circling drone.
   - *Action:* `DO_SET_ROI (Lighthouse)` → `NAV_LOITER_TURNS`. The drone circles while the gimbal automatically tilts and pans to keep the lighthouse in the center of the frame.
2. **Static Security Watch:**
   - *Scenario:* A plane is patrolling a perimeter but needs to keep its camera locked on a specific gate.
   - *Action:* `DO_SET_ROI (Gate)` followed by a patrol mission.

Key Parameters

- `MNT1_TYPE`: Defines the type of gimbal hardware.
- `WP_YAW_BEHAVE`: Must be aware that ROI will override the default "Face Next Waypoint" behavior.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:1975**: `do_roi` implementation.
- **libraries/AP_Mount/AP_Mount.cpp**: Core tracking mathematics.

DO_DIGICAM_CONFIGURE (202) / DO_DIGICAM_CONTROL (ID 203)

Summary

The `DO_DIGICAM_CONFIGURE` and `DO_DIGICAM_CONTROL` commands are the legacy interface for interacting with onboard digital cameras. While modern systems prefer the MAVLink Camera Protocol (v2), these commands remain vital for supporting a vast array of existing hardware, including CHDK-enabled Canon cameras, Sony NEX series via IR/Multi-port, and simple PWM-triggered DSLRs.

Status

Legacy / Supported (All Vehicles with AP_Camera enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

ArduPilot uses a highly efficient packing strategy to store these complex commands:

- **202 (Configure):** Stored in the `digicam_configure` struct. Includes Shutter Speed, Aperture, and ISO.
- **203 (Control):** Stored in the `digicam_control` struct. Includes Zoom absolute, Zoom step, and Focus locking.
- **Packing:** Both utilize the standard 15-byte mission storage limit by bit-packing smaller values where possible.

Execution (Engineer's View)

Triggering and Handshaking

The commands are routed through the `AP_Camera` library (AP_Camera.cpp).

1. **Direct Shutter:** If `Shooting Command (Param 5)` is set to 1, the autopilot triggers the shutter.
2. **Focus Lock:** ArduPilot supports a "Half-press" logic for cameras that require focusing before the final capture.
3. **Engine Cut-off:** A unique feature of `DO_DIGICAM_CONFIGURE` is the **Engine Cut-off Time (Param 7)**. For gas-powered vehicles with extreme vibration, the autopilot can momentarily kill the engine or reduce throttle during the capture to ensure a blur-free image.

Data Fields (MAVLink)

DO_DIGICAM_CONTROL (203)

- `param1` **(Session):** 0:Ignore, 1:Show Lens, 2:Hide Lens.
- `param2` **(Zoom):** Absolute position.
- `param3` **(Step):** Zoom step (offset).
- `param4` **(Focus):** 0:Ignore, 1:Lock, 2:Unlock.
- `param5` **(Shot):** 1:Take picture.

Theory: The Latency of Mechanical Shutters

Unlike digital sensors, mechanical shutters have **Trigger Latency** ($\Delta t_{shutter}$).

- **The Drift:** If a drone is flying at 20 m/s and the shutter has a 200ms lag, the photo will be taken 4 meters past the intended coordinate.
- **ArduPilot Compensation:** High-end configurations use the `CAM_FEEDBACK_PIN` (Hot Shoe). When the shutter actually fires, the camera sends a signal back to the flight controller, which then captures the *exact* GPS/IMU state for that microsecond, ensuring sub-centimeter mapping accuracy.

Practical Use Cases

1. **Vibration-Sensitive Long Exposure:**
    - *Scenario:* A high-altitude plane taking photos in low light.
    - *Action:* `DO_DIGICAM_CONFIGURE (Engine Cut-off: 0.5s)`. The motor stops for half a second, the photo is taken in still air, and the motor restarts automatically.
2. **Canon CHDK Integration:**
    - *Scenario:* Using a cheap Canon Point-and-Shoot for mapping.
    - *Action:* `DO_DIGICAM_CONTROL` sends the PWM pulse required by the CHDK script to trigger the capture.

Key Parameters

- `CAM_TRIGG_TYPE`: Must be set to `Relay` or `PWM`.
- `CAM_DURATION`: The length of the shutter pulse.

Key Codebase Locations

- **libraries/AP_Mission/AP_Mission_Commands.cpp:115**: Command parsing.
- **libraries/AP_Camera/AP_Camera.cpp**: Core execution logic.

DO_MOUNT_CONTROL (ID 205)

Summary

The `DO_MOUNT_CONTROL` command allows the mission script to set specific Pitch, Roll, and Yaw angles for a camera mount or antenna gimbal. Unlike ROI (which targets a coordinate), `MOUNT_CONTROL` targets specific body-relative or Earth-relative angles.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Pitch (Param 1):** Angle in degrees.
- **Roll (Param 2):** Angle in degrees.
- **Yaw (Param 3):** Angle in degrees.
- **Mechanism:** Stored in the `mount_control` content struct.

Execution (Engineer's View)

Controlling the Mount

ArduPilot directs these commands to the `AP_Mount` library (AP_Mount.cpp).

1. **Angle Mapping:** The command defines the target orientation.
2. **Mount Mode:** Encountering this command typically switches the mount to `MAVLINK_TARGETING` mode.
3. **Airframe Yaw Integration:** A critical feature in ArduCopter: If the vehicle has a camera mount that does *not* support a pan (Yaw) axis (e.g., a 2-axis gimbal), ArduPilot will rotate the **entire vehicle airframe** to achieve the requested Yaw angle (mode_auto.cpp).

Data Fields (MAVLink)

- `param1` **(Pitch):** Target pitch angle (deg).
- `param2` **(Roll):** Target roll angle (deg).
- `param3` **(Yaw):** Target yaw angle (deg).
- `param7` **(Mode):** Selection between angle and rate control (if supported).

Theory: Body-Relative vs. Earth-Relative

- **Body-Relative:** The angles are relative to the drone's nose. If the drone turns 90 degrees, the camera turns 90 degrees.
- **Earth-Relative (North-Up):** The camera locks onto a compass heading. If the drone turns, the gimbal compensates to stay pointed North.
- **ArduPilot Default:** `MAV_CMD_DO_MOUNT_CONTROL` is typically interpreted as Earth-Relative for Pitch (tilt) and Yaw (pan), but Body-Relative for Roll.

Practical Use Cases

1. **Vertical Mapping (Nadir):**
   - *Scenario:* A surveyor needs the camera to point straight down for the entire flight.
   - *Action:* `DO_MOUNT_CONTROL (Pitch: -90, Roll: 0, Yaw: 0)`.
2. **Forward Scouting:**
   - *Scenario:* A search and rescue drone needs to look 15 degrees below the horizon and 45 degrees to the right to scan a coastline.
   - *Action:* `DO_MOUNT_CONTROL (Pitch: -15, Yaw: 45)`.

Key Parameters

- `MNT1_TYPE`: Enables the gimbal driver.
- `MNT1_PITCH_MIN/MAX`: Defines the mechanical limits to prevent servo damage.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount.cpp:555**: Command intake.
- **ArduCopter/mode_auto.cpp:1987**: Yaw-override logic for 2-axis gimbals.

DO_SET_CAM_TRIGG_DIST (ID 206)

Summary

The `DO_SET_CAM_TRIGG_DIST` command enables automatic camera triggering based on the 2D distance traveled by the vehicle. This is the primary mechanism for aerial mapping (photogrammetry), ensuring consistent image overlap regardless of groundspeed fluctuations caused by wind.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Distance (Param 1):** The distance in meters between triggers.
- **Shutter (Param 2):** Shutter integration time (typically unused by ArduPilot, which relies on hardware-specific drivers).
- **Trigger Once (Param 3):** If set to 1, the camera triggers immediately upon receiving the command.
- **Mechanism:** Stored in the `cam_trigg_dist` content struct within `AP_Mission`.

Execution (Engineer's View)

Triggering Logic

The command utilizes the `AP_Camera` library (AP_Camera.cpp).

1. **Distance Threshold:** The autopilot records the GPS position of the *last* trigger.
2. **Continuous Check:** Every loop, it calculates the 2D horizontal distance (d) from the last trigger point.
3. **Firing:** When $d \geq \text{TriggDist}$, the autopilot sends a signal to the camera shutter (via Relay, PWM, MAVLink, or DroneCAN) and updates the last-trigger position.
4. **Disabling:** Sending this command with `Param 1 = 0` disables automatic triggering.

Data Fields (MAVLink)

- `param1` **(Distance):** Meters between shots. 0 to disable.
- `param2` **(Shutter):** Integration time (ms).
- `param3` **(Trigger):** 1 to trigger one shot immediately.
- `param4` to `param7`: Unused.

Theory: Photogrammetry Overlap

Mapping accuracy depends on **Frontal Overlap** (O_f). For a camera with sensor height H_s and focal length f, flying at altitude A with a groundspeed V_g:

- **Ground Sample Distance (GSD):** The real-world size of one pixel.
- **Trigger Distance (D):**

$$D = \text{Footprint}_{height} \cdot (1 - O_f)$$

Using `DO_SET_CAM_TRIGG_DIST` ensures that even if the drone slows down when flying into a headwind, the images are still taken at the mathematically correct spatial intervals to maintain the required O_f for 3D reconstruction.

Practical Use Cases

1. **Ortho-Mosaic Mapping:**
   - *Scenario:* Mapping a farm at 80\% overlap.
   - *Action:* Mission starts with `DO_SET_CAM_TRIGG_DIST (20m)`. The drone flies the survey grid, and the camera fires every 20 meters.
2. **Corridor Inspection:**
   - *Scenario:* Inspecting a pipeline.
   - *Action:* `DO_SET_CAM_TRIGG_DIST (50m)` ensures high-resolution coverage without filling the SD card with redundant images.

Key Parameters

- `CAM_TRIGG_TYPE`: Defines if the trigger is via Relay, PWM, or MAVLink.
- `CAM_DURATION`: How long the shutter signal is held high.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:291**: Implementation of distance-based triggering.
- **ArduCopter/mode_auto.cpp**: Integration with the mission engine.

DO_FENCE_ENABLE (ID 207)

Summary

The `DO_FENCE_ENABLE` command allows the mission script to dynamically enable or disable the vehicle's Geofence system. This is particularly useful for missions where a drone must transition from a restricted "test area" into an open flight corridor, or vice-versa.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command from the GCS.

Mission Storage (AP_Mission)

- **Enable (Param 1):**
    - 0: Disable all fences.
    - 1: Enable all fences.
    - 2: Disable "Floor" only (Altitude minimum).
- **Packing:** Stored in the internal `p1` field.

Execution (Engineer's View)

Dynamic Fencing Logic

Execution is handled by the `AC_Fence` library (AC_Fence.cpp).

1. **State Change:** The command updates the internal fence enable bitmask.
2. **Safety Verification:** If enabling the fence, the autopilot immediately checks the current vehicle position against the fence boundaries.
3. **Breech Handling:** If the vehicle is *already* outside the boundary when the fence is enabled via mission command, the autopilot will immediately trigger a **Fence Failsafe** (typically RTL or Land).

Data Fields (MAVLink)

- `param1` **(State):** 0: Disable, 1: Enable, 2: Disable Floor.
- `param2` to `param7`: Unused.

Theory: Adaptive Airspace

Dynamic fencing enables **Adaptive Airspace** management.

- **Corridors:** A mission can be "fenced" into a narrow pipe for a BVLOS transit. Once it reaches a designated high-altitude workspace, the `DO_FENCE_ENABLE (Disable)` command can be used to allow for freer movement during manual inspection tasks.
- **Payload Protection:** A drone carrying hazardous material might enable a very tight fence only during the transport phase, then disable it for recovery once the payload is released.

Practical Use Cases

1. **Transitioning to Open Sea:**
    - *Scenario:* A search-and-rescue drone launches from a crowded beach.
    - *Action:* Mission starts with Geofence Enabled. Once the drone is 500m offshore, `DO_FENCE_ENABLE (Disable)` is triggered to allow it to scan a wide area without nuisance alerts.
2. **Autonomous Testing:**
    - *Scenario:* Testing a new Lua script.
    - *Action:* The script is only allowed to run while the drone is inside a "Safety Box." The mission enables the box before starting the script and disables it upon completion.

Key Parameters

- `FENCE_ENABLE`: Global master switch for fencing.
- `FENCE_ACTION`: Determines what happens when a fence is breached (RTL, Land, etc.).

Key Codebase Locations

- **libraries/AC_Fence/AC_Fence.cpp**: Core fence logic.
- **ArduCopter/mode_auto.cpp:1008**: Mission command intake.

DO_PARACHUTE (ID 208)

Summary

The `MAV_CMD_DO_PARACHUTE` command triggers or manages the vehicle's emergency recovery parachute. In a mission context, this is rarely used for standard recovery (which is usually an RTL) but is vital for "Total Safety" missions or testing environments where a soft impact must be guaranteed at a specific stage.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct "PANIC" command from the GCS.

Mission Storage (AP_Mission)

- **Action (Param 1):**
  - 0: Disable parachute release.
  - 1: Enable parachute (Armed).
  - 2: Release parachute (Fire immediately).
- **Packing:** Stored in the internal `p1` field.

Execution (Engineer's View)

Emergency Logic

The command utilizes the `AP_Parachute` library (AP_Parachute.cpp).

1. **Safety Interlocks:** If `Release (2)` is commanded, the autopilot **instantly** stops all flight motors (ArduCopter/GCS_Mavlink.cpp:978). This prevents the parachute lines from tangling in the props.
2. **Deployment:** A high signal is sent to the assigned parachute relay or servo.
3. **Logging:** A "PARACHUTE" event is recorded in the DataFlash log.

Data Fields (MAVLink)

- `param1` **(Action):** 0: Disable, 1: Enable, 2: Release.
- `param2` to `param7`: Unused.

Theory: The Point of No Return

Parachute deployment is a **Terminating Event**.

- **Motor Inhibition:** Once a parachute is fired, ArduPilot will not allow the motors to restart until the vehicle has been disarmed and rebooted.
- **Altitude Constraint:** Parachutes require a minimum altitude to inflate. ArduPilot uses the `CHUTE_ALT_MIN` parameter to prevent deployment if the vehicle is too low (which could lead to a prop-entanglement before inflation).

Practical Use Cases

1. **Sensitive Equipment Recovery:**
    - *Scenario:* A drone is carrying a $100k prototype sensor.
    - *Action:* At the end of the mission, instead of a standard landing, the mission triggers `DO_PARACHUTE (Release)` over a designated soft-target area to ensure the sensor is never subjected to a landing impact.
2. **Safety Drills:**
    - *Scenario:* Testing a new airframe.
    - *Action:* The GCS "Kill Switch" is mapped to `DO_PARACHUTE (Release)`.

Key Parameters

- `CHUTE_ENABLED`: Global master switch.
- `CHUTE_TYPE`: Defines if the trigger is a Relay or Servo.
- `CHUTE_SERVO_ON`: The PWM value used to fire the chute.

Key Codebase Locations

- **libraries/AP_Parachute/AP_Parachute.cpp**: Core deployment logic.
- **ArduCopter/GCS_Mavlink.cpp:978**: Critical motor-kill logic.

DO_INVERTED_FLIGHT (ID 210)

Summary

The `DO_INVERTED_FLIGHT` command instructions the autopilot to fly the aircraft upside-down. This is primarily used in ArduPlane for aerobatics or specialized flight maneuvers.

Status

Supported (ArduPlane Only)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command from the GCS.

Mission Storage (AP_Mission)

- **Inverted (Param 1):**
    - 0: Normal flight (Upright).
    - 1: Inverted flight (Upside-down).
- **Packing:** Stored in the internal `p1` field.

Execution (Engineer's View)

ArduPlane Implementation

In Plane, the command updates the `auto_state.inverted_flight` flag (commands_logic.cpp).

1. **Attitude Target:** When inverted flight is enabled, the AHRS (Attitude and Heading Reference System) redefines the Roll target as 180 degrees.
2. **Controller Inversion:** The PID controllers for Aileron and Elevator must account for the inverted state.
    - **Pitch:** Pulling "Up" on the stick while inverted will cause the aircraft to move towards the ground.
    - **Self-Leveling:** The autopilot automatically handles the inversion of these vectors to ensure that standard "Fly-By-Wire" (FBW) controls remain intuitive for the pilot (if in a semi-autonomous mode) or consistent for the mission engine.
3. **Oil/Fuel Handling:** This command is only recommended for airframes with inverted-flight capable propulsion systems (e.g., fuel-injected engines with header tanks or electric systems).

Data Fields (MAVLink)

- `param1` **(State):** 0: Normal, 1: Inverted.
- `param2` to `param7`: Unused.

Theory: The Lift Vector Inversion

In normal flight, lift (L) acts upwards to oppose gravity (W).

$$L = W$$

In inverted flight, the aircraft must maintain a negative angle of attack (α) to generate lift "upwards" relative to the Earth, even though the wing is upside down.

$$L_{inverted} = \frac{1}{2}\rho v^2 S C_{L,inverted}$$

ArduPilot's navigation controller (L1) calculates the required bank angle to maintain the ground track while accounting for the reduced lift efficiency of most non-symmetrical airfoils when flying inverted.

Practical Use Cases

1. **Aerobatic Display:**
    - *Scenario:* An autonomous drone show involving complex maneuvers.
    - *Action:* `WAYPOINT (A)` → `DO_INVERTED_FLIGHT (1)` → `WAYPOINT (B)`. The plane rolls 180 degrees and flies the segment between A and B upside-down.
2. **Sensor Calibration:**
    - *Scenario:* Calibrating an IMU or magnetometer by exposing it to symmetrical forces.
    - *Action:* Flying a specific leg upright, then repeating it inverted to cancel out bias.

Key Parameters

- `ROLL_LIMIT_DEG`: Still applies while inverted.

Key Codebase Locations

- **ArduPlane/commands_logic.cpp:134**: Mission command intake.
- **ArduPlane/attitude.cpp**: High-level attitude control logic for inversion.

DO_GRIPPER (ID 211)

Summary

The `DO_GRIPPER` command manages the state of a mechanical cargo gripper (claw, magnetic latch, or drop-hook). This is a dedicated actuator command that is safer and more descriptive than a raw `DO_SET_SERVO` because it integrates with the autopilot's landing and failsafe logic.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Instance (Param 1):** The gripper ID (ArduPilot typically supports 1 primary gripper).
- **Action (Param 2):**
    - 0: Release (Open).
    - 1: Grab (Close).
- **Packing:** Stored in the internal `gripper` content struct.

Execution (Engineer's View)

Gripper Logic

Execution is handled by the `AP_Gripper` library (AP_Gripper.cpp).

1. **Safety Interlocks:** ArduPilot can be configured to prevent a "Release" action if the vehicle is not within a safe altitude range or if a "Touchdown" has not been detected.
2. **Actuator Type:** `AP_Gripper` supports multiple hardware backends:
    - **Servo:** Moves a servo to a specific PWM (Open/Closed).
    - **EPM (Electro-Permanent Magnet):** Sends a pulse to flip the magnetic polarity (no power required to hold).
3. **Completion:** The mission script advances immediately after the command is sent to the actuator; it does not wait for a "sensor confirmed closed" signal unless combined with a `CONDITION_DELAY`.

Data Fields (MAVLink)

- `param1` **(ID):** Gripper instance number.
- `param2` **(Action):** 0: Release, 1: Grab.
- `param3` to `param7`: Unused.

Theory: Energy States of Actuators

- **Standard Servos:** Require continuous power to maintain a "Grab" state against a heavy load. This causes heat and battery drain.
- **EPMs:** Use a high-current pulse to re-align the magnetic domains of an Alnico core. Once "Grabbed," the payload is held by a permanent magnetic field with zero power consumption. This is the preferred gripper type for long-endurance ArduPilot missions.

Practical Use Cases

1. **Automated Package Delivery:**
    - *Scenario:* A drone delivering a box to a backyard.
    - *Action:* `NAV_WAYPOINT` → `NAV_PAYLOAD_PLACE` → `DO_GRIPPER (Release)`.
2. **Sample Collection:**
    - *Scenario:* A rover picking up a rock.
    - *Action:* `DO_GRIPPER (Grab)` once the rover is over the target.

Key Parameters

- `GRIP_ENABLE`: Global switch.
- `GRIP_TYPE`: Selection between Servo and EPM.
- `GRIP_CAN_ON_PWM`: The PWM value used to Close the gripper.

Key Codebase Locations

- **libraries/AP_Gripper/AP_Gripper.cpp**: Core hardware interface.
- **ArduCopter/mode_auto.cpp**: Mission integration.

DO_AUTOTUNE_ENABLE (ID 212)

Summary

The `DO_AUTOTUNE_ENABLE` command triggers ArduPilot's automated PID tuning process during a mission. This allows the vehicle to optimize its control gains in real-time while flying a specific leg of a mission, which is useful for airframes whose dynamics change significantly with different payloads.

Status

Supported (ArduPlane Only)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command from the GCS.

Mission Storage (AP_Mission)

- **Enable (Param 1):**
  - 0: Disable Autotune.
  - 1: Enable Autotune.
- **Packing:** Stored in the internal `p1` field.

Execution (Engineer's View)

Tuning Logic

The command utilizes the `AP_AutoTune` library (GCS_Mavlink.cpp:1207).

1. **Safety Verification:** The aircraft must be in a stable flight state (typically FBWA or Cruise) for Autotune to be effective.
2. **Perturbation:** When enabled, the autopilot injects small, controlled step-inputs into the Roll and Pitch axes.
3. **Analysis:** It monitors the vehicle's response (Rate and Acceleration) to calculate the ideal P, I, and D gains.
4. **Completion:** The mission advances immediately. The Autotune process runs in the background. It is common to follow this command with a long, straight `WAYPOINT` to give the tuner enough time to converge.

Data Fields (MAVLink)

- `param1` **(State):** 0: Disable, 1: Enable.
- `param2` to `param7`: Unused.

Theory: System Identification

AutoTune is a form of **Online System Identification**.

- **Excitation:** The step-inputs "excite" the airframe's natural frequencies.
- **Damping Ratio:** The algorithm looks for the "Overshoot" and "Settling Time" to ensure the resulting PID gains provide a damping ratio (ζ) near 0.707 (Critically Damped).

- **Risk:** If Autotune is enabled on a poorly balanced or structurally weak airframe, the perturbations can trigger oscillations. ArduPilot's tuner includes "divergence protection" to automatically abort if the aircraft's attitude becomes unstable.

Practical Use Cases

1. **Post-Payload Calibration:**
   - *Scenario:* A plane drops a heavy 2kg sensor mid-flight.
   - *Action:* The mission triggers `DO_AUTOTUNE_ENABLE (1)` on the return leg to re-calibrate the PIDs for the now much lighter (and potentially differently balanced) airframe.
2. **Maiden Flight Script:**
   - *Scenario:* Automating the first flight of a new aircraft.
   - *Action:* Mission includes a 5km straight leg with Autotune enabled to ensure a perfect tune before the first landing.

Key Parameters

- `AUTOTUNE_LEVEL`: Defines how "aggressive" the resulting tune should be.
- `AUTOTUNE_AXES`: Selection of which axes to tune (Roll, Pitch, Yaw).

Key Codebase Locations

- **ArduPlane/GCS_Mavlink.cpp:1207**: Autotune trigger logic.
- **libraries/AP_AutoTune/AP_AutoTune.cpp**: The core gain-calculation engine.

DO_SET_RESUME_REPEAT_DIST (ID 215)

Summary

The `DO_SET_RESUME_REPEAT_DIST` command defines a "Rewind Distance" for mission resumption. If a mission is interrupted (e.g., due to a mode change or a battery failsafe), and then resumed, the autopilot will not simply fly to the next waypoint. Instead, it will backtrack along the mission path by the specified distance before continuing forward.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Distance (Param 1):** The backtrack distance in meters.
- **Packing:** Stored in the internal `_repeat_dist` variable in the `AP_Mission` library.

Execution (Engineer's View)

Resumption Logic

ArduPilot handles this command within the `AP_Mission::resume()` function (AP_Mission.cpp).

1. **Interrupt Discovery:** The autopilot tracks the last successful waypoint passed (`LAST_WP_PASSED`).
2. **Rewind Calculation:** If `_repeat_dist > 0`, the autopilot calls `calc_rewind_pos()`.
    - It looks at the segment between the last WP and the current WP.
    - It calculates a 3D coordinate that is `Param1` meters *backwards* along that segment.
3. **Path Re-entry:** The vehicle first flies to this calculated rewind point and then resumes the original mission track.

Data Fields (MAVLink)

- `param1` **(Distance):** Rewind distance in meters.
- `param2` to `param7`: Unused.

Theory: The Overlap Requirement

In mapping and sensor missions, an interrupt usually results in a "data gap."

- **The Hazard:** Most autopilots resume at the *next* waypoint, leaving a section of the mission un-scanned.
- **The Solution:** By setting a repeat distance (e.g., 50 meters), the pilot ensures that the drone "re-scans" the last section of the previous leg, providing a overlap buffer that guarantees data continuity for photogrammetry or LiDAR processing.

Practical Use Cases

1. **Battery Swap Recovery:**

    - *Scenario:* A mapping drone triggers a low-battery RTL in the middle of a 1km leg.
    - *Action:* The mission includes `DO_SET_RESUME_REPEAT_DIST (100m)`. After the pilot swaps the battery and clicks "Resume," the drone flies back to the point 100m *before* it left the track, ensuring the map has no holes.
2. **Loss of Signal (LOS) Buffer:**
    - *Scenario:* A drone is inspecting a long bridge and loses telemetry.
    - *Action:* Backtracking ensures that any missed photos are re-taken upon reconnection.

Key Parameters

- `MIS_RESTART`: Affects whether the mission resets entirely or supports this resume logic.

Key Codebase Locations

- **libraries/AP_Mission/AP_Mission.cpp:158**: `resume` logic and rewind calculation.

DO_SPRAYER (ID 216)

Summary

The `DO_SPRAYER` command provides mission-level control for agricultural liquid sprayers. It is used to enable or disable the pump and manage the flow rate based on the vehicle's ground speed to ensure even chemical application.

Status

Supported (ArduCopter and ArduPlane with AC_Sprayer enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Enable (Param 1):**
  - 0: Disable Sprayer (Pump Off).
  - 1: Enable Sprayer (Pump On).
- **Packing:** Stored in the internal `p1` field.

Execution (Engineer's View)

Sprayer Control Logic

Execution is handled by the `AC_Sprayer` library (AC_Sprayer.cpp).

1. **Velocity Compensation:** One of the most advanced features of `AC_Sprayer` is speed-scaling. The pump's PWM output is modulated based on groundspeed (V_g).

$$\mathrm{PWM}_{out} = \mathrm{BaseFlow} \cdot \left(\frac{V_{ground}}{V_{nominal}} \right)$$

2. **Spin-up/down:** The controller manages the ramp-up time for the pump to prevent motor surges.

3. **Automatic Cutoff:** ArduPilot automatically disables the sprayer if the vehicle comes to a stop or enters a failsafe state, preventing chemical pooling.

Data Fields (MAVLink)

- `param1` **(State):** 0: Disable, 1: Enable.
- `param2` to `param7`: Unused.

Theory: Linear Application Rate

In precision agriculture, the goal is to apply a specific volume of liquid per unit of area (L/Ha).

- **The Challenge:** Drones decelerate at waypoints. If the pump stayed at a constant speed, the area near the waypoint would receive a massive overdose of chemicals.
- **The Solution:** The `AC_Sprayer` library integrates with the navigation controller to vary the pump speed in real-time, maintaining a constant **Linear Application Rate** regardless of flight dynamics.

Practical Use Cases

1. **Crop Mapping & Spraying:**
   - *Scenario:* Spraying a rectangular field.
   - *Action:* `WAYPOINT (A)` → `DO_SPRAYER (1)` → `WAYPOINT (B)`. The drone flies the transect, spraying only between A and B, and adjusting the flow as it accelerates out of A and decelerates into B.
2. **Mosquito Abatement:**
   - *Scenario:* Targeting a specific swampy area.
   - *Action:* Mission uses ROI to circle the swamp while `DO_SPRAYER` manages the payload.

Key Parameters

- `SPRAY_ENABLE`: Global switch.
- `SPRAY_PUMP_RATE`: Nominal pump speed (\%).
- `SPRAY_SPEED_MIN`: Groundspeed below which the pump is shut off.

Key Codebase Locations

- **libraries/AC_Sprayer/AC_Sprayer.cpp**: Flow-scaling and pump control.

DO_SEND_SCRIPT_MESSAGE (ID 217)

Summary

The `DO_SEND_SCRIPT_MESSAGE` command provides a bridge between the static Mission Engine and ArduPilot's dynamic **Lua Scripting** environment. It allows a mission to pass numeric data to a running script, which can then perform complex logic, such as changing flight parameters, interacting with custom hardware, or modifying the mission on-the-fly.

Status

Supported (All Vehicles with AP_Scripting enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload.

Mission Storage (AP_Mission)

- **Target (Param 1):** An ID used by the Lua script to identify the specific message.
- **Values (Param 2-4):** Three numeric values (p1, p2, p3) passed to the script.
- **Packing:** Stored in the `scripting` content struct.

Execution (Engineer's View)

Scripting Bridge

The command utilizes the `AP_Scripting` library (AP_Scripting.cpp).

1. **Event Generation:** When the mission reaches this item, the autopilot generates a `MISSION_ITEM_REACHED` event internally.
2. **Script Polling:** A Lua script running on the flight controller uses the `mission:get_last_script_message()` binding to retrieve the data.
3. **Action:** The script parses the `Param 1` ID and executes the corresponding custom code.
4. **Completion:** The mission advances immediately. The "wait" for the script to finish must be handled manually via a `CONDITION_DELAY` or a script-controlled mission override.

Data Fields (MAVLink)

- `param1` **(ID):** Message identifier.
- `param2` **(p1):** First numeric parameter.
- `param3` **(p2):** Second numeric parameter.
- `param4` **(p3):** Third numeric parameter.
- `param5` to `param7`: Unused.

Theory: Extending the State Machine

Missions are typically limited to the commands defined in the MAVLink spec. `DO_SEND_SCRIPT_MESSAGE` transforms the Mission Engine into a **Programmable Logic Controller (PLC)**.

- **Custom Payloads:** A drone carrying a non-standard sensor (e.g., a spectral scanner) can use this command to tell a Lua script to start a specific calibration routine.

- **External Comms:** A script can listen for this message and then send a custom HTTP/UDP packet to an onboard companion computer or a cloud server.

Practical Use Cases

1. **Dynamic Parameter Tuning:**
    - *Scenario:* A plane needs different PID gains for a low-altitude "nap-of-the-earth" segment.
    - *Action:* `DO_SEND_SCRIPT_MESSAGE (ID: 10, p1: 1)` triggers a Lua script to call `param:set('ATC_RAT_RLL_P', 0.15)`.
2. **Robotic Integration:**
    - *Scenario:* A drone landing on a moving rover needs to "Handshake" with the rover's local network.
    - *Action:* Mission triggers the message, and the script handles the custom socket communication.

Key Parameters

- `SCR_ENABLE`: Enables the Lua scripting engine.
- `SCR_VM_I_COUNT`: Instruction count (performance) for scripts.

Key Codebase Locations

- **libraries/AP_Scripting/AP_Scripting.cpp**: Lua binding implementation.
- **ArduCopter/mode_auto.cpp**: Command-to-script event routing.

DO_AUX_FUNCTION (ID 218)

Summary

The `DO_AUX_FUNCTION` command allows a mission to trigger any of ArduPilot's numerous "Auxiliary Functions." These are the same functions typically assigned to physical switches on a radio transmitter (e.g., "Save Waypoint," "Camera Trigger," "Arm/Disarm"). This command effectively allows the mission to "flick a virtual switch."

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Function (Param 1):** The ID of the auxiliary function (e.g., 7 = Save Waypoint).
- **Switch Position (Param 2):**
    - 0: Low (Off).
    - 1: Middle.
    - 2: High (On).
- **Packing:** Stored in the `auxfunction` content struct.

Execution (Engineer's View)

Universal Switch Logic

Execution is handled by the `RC_Channel` library's auxiliary function dispatcher.

1. **Virtual Input:** ArduPilot treats the mission command as a virtual RC channel.
2. **Mapping:** The `Param 1` value is matched against the list of internal functions (e.g., `AUXSW_MOTOR_INTERLOCK`, `AUXSW_GRIPPER`).
3. **Action:** The autopilot executes the code associated with that switch transition (e.g., if set to `High`, it runs the `on_switch_high()` method for that function).

Data Fields (MAVLink)

- `param1` **(Function):** Auxiliary function ID.
- `param2` **(Position):** 0: Low, 1: Mid, 2: High.
- `param3` to `param7`: Unused.

Theory: Abstracted Logic

`DO_AUX_FUNCTION` represents the ultimate in **Logic Abstraction**.

- **The Power:** ArduPilot has over 100 aux functions. Instead of creating 100 individual mission commands, this one command provides access to all of them.
- **Unified Control:** It ensures that whether an action is triggered by a human on a radio, a GCS button, or an autonomous mission, it always calls the same verified block of C++ code.

Practical Use Cases

1. **Waypoint Saving:**
    - *Scenario:* A pilot is flying a survey and wants the mission to automatically record its own progress.
    - *Action:* `DO_AUX_FUNCTION (Function: 7, Position: 2)`.
2. **Emergency Stop (Motor Interlock):**
    - *Scenario:* A high-risk mission segment (e.g., flying through a tunnel).
    - *Action:* A script can inject `DO_AUX_FUNCTION (Function: 32, Position: 0)` to instantly kill the motors if a failure is detected.

Key Parameters

- `RCx_OPTION`: Used to identify the ID of specific functions.

Key Codebase Locations

- **libraries/RC_Channel/RC_Channel.cpp**: Auxiliary function dispatcher.
- **libraries/AP_Mission/AP_Mission.cpp**: Command-to-RC-logic routing.

DO_GUIDED_LIMITS (ID 222)

Summary

The `DO_GUIDED_LIMITS` command sets safety constraints for Guided Mode. It defines the maximum time and distance an external controller (e.g., an onboard companion computer or an GCS script) is allowed to control the vehicle before the autopilot automatically intervenes and terminates the command.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct setup command.

Mission Storage (AP_Mission)

- **Time Limit (Param 1):** Max time in seconds for external control.
- **Alt Min (Param 2):** Minimum altitude (meters).
- **Alt Max (Param 3):** Maximum altitude (meters).
- **Horiz Max (Param 4):** Maximum horizontal distance from the command start point (meters).
- **Packing:** Stored in the `guided_limits` content struct.

Execution (Engineer's View)

Safety Buffer Logic

ArduPilot handles this in the `Guided` flight mode logic (mode_auto.cpp).

1. **Handover:** When the mission reaches `NAV_GUIDED_ENABLE` (or enters Guided mode via MAVLink), these limits are activated.
2. **Continuous Monitoring:** Every loop, the autopilot checks:
   - **Time:** `millis() - start_time > Param1`.
   - **Altitude:** `current_alt < Param2` OR `current_alt > Param3`.
   - **Radius:** `2D_Distance(start_pos, current_pos) > Param4`.
3. **Failsafe:** If any limit is breached, the autopilot instantly terminates Guided mode and typically returns to the mission or enters a failsafe state (RTL).

Data Fields (MAVLink)

- `param1` **(Time):** Max execution time (s).
- `param2` **(Alt Min):** Min altitude (m).
- `param3` **(Alt Max):** Max altitude (m).
- `param4` **(Radius):** Max horizontal distance (m).

Theory: The Sandbox Principal

`DO_GUIDED_LIMITS` implements a **Sandbox** for external intelligence.

- **The Problem:** Companion computer scripts (running ROS or Python) can crash or "go rogue" due to bugs.

- **The Solution:** The flight controller (which is the ultimate arbiter of safety) creates a "Virtual Box" and a "Watchdog Timer." The companion computer is free to experiment inside the box, but the flight controller will "reel it back in" the moment it tries to exit the safe boundaries.

Practical Use Cases

1. **AI Landing Research:**
   - *Scenario:* Testing a new vision-based landing algorithm on a Raspberry Pi.
   - *Action:* `DO_GUIDED_LIMITS (Time: 30s, Alt Min: 2m)`. If the AI fails to land within 30 seconds or drops below 2 meters unexpectedly, the flight controller takes over.
2. **GCS "Follow-Me":**
   - *Scenario:* A tablet app is commanding the drone to follow a cyclist.
   - *Action:* Use horizontal limits to ensure the drone never wanders too far from the cyclist if the app loses the tracking lock.

Key Parameters

- `GUID_OPTIONS`: Can affect how Guided mode behaves when limits are hit.

Key Codebase Locations

- **ArduCopter/mode_auto.cpp:785**: Mission command intake.
- **ArduCopter/mode_guided.cpp**: Limit enforcement logic.

DO_ENGINE_CONTROL (ID 223)

Summary

The `DO_ENGINE_CONTROL` command provides a mission-level interface for Internal Combustion Engines (ICE). It is used to start or stop the engine, manage "Cold Start" procedures, and control height-based delays for engine engagement. This is critical for high-endurance large-scale drones and hybrid power systems.

Status

Supported (ArduPlane and ArduCopter with ICE enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Start/Stop (Param 1):** 1: Start Engine, 0: Stop Engine.
- **Cold Start (Param 2):** 1: Cold Start (enables extended cranking/glow plug time), 0: Warm Start.
- **Height Delay (Param 3):** Height in meters to wait before attempting a start (unit converted to cm in storage).
- **Options (Param 4):** Bitmask for start behavior (e.g., Allow start while disarmed).
- **Mechanism:** Stored in the internal `do_engine_control` content struct.

Execution (Engineer's View)

ICE State Machine

The command is handled by the `AP_ICEngine` library.

1. **Cranking Logic:** When `Start (1)` is commanded, the autopilot manages the Starter Motor (via a Relay or ESC) and Ignition system.
2. **Telemetry Check:** ArduPilot monitors the engine RPM via an onboard sensor. The "Start" is only considered successful when the RPM exceeds the `ICE_START_RPM`.
3. **Kill Switch:** The `Stop (0)` command instantly terminates the ignition/fuel relay, ensuring a rapid shutdown for safety or post-mission power-down.
4. **VTOL Integration:** In hybrid QuadPlanes, this command can be used to start the generator engine once the aircraft is clear of the ground and transitioning to fixed-wing flight.

Data Fields (MAVLink)

- `param1` **(Start):** 1: Start, 0: Stop.
- `param2` **(Cold):** 1: Cold Start, 0: Warm.
- `param3` **(Height):** Delay start until this height (meters).
- `param4` **(Options):** Bit 0: Allow start while disarmed.
- `param5` to `param7`: Unused.

Theory: Combustion Safety

Starting a combustion engine autonomously is a significant safety risk.

- **Vibration:** Engines produce high-frequency vibration that can saturate the EKF. ArduPilot's engine controller includes logic to check the "Health" of the IMUs during the cranking phase.
- **Height Delays:** The `Height Delay (Param 3)` is a critical safety feature. It prevents a hot, powerful engine from starting on the ground where it could injure people. The engine is only engaged once the vehicle is safely "In the Air."

Practical Use Cases

1. **Hybrid Range Extension:**
    - *Scenario:* A VTOL drone launches using battery power for noise abatement.
    - *Action:* Mission includes `DO_ENGINE_CONTROL (Start, Height Delay: 20m)`. The engine starts only after the drone is high enough to minimize ground noise.
2. **Autonomous Landing Shutdown:**
    - *Scenario:* A gas-powered drone completes its mission.
    - *Action:* `NAV_LAND` → `[CONDITION_DELAY](/mission-planning/condition-commands.html#CONDITION_DELAY) (5s)` → `DO_ENGINE_CONTROL (Stop)`. Ensures the engine is cool and stopped before humans approach the airframe.

Key Parameters

- `ICE_START_RPM`: RPM threshold to consider the engine "Running."
- `ICE_PIN`: Relay pin used for the Starter.
- `ICE_PWM_IGN`: PWM value to enable ignition.

Key Codebase Locations

- **libraries/AP_ICEngine/AP_ICEngine.cpp**: The core ICE driver and state machine.
- **ArduPlane/commands_logic.cpp:183**: Plane mission integration.

DO_GIMBAL_MANAGER_PITCHYAW (ID 1000)

Summary

The `DO_GIMBAL_MANAGER_PITCHYAW` command provides advanced, high-level control for modern MAVLink gimbals. It allows for simultaneous control of both angles and rates, and includes flags for specifying the frame of reference (Body vs. Earth). This is the "Modern" version of the legacy `DO_MOUNT_CONTROL`.

Status

Supported (All Vehicles with Gimbal Manager backends)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Pitch (Param 1):** Angle in degrees.
- **Yaw (Param 2):** Angle in degrees.
- **Pitch Rate (Param 3):** Deg/s.
- **Yaw Rate (Param 4):** Deg/s.
- **Flags (Param 5):** Bitmask for coordinate frames and modes.
- **Gimbal ID (Param 7):** Target instance.
- **Packing:** Stored in the `gimbal_manager_pitchyaw` content struct.

Execution (Engineer's View)

Gimbal Manager Protocol

Unlike simple servo gimbals, modern gimbals (like the DJI H20T or specialized Gremsy units) handle their own stabilization. ArduPilot acts as a "Manager."

1. **Setpoints:** ArduPilot extracts the 5D setpoint (Angle P, Angle Y, Rate P, Rate Y, Frame).
2. **Targeting:** The command is forwarded to the specific Gimbal ID via the MAVLink Gimbal Protocol v2.
3. **Frame Switching:**
   - **Body Frame:** The camera follows the drone's nose.
   - **Earth Frame:** The camera stays locked to a compass heading.
4. **Rate Overlays:** The inclusion of "Rate" parameters allows for smooth "Cinematic Pans" where the gimbal moves at a constant speed to a target angle, rather than jumping instantly.

Data Fields (MAVLink)

- `param1` **(Pitch):** deg.
- `param2` **(Yaw):** deg.
- `param3` **(PRate):** deg/s.
- `param4` **(YRate):** deg/s.
- `param5` **(Flags):** See GIMBAL_MANAGER_FLAGS.

Theory: Rate vs. Position Control

In control systems, **Position Control** (P) is prone to "jerk" ($J = \frac{da}{dt}$).

- **The Improvement:** By specifying a **Rate Limit** (ω), the gimbal manager implements a **Velocity-Limited Position Controller**.
- **Mathematics:** The target angle $\theta(t)$ follows a ramp:

$$\theta(t) = \theta_0 + \omega \cdot t$$

  until $\theta(t) = \theta_{target}$. This results in the smooth, professional camera movements seen in high-end cinema drones.

Practical Use Cases

1. **Survey Pivot:**
    - *Scenario:* A triple-lens camera needs to pan from visual to thermal while maintaining a steady rate to avoid motion blur.
    - *Action:* `DO_GIMBAL_MANAGER_PITCHYAW (Yaw: 90, YawRate: 5)`. The camera slowly turns to the right over 18 seconds.
2. **Tracking a Moving Target:**
    - *Scenario:* The GCS is streaming a target velocity.
    - *Action:* The command uses the Rate parameters to "Lead" the target.

Key Parameters

- `MNT1_TYPE`: Must be set to `MAVLinkV2`.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5519**: Command forwarding logic.
- **libraries/AP_Mount/AP_Mount.cpp**: High-level mount coordination.

DO_WINCH (ID 42600)

Summary

The `DO_WINCH` command provides advanced control for robotic winches slung beneath a vehicle. It supports releasing a specific length of cable, controlling the cable's rate of descent, or "relaxing" the motor. This is the primary command for air-to-ground delivery systems where the drone must remain high while the payload descends.

Status

Supported (ArduCopter)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Action (Param 2):**
  - 0: Relax (Motor powered off).
  - 1: Position (Release length).
  - 2: Rate (Descent speed).
- **Length (Param 3):** Distance in meters (Negative to reel in).
- **Rate (Param 4):** Speed in m/s.
- **Packing:** Stored in the `winch` content struct.

Execution (Engineer's View)

Winch Control State Machine

The command is handled by the `AP_Winch` library (AP_Winch.cpp).

1. **Safety Verification:** ArduPilot checks the `WINCH_TYPE`. It supports PWM winches (servos), DroneCAN winches, and Daedalus-protocol winches.
2. **Position Control:** If `Action: 1` is selected, the autopilot uses a PID loop to move the cable to the absolute length requested. This typically requires an encoder on the winch motor.
3. **Rate Control:** If `Action: 2` is selected, the autopilot maintains a steady cable velocity. This is safer for heavy payloads to prevent "yanking" the drone out of the air if the cable snags.
4. **Completion:** The mission advances immediately. The winch continues to work in the background. It is highly recommended to follow a winch command with a `CONDITION_DELAY` if the next mission leg requires the payload to be fully deployed or retracted.

Data Fields (MAVLink)

- `param1` **(Instance):** Winch number (typically 0).
- `param2` **(Action):** 0:Relax, 1:Length, 2:Rate.
- `param3` **(Length):** m.
- `param4` **(Rate):** m/s.

Theory: Tension and Pendulums

Slung loads introduce **Pendulum Dynamics** into the drone's flight model.

- **Oscillation:** A long cable (L) has a natural frequency $f = \frac{1}{2\pi}\sqrt{\frac{g}{L}}$.
- **Damping:** ArduPilot's position controller is aware of the cable length if `DO_WINCH` is used, allowing the EKF to potentially compensate for the shifting center of gravity.
- **Safety:** The `Relax (0)` action is critical for emergencies. If the payload snags on a tree, relaxing the winch allows the drone to fly away rather than being pulled into the ground.

Practical Use Cases

1. **Cable-Drop Delivery:**
   - *Scenario:* Delivering a parcel to a balcony.
   - *Action:* Drone hovers 10m above the balcony. `DO_WINCH (Action: 1, Length: 10m)` lowers the parcel. `[DO_GRIPPER](/mission-planning/do-commands.html#DO_GRIPPER) (Release)` drops it. `DO_WINCH (Action: 1, Length: -10m)` reels the hook back in.
2. **Water Sampling:**
   - *Scenario:* Lowering a sensor into a lake.
   - *Action:* `DO_WINCH (Action: 2, Rate: 0.5m/s)` ensures a smooth entry for the sensor.

Key Parameters

- `WINCH_TYPE`: Hardware selection.
- `WINCH_MAX_LENGTH`: Safety limit to prevent the motor from unspooling the entire drum.

Key Codebase Locations

- **libraries/AP_Winch/AP_Winch.cpp**: Main driver logic.
- **ArduCopter/mode_auto.cpp:1997**: Mission command intake.

SET_CAMERA_ZOOM (ID 531)

Summary

The `SET_CAMERA_ZOOM` command controls the focal length of an onboard camera during a mission. It allows for both incremental (rate-based) and absolute (percentage-based) zoom adjustments, enabling detailed inspections or wide-area reconnaissance from the same mission script.

Status

Supported (All Vehicles with AP_Camera enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command from the GCS.

Mission Storage (AP_Mission)

- **Zoom Type (Param 1):**
    - 0: Step (Not supported in storage).
    - 1: Continuous (Rate-based).
    - 2: Range (Percentage-based 0-100).
- **Zoom Value (Param 2):** The value corresponding to the requested type.
- **Packing:** Stored in the `set_camera_zoom` content struct.

Execution (Engineer's View)

Camera Driver Interaction

The command is routed through the `AP_Camera` library (AP_Camera.cpp).

1. **Rate-Based Zoom:** If `Continuous (1)` is selected, the autopilot commands the lens motor to move at a specific speed. This is typically used with gimbal-integrated cameras where the operator wants to zoom in "until clear."
2. **Percentage Zoom:** If `Range (2)` is selected, the autopilot maps the 0.0-100.0 input to the camera's internal zoom range ($Z_{min} \to Z_{max}$).
3. **Backend Support:** This command works with MAVLink-enabled cameras (like SToRM32 or Tarot), DroneCAN cameras, and specialized drivers (like the Gremsy or Sony cameras). If the camera backend does not support zoom, the command is ignored.

Data Fields (MAVLink)

- `param1` **(Type):** 0:Step, 1:Continuous, 2:Range.
- `param2` **(Value):** Rate (speed) or Percentage.
- `param3` to `param7`: Unused.

Theory: Magnification vs. Resolution

Zooming in does not increase the sensor's physical resolution; it changes the **Field of View (FOV)**.

- **GSD Impact:** Doubling the zoom ($2\times$) effectively halves the Ground Sample Distance (GSD), assuming the altitude remains constant.

- **Vibration Sensitivity:** As FOV decreases (Zoom increases), the image becomes exponentially more sensitive to high-frequency vibration ($Z_{sens} \propto \frac{1}{\text{FOV}}$). This often requires the gimbal's PID gains to be adjusted dynamically (handled automatically by advanced backends).

Practical Use Cases

1. **In-Mission Target Detail:**
   - *Scenario:* A drone is surveying a fence. It detects a breach at a waypoint.
   - *Action:* `SET_CAMERA_ZOOM (Type: 2, Value: 80\%)`. The drone zooms in to capture high-detail evidence before continuing the survey.
2. **Dynamic Reconnaissance:**
   - *Scenario:* A search-and-rescue plane is looking for a boat.
   - *Action:* The plane orbits at $1\times$ zoom to cover the area. Once a suspect object is found, it uses a script to trigger `SET_CAMERA_ZOOM` to $10\times$ for confirmation.

Key Parameters

- `CAM_TYPE`: Selection of camera hardware.
- `MNT1_TYPE`: Often required as gimbals handle the physical lens motors.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:297**: Mission command intake.
- **libraries/AP_Camera/AP_Camera_MAVLinkCamV2.cpp**: Translation to MAVLink camera protocol.

SET_CAMERA_FOCUS (ID 532)

Summary

The `SET_CAMERA_FOCUS` command manages the focus state of an onboard camera. It supports triggering Auto-Focus (AF) routines or setting manual focus levels, ensuring sharp imagery for automated inspections where the distance to the subject varies.

Status

Supported (All Vehicles with AP_Camera enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command from the GCS.

Mission Storage (AP_Mission)

- **Focus Type (Param 1):**
  - 0: Auto.
  - 1: Continuous (Manual rate).
  - 2: Range (Manual percentage 0-100).
- **Focus Value (Param 2):** The value corresponding to the manual types.
- **Packing:** Stored in the `set_camera_focus` content struct.

Execution (Engineer's View)

Focus Management

The command is processed by the `AP_Camera` library (AP_Camera.cpp).

1. **Auto-Focus Trigger:** If `Auto (0)` is selected, ArduPilot commands the camera backend to perform a "One-shot AF" or "Continuous AF" depending on the camera's internal capabilities.
2. **Manual Overrides:** If `Range (2)` is selected, ArduPilot maps the input to the lens's focal range ($F_{min} \rightarrow F_{max}$).
3. **Lens Compatibility:** This command requires a camera with an electronically controlled focus motor (e.g., Sony block cameras, Gremsy-integrated sensors). Fixed-focus mapping cameras (like those used for photogrammetry) will ignore this command.

Data Fields (MAVLink)

- `param1` **(Type):** 0:Auto, 1:Continuous, 2:Range, 3:Meters (Rarely supported).
- `param2` **(Value):** Rate or Percentage.
- `param3` to `param7`: Unused.

Theory: The Circle of Confusion

Focusing is the process of minimizing the **Circle of Confusion** (c) on the image sensor.

- **Depth of Field (DOF):** At high zoom levels, the DOF becomes extremely shallow.
- **Vibration:** Out-of-focus images cannot be corrected in post-processing. `SET_CAMERA_FOCUS` is used to "Lock" focus before a high-vibration segment to prevent the camera's internal AF from

"hunting" due to motion blur.

Practical Use Cases

1. **Macro Inspection:**
   - *Scenario:* A drone is inspecting a weld on a bridge at a distance of 1.5 meters.
   - *Action:* `SET_CAMERA_FOCUS (Auto)` followed by `IMAGE_START_CAPTURE`.
2. **Infinity Lock:**
   - *Scenario:* Mapping from 100m altitude.
   - *Action:* Mission starts with `SET_CAMERA_FOCUS (Range: 100\%)` to lock the lens at infinity, preventing hunting during flight.

Key Parameters

- `CAM_TYPE`: Selection of camera hardware.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:307**: Focus control implementation.

SET_CAMERA_SOURCE (ID 534)

Summary

The `SET_CAMERA_SOURCE` command allows the mission to dynamically switch between different sensors (lenses) on a multi-sensor camera system. This is common on modern dual-sensor payloads that feature both a visual (RGB) and a thermal (EO/IR) sensor.

Status

Supported (All Vehicles with AP_Camera enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Instance (Param 1):** The camera ID (1-6). 0 for "All."
- **Primary Source (Param 2):** Selection for the primary stream (e.g., RGB).
- **Secondary Source (Param 3):** Selection for the secondary stream (e.g., IR).
- **Packing:** Stored in the `set_camera_source` content struct.

Execution (Engineer's View)

Multicam Logic

The command is handled by `AP_Camera::set_camera_source` (AP_Camera.cpp).

1. **Index Translation:** ArduPilot maps "Camera 1" to the first available hardware driver.
2. **Source Selection:** The autopilot sends a command to the camera backend (MAVLink or specialized driver) to reconfigure the video stream or the image capture target.
3. **Use Cases:**
   - Switching from Wide-Angle to Telephoto on a triple-lens system.
   - Switching from Visual to Night-Vision (Thermal) for a search leg.

Data Fields (MAVLink)

- `param1` **(Instance):** Camera instance number.
- `param2` **(Primary):** 0:No change, 1:RGB, 2:IR, 3:NDVI.
- `param3` **(Secondary):** Same as above.
- `param4` to `param7`: Unused.

Theory: Sensor Fusion

In modern autonomous flight, the "Source" is not just a video feed; it defines the vehicle's **Primary Intelligence Input**.

- **RGB:** Used for photogrammetry and human monitoring.
- **IR:** Used for finding heat signatures (SAR) or hot-spots (Industrial Inspection).
- **The Switch:** `SET_CAMERA_SOURCE` allows a single drone to perform two missions in one flight by reconfiguring its brain for the task at hand.

Practical Use Cases

1. **Thermal Inspection:**
    - *Scenario:* Inspecting high-voltage lines.
    - *Action:* Mission flies to the pole. `SET_CAMERA_SOURCE (IR)` is used to check for heat, then `SET_CAMERA_SOURCE (RGB)` is used to take high-res visual confirmation of the hardware.
2. **Multi-Spectrally Survey:**
    - *Scenario:* Mapping a field for crop health.
    - *Action:* Mission alternates sources to capture both RGB and NDVI (Normalized Difference Vegetation Index) data.

Key Parameters

- `CAM_TYPE`: Selection of camera hardware.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:329**: Logic for multi-sensor switching.

IMAGE_START_CAPTURE (2000) / IMAGE_STOP_CAPTURE (ID 2001)

Summary

The `IMAGE_START_CAPTURE` and `IMAGE_STOP_CAPTURE` commands provide precise control over high-resolution still image acquisition. They support single-shot, interval-based (time), and burst-based capture modes, making them the standard choice for professional survey and reconnaissance missions.

Status

Supported (All Vehicles with AP_Camera enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Instance (Param 1):** The target camera (1-6). 0 for "All."
- **Interval (Param 2):** Time between shots in seconds.
- **Total Images (Param 3):** Number of images to take. 0 for "Continuous/Unlimited."
- **Start Seq (Param 4):** Sequence number for the first image (often ignored by ArduPilot storage).
- **Packing:** Stored in the `image_start_capture` content struct.

Execution (Engineer's View)

Capture Logic

Execution is handled by `AP_Camera` (AP_Camera.cpp).

1. **Single vs. Multiple:**
   - If `Param 3 = 1`, ArduPilot triggers a single `take_picture()` event.
   - If `Param 3 > 1` or `0`, it initializes a timer-based loop `take_multiple_pictures()`.
2. **Hardware Handshake:** For MAVLink-enabled cameras (e.g., Sony Airpeak or specialized gimbals), the autopilot sends the `IMAGE_START_CAPTURE` packet down the bus to the camera. For simple cameras, it uses the Relay/PWM shutter trigger.
3. **Termination:** `IMAGE_STOP_CAPTURE` (2001) instantly kills any running timers and sends a "Stop" packet to MAVLink backends.

Data Fields (MAVLink)

- `param1` **(ID):** Camera ID.
- `param2` **(Interval):** s.
- `param3` **(Total):** Count.
- `param4` **(Seq):** Start number.

Theory: The Data Pipeline

Still images capture significantly more detail than video frames because they use the sensor's full resolution without compression artifacts.

- **Storage Bandwidth:** Capturing 42MP images every 1.5 seconds requires a high-speed UHS-II SD card. ArduPilot's mission engine handles the *triggering*, but the camera backend manages the actual file I/O.
- **Feedback:** ArduPilot listens for `CAMERA_IMAGE_CAPTURED` messages from smart backends to log the precise GPS coordinate where each image was actually written to disk.

Practical Use Cases

1. **High-Res Photogrammetry:**
   - *Scenario:* Mapping a construction site.
   - *Action:* `IMAGE_START_CAPTURE (Interval: 2s, Total: 0)`. The drone flies the grid, snapping every 2 seconds until the mission ends.
2. **Point of Interest Burst:**
   - *Scenario:* Inspecting a cracked turbine blade.
   - *Action:* Mission hovers at the blade. `IMAGE_START_CAPTURE (Interval: 0.5s, Total: 10)`. The drone captures 10 high-speed shots to ensure a clear image is obtained despite vibration.

Key Parameters

- `CAM_TRIGG_TYPE`: Defines the physical trigger method.
- `CAM_FEEDBACK_PIN`: Used to log image coordinates from a camera's "Hot Shoe" signal.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:350**: Still image capture handler.
- **libraries/AP_Camera/AP_Camera_MAVLink.cpp**: MAVLink camera bridge.

VIDEO_START_CAPTURE (2500) / VIDEO_STOP_CAPTURE (ID 2501)

Summary

The `VIDEO_START_CAPTURE` and `VIDEO_STOP_CAPTURE` commands control the recording state of onboard video sensors. They allow a mission to automatically record only the segments of flight that are relevant to the objective, saving storage space and power.

Status

Supported (All Vehicles with AP_Camera enabled)

Directionality

- **RX (Receive):** The vehicle receives this command as part of a mission upload or as a direct command.

Mission Storage (AP_Mission)

- **Stream ID (Param 1):** The target video stream or camera instance.
- **Packing:** Stored in the `video_start_capture` content struct.

Execution (Engineer's View)

Recording Logic

Execution is handled by `AP_Camera::record_video` (AP_Camera.cpp).

1. **Direct Control:** If the camera is a simple GoPro-style camera triggered via a PWM signal, ArduPilot moves the servo/pin to the "Record" position.
2. **MAVLink Control:** For high-end cameras (e.g., DJI, Sony), ArduPilot sends the standard MAVLink command to the camera's component ID.
3. **Conflict Management:** If a "Start" is commanded while the camera is already recording, ArduPilot typically ignores the second command to prevent file corruption.

Data Fields (MAVLink)

- `param1` **(ID):** Video stream/camera ID.
- `param2` **(Freq):** Record frequency (FPS). Typically ignored (set in camera menu).
- `param3` to `param7`: Unused.

Theory: Bandwidth vs. Detail

Video recording is a continuous energy drain on both the battery and the onboard processor.

- **The Overload:** Modern 4K/60fps video generates significant electromagnetic interference (EMI).
- **Autonomous Recording:** By using mission commands to start recording *after* the takeoff and *stop* before the landing, the pilot minimizes the noise exposure during the most critical flight phases (where GPS/Compass health is most vital).

Practical Use Cases

1. **Evidence Collection:**

    - *Scenario:* A security drone patrolling a site.
    - *Action:* `WAYPOINT (Start of Patrol)` → `VIDEO_START_CAPTURE`. The drone records the entire patrol leg and stops once it begins its return-to-launch.
2. **Cinematic Reveal:**
    - *Scenario:* A drone flying a specific path for a film shot.
    - *Action:* `VIDEO_START_CAPTURE` is triggered precisely at the start of the camera movement.

Key Parameters

- `CAM_TYPE`: Selection of camera hardware.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:411**: Implementation of video recording logic.
- **libraries/AP_Camera/AP_Camera_Gopro.cpp**: Specialized GoPro control.

OTHER

JUMP_TAG (600) / DO_JUMP_TAG (ID 601)

Summary

The `JUMP_TAG` and `DO_JUMP_TAG` commands provide an advanced, identifier-based flow control system for missions. Unlike the standard `DO_JUMP` (which uses sequence numbers), the tag-based system uses a "Named Marker" (a numeric tag). This allows the mission to be edited—inserting or deleting waypoints—without breaking the jump logic, as the autopilot searches for the tag rather than a fixed index.

Status

Supported (All Vehicles)

Directionality

- **RX (Receive):** The vehicle receives these commands as part of a mission upload.

Mission Storage (AP_Mission)

- **Tag (Param 1):** A user-defined integer ID (1-255).
- **Repeat (Param 2):** (601 only) Number of times to jump to that tag.
- **Search Logic:** When `DO_JUMP_TAG` is reached, the autopilot scans the entire mission command list from the beginning to find the first `JUMP_TAG` with a matching ID (AP_Mission.cpp:2279).

Execution (Engineer's View)

Robust Flow Control

1. **Tag Discovery:** The mission engine calls `find_tag_index(tag)`. This is a $O(N)$ operation where N is the number of mission items.
2. **Persistence:** Like standard jumps, ArduPilot tracks the `num_times_run` for each `DO_JUMP_TAG` using the command's unique mission index.
3. **Failure State:** If a `DO_JUMP_TAG` points to a tag that does not exist in the mission list, ArduPilot logs a warning and the mission **completes immediately** for safety.

Data Fields (MAVLink)

JUMP_TAG (600)

- `param1` **(Tag ID):** The identifier for this marker.

DO_JUMP_TAG (601)

- `param1` **(Tag ID):** The target marker to jump to.
- `param2` **(Repeat):** Number of times to jump.

Theory: Semantic vs. Positional Addressing

In computer science, `DO_JUMP` is equivalent to a **GOTO** statement with a line number. `DO_JUMP_TAG` is equivalent to a **GOTO** with a **Label**.

- **Maintainability:** Positional jumps are "Brittle." If you add a waypoint at the start of a mission, every `DO_JUMP` that follows must be manually updated to point to the new shifted indices.

- **Flexibility:** `DO_JUMP_TAG` is "Semantic." The jump points to the *meaning* of the location (e.g., "Start of Search Area"), ensuring the mission logic remains valid regardless of minor path adjustments.

Practical Use Cases

1. **Multi-Stage Search:**
   - *Scenario:* A drone needs to orbit a site, then fly a grid, then repeat.
   - *Action:* Place `JUMP_TAG (ID: 50)` at the start of the orbit. Use `DO_JUMP_TAG (Target: 50, Repeat: 5)` at the end of the grid.
2. **Dynamic Scripting:**
   - *Scenario:* A GCS script wants to loop a specific section of a mission that was just uploaded.
   - *Action:* The script searches for the Tag ID instead of parsing the whole mission list for indices.

Key Parameters

- `MIS_RESTART`: Resetting the mission will reset the tag counters.

Key Codebase Locations

- **libraries/AP_Mission/AP_Mission.cpp:2279**: Tag discovery loop.
- **libraries/AP_Mission/AP_Mission.cpp:2176**: Mission re-indexing and tag validation.