MAVLINK MESSAGE REFERENCE

ARDUPILOT IMPLEMENTATION GUIDE

GENERATED 2025 // MAVLINK HUD SYSTEMS

MAVLink Message Reference

SUPPORTED (BIDIRECTIONAL)

SYSTEM

TIMESYNC (ID 111) SUPPORTED (CRITICAL)

Summary

The `TIMESYNC` message is the primary mechanism for high-precision time synchronization between ArduPilot and external MAVLink nodes (like a Ground Control Station or a Companion Computer). It allows systems to calculate the network latency (Round Trip Time) and the clock offset between the devices, ensuring that logs and events are perfectly aligned.

Status

Supported (Critical)

Directionality

- **TX (Transmit):** All Vehicles (Requesting or Responding to sync)
- **RX (Receive):** All Vehicles (Handling sync requests/responses)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_timesync` in libraries/GCS_MAVLink/GCS_Common.cpp:3617.

Protocol Logic

The message uses two fields, `tc1` and `ts1`, to implement a "Ping-Pong" sync:

1. **Request (`tc1 == 0`):** An external system sends a packet with its current time in `ts1`. ArduPilot immediately responds by sending a packet back where `ts1` is the same, but `tc1` is ArduPilot's current system time (in nanoseconds).
2. **Response (`tc1 != 0`):** If ArduPilot receives a packet where `tc1` is non-zero, it recognizes it as a response to a request it previously sent. It calculates the Round Trip Time (RTT) and logs the sync event.

Implementation Detail

ArduPilot acts as the "Time Master" in most setups. It does **not** adjust its internal clock based on `TIMESYNC` from a GCS; instead, it provides the reference time for the GCS to adjust its own offsets.

Data Fields

- **`tc1`:** Time sync timestamp 1 (nanoseconds).
- **`ts1`:** Time sync timestamp 2 (nanoseconds).

Practical Use Cases

1. **Companion Computer Log Alignment:**
   - *Scenario:* A Jetson Nano is recording video while the drone is flying.
   - *Action:* The Jetson Nano uses `TIMESYNC` to determine exactly how many milliseconds its system clock is ahead or behind the flight controller. It then uses this offset to stamp the video frames with the vehicle's `time_boot_ms`.
2. **Precision Camera Triggering:**
   - *Scenario:* A surveyor is using a mapping camera that triggers via a MAVLink signal.

    - *Action:* By using `TIMESYNC`, the camera knows exactly when the "Trigger" message was sent by the autopilot, allowing it to record a capture timestamp that accounts for link latency.
3. **Link Quality Debugging:**
    - *Scenario:* A pilot is experiencing "laggy" controls over a long-range radio.
    - *Action:* A developer monitors the RTT calculated from `TIMESYNC`. If the RTT spikes from 100ms to 500ms, it indicates a bottleneck or interference on the telemetry link.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3617**: Primary sync handler.
- **libraries/GCS_MAVLink/GCS_Common.cpp:3677**: Transmission function (`send_timesync`).

POWER_STATUS (ID 125)

SUPPORTED

Summary

The `POWER_STATUS` message provides real-time diagnostics for the flight controller's internal power rails. It reports the board's supply voltage (Vcc), the servo rail voltage, and a series of health flags that indicate whether power sources (like a USB cable or a Power Brick) are connected and valid. This message is critical for identifying brownout risks and verifying power redundancy in professional flight systems.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports internal power health)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is in `GCS_MAVLINK::send_power_status` within libraries/GCS_MAVLink/GCS_Common.cpp:216.

Data Sourcing

- **Voltages:** Sourced from the Hardware Abstraction Layer (`hal.analogin`).
  - `vcc`: The 5V logic rail voltage in millivolts. Typically ~5000mV.
  - `vservo`: The voltage on the servo output rail in millivolts. Used to monitor the health of the BEC or battery powering the servos.
- **Flags:** A bitmask (`MAV_POWER_STATUS`) updated in libraries/AP_HAL_ChibiOS/AnalogIn.cpp:792.
  - `MAV_POWER_STATUS_BRICK_VALID`: Indicates a healthy power supply from the primary battery brick.
  - `MAV_POWER_STATUS_USB_CONNECTED`: Indicates a USB cable is powering the board.
  - `MAV_POWER_STATUS_CHANGED`: Set if the power configuration changed during flight (e.g., a battery was lost).

Data Fields

- `Vcc`: 5V rail voltage in millivolts.
- `Vservo`: Servo rail voltage in millivolts.
- `flags`: Power status flags (`MAV_POWER_STATUS`).

Practical Use Cases

1. **Brownout Prevention:**
   - *Scenario:* A pilot is using high-torque servos that pull a lot of current from the 5V rail.
   - *Action:* The GCS monitors the `vcc` field. If the voltage drops below 4.5V during aggressive maneuvers, the GCS triggers a loud "Low Vcc Warning" to warn the pilot of an imminent crash.
2. **Redundancy Verification:**
   - *Scenario:* A high-end drone has dual power bricks for reliability.
   - *Action:* Before takeoff, the pilot checks the `flags` field in the "Status" tab to ensure both `MAV_POWER_STATUS_BRICK_VALID` and a secondary source bit are set.

3. **USB Debugging:**
   - *Scenario:* A user is trying to configure a drone but can't get the motors to spin.
   - *Action:* The GCS shows `USB_CONNECTED` is active. Since ArduPilot generally prohibits motor arming while on USB for safety, the pilot knows they must disconnect the cable and use telemetry to test the motors.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:216**: Primary message constructor.
- **libraries/AP_HAL_ChibiOS/AnalogIn.cpp:792**: Hardware-level power flag logic.

SERIAL_CONTROL (ID 126)

SUPPORTED

Summary

The `SERIAL_CONTROL` message allows a Ground Control Station (GCS) or companion computer to perform low-level read/write operations on the vehicle's serial ports. It serves as a "tunnel," allowing remote configuration of devices (like GPS modules or radios) attached to the autopilot without physical access.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Response data)
- **RX (Receive):** All Vehicles (Write data / Request read)

Mechanics

The implementation is in `GCS_MAVLINK::handle_serial_control` in libraries/GCS_MAVLink/GCS_serial_control.cpp:33.

Operations

1. **Port Selection:** The `device` field targets a specific port (e.g., `SERIAL_CONTROL_DEV_GPS1`).
2. **Locking:** Setting `SERIAL_CONTROL_FLAG_EXCLUSIVE` locks the port, preventing the normal driver (like AP_GPS) from reading/writing. This allows the GCS to take full control.
3. **Writing:** Data in the `data` field is written to the UART.
4. **Reading:** If `SERIAL_CONTROL_FLAG_RESPOND` is set, the autopilot waits up to `timeout` milliseconds for data to become available on the UART, then sends it back in a `SERIAL_CONTROL` reply (with `SERIAL_CONTROL_FLAG_REPLY` set).

Data Fields

- **`device`**: Serial control device type (`SERIAL_CONTROL_DEV` enum). 0=Telem1, 2=GPS1, 10=Shell, 100+=Serial0...9.
- **`flags`**: Bitmap of flags (`SERIAL_CONTROL_FLAG` enum).
  - 1: Reply (set by vehicle).
  - 2: Respond (request response).
  - 4: Exclusive (lock port).
  - 8: Blocking (wait for buffer space).
  - 16: Multi (allow multiple response packets).
- **`timeout`**: Timeout for reply data in milliseconds.
- **`baudrate`**: Baudrate of transfer. 0 to keep current.
- **`count`**: How many bytes of data are valid.
- **`data`**: Data payload (up to 70 bytes).

Practical Use Cases

1. **u-blox Pass-Through:**
   - *Scenario:* A user wants to debug a GPS configuration using u-center.
   - *Action:* Mission Planner sends `SERIAL_CONTROL` messages to lock the GPS port and tunnel traffic between u-center (on PC) and the GPS module (on drone) via the MAVLink telemetry

link.

2. **Remote Shell:**
    - *Scenario:* A developer needs to run NSH commands on Pixhawk.
    - *Action:* Sending `device=SERIAL_CONTROL_DEV_SHELL` routes data to the NuttX Shell (NSH), allowing a remote terminal session.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_serial_control.cpp:33**: Implementation of the handler.

AUTOPILOT_VERSION (ID 148)

SUPPORTED (CRITICAL)

Summary

The `AUTOPILOT_VERSION` message is a critical diagnostic packet that defines the vehicle's identity and capabilities. It reports the firmware version (e.g., ArduCopter 4.5.1), the specific Git SHA hash of the build, and a bitmask of supported MAVLink features (e.g., whether the drone supports FTP, Mission Commands, or Geofencing). Ground Control Stations use this message during the initial connection handshake to tailor the user interface to the vehicle's specific features.

Status

Supported (Critical)

Directionality

- **TX (Transmit):** All Vehicles (Identity and Capability report)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_autopilot_version` within libraries/GCS_MAVLink/GCS_Common.cpp:2912.

Data Sourcing

- **Version Info:** Sourced from the `AP_FWVersion` library.
    - `flight_sw_version`: A packed integer representing Major, Minor, and Patch versions.
    - `middleware_sw_version`: Often contains the version of the underlying HAL (e.g., ChibiOS).
    - `flight_custom_version`: Stores the 8-byte Git SHA hash of the ArduPilot source tree.
- **Capabilities:** A 64-bit bitmask calculated in `GCS_MAVLink::capabilities()` (GCS_Common.cpp:6980).
    - Flags include `MAV_PROTOCOL_CAPABILITY_MISSION_INT`, `MAV_PROTOCOL_CAPABILITY_PARAM_FLOAT`, and `MAV_PROTOCOL_CAPABILITY_COMMAND_INT`.

Trigger Logic

ArduPilot typically sends this message:

1. **On Request:** In response to `MAV_CMD_REQUEST_AUTOPILOT_CAPABILITIES`.
2. **Handshake:** Automatically as part of the initial connection sequence with most modern Ground Control Stations.

Data Fields

- `capabilities`: Bitmap of capabilities.
- `flight_sw_version`: Firmware version number.
- `middleware_sw_version`: Middleware version number.
- `os_sw_version`: Operating system version number.
- `board_version`: HW / board version (last 8 bytes should be silicon ID, if any).
- `flight_custom_version`: Custom version field, commonly the first 8 bytes of the git hash.
- `middleware_custom_version`: Custom version field, commonly the first 8 bytes of the git hash.
- `os_custom_version`: Custom version field, commonly the first 8 bytes of the git hash.
- `vendor_id`: ID of the board vendor.

- `product_id`: ID of the product.
- `uid`: UID if provided by hardware.
- `uid2`: UID if provided by hardware.

Practical Use Cases

1. **UI Customization:**
   - *Scenario:* A pilot connects a "Blimp" to Mission Planner.
   - *Action:* Mission Planner reads `AUTOPILOT_VERSION`, identifies the vehicle as a Blimp, and hides Copter-specific widgets (like "Motor Test") while showing Blimp-specific controls.
2. **Firmware Integrity Verification:**
   - *Scenario:* An industrial operator needs to ensure all drones in a fleet are running an identical, certified firmware build.
   - *Action:* A script queries `AUTOPILOT_VERSION` and compares the `flight_custom_version` (Git SHA) against the certified hash. If they don't match, the drone is grounded.
3. **Feature Discovery:**
   - *Scenario:* A developer is adding support for "MAVLink FTP" to a custom companion computer.
   - *Action:* The computer checks the `capabilities` bitmask. If `MAV_PROTOCOL_CAPABILITY_FTP` is not set, the computer falls back to standard parameter requests.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2912**: Implementation of the version report.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6980**: Capability bitmask calculation logic.
- **libraries/AP_Common/AP_FWVersion.h**: Internal structure for version tracking.

MEMINFO (ID 152)

SUPPORTED

Summary

The `MEMINFO` message provides diagnostics regarding the flight controller's internal memory usage. It reports the amount of free heap memory available to the system, which is critical for ensuring the stability of long-running tasks, Lua scripts, and communication buffers.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports memory health)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is in `GCS_MAVLINK::send_meminfo` within libraries/GCS_MAVLink/GCS_Common.cpp:208.

Data Sourcing

- **Available Memory:** Sourced via `hal.util->available_memory()`. This returns the number of bytes currently free in the system heap.
- **Fields:**
    - `brkval`: Historically used for the heap break value. In modern ChibiOS-based ArduPilot, this is typically set to 0.
    - `freemem`: The primary field, reporting free memory in bytes. Note: The standard MAVLink message uses a 16-bit field for `freemem`, but ArduPilot uses an extension field (`freemem32`) to support modern MCUs with more than 64KB of RAM.

Data Fields

- `brkval`**:** Heap top.
- `freemem`**:** Free memory (16-bit field, legacy).
- `freemem32`**:** Free memory (32-bit field, preferred).

Practical Use Cases

1. **Lua Script Debugging:**
    - *Scenario:* A developer is writing a complex Lua script to perform autonomous package delivery.
    - *Action:* The developer monitors the `freemem` field in the GCS. If the free memory steadily decreases over time, it indicates a "memory leak" in the script (e.g., global variables being created inside a loop).
2. **State-of-Health Monitoring:**
    - *Scenario:* An industrial drone is performing a 2-hour autonomous inspection.
    - *Action:* The GCS logs the `MEMINFO` message. If free memory drops below a safety threshold (e.g., 10\% of total RAM), the GCS triggers a "Low Memory Warning," advising the operator to land and reboot.
3. **Firmware Stress Testing:**

    - *Scenario:* Developers are testing a new feature that uses large internal buffers (e.g., Terrain following or advanced ADSB).
    - *Action:* By checking `MEMINFO` during flight, they can verify that the system has enough headroom to handle peak processing loads without crashing.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:208**: Implementation of the memory report.
- **libraries/AP_HAL/Util.h**: Defines the `available_memory()` interface.

HWSTATUS (ID 165)

SUPPORTED

Summary

The `HWSTATUS` message reports basic hardware health metrics, specifically the board input voltage and I2C error counts.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports hardware status)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission is handled by `GCS_MAVLINK::send_hwstatus` within libraries/GCS_MAVLink/GCS_Common.cpp:5722.

Data Source

- `Vcc`: Retrieved from `hal.analogin->board_voltage()`.
- `I2Cerr`: Hardcoded to 0 in modern ArduPilot versions (I2C errors are now typically tracked per-bus in other diagnostic messages).

Data Fields

- `Vcc`: Board voltage (mV).
- `I2Cerr`: I2C error count (legacy field, often 0).

Practical Use Cases

1. **Brownout Detection:**
   - *Scenario:* A user suspects their flight controller is rebooting in flight.
   - *Action:* Reviewing the `HWSTATUS.Vcc` log shows dips below 4.5V, indicating an inadequate power supply or BEC brownout.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5722**: Implementation of the sender.

MESSAGE_INTERVAL (ID 244)

SUPPORTED

Summary

The `MESSAGE_INTERVAL` message provides information about the current frequency (interval) of a specific MAVLink message on a telemetry link. In ArduPilot, this message is used to respond to Ground Control Station queries about stream rates, allowing the GCS to verify that the vehicle is sending data at the requested speed.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports current message interval)
- **RX (Receive):** None (Use `MAV_CMD_SET_MESSAGE_INTERVAL` to change rates)

Transmission (TX)

ArduPilot sends `MESSAGE_INTERVAL` primarily in response to the `MAV_CMD_GET_MESSAGE_INTERVAL` command. The logic is implemented in `GCS_MAVLINK::handle_command_get_message_interval` within libraries/GCS_MAVLink/GCS_Common.cpp:3241.

Response Logic

- **Active Message:** If the requested message ID is being streamed, ArduPilot sends the interval in microseconds.
- **Disabled Message:** If the message is currently disabled, it sends `-1`.
- **Unsupported Message:** If the message ID is unknown to the firmware, it sends `0`.

Reception (RX)

ArduPilot **does not** handle the `MESSAGE_INTERVAL` message if received from an external system. To *change* a message rate, a GCS must use the command protocol:

- `MAV_CMD_SET_MESSAGE_INTERVAL` **(511):** Used to request a specific message ID at a specific interval. Handled in `GCS_MAVLINK::set_message_interval` (GCS_Common.cpp:3126).

Data Fields

- `message_id`: The ID of the message for which this interval is being reported.
- `interval_us`: The interval between two messages, in microseconds. A value of -1 indicates this message is disabled, 0 indicates the message is not supported, > 0 indicates the interval in microseconds.

Practical Use Cases

1. **Dynamic HUD Refresh Rates:**
   - *Scenario:* A GCS wants to show high-speed orientation data ($50Hz$) during takeoff but reduce it to $10Hz$ during cruise to save battery and bandwidth.
   - *Action:* The GCS sends `MAV_CMD_SET_MESSAGE_INTERVAL` for `ATTITUDE` (30). It then sends `MAV_CMD_GET_MESSAGE_INTERVAL` to verify ArduPilot has successfully applied the $20000\mu s$

(50Hz) interval.

2. **Telemetry Bandwidth Management:**
    - *Scenario:* A pilot is using a low-speed satellite link.
    - *Action:* The GCS queries the intervals of all active streams. It can then identify high-bandwidth messages that aren't strictly necessary and disable them to prioritize critical telemetry.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3241**: Logic for responding with the current interval.
- **libraries/GCS_MAVLink/GCS_Common.cpp:3126**: Logic for setting a new interval.

NAMED_VALUE_FLOAT (ID 251)

SUPPORTED (RX & TX)

Summary

Key-value pair of a string name and a float value. Used extensively for debugging and Lua scripting integration.

Status

Supported (RX & TX)

Directionality

- **TX (Transmit):** All Vehicles - Sends debug/script data.
- **RX (Receive):** All Vehicles - Receives debug/script data.

Transmission (TX)

Lua scripts often use `gcs:send_named_float('my_var', value)` to output debug data to the GCS graph.

Source: libraries/GCS_MAVLink/GCS.cpp

Reception (RX)

Handled by `GCS_MAVLink::handle_named_value`. Useful for injecting data into scripts or logging external events.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Data Fields

- `time_boot_ms`: Timestamp.
- `name`: Name string (10 chars max).
- `value`: Float value.

Practical Use Cases

1. **Lua Script Debugging:**
    - *Scenario:* Developing a complex Lua script for thermal soaring.
    - *Action:* The script sends `gcs:send_named_float('vario', climb_rate)` to visualize the internal climb rate filter on the GCS tuning graph.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS.cpp:122**: Sending.

STATUSTEXT (ID 253)

SUPPORTED (RX & TX)

Summary

Text message status report. This is the primary way ArduPilot communicates errors, warnings, and info messages to the user.

Status

Supported (RX & TX)

Directionality

- **TX (Transmit):** All Vehicles - Sends status messages.
- **RX (Receive):** All Vehicles - Receives status messages (e.g., from companion computer).

Transmission (TX)

ArduPilot sends this message whenever `gcs().send_text()` is called. This covers everything from "Armed" messages to "EKF Variance" errors.

Reception (RX)

Handled by `GCS_MAVLink::handle_statustext`.

- **Logging:** Logs the text to the Dataflash log.
- **Forwarding:** Forwards the message to other active MAVLink channels (e.g., forwarding a message from a companion computer to the GCS).

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Data Fields

- `severity`: Severity level (MAV_SEVERITY_EMERGENCY to MAV_SEVERITY_DEBUG).
- `text`: Text string (50 chars max).

Practical Use Cases

1. **Failsafe Notification:**
   - *Scenario:* Battery low.
   - *Action:* ArduPilot sends `STATUSTEXT` (Severity: CRITICAL) "Battery Low!". The GCS speaks this alert to the pilot.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4328**: Handler.

DEVICE_OP_READ (ID 11000)

SUPPORTED

Summary

The `DEVICE_OP_READ` message provides a mechanism for low-level debugging of I2C and SPI devices connected to the Autopilot. It allows a developer to read raw registers from sensors or peripherals directly via MAVLink, bypassing the standard driver abstractions.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Executes the read operation)

Usage

This message is typically sent by a developer tool (like MAVProxy) to diagnose hardware issues (e.g., "Is the compass responding on I2C bus 0 at address 0x1E?").

Core Logic

The handler is implemented in `GCS_MAVLINK::handle_device_op_read` within libraries/GCS_MAVLink/GCS_DeviceOp.cpp:34.

1. **Device Lookup:** It resolves the device using `hal.i2c_mgr->get_device()` or `hal.spi->get_device()` based on the `bustype`.
2. **Semaphore:** It attempts to take the bus semaphore to ensure atomic access.
3. **Read:** It performs the read operation (`read_bank_registers` or `transfer_bank`).
4. **Reply:** It sends a `DEVICE_OP_READ_REPLY` (11001) with the data or an error code.

Data Fields

- **`target_system` / `target_component`:** Targeted component.
- **`request_id`:** ID to match request with reply.
- **`bustype`:** `DEVICE_OP_BUSTYPE_I2C` (0) or `DEVICE_OP_BUSTYPE_SPI` (1).
- **`bus`:** I2C Bus ID.
- **`address`:** I2C Device Address.
- **`busname`:** SPI Bus Name (string).
- **`regstart`:** First register to read.
- **`count`:** Number of bytes to read.
- **`bank`:** Bank number (for devices with banked registers).

Practical Use Cases

1. **Sensor Debugging:**
    - *Scenario:* A new Compass sensor is not being detected.
    - *Action:* The developer sends `DEVICE_OP_READ` to the sensor's "WHO_AM_I" register (e.g., 0x75). If the reply contains the correct ID, the hardware is working, and the issue is likely in the driver initialization.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_DeviceOp.cpp:34**: Implementation of the handler.

DEVICE_OP_READ_REPLY (ID 11001)

SUPPORTED

Summary

The `DEVICE_OP_READ_REPLY` message is the response to a `DEVICE_OP_READ` (11000) command. It returns the requested register data from an I2C or SPI device, along with a result code indicating success or failure.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports read results to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `GCS_MAVLINK` library immediately after attempting the requested device read operation.

Core Logic

The implementation is in `GCS_MAVLINK::handle_device_op_read` within libraries/GCS_MAVLink/GCS_DeviceOp.cpp:77.

It populates the `data` buffer with the bytes read from the device.

Data Fields

- **`request_id`**: ID matching the request.
- **`result`**: 0=Success, 1=Unknown bus, 2=Unknown device, 3=Semaphore error, 4=Read error, 5=Buffer overflow.
- **`regstart`**: The starting register that was read.
- **`count`**: Number of bytes read.
- **`data`**: Raw data buffer (up to 128 bytes).
- **`bank`**: Bank number.

DEVICE_OP_WRITE (ID 11002) SUPPORTED

Summary

The `DEVICE_OP_WRITE` message provides a mechanism for low-level debugging of I2C and SPI devices. It allows a developer to write raw values to registers on sensors or peripherals directly via MAVLink.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Executes the write operation)

Usage

WARNING: Using this message allows direct hardware manipulation. Writing incorrect values to sensor registers can crash drivers, disable sensors, or cause physical damage. Use with extreme caution.

Core Logic

The handler is implemented in `GCS_MAVLINK::handle_device_op_write` within libraries/GCS_MAVLink/GCS_DeviceOp.cpp:101.

1. **Device Lookup:** Resolves the device via I2C bus/address or SPI name.
2. **Write:** Performs `write_bank_register` or `transfer_bank` (if `regstart == 0xff`).
3. **Reply:** Sends `DEVICE_OP_WRITE_REPLY` (11003) with the result code.

Data Fields

- **`target_system`** / **`target_component`**: Targeted component.
- **`request_id`**: ID to match request with reply.
- **`bustype`**: `DEVICE_OP_BUSTYPE_I2C` (0) or `DEVICE_OP_BUSTYPE_SPI` (1).
- **`bus`**: I2C Bus ID.
- **`address`**: I2C Device Address.
- **`busname`**: SPI Bus Name (string).
- **`regstart`**: First register to write. Set to 0xFF for raw transfer.
- **`count`**: Number of bytes to write.
- **`data`**: Raw data buffer (up to 128 bytes).
- **`bank`**: Bank number.

DEVICE_OP_WRITE_REPLY (ID 11003)

SUPPORTED

Summary

The `DEVICE_OP_WRITE_REPLY` message is the response to a `DEVICE_OP_WRITE` (11002) command. It indicates whether the low-level register write operation was successful.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports write status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `GCS_MAVLINK` library immediately after attempting the requested device write operation.

Core Logic

The implementation is in `GCS_MAVLINK::handle_device_op_write` within libraries/GCS_MAVLink/GCS_DeviceOp.cpp:140.

Data Fields

- **`request_id`:** ID matching the request.
- **`result`:** 0=Success, 1=Unknown bus, 2=Unknown device, 3=Semaphore error, 4=Write error.

SECURE_COMMAND (ID 11004)

SUPPORTED

Summary

The `SECURE_COMMAND` message allows a Ground Control Station (GCS) to perform sensitive, privileged operations on the Autopilot or connected peripherals. These operations, such as updating bootloader keys or configuring Remote ID, are protected by a digital signature to prevent unauthorized access.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Executes the secure command)

Usage

The command payload must be signed using a private key corresponding to a public key already known to the Autopilot. The signature covers the sequence number, operation, data, and a session key (retrieved via `SECURE_COMMAND_GET_SESSION_KEY`).

Core Logic

The handler is implemented in `AP_CheckFirmware::handle_secure_command` within libraries/AP_CheckFirmware/AP_CheckFirmware_secure_command.cpp:270.

1. **Verification:** It calls `check_signature()` to verify the command payload against the stored public keys.
2. **Execution:** If valid, it performs the requested operation (e.g., updating the bootloader's key table).
3. **Reply:** It sends a `SECURE_COMMAND_REPLY` (11005) with the result.

Operations (`SECURE_COMMAND_OP`)

- `GET_SESSION_KEY`: Retrieve a temporary session key.
- `GET_REMOTEID_SESSION_KEY`: Retrieve session key for Remote ID module.
- `REMOVE_PUBLIC_KEYS`: Clear public keys.
- `GET_PUBLIC_KEYS`: Read stored public keys.
- `SET_PUBLIC_KEYS`: Write new public keys.
- `GET_REMOTEID_CONFIG`: Read Remote ID config.
- `SET_REMOTEID_CONFIG`: Write Remote ID config.
- `FLASH_BOOTLOADER`: Update the bootloader.

Data Fields

- **`target_system` / `target_component`:** Targeted component.
- **`sequence`:** Sequence number to prevent replay attacks.
- **`operation`:** Operation ID (`SECURE_COMMAND_OP`).
- **`data_length`:** Length of data payload.
- **`sig_length`:** Length of signature.
- **`data`:** Payload followed by Signature (max 220 bytes total).

SECURE_COMMAND_REPLY (ID 11005) SUPPORTED

Summary

The `SECURE_COMMAND_REPLY` message is the response to a `SECURE_COMMAND` (11004). It indicates the success or failure of the requested secure operation and returns any requested data (e.g., session keys, public keys).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports result to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_CheckFirmware` library immediately after processing the secure command.

Core Logic

The implementation is in `AP_CheckFirmware::handle_secure_command` within libraries/AP_CheckFirmware/AP_CheckFirmware_secure_command.cpp:386.

Data Fields

- **`sequence`**: Sequence number from the request.
- **`operation`**: Operation ID from the request.
- **`result`**: Result code (`MAV_RESULT`).
  - `ACCEPTED`: Success.
  - `DENIED`: Signature verification failed.
  - `UNSUPPORTED`: Unknown operation.
  - `FAILED`: Internal error (e.g., flash write failed).
- **`data_length`**: Length of return data.
- **`data`**: Return data (up to 220 bytes), such as the Session Key.

OSD_PARAM_CONFIG (ID 11033)

SUPPORTED

Summary

The `OSD_PARAM_CONFIG` message allows a Ground Control Station (GCS) to configure the "Parameter Tuning Screens" of the onboard OSD. This feature enables pilots to tune specific parameters (like PIDs or Rates) using the RC transmitter sticks and the OSD menu, without needing a GCS connected.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Configures OSD screens)

Usage

The GCS sends this message to define which parameters appear on which OSD screen and in what order.

Core Logic

The handler is implemented in `AP_OSD::handle_write_msg` (which delegates to `AP_OSD_ParamScreen`) within libraries/AP_OSD/AP_OSD.cpp:654.

It writes the configuration to the `OSD_ParamScreen` backend.

Data Fields

- **`target_system`** / **`target_component`**: Targeted component.
- **`request_id`**: ID to match request with reply.
- **`osd_screen`**: OSD Screen index (1-based).
- **`osd_screen_param_index`**: Index within the screen (0-based).
- **`param_id`**: Parameter name (string).
- **`config_type`**: `OSD_PARAM_CONFIG_TYPE` (e.g., Min, Max, Step, Value).
- **`min_value`**: Minimum value for tuning.
- **`max_value`**: Maximum value for tuning.
- **`increment`**: Step size for tuning.

Practical Use Cases

1. **Field Tuning:**
   - *Scenario:* A pilot is tuning a new racing drone at the field.
   - *Action:* They use the GCS to map `ATC_RAT_RLL_P`, `ATC_RAT_RLL_I`, and `ATC_RAT_RLL_D` to OSD Screen 1 using `OSD_PARAM_CONFIG`. Now, they can land, adjust PIDs using their goggles and sticks, and take off again instantly.
2. **Mission Configuration:**
   - *Scenario:* A survey drone needs adjustable overlap.
   - *Action:* The integrator maps the custom script parameter `SURVEY_OVERLAP` to the OSD menu, allowing the operator to change the mission parameter without a laptop.

OSD_PARAM_CONFIG_REPLY (ID 11034)

SUPPORTED

Summary

The `OSD_PARAM_CONFIG_REPLY` message is the response to an `OSD_PARAM_CONFIG` (11033) command. It indicates whether the requested OSD parameter configuration was accepted or rejected.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports config result to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_OSD` library immediately after processing the configuration command.

Core Logic

The implementation is in `AP_OSD_ParamScreen::handle_write_msg` within libraries/AP_OSD/AP_OSD_ParamScreen.cpp:597.

Data Fields

- `request_id`: ID matching the request.
- `result`: `OSD_PARAM_CONFIG_ERROR` enum (Success, Invalid Screen, Invalid Parameter, etc.).

Practical Use Cases

1. **Validation:**
   - *Scenario:* A user tries to map a non-existent parameter "FOO_BAR" to the OSD.
   - *Action:* The Autopilot replies with `OSD_PARAM_CONFIG_REPLY` containing `OSD_PARAM_INVALID_PARAMETER`. The GCS displays an error message to the user: "Parameter not found."
2. **Sync Confirmation:**
   - *Scenario:* A GCS is restoring a saved OSD layout.
   - *Action:* It sends 10 config messages in rapid succession. It waits for 10 `OSD_PARAM_CONFIG_REPLY` messages with `SUCCESS` to confirm the layout has been fully applied to the flight controller.

OSD_PARAM_SHOW_CONFIG (ID 11035)

SUPPORTED

Summary

The `OSD_PARAM_SHOW_CONFIG` message allows a Ground Control Station (GCS) to query the current configuration of the OSD Parameter Tuning screens. This allows the GCS to display or edit the current layout of the OSD menu.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Processes the query)

Usage

The GCS requests information about a specific entry in the OSD parameter list.

Core Logic

The handler is implemented in `AP_OSD_ParamScreen::handle_read_msg` within libraries/AP_OSD/AP_OSD_ParamScreen.cpp:606.

It looks up the parameter configured at the requested index and replies with its name, limits, and type.

Data Fields

- `target_system` / `target_component`: Targeted component.
- `request_id`: ID to match request with reply.
- `osd_screen`: OSD Screen index.
- `osd_screen_param_index`: Index within the screen.

OSD_PARAM_SHOW_CONFIG_REPLY (ID 11036)

SUPPORTED

Summary

The `OSD_PARAM_SHOW_CONFIG_REPLY` message is the response to an `OSD_PARAM_SHOW_CONFIG` (11035) query. It returns the details of a specific parameter configured on the OSD screen, including its name, min/max limits, and tuning increment.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports config to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_OSD` library immediately after processing the query.

Core Logic

The implementation is in `AP_OSD_ParamScreen::handle_read_msg` within libraries/AP_OSD/AP_OSD_ParamScreen.cpp:606.

Data Fields

- **`request_id`**: ID matching the request.
- **`result`**: `OSD_PARAM_CONFIG_ERROR` enum.
- **`param_id`**: Parameter name string.
- **`config_type`**: Config type (`OSD_PARAM_CONFIG_TYPE`).
- **`min_value`**: Minimum value.
- **`max_value`**: Maximum value.
- **`increment`**: Step size.

MCU_STATUS (ID 11039) SUPPORTED

Summary

The `MCU_STATUS` message reports the health telemetry of the Flight Controller's microcontroller unit (MCU). This includes the core temperature and the input voltage to the MCU rail (VDD_5V or VDD_3V3 depending on board design).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports MCU health to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `GCS_MAVLINK` library using data from the HAL (Hardware Abstraction Layer).

Core Logic

The implementation is in `GCS_MAVLINK::send_mcu_status` within libraries/GCS_MAVLink/GCS_Common.cpp:244.

It reads directly from the `hal.analogin` interface:

- `mcu_temperature()`: Internal temperature sensor of the STM32.
- `mcu_voltage()`: Voltage measured on the VDD pin or internal reference.

Data Fields

- **`MCU_Temperature`:** Temperature in centi-degrees Celsius (degC * 100).
- **`MCU_Voltage`:** Voltage in milli-volts (mV).
- **`MCU_Voltage_min`:** Minimum voltage recorded since boot (mV).
- **`MCU_Voltage_max`:** Maximum voltage recorded since boot (mV).
- **`id`:** MCU instance ID (usually 0).

Practical Use Cases

1. **Overheating Warning:**
   - *Scenario:* A flight controller is enclosed in a sealed box with no airflow.
   - *Action:* The GCS monitors `MCU_Temperature`. If it exceeds 85°C, it warns the pilot of potential thermal shutdown or IMU degradation.
2. **Brownout Detection:**
   - *Scenario:* The 5V BEC powering the flight controller is overloaded by a servo.
   - *Action:* The `MCU_Voltage_min` field drops below 4500mV. Post-flight logs reveal the dip, explaining why the GPS glitched (as its voltage also sagged).

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:244**: Implementation of the sender.

TELEMETRY

ATTITUDE (ID 30)

SUPPORTED

Summary

The `ATTITUDE` message provides the vehicle's orientation in three-dimensional space using Euler angles (Roll, Pitch, and Yaw). It also includes the angular velocity (rotation rates) for each axis. This is the primary telemetry packet used by Ground Control Stations to drive the Artificial Horizon and heading indicators.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports orientation to GCS)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_attitude` within libraries/GCS_MAVLink/GCS_Common.cpp:5816.

Data Sourcing

- **State Estimator:** Data is retrieved directly from the `AP_AHRS` (Attitude and Heading Reference System) library, which fuses data from the IMU, Compass, and GPS.
- **Format:**
  - `roll`, `pitch`, `yaw`: Provided in radians.
  - `rollspeed`, `pitchspeed`, `yawspeed`: Angular velocity in rad/s.
- **Timestamp:** Uses `AP_HAL::millis()` since boot.

Scheduling

- Sent as part of the `MSG_ATTITUDE` stream.
- Triggered in `GCS_Common.cpp:6065` within the `try_send_message` loop.
- High-frequency message: Typically streamed at 20Hz - 50Hz for a smooth UI experience.

Data Fields

- **`time_boot_ms`**: Timestamp (milliseconds since system boot).
- **`roll`**: Roll angle (rad, -pi..+pi).
- **`pitch`**: Pitch angle (rad, -pi..+pi).
- **`yaw`**: Yaw angle (rad, -pi..+pi).
- **`rollspeed`**: Roll angular speed (rad/s).
- **`pitchspeed`**: Pitch angular speed (rad/s).
- **`yawspeed`**: Yaw angular speed (rad/s).

Practical Use Cases

1. **Artificial Horizon:**
   - *Scenario:* A pilot is flying FPV (First Person View) through thick fog.
   - *Action:* The GCS HUD uses the `roll` and `pitch` values to draw a virtual horizon, allowing the pilot to maintain level flight without visual cues from the camera.

2. **Heading Lock Verification:**
    - *Scenario:* During an AUTO mission, the user wants to ensure the drone is pointing toward the next waypoint.
    - *Action:* The GCS monitors the `yaw` field and compares it against the "Desired Yaw" to verify the flight controller is tracking the path correctly.
3. **Vibration/Buffeting Detection:**
    - *Scenario:* A plane is flying in high turbulence.
    - *Action:* A developer analyzes the `rollspeed` and `pitchspeed` fields to quantify how much the airframe is being shaken by external forces.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5816**: Implementation of `send_attitude`.
- **libraries/AP_AHRS/AP_AHRS.h**: Source of orientation data.

ATTITUDE_QUATERNION (ID 31)

SUPPORTED

Summary

The `ATTITUDE_QUATERNION` message provides the vehicle's orientation using unit quaternions (q_w, q_x, q_y, q_z). While Euler angles (used in the `ATTITUDE` message) are more intuitive for human pilots, quaternions are mathematically superior for 3D visualization and navigation algorithms because they avoid "Gimbal Lock"—a state where orientation becomes ambiguous at 90 degrees of pitch.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Advanced orientation telemetry)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_attitude_quaternion` within libraries/GCS_MAVLink/GCS_Common.cpp:5833.

Data Sourcing

- **Source:** Data is retrieved from the `AP_AHRS` library via `AP::[ahrs](/field-manual/mavlink-interface/ahrs.html)().get_quaternion()`.
- **Fields:**
  - `q1` (q_w), `q2` (q_x), `q3` (q_y), `q4` (q_z): Normalized components of the attitude quaternion.
  - `rollspeed`, `pitchspeed`, `yawspeed`: Angular velocity in rad/s (same as ID 30).
- **Timestamp:** Uses `AP_HAL::millis()` since boot.

Scheduling

- Sent as part of the `MSG_ATTITUDE_QUATERNION` stream.
- Triggered in `GCS_Common.cpp:6070` within the `try_send_message` loop.

Data Fields

- **`time_boot_ms`:** Timestamp (milliseconds since system boot).
- **`q1`:** Quaternion component 1, w (1 in null-rotation).
- **`q2`:** Quaternion component 2, x (0 in null-rotation).
- **`q3`:** Quaternion component 3, y (0 in null-rotation).
- **`q4`:** Quaternion component 4, z (0 in null-rotation).
- **`rollspeed`:** Roll angular speed (rad/s).
- **`pitchspeed`:** Pitch angular speed (rad/s).
- **`yawspeed`:** Yaw angular speed (rad/s).

Practical Use Cases

1. **3D Model Rendering:**
   - *Scenario:* A Ground Control Station displays a high-fidelity 3D model of the drone that rotates in real-time.

    - *Action:* The renderer uses the quaternion data to update the model's rotation matrix. This ensures smooth movement even during vertical climbs or aerobatic maneuvers where Euler angles might glitch.
2. **VR/AR HUDs:**
    - *Scenario:* A pilot is using AR (Augmented Reality) glasses to see the drone's attitude overlaid on their real-world vision.
    - *Action:* Quaternions are used to align the virtual horizon precisely with the headset's inertial frame without complex trigonometric conversions.
3. **Advanced Log Analysis:**
    - *Scenario:* A researcher is analyzing the stability of a new airframe design.
    - *Action:* By using quaternions, the researcher can accurately calculate the error between "Desired" and "Actual" orientation throughout the entire sphere of rotation.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5833**: Implementation of `send_attitude_quaternion`.
- **libraries/AP_AHRS/AP_AHRS.h**: Source of quaternion data.

LOCAL_POSITION_NED (ID 32)

SUPPORTED

Summary

The LOCAL_POSITION_NED message provides the vehicle's position and velocity in a local North-East-Down (NED) coordinate system. Unlike GLOBAL_POSITION_INT, which uses Latitude/Longitude, this message uses meters relative to a fixed local "Origin" (usually the EKF origin established at boot or GPS lock). It is essential for navigation in environments where relative distance is more critical than absolute global coordinates.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Local state telemetry)
- **RX (Receive):** None (Use VISION_POSITION_ESTIMATE or ODOMETRY for external input)

Transmission (TX)

The transmission logic resides in GCS_MAVLINK::send_local_position within libraries/GCS_MAVLink/GCS_Common.cpp:2979.

Data Sourcing

- **State Estimator:** Data is retrieved from the AP_AHRS library.
- **Position:** Sourced via [ahrs](/field-manual/mavlink-interface/ahrs.html).get_relative_position_NED_origin(). This returns meters North, East, and Down relative to the EKF origin.
- **Velocity:** Sourced via ahrs.get_velocity_NED(), providing ground speed components in m/s.
- **Timestamp:** Uses AP_HAL::millis() since boot.

Scheduling

- Sent as part of the MSG_LOCAL_POSITION stream.
- Triggered in GCS_Common.cpp:6134 within the try_send_message loop.

Data Fields

- **time_boot_ms**: Timestamp (milliseconds since system boot).
- **x**: X Position (meters).
- **y**: Y Position (meters).
- **z**: Z Position (meters).
- **vx**: X Speed (m/s).
- **vy**: Y Speed (m/s).
- **vz**: Z Speed (m/s).

Practical Use Cases

1. **Indoor Flight Visualization:**
    - *Scenario:* A drone is flying in a warehouse using Optical Flow and has no GPS.

    - *Action:* The GCS uses `LOCAL_POSITION_NED` to draw the drone's path on a blank grid, as Latitude/Longitude are either unavailable or inaccurate.
2. **Swarm Coordination:**
    - *Scenario:* Multiple drones are performing a light show.
    - *Action:* An orchestrating computer monitors the NED coordinates of all drones to ensure they maintaining the correct relative spacing in their "stage" coordinate system.
3. **Visual Odometry Debugging:**
    - *Scenario:* A developer is integrating a Realsense T265 camera for external positioning.
    - *Action:* The developer compares the vehicle's reported `LOCAL_POSITION_NED` (the EKF's fused result) against the raw camera output to tune the EKF's trust in the external sensor.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2979**: Implementation of `send_local_position`.
- **libraries/AP_AHRS/AP_AHRS.h**: Provides the relative position and velocity data.

GLOBAL_POSITION_INT (ID 33)

SUPPORTED

Summary

The `GLOBAL_POSITION_INT` message is the primary source of absolute geographic positioning data in MAVLink. It provides the vehicle's Latitude, Longitude, and Altitude (both AMSL and relative to Home) using integer representation for high efficiency and precision. It is the core message used by Ground Control Stations for map plotting and altitude display.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Primary position telemetry)
- **RX (Receive):** All Vehicles (Used for "Follow Me" and Multi-Vehicle Avoidance)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_global_position_int` within libraries/GCS_MAVLink/GCS_Common.cpp:5872.

Data Sourcing

- **Coordinate Source:** Data is fused by the EKF and provided by the `AP_AHRS` library via `[ahrs](/field-manual/mavlink-interface/ahrs.html).get_location()`.
- **Format:**
  - `lat`, `lon`: Sourced in $degrees \times 10^7$.
  - `alt`: Altitude Above Mean Sea Level (AMSL) in millimeters.
  - `relative_alt`: Altitude relative to the Home position in millimeters.
  - `heading`: Vehicle yaw sourced from `ahrs.yaw_sensor` in centidegrees (0 to 35999).
  - `vx`, `vy`, `vz`: Ground speed components in cm/s.

Scheduling

- Sent as part of the `MSG_GLOBAL_POSITION_INT` stream.
- Triggered in `GCS_Common.cpp:6102` within the `try_send_message` loop.

Reception (RX)

ArduPilot handles this message in various specialized libraries, most notably `AP_Follow` and `AP_Avoidance`.

- **Follow Me:** In libraries/AP_Follow/AP_Follow.cpp, ArduPilot receives `GLOBAL_POSITION_INT` from a "Lead" vehicle or a GCS to track and follow its position in real-time.
- **ADSB/Avoidance:** Used to track the positions of other MAVLink-enabled vehicles in the vicinity to trigger collision avoidance maneuvers.

Data Fields

- **`time_boot_ms`:** Timestamp (milliseconds since system boot).
- **`lat`:** Latitude, expressed as degrees * 1E7.

- `lon`: Longitude, expressed as degrees * 1E7.
- `alt`: Altitude (MSL). Note that virtually all GPS modules provide both WGS84 and MSL.
- `relative_alt`: Altitude above the home position.
- `vx`: Ground X Speed (Latitude, positive north).
- `vy`: Ground Y Speed (Longitude, positive east).
- `vz`: Ground Z Speed (Altitude, positive down).
- `hdg`: Vehicle heading (max 65535, valid 0-35999).

Practical Use Cases

1. **GCS Map Display:**
   - *Scenario:* A pilot is flying a 10km long-range mission.
   - *Action:* The GCS uses `lat` and `lon` to update the vehicle's position on the map at high frequency (typically 5Hz - 10Hz).
2. **Terrain Following Verification:**
   - *Scenario:* A drone is flying 10m above a sloping hill.
   - *Action:* The pilot monitors `relative_alt` to ensure the vehicle is maintaining the correct height relative to the take-off point, regardless of the absolute AMSL altitude.
3. **Lead-Follow Swarming:**
   - *Scenario:* A "Follower" drone is slaved to a "Leader" drone.
   - *Action:* The Follower receives the Leader's `GLOBAL_POSITION_INT` and calculates the required offset to maintain its position in the formation.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5872**: TX implementation.
- **libraries/AP_Follow/AP_Follow.cpp**: RX implementation for vehicle following.
- **libraries/AP_AHRS/AP_AHRS.h:586**: Source of `yaw_sensor` data.

VFR_HUD (ID 74)

SUPPORTED

Summary

The `VFR_HUD` (Visual Flight Rules Head-Up Display) message is a bandwidth-optimized packet designed specifically to drive the dashboard instruments of a Ground Control Station. It aggregates the most critical flight metrics—speed, altitude, heading, throttle, and vertical speed—into a single 20-byte payload, reducing the need for the GCS to parse multiple high-frequency streams like `ATTITUDE` and `GLOBAL_POSITION_INT`.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Summary telemetry)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_vfr_hud` within libraries/GCS_MAVLink/GCS_Common.cpp:3390.

Data Sourcing

ArduPilot uses vehicle-specific virtual functions to populate the fields:

- `airspeed`: Sourced from the `AP_Airspeed` library. If no airspeed sensor is present, ArduPilot may fall back to an EKF wind-speed estimate.
- `groundspeed`: Provided by `AP::[ahrs](/field-manual/mavlink-interface/ahrs.html)().groundspeed()` in m/s.
- `heading`: Provided by `AP::ahrs().yaw_sensor` in degrees (0-360).
- `throttle`: Represented as a percentage (0-100). For Copters, this is derived from the motor mixer's average output.
- `alt`: Current altitude Above Mean Sea Level (AMSL) in meters.
- `climb`: Vertical velocity (climb rate) in m/s.

Bandwidth Optimization

`VFR_HUD` is the "Swiss Army Knife" of telemetry. By streaming this single message at 5Hz-10Hz, a GCS can maintain a fully functional HUD even on very low-bandwidth links (like long-range 915MHz radios) where sending full position and attitude packets would cause lag.

Data Fields

- `airspeed`: Current airspeed in m/s.
- `groundspeed`: Current ground speed in m/s.
- `heading`: Current heading in degrees, in compass units (0..360, 0=north).
- `throttle`: Current throttle setting in integer percent, 0 to 100.
- `alt`: Current altitude (MSL), in meters.
- `climb`: Current climb rate in meters/second.

Practical Use Cases

1. **Dashboard Telemetry:**
   - *Scenario:* A pilot is flying a long-range FPV mission.
   - *Action:* The GCS HUD uses `VFR_HUD` as the primary source for the Speed Tape, Altimeter Tape, and Compass Rose.
2. **Stall Prevention:**
   - *Scenario:* A fixed-wing plane is performing an autonomous climb.
   - *Action:* The pilot monitors the `airspeed` field in the HUD. If it drops toward the "Stall Speed" (calculated GCS-side), the pilot can take manual control or adjust the throttle.
3. **Variometer Feedback:**
   - *Scenario:* A glider pilot is looking for thermals.
   - *Action:* The `climb` rate field is used to drive an audio variometer (beeping) on the ground station, helping the pilot identify rising air.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3390**: Common implementation.
- **ArduPlane/GCS_Mavlink.cpp:277**: Plane-specific airspeed and throttle logic.
- **ArduCopter/GCS_Mavlink.cpp:246**: Copter-specific throttle logic.

BATTERY_STATUS (ID 147)

SUPPORTED (CRITICAL)

Summary

The BATTERY_STATUS message provides a comprehensive report on the vehicle's power source health. Unlike the simpler SYS_STATUS (which only reports total voltage and current), BATTERY_STATUS supports multiple battery instances and provides detailed information including individual cell voltages, temperature, and remaining capacity. This is the primary message used by Ground Control Stations for advanced battery telemetry and "Smart Battery" integration.

Status

Supported (Critical)

Directionality

- **TX (Transmit):** All Vehicles (Broadcasts battery state)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is centered in `GCS_MAVLINK::send_battery_status` within libraries/GCS_MAVLink/GCS_Common.cpp:266.

Data Sourcing

- **Multiple Instances:** ArduPilot iterates through all active battery monitors configured via the `BATT_MONITOR` parameters. It sends a separate `BATTERY_STATUS` packet for each, using the `id` field to distinguish them.
- **Cell Voltages:**
    - **Standard (1-10):** The message reports up to 10 cell voltages in the `voltages` array (in millivolts).
    - **Extended (11-14):** ArduPilot uses MAVLink extensions to report cells 11 through 14 in the `voltages_ext` field.
- **Temperature:** If the battery monitor backend supports it (e.g., SMBus or DroneCAN), the internal battery temperature is reported in centidegrees Celsius.
- **Current and Capacity:** Sourced from the `AP_BattMonitor` library, providing `current_consumed` (mAh) and `battery_remaining` (percentage).

Data Fields

- **`id`**: Battery ID.
- **`battery_function`**: Function of the battery.
- **`type`**: Type (chemistry) of the battery.
- **`temperature`**: Temperature in centi-degrees Celsius.
- **`voltages`**: Battery voltage of cells, in millivolts (1 = 1 millivolt).
- **`current_battery`**: Battery current, in 10*milliamperes (1 = 10 milliampere), -1: autopilot does not measure the current.
- **`current_consumed`**: Consumed charge, in milliampere hours (1 = 1 mAh), -1: autopilot does not provide consumption estimate.
- **`energy_consumed`**: Consumed energy, in 100*Joules (interrim test).
- **`battery_remaining`**: Remaining battery energy. (0\%: 0, 100\%: 100), -1: autopilot does not estimate the remaining battery.

- `time_remaining`: Remaining battery time, 0: autopilot does not provide remaining battery time estimate.
- `charge_state`: State for additional protection, see MAV_BATTERY_CHARGE_STATE.
- `voltages_ext`: Battery voltages for cells 11-14.
- `mode`: Battery mode.
- `fault_bitmask`: Fault/health indications.

Practical Use Cases

1. **Individual Cell Health Monitoring:**
    - *Scenario:* A high-value octocopter is using a 12S LiPo battery.
    - *Action:* The GCS monitors the `voltages` array. If one cell drops significantly below the others (e.g., 3.2V vs 3.7V), the GCS triggers a "Battery Imbalance" warning, allowing the pilot to land before the cell fails.
2. **Smart Battery Integration:**
    - *Scenario:* A drone is equipped with a Tattu or Grepow Smart Battery that communicates via SMBus.
    - *Action:* ArduPilot reads the battery's internal cycle count and temperature and relays this via `BATTERY_STATUS`. The GCS displays this information, helping the operator track the long-term health of their battery fleet.
3. **Dual Battery Redundancy:**
    - *Scenario:* A drone uses two parallel batteries for redundancy.
    - *Action:* The GCS displays two separate battery widgets (ID 0 and ID 1). If one battery starts drawing significantly more current than the other, it indicates a connector issue or an aging pack.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:266**: Implementation of the MAVLink packet construction.
- **libraries/AP_BattMonitor/AP_BattMonitor.cpp**: Primary source for all battery-related data.

AHRS (ID 163) SUPPORTED

Summary

The `AHRS` message reports the internal state of the Attitude Heading Reference System (AHRS), specifically focusing on gyro drift estimates and attitude error metrics. It is useful for monitoring the health and convergence of the EKF or DCM estimator.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports AHRS health)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission is handled by `GCS_MAVLINK::send_ahrs` within libraries/GCS_MAVLink/GCS_Common.cpp:2424.

Data Fields

- `omegaIx`: X gyro drift estimate (rad/s).
- `omegaIy`: Y gyro drift estimate (rad/s).
- `omegaIz`: Z gyro drift estimate (rad/s).
- `accel_weight`: Average accel_weight (0 to 1). *Currently sent as 0.*
- `renorm_val`: Average renormalisation value (0 to 1). *Currently sent as 0.*
- `error_rp`: Average error_roll_pitch value (0 to 1).
- `error_yaw`: Average error_yaw value (0 to 1).

Practical Use Cases

1. **Vibration Diagnosis:**
   - *Scenario:* A user experiences "EKS FAIL" warnings.
   - *Action:* Analyzing `omegaIx/y/z` in the logs can reveal if the gyro bias estimates are fluctuating wildly, often a sign of excessive vibration or a failing IMU.
2. **Magnetic Interference:**
   - *Scenario:* A rover's heading drifts during a turn.
   - *Action:* `error_yaw` increasing indicates the compass reading disagrees with the gyro integration, potentially due to magnetic interference from motors.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2424**: Implementation of the sender.

EFI_STATUS (ID 225)

SUPPORTED (RX & TX)

Summary

Status of an Electronic Fuel Injection (EFI) engine.

Status

Supported (RX & TX)

Directionality

- **TX (Transmit):** All Vehicles - Forwards EFI status to GCS.
- **RX (Receive):** All Vehicles - Receives status from EFI hardware (e.g., via CAN or Serial).

Transmission (TX)

ArduPilot streams this message to the GCS to report engine health.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Reception (RX)

Handled by `AP_EFI::handle_EFI_message`. Allows a serial-connected EFI unit to inject its status into the autopilot.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Data Fields

- `health`: Health flags.
- `ecu_index`: ECU instance.
- `rpm`: Engine RPM.
- `fuel_consumed`: Fuel consumed (cm^3).
- `fuel_flow`: Fuel flow rate (cm^3/min).
- `engine_load`: Engine load percentage.
- `throttle_position`: Throttle position percentage.
- `spark_dwell_time`: Spark dwell time.
- `barometric_pressure`: Barometric pressure (kPa).
- `intake_manifold_pressure`: Intake pressure (kPa).
- `intake_air_temperature`: Intake temp (degC).
- `cylinder_head_temperature`: CHT (degC).
- `ignition_timing`: Ignition timing (deg).
- `injection_time`: Injection time.
- `exhaust_gas_temperature`: EGT (degC).
- `throttle_out`: Throttle output percentage.
- `pt_compensation`: Pressure/Temp compensation.

Practical Use Cases

1. **Gas Engine Monitoring:**
    - *Scenario:* A large gasoline-powered VTOL plane.

    - *Action:* The pilot monitors `cylinder_head_temperature` and `rpm` to ensure the engine is not overheating during hover.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Handler and Streamer.

HOME_POSITION (ID 242)

SUPPORTED (CRITICAL)

Summary

The `HOME_POSITION` message defines the vehicle's reference point for return-to-launch (RTL) maneuvers and distance-from-home calculations. It provides the global coordinates (Latitude, Longitude, Altitude) of the take-off location or a user-defined mission origin. This message is critical for GCS situational awareness and for ensuring the drone has a valid recovery point before embarking on an autonomous mission.

Status

Supported (Critical)

Directionality

- **TX (Transmit):** All Vehicles (Reports the current home location)
- **RX (Receive):** None (Use `SET_HOME_POSITION` (243) to override the home location)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_home_position` within libraries/GCS_MAVLink/GCS_Common.cpp:3032.

Data Sourcing

- **AHRS Home:** Data is retrieved from the `AP_AHRS` library via `get_home()`.
- **Coordinate Format:**
  - **Latitude/Longitude:** Sourced in $degrees \times 10^7$.
  - **Altitude:** Sourced as MSL altitude in millimeters.
  - **Relative Position:** Includes the X, Y, and Z distance (in meters) from the local EKF origin to the Home position.

Trigger Logic

ArduPilot sends this message:

1. **Periodically:** As part of the `MSG_HOME` telemetry stream (typically at a low rate like 0.5Hz or 1Hz).
2. **On Demand:** In response to a `MAV_CMD_GET_HOME_POSITION` command.
3. **On Set:** Automatically whenever the Home position is initialized or updated (e.g., at the moment of arming).

Data Fields

- `latitude` : Latitude (WGS84), in degrees * 1E7.
- `longitude` : Longitude (WGS84, in degrees * 1E7.
- `altitude` : Altitude (MSL), in meters * 1000 (positive for up).
- `x` : Local X position of this position in the local coordinate frame.
- `y` : Local Y position of this position in the local coordinate frame.
- `z` : Local Z position of this position in the local coordinate frame.
- `q` : World to surface normal and heading transformation of the takeoff position. Used to indicate the heading and slope of the ground.
- `approach_x` : Local X position of the end of the approach vector. Multicopters should set this position based on their takeoff path. Grass-landing fixed wing aircraft should set it the same way as

multicopters. Runway-landing fixed wing aircraft should set it to the opposite direction of the takeoff, assuming the takeoff happened from the threshold / touchdown zone.

- `approach_y`: Local Y position of the end of the approach vector. Multicopters should set this position based on their takeoff path. Grass-landing fixed wing aircraft should set it the same way as multicopters. Runway-landing fixed wing aircraft should set it to the opposite direction of the takeoff, assuming the takeoff happened from the threshold / touchdown zone.
- `approach_z`: Local Z position of the end of the approach vector. Multicopters should set this position based on their takeoff path. Grass-landing fixed wing aircraft should set it the same way as multicopters. Runway-landing fixed wing aircraft should set it to the opposite direction of the takeoff, assuming the takeoff happened from the threshold / touchdown zone.
- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).

Practical Use Cases

1. **RTL Safety Check:**
    - *Scenario:* A pilot is about to fly a long-range mission.
    - *Action:* The GCS checks for the presence of the `HOME_POSITION` message. If not received, the GCS shows a "Home Not Set" warning and may prevent the pilot from arming, ensuring the drone won't attempt to return to an unknown location.
2. **Distance-to-Home Display:**
    - *Scenario:* A drone is 2km away from the pilot.
    - *Action:* The GCS HUD calculates the distance between the vehicle's current `GLOBAL_POSITION_INT` (33) and the `HOME_POSITION` (242) to show a live "Distance" readout to the pilot.
3. **Mission Origin Alignment:**
    - *Scenario:* A surveyor is using a pre-planned mission that relies on relative altitudes.
    - *Action:* The GCS uses the `HOME_POSITION` altitude as the $0m$ reference point, ensuring the mission's vertical steps are executed at the correct heights above the ground.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3032**: Implementation of the MAVLink packet construction.
- **libraries/AP_AHRS/AP_AHRS.h**: Provides the `get_home()` interface.

AIS_VESSEL (ID 301)

SUPPORTED

Summary

The `AIS_VESSEL` message reports the position, velocity, and identification of a marine vessel detected by an onboard AIS (Automatic Identification System) receiver. This allows the Autopilot and Ground Control Station to track maritime traffic and, in some cases, perform collision avoidance.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports detected vessels to GCS)
- **RX (Receive):** None (Autopilot consumes raw NMEA from AIS hardware, not MAVLink)

Transmission (TX)

The message is generated by the `AP_AIS` library, which parses incoming data from an AIS receiver connected to a serial port.

Core Logic

The implementation is in `AP_AIS::send` within libraries/AP_AIS/AP_AIS.cpp:264.

1. **Database:** The library maintains a list of detected vessels (`_list`).
2. **Scheduling:** It iterates through the list and transmits a message for a vessel if:
   - The data has updated since the last transmission.
   - 30 seconds have elapsed (periodic refresh).
3. **TSLC:** It calculates `tslc` (Time Since Last Communication) to let the GCS know how stale the data is.

Data Fields

- **`MMSI`:** Mobile Marine Service Identity.
- **`lat`:** Latitude (deg * 1E7).
- **`lon`:** Longitude (deg * 1E7).
- **`COG`:** Course over ground (deg * 100).
- **`heading`:** True heading (deg * 100).
- **`SOG`:** Speed over ground (cm/s).
- **`callsign`:** The vessel callsign.
- **`name`:** The vessel name.
- **`width`:** Width of the vessel (meters).
- **`length`:** Length of the vessel (meters).
- **`type`:** Type of vessel.
- **`dimension_bow`:** Distance from GPS to bow (meters).
- **`dimension_stern`:** Distance from GPS to stern (meters).
- **`dimension_port`:** Distance from GPS to port (meters).
- **`dimension_starboard`:** Distance from GPS to starboard (meters).
- **`tslc`:** Time since last communication (seconds).
- **`flags`:** Bitmask to indicate various statuses including valid data fields.
- **`rot`:** Rate of turn (deg/min).

Practical Use Cases

1. **Maritime Patrol:**
   - *Scenario:* A drone is inspecting a harbor.
   - *Action:* The GCS displays all ships on the map with their names and vectors (SOG/COG), allowing the pilot to identify vessels without visual confirmation.
2. **Collision Avoidance:**
   - *Scenario:* An autonomous boat is navigating a channel.
   - *Action:* The Autopilot uses the `lat/lon` and `SOG` of surrounding AIS targets to calculate Time to Closest Point of Approach (TCPA) and steer clear of moving ships.

Key Codebase Locations

- **libraries/AP_AIS/AP_AIS.cpp:264**: Implementation of the sender.

GENERATOR_STATUS (ID 373)

SUPPORTED

Summary

The `GENERATOR_STATUS` message provides telemetry for an onboard electrical generator (e.g., RichenPower hybrid generator). It reports the operational state (Idle, Generating, Fault), electrical output, and maintenance metrics.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports generator state to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_Generator` library. Specific backends (like `AP_Generator_RichenPower`) implement the logic to populate this message from the generator's serial telemetry.

Core Logic

The implementation example is in `AP_Generator_RichenPower::send_generator_status` within libraries/AP_Generator/AP_Generator_RichenPower.cpp:438.

It maps internal generator states (Idle, Run, Charge) to the standard `MAV_GENERATOR_STATUS_FLAG` bitmask.

Data Fields

- **`status`**: Status flags.
- **`generator_speed`**: Speed of electrical generator or alternator.
- **`battery_current`**: Current into/out of battery.
- **`load_current`**: Current going to the UAV.
- **`power_generated`**: The power being generated.
- **`bus_voltage`**: Voltage of the bus seen at the generator, or battery voltage if battery is active.
- **`rectifier_temperature`**: The temperature of the rectifier or other inverter.
- **`generator_temperature`**: The temperature of the mechanical motor, fuel cell core or generator.
- **`bat_current_setpoint`**: The target battery current.
- **`runtime`**: Seconds this generator has run since it was rebooted.
- **`time_until_maintenance`**: Seconds until this generator requires maintenance. A negative value indicates maintenance is past-due.

Practical Use Cases

1. **Hybrid Drones:**
   - *Scenario:* A petrol-electric hybrid multirotor.
   - *Action:* The GCS displays the `generator_speed` and `bus_voltage`. If the generator stalls (`status` becomes `OFF` while in air), the GCS triggers a critical alarm, warning the pilot they are running on reserve battery power only.
2. **Maintenance Tracking:**

    - *Scenario:* Fleet management.
    - *Action:* The `time_until_maintenance` field allows ground crews to schedule oil changes or engine service proactively.

Key Codebase Locations

- **libraries/AP_Generator/AP_Generator_RichenPower.cpp:438**: Implementation of the sender.

RELAY_STATUS (ID 376)

SUPPORTED

Summary

The `RELAY_STATUS` message reports the current state (On/Off) of the vehicle's relay pins. Relays are digital switches often used to control lights, camera triggers, or power to peripherals.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports relay states to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_Relay` library. It is typically sent at low frequency or upon change.

Core Logic

The implementation is in `AP_Relay::send_relay_status` within libraries/AP_Relay/AP_Relay.cpp:655.

It supports reporting up to 16 relays using bitmasks.

Data Fields

- **`time_boot_ms`:** Timestamp (milliseconds since system boot).
- **`on`:** Relay state. 1 bit per relay. 0: off, 1: on.
- **`present`:** Relay present. 1 bit per relay. 0: not configured, 1: configured.

Practical Use Cases

1. **Remote Switch Verification:**
   - *Scenario:* A user toggles a switch on their RC transmitter to turn on landing lights (connected to Relay 1).
   - *Action:* The GCS receives `RELAY_STATUS`. It sees Bit 0 of `on` go high and updates the UI indicator to show the lights are ON.

Key Codebase Locations

- **libraries/AP_Relay/AP_Relay.cpp:655**: Implementation of the sender.

AOA_SSA (ID 11020)

SUPPORTED (PLANE ONLY)

Summary

The `AOA_SSA` message reports the vehicle's Angle of Attack (AoA) and Side Slip Angle (SSA). This is critical for fixed-wing aircraft to monitor aerodynamic performance and prevent stalls.

Status

Supported (Plane Only)

Directionality

- **TX (Transmit):** Autopilot (Reports AoA/SSA to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the **ArduPlane** firmware's `GCS_Mavlink` module.

Core Logic

The implementation is in `GCS_MAVLINK_Plane::send_aoa_ssa` within ArduPlane/GCS_Mavlink.cpp:190.

It pulls data directly from the AHRS:

- `ahrs.getAOA()`: Angle of Attack in degrees.
- `ahrs.getSSA()`: Side Slip Angle in degrees.

These values may be synthesized from inertial data and airspeed, or measured directly by an AoA sensor (like a vane).

Data Fields

- **`time_usec`**: Timestamp (us since UNIX epoch).
- **`AOA`**: Angle of Attack (degrees).
- **`SSA`**: Side Slip Angle (degrees).

Practical Use Cases

1. **Stall Warning:**
   - *Scenario:* A pilot is flying a glider near its limits.
   - *Action:* The GCS monitors `AOA`. If it exceeds the critical angle (e.g., 15 degrees), an audible alarm is triggered.
2. **Turn Coordination:**
   - *Scenario:* Tuning yaw dampers.
   - *Action:* Analyzing the `SSA` log to minimize sideslip during turns.

Key Codebase Locations

- **ArduPlane/GCS_Mavlink.cpp:190**: Implementation of the sender.

`ESC_TELEMETRY_1_TO_4` (ID 11030)

SUPPORTED

Summary

The `ESC_TELEMETRY_1_TO_4` message provides real-time telemetry data for the first four Electronic Speed Controllers (ESCs). This includes voltage, current, RPM, and temperature, which are critical for monitoring propulsion health.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports ESC status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_ESC_Telem` library. It aggregates data from various sources (BLHeli_S/32, DroneCAN, DShot) and standardizes it into MAVLink messages.

Core Logic

The implementation is in `AP_ESC_Telem::send_esc_telemetry_mavlink` within libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428.

It groups ESCs into blocks of 4. If `ESC_TELEMETRY_1_TO_4` is full, it moves to `ESC_TELEMETRY_5_TO_8` (11031), and so on, up to 32 ESCs.

Data Fields

- `temperature`: Array of 4 temperatures in degC.
- `voltage`: Array of 4 voltages in centi-volts (V * 100).
- `current`: Array of 4 currents in centi-amps (A * 100).
- `totalcurrent`: Array of 4 consumed energy values in mAh.
- `rpm`: Array of 4 RPM values.
- `count`: Array of 4 error counts (or packet counts depending on ESC type).

Practical Use Cases

1. **Propulsion Failure Detection:**
   - *Scenario:* A motor bearing seizes mid-flight.
   - *Action:* The current for that motor spikes while RPM drops. The GCS logs this anomaly, helping post-flight diagnostics.
2. **Battery Management:**
   - *Scenario:* Long-range flight.
   - *Action:* The GCS sums the `totalcurrent` (mAh) from all ESCs to cross-check the battery monitor's consumption estimate.

Key Codebase Locations

- **libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428**: Implementation of the sender.

ESC_TELEMETRY_5_TO_8 (ID 11031) SUPPORTED

Summary

The `ESC_TELEMETRY_5_TO_8` message provides real-time telemetry data for Electronic Speed Controllers (ESCs) 5 through 8. It follows the same structure and logic as `ESC_TELEMETRY_1_TO_4` (11030).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports ESC status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_ESC_Telem` library.

Core Logic

The implementation is in `AP_ESC_Telem::send_esc_telemetry_mavlink` within libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428.

It is populated if the system has more than 4 active ESCs (e.g., an Octocopter).

Data Fields

- `temperature`: Array of 4 temperatures in degC.
- `voltage`: Array of 4 voltages in centi-volts.
- `current`: Array of 4 currents in centi-amps.
- `totalcurrent`: Array of 4 consumed energy values in mAh.
- `rpm`: Array of 4 RPM values.
- `count`: Array of 4 error counts.

ESC_TELEMETRY_9_TO_12 (ID 11032) SUPPORTED

Summary

The `ESC_TELEMETRY_9_TO_12` message provides real-time telemetry data for Electronic Speed Controllers (ESCs) 9 through 12. It follows the same structure and logic as `ESC_TELEMETRY_1_TO_4` (11030).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports ESC status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_ESC_Telem` library.

Core Logic

The implementation is in `AP_ESC_Telem::send_esc_telemetry_mavlink` within libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428.

It is populated if the system has more than 8 active ESCs (e.g., a Dodecacopter or Dodeca-Hexa).

Data Fields

- **`temperature`:** Array of 4 temperatures in degC.
- **`voltage`:** Array of 4 voltages in centi-volts.
- **`current`:** Array of 4 currents in centi-amps.
- **`totalcurrent`:** Array of 4 consumed energy values in mAh.
- **`rpm`:** Array of 4 RPM values.
- **`count`:** Array of 4 error counts.

ESC_TELEMETRY_13_TO_16 (ID 11040)

SUPPORTED

Summary

The `ESC_TELEMETRY_13_TO_16` message provides real-time telemetry data for Electronic Speed Controllers (ESCs) 13 through 16. It follows the same structure and logic as `ESC_TELEMETRY_1_TO_4` (11030).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports ESC status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_ESC_Telem` library.

Core Logic

The implementation is in `AP_ESC_Telem::send_esc_telemetry_mavlink` within libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428.

It is populated if the system has more than 12 active ESCs.

Data Fields

- `temperature`**:** Array of 4 temperatures in degC.
- `voltage`**:** Array of 4 voltages in centi-volts.
- `current`**:** Array of 4 currents in centi-amps.
- `totalcurrent`**:** Array of 4 consumed energy values in mAh.
- `rpm`**:** Array of 4 RPM values.
- `count`**:** Array of 4 error counts.

ESC_TELEMETRY_17_TO_20 (ID 11041)

SUPPORTED

Summary

The `ESC_TELEMETRY_17_TO_20` message provides real-time telemetry data for Electronic Speed Controllers (ESCs) 17 through 20. It follows the same structure and logic as `ESC_TELEMETRY_1_TO_4` (11030).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports ESC status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_ESC_Telem` library.

Core Logic

The implementation is in `AP_ESC_Telem::send_esc_telemetry_mavlink` within libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428.

It is populated if the system has more than 16 active ESCs.

Data Fields

- **`temperature`**: Array of 4 temperatures in degC.
- **`voltage`**: Array of 4 voltages in centi-volts.
- **`current`**: Array of 4 currents in centi-amps.
- **`totalcurrent`**: Array of 4 consumed energy values in mAh.
- **`rpm`**: Array of 4 RPM values.
- **`count`**: Array of 4 error counts.

ESC_TELEMETRY_21_TO_24 (ID 11042)

SUPPORTED

Summary

The `ESC_TELEMETRY_21_TO_24` message provides real-time telemetry data for Electronic Speed Controllers (ESCs) 21 through 24. It follows the same structure and logic as `ESC_TELEMETRY_1_TO_4` (11030).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports ESC status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_ESC_Telem` library.

Core Logic

The implementation is in `AP_ESC_Telem::send_esc_telemetry_mavlink` within libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428.

It is populated if the system has more than 20 active ESCs.

Data Fields

- **`temperature`:** Array of 4 temperatures in degC.
- **`voltage`:** Array of 4 voltages in centi-volts.
- **`current`:** Array of 4 currents in centi-amps.
- **`totalcurrent`:** Array of 4 consumed energy values in mAh.
- **`rpm`:** Array of 4 RPM values.
- **`count`:** Array of 4 error counts.

ESC_TELEMETRY_25_TO_28 (ID 11043)

SUPPORTED

Summary

The `ESC_TELEMETRY_25_TO_28` message provides real-time telemetry data for Electronic Speed Controllers (ESCs) 25 through 28. It follows the same structure and logic as `ESC_TELEMETRY_1_TO_4` (11030).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports ESC status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_ESC_Telem` library.

Core Logic

The implementation is in `AP_ESC_Telem::send_esc_telemetry_mavlink` within libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428.

It is populated if the system has more than 24 active ESCs.

Data Fields

- `temperature`: Array of 4 temperatures in degC.
- `voltage`: Array of 4 voltages in centi-volts.
- `current`: Array of 4 currents in centi-amps.
- `totalcurrent`: Array of 4 consumed energy values in mAh.
- `rpm`: Array of 4 RPM values.
- `count`: Array of 4 error counts.

ESC_TELEMETRY_29_TO_32 (ID 11044)

SUPPORTED

Summary

The `ESC_TELEMETRY_29_TO_32` message provides real-time telemetry data for Electronic Speed Controllers (ESCs) 29 through 32. It follows the same structure and logic as `ESC_TELEMETRY_1_TO_4` (11030).

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports ESC status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_ESC_Telem` library.

Core Logic

The implementation is in `AP_ESC_Telem::send_esc_telemetry_mavlink` within libraries/AP_ESC_Telem/AP_ESC_Telem.cpp:428.

It is populated if the system has more than 28 active ESCs (e.g., massive Drone Light Show swarms or complex VTOLs).

Data Fields

- `temperature`: Array of 4 temperatures in degC.
- `voltage`: Array of 4 voltages in centi-volts.
- `current`: Array of 4 currents in centi-amps.
- `totalcurrent`: Array of 4 consumed energy values in mAh.
- `rpm`: Array of 4 RPM values.
- `count`: Array of 4 error counts.

SENSORS

GPS_RAW_INT (ID 24)

SUPPORTED

Summary

The `GPS_RAW_INT` message provides raw satellite positioning data from the vehicle's primary GPS receiver. It includes latitude, longitude, altitude (MSL), and velocity components, along with quality indicators like the number of satellites visible and horizontal dilution of precision (HDOP).

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Streams GPS 0 data)
- **RX (Receive):** None (Use `GPS_INPUT` or `HIL_GPS` for external input)

Transmission (TX)

The primary transmission logic is in `AP_GPS::send_mavlink_gps_raw` within libraries/AP_GPS/AP_GPS.cpp:1370.

Data Sourcing

- **Primary Instance:** This message specifically sends data from the first GPS instance (`instance 0`). For second GPS units, ArduPilot uses the ArduPilot-specific message `GPS2_RAW` (ID 124).
- **Coordinate Format:** Latitude and Longitude are sent as `int32_t` with a scale of 1E7.
- **Altitude:** Sourced as Altitude above Mean Sea Level (MSL) in millimeters.
- **Velocity:** Sourced from the GPS driver's velocity vector, converted to cm/s for ground speed and centidegrees for course over ground.

Scheduling

- Sent as part of the `MSG_GPS_RAW` stream.
- Triggered in `GCS_Common.cpp:6204` within the `try_send_message` loop.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `fix_type`: GPS fix type (`GPS_FIX_TYPE`).
- `lat`: Latitude (WGS84, EGM96 ellipsoid), in degrees * 1E7.
- `lon`: Longitude (WGS84, EGM96 ellipsoid), in degrees * 1E7.
- `alt`: Altitude (MSL). Positive for up.
- `eph`: GPS HDOP horizontal dilution of precision in cm (m*100). If unknown, set to: UINT16_MAX.
- `epv`: GPS VDOP vertical dilution of precision in cm (m*100). If unknown, set to: UINT16_MAX.
- `vel`: GPS ground speed (m/s * 100). If unknown, set to: UINT16_MAX.
- `cog`: Course over ground (NOT heading, but direction of movement) in degrees * 100, 0.0..359.99 degrees. If unknown, set to: UINT16_MAX.
- `satellites_visible`: Number of satellites visible. If unknown, set to 255.

Practical Use Cases

1. **Map Plotting:**

    - *Scenario:* A tablet running QGroundControl needs to show the drone's icon on a satellite map.
    - *Action:* QGC reads `lat` and `lon` from `GPS_RAW_INT` to position the icon.
2. **Signal Quality Monitoring:**
    - *Scenario:* A pilot is flying near tall buildings and wants to ensure the GPS link is stable.
    - *Action:* The HUD monitors `satellites_visible` and `eph` (HDOP). If satellites drop below 6 or HDOP rises above 200 (2.0), the GCS warns the pilot of "Poor GPS Health".
3. **Antenna Tracking:**
    - *Scenario:* A long-range antenna tracker needs to point a directional antenna at the drone.
    - *Action:* The tracker calculates the heading from its own position to the `lat`/`lon` reported by the vehicle.

Key Codebase Locations

- **libraries/AP_GPS/AP_GPS.cpp:1370**: Implementation of the MAVLink packet construction.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6204**: Scheduling and stream control.

SCALED_IMU (ID 26)

SUPPORTED

Summary

The `SCALED_IMU` message provides high-frequency acceleration and rotation data from the vehicle's primary inertial sensors. Unlike legacy MAVLink implementations that distinguish between "Raw" (ADC values) and "Scaled" (physics units), ArduPilot's `RAW_IMU` and `SCALED_IMU` both provide physics-ready values (mG and rad/s). The primary distinction in ArduPilot is the **timestamp format** and **instance mapping**.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Outgoing telemetry)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_scaled_imu` within libraries/GCS_MAVLink/GCS_Common.cpp:2259.

Data Sourcing

- **Primary Instance:** `SCALED_IMU` (ID 26) transmits data specifically from **IMU 0**.
- **Time Source:** Uses a millisecond timestamp (`AP_HAL::millis()`).
- **Scaling:**
  - **Accelerometer:** Scaled to milli-G (mG).
  - **Gyroscope:** Scaled to millirad/s (rad/s * 1000).
  - **Magnetometer:** Scaled to milli-Gauss (mGauss).

Stream Configuration

In ArduPilot's default `RAW_SENSORS` stream (typically `SRx_RAW_SENS`), `RAW_IMU` (27) is used for IMU 0 to take advantage of microsecond precision. Consequently, `SCALED_IMU` (26) is often redundant for the primary sensor and may not be active in default configurations. However, `SCALED_IMU2` (115) and `SCALED_IMU3` (129) are the standard way secondary and tertiary IMUs are reported.

Data Fields

- **`time_boot_ms`:** Timestamp (milliseconds since system boot).
- **`xacc`:** X acceleration (mg).
- **`yacc`:** Y acceleration (mg).
- **`zacc`:** Z acceleration (mg).
- **`xgyro`:** Angular speed around X axis (millirad /sec).
- **`ygyro`:** Angular speed around Y axis (millirad /sec).
- **`zgyro`:** Angular speed around Z axis (millirad /sec).
- **`xmag`:** X Magnetic field (milli tesla).
- **`ymag`:** Y Magnetic field (milli tesla).
- **`zmag`:** Z Magnetic field (milli tesla).

Practical Use Cases

1. **Vibration Analysis:**
   - *Scenario:* A builder is worried about motor balance and wants to check "Vibe" levels in real-time.
   - *Action:* The GCS graphs the `xacc`, `yacc`, and `zacc` fields at high frequency (e.g., 50Hz) to visualize mechanical noise.
2. **Orientation Verification:**
   - *Scenario:* A user has mounted the flight controller sideways.
   - *Action:* By observing rotation rates (`xgyro`, `ygyro`, `zgyro`) while physically rotating the drone, the user can verify if the `BOARD_ORIENTATION` parameter is set correctly.
3. **Peripheral Monitoring:**
   - *Scenario:* Monitoring the health of a redundant IMU system.
   - *Action:* Comparing the outputs of `SCALED_IMU` (IMU 0) against `SCALED_IMU2` (IMU 1) to check for sensor drift or hardware failure.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2259**: Implementation of `send_scaled_imu`.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6305**: The message scheduler case.

RAW_IMU (ID 27)

SUPPORTED

Summary

The `RAW_IMU` message is the high-precision companion to `SCALED_IMU`. In ArduPilot, both messages provide physics-scaled values (mG and rad/s), but `RAW_IMU` leverages a 64-bit microsecond timestamp (`time_usec`), making it the preferred message for high-frequency telemetry used in vibration analysis and IMU debugging.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Outgoing telemetry)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_raw_imu` within libraries/GCS_MAVLink/GCS_Common.cpp:2150.

Data Sourcing

- **Primary Instance:** This message specifically transmits data from **IMU 0**.
- **Timestamp:** Uses `AP_HAL::micros64()`, providing the best possible temporal resolution for the data.
- **Scaling:**
  - **Accelerometer:** Scaled to milli-G (mG).
  - **Gyroscope:** Scaled to millirad/s (rad/s * 1000).
  - **Magnetometer:** Scaled to milli-Gauss (mGauss).
- **Temperature:** Includes the IMU's internal temperature sensor data in centidegrees Celsius.

Scheduling

- Sent as part of the `MSG_RAW_IMU` stream.
- This message is usually prioritized in the "Raw Sensors" stream configuration to ensure ground control stations have high-fidelity sensor logs.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `xacc`: X acceleration (raw).
- `yacc`: Y acceleration (raw).
- `zacc`: Z acceleration (raw).
- `xgyro`: Angular speed around X axis (raw).
- `ygyro`: Angular speed around Y axis (raw).
- `zgyro`: Angular speed around Z axis (raw).
- `xmag`: X Magnetic field (raw).
- `ymag`: Y Magnetic field (raw).
- `zmag`: Z Magnetic field (raw).
- `id`: Id. Optional, default: 0.

- `temperature`: Temperature (centidegrees). Optional, default: 0.

Practical Use Cases

1. **FFT Tuning:**
   - *Scenario:* A pilot is setting up a Notch Filter to eliminate motor noise.
   - *Action:* The GCS captures a burst of `RAW_IMU` data and performs a Fast Fourier Transform (FFT) to identify the resonant frequencies of the frame. The microsecond timestamps are critical for accurate frequency mapping.
2. **IMU Health Check:**
   - *Scenario:* During pre-flight, the GCS detects "IMU Mismatch".
   - *Action:* The operator compares `RAW_IMU` (IMU 0) against other IMU messages to see if a particular sensor is producing erratic or noisy values while stationary.
3. **Blackbox Logging (Remote):**
   - *Scenario:* A developer is testing a vehicle too small for a high-speed SD card.
   - *Action:* The telemetry link is saturated with `RAW_IMU` packets, allowing the developer to reconstruct the flight dynamics on a remote ground station with high precision.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2150**: Implementation of `send_raw_imu`.
- **libraries/GCS_MAVLink/GCS.h:360**: Declaration of the send function.

SCALED_PRESSURE (ID 29) SUPPORTED

Summary

The `SCALED_PRESSURE` message provides calibrated environmental data, specifically absolute pressure (for altitude) and differential pressure (for airspeed). This message is the primary source for the Altimeter and Airspeed indicators in a Ground Control Station.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports altitude/airspeed data)
- **RX (Receive):** Antenna Tracker (Syncs altitude for tracking)

Transmission (TX)

The transmission logic is implemented in `GCS_MAVLINK::send_scaled_pressure` within libraries/GCS_MAVLink/GCS_Common.cpp:2354.

Data Sourcing

- **Absolute Pressure (`press_abs`):** Sourced from the primary barometer (Index 0) via `AP_Baro`. Values are in Hectopascals (hPa).
- **Differential Pressure (`press_diff`):** Sourced from the primary airspeed sensor (Index 0) via `AP_Airspeed`. Values are in hPa.
- **Temperature:** Includes the barometer's ambient temperature reading in centidegrees Celsius.

Reception (RX)

While most vehicles only send this data, the **Antenna Tracker** firmware receives it.

- **Handler:** `Tracker::tracking_update_pressure` in AntennaTracker/tracking.cpp.
- **Purpose:** The tracker compares its local pressure against the vehicle's pressure to calculate a high-precision relative altitude, which is critical for aiming the directional antenna accurately at the drone.

Data Fields

- `time_boot_ms`: Timestamp (milliseconds since system boot).
- `press_abs`: Absolute pressure (hectopascal).
- `press_diff`: Differential pressure 1 (hectopascal).
- `temperature`: Temperature (centidegrees Celsius).

Practical Use Cases

1. **HUD Altimeter:**
   - *Scenario:* A pilot is flying in a mountainous area.
   - *Action:* The GCS uses `press_abs` to calculate the vehicle's barometric altitude, providing a stable height reference that doesn't suffer from GPS vertical jitter.
2. **Airspeed Verification:**
   - *Scenario:* A fixed-wing pilot is flying in high winds.

    - *Action:* The GCS monitors `press_diff`. If the reading stays near zero while flying fast, the GCS warns of a "Pitot Tube Blockage".
3. **Automatic Antenna Aiming:**
    - *Scenario:* A long-range mission requires a high-gain antenna.
    - *Action:* The Antenna Tracker uses received `SCALED_PRESSURE` packets to maintain a vertical lock on the vehicle's position.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2308**: Core logic for populating pressure data.
- **libraries/AP_Baro/AP_Baro.h**: Source of absolute pressure data.
- **libraries/AP_Airspeed/AP_Airspeed.h**: Source of differential pressure data.

OPTICAL_FLOW (ID 100)

SUPPORTED

Summary

The `OPTICAL_FLOW` message provides 2D velocity data based on the optical movement of the ground beneath the vehicle. It is a critical sensor message for GPS-denied navigation, allowing the flight controller to maintain a stable horizontal position (loiter) indoors or in deep urban canyons. ArduPilot can both receive this data from external MAVLink sensors (like the PX4Flow) and transmit it to Ground Control Stations for real-time visualization of sensor health.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports sensor data to GCS)
- **RX (Receive):** All Vehicles (Accepts data from external MAVLink flow sensors)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_optical_flow` in libraries/GCS_MAVLink/GCS_Common.cpp:4047.

Data Processing

1. **Backend Dispatch:** The message is passed to the `AP_OpticalFlow` library, which identifies the `MAV` backend.
2. **Accumulation:** In AP_OpticalFlow_MAV.cpp, the `flow_x`, `flow_y`, and `quality` values are integrated into a running sum.
3. **EKF Fusion:** During the next EKF (Extended Kalman Filter) update, these integrated values are converted into body-frame velocity estimates. The `quality` field is used to determine how much the EKF should "trust" the data (e.g., quality < 50 may be ignored).

Transmission (TX)

ArduPilot relays optical flow data to the GCS via the `MSG_OPTICAL_FLOW` stream.

- **Purpose:** This allows pilots to verify that the flow sensor is working correctly (e.g., seeing the "Flow" graph move when they physically tilt the drone) without needing to connect directly to the sensor hardware.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `sensor_id`: Sensor ID.
- `flow_x`: Flow in x-sensor direction (deci-pixels).
- `flow_y`: Flow in y-sensor direction (deci-pixels).
- `flow_comp_m_x`: Flow in x-sensor direction, angular-speed compensated (m).
- `flow_comp_m_y`: Flow in y-sensor direction, angular-speed compensated (m).
- `quality`: Optical flow quality / confidence. 0: bad, 255: maximum quality.
- `ground_distance`: Ground distance (m). Positive value: distance known. Negative value: Unknown distance.

- `flow_rate_x`: Flow rate about X axis (rad/s).
- `flow_rate_y`: Flow rate about Y axis (rad/s).

Practical Use Cases

1. **Stable Indoor Hover:**
   - *Scenario:* A pilot is flying a drone inside a high-ceiling warehouse where GPS is unavailable.
   - *Action:* The drone uses the `OPTICAL_FLOW` message from an onboard camera to counter drift, allowing the pilot to take their hands off the sticks while the drone holds position perfectly.
2. **Precision Landing:**
   - *Scenario:* A drone is landing on a small target with a specific visual pattern.
   - *Action:* As the drone nears the ground, the optical flow sensor provides high-resolution horizontal velocity data that is more accurate than GPS, allowing for a smoother and more precise touchdown.
3. **Terrain Following (Low Alt):**
   - *Scenario:* A drone is flying 1m above the ground.
   - *Action:* While primarily using a rangefinder for height, the optical flow data (combined with height) provides a robust velocity reference that is unaffected by atmospheric pressure changes.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4047**: Entry point for flow data.
- **libraries/AP_OpticalFlow/AP_OpticalFlow_MAV.cpp**: MAVLink backend implementation.
- **libraries/AP_OpticalFlow/AP_OpticalFlow.cpp**: Frontend sensor management.

GLOBAL_VISION_POSITION_ESTIMATE (ID 101)

SUPPORTED

Summary

The `GLOBAL_VISION_POSITION_ESTIMATE` message is used by external positioning systems (like Vicon or SLAM) to provide the vehicle with a globally-referenced position and attitude estimate. While the MAVLink standard distinguishes this from local estimates, ArduPilot unifies both into its "External Navigation" pipeline to support high-precision flight in environments where GPS is unavailable or untrusted.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Enables external navigation input)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_global_vision_position_estimate` in libraries/GCS_MAVLink/GCS_Common.cpp:3859.

Data Processing

1. **Decoding:** The message is decoded and passed to a common handler for vision data (GCS_Common.cpp:3919).
2. **Library Integration:** The data is forwarded to the `AP_VisualOdom` library.
3. **EKF Fusion:** The `AP_VisualOdom_MAV` backend converts the meters-based X, Y, and Z coordinates into a format suitable for the EKF (Extended Kalman Filter). The EKF then treats this as a primary position source (External Navigation) via `AP_AHRS::writeExtNavData`.

Data Fields

- `usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `x`: Global X position (meters).
- `y`: Global Y position (meters).
- `z`: Global Z position (meters).
- `roll`: Roll angle (rad).
- `pitch`: Pitch angle (rad).
- `yaw`: Yaw angle (rad).
- `covariance`: Pose covariance matrix upper right triangle (first six entries are the first ROW, next five entries are the second ROW, etc.).
- `reset_counter`: Estimate reset counter. This should be incremented when the estimate jumps in a discontinuous creation (e.g. at the start of a mission or when the system recovers from a tracking failure).

Practical Use Cases

1. **Vicon/Optitrack Integration:**
   - *Scenario:* A researcher is flying a drone in a motion capture laboratory.
   - *Action:* The lab computer tracks the drone and streams `GLOBAL_VISION_POSITION_ESTIMATE` at 50Hz-100Hz. The drone flies with millimeter precision, even with no GPS lock.

2. **Autonomous Greenhouse Monitoring:**
    - *Scenario:* A drone is navigating between rows of crops in a greenhouse using a pre-mapped SLAM environment.
    - *Action:* The onboard SLAM computer sends global-referenced coordinates to the autopilot, allowing the drone to follow complex inspection routes autonomously.
3. **GPS-Denied Surveying:**
    - *Scenario:* A drone is surveying the inside of a large storage tank.
    - *Action:* A laser-based positioning system provides absolute coordinates to the flight controller, ensuring the survey data is spatially accurate.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3859**: Message entry point.
- **libraries/AP_VisualOdom/AP_VisualOdom_MAV.cpp:26**: MAVLink backend for visual odometry.

VISION_POSITION_ESTIMATE (ID 102) SUPPORTED

Summary

The `VISION_POSITION_ESTIMATE` message is the standard MAVLink packet for providing local X, Y, and Z position data from an external vision system (e.g., SLAM or Visual Odometry). ArduPilot uses this message to enable stable flight and autonomous navigation in GPS-denied environments.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Enables external navigation input)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_vision_position_estimate` in libraries/GCS_MAVLink/GCS_Common.cpp:3850.

Data Processing

1. **Unification:** In ArduPilot's implementation, `VISION_POSITION_ESTIMATE` and `GLOBAL_VISION_POSITION_ESTIMATE` (101) are handled by the same internal pipeline (GCS_Common.cpp:3919).
2. **State Estimation:** The data is forwarded to the `AP_VisualOdom` library.
3. **EKF Injection:** The message provides position (X, Y, Z in meters) and attitude (Roll, Pitch, Yaw in radians). These are injected into the EKF as "External Navigation" data. This allows the vehicle to hold its position based purely on camera-derived movement.

Data Fields

- `usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `x`: Global X position (meters).
- `y`: Global Y position (meters).
- `z`: Global Z position (meters).
- `roll`: Roll angle (rad).
- `pitch`: Pitch angle (rad).
- `yaw`: Yaw angle (rad).
- `covariance`: Pose covariance matrix upper right triangle (first six entries are the first ROW, next five entries are the second ROW, etc.).
- `reset_counter`: Estimate reset counter. This should be incremented when the estimate jumps in a discontinuous creation (e.g. at the start of a mission or when the system recovers from a tracking failure).

Practical Use Cases

1. **Companion Computer SLAM:**
    - *Scenario:* An Intel Realsense T265 camera is connected to a Jetson Nano onboard the drone.
    - *Action:* The Jetson Nano runs a SLAM algorithm and sends `VISION_POSITION_ESTIMATE` to the flight controller at 30Hz. The pilot can then switch to "Position Hold" or "Auto" mode

indoors.

2. **Swarm Robotics in Studios:**
    - *Scenario:* Multiple drones are performing synchronized dance routines in a studio equipped with infrared cameras.
    - *Action:* The central studio controller tracks each drone and sends local coordinates, ensuring no collisions and perfect synchronization.
3. **Visual Docking:**
    - *Scenario:* A drone is attempting to land on a moving rover with a visual target.
    - *Action:* A visual tracking system calculates the drone's position relative to the target and provides it via MAVLink, allowing for a precise automated landing.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3850**: Message entry point.
- **libraries/AP_VisualOdom/AP_VisualOdom.cpp:193**: Common pose handling logic.
- **libraries/AP_VisualOdom/AP_VisualOdom_MAV.cpp:26**: MAVLink backend for visual odometry.

VISION_SPEED_ESTIMATE (ID 103) SUPPORTED

Summary

The `VISION_SPEED_ESTIMATE` message is used by external sensors to provide 3D velocity data (X, Y, Z in meters per second) to the flight controller. This is often used as a supplement to position estimates or as a standalone source for velocity fusion in the EKF (Extended Kalman Filter), allowing the vehicle to maintain stability based on perceived movement even if absolute global or local position is unknown.

Status

Supported

Directionality

- **TX (Transmit):** None (Except SITL for simulation)
- **RX (Receive):** All Vehicles (Enables external velocity input)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_vision_speed_estimate` in libraries/GCS_MAVLink/GCS_Common.cpp:3964.

Data Processing

1. **Decoding:** The message is decoded into X, Y, and Z velocity components (m/s).
2. **Visual Odometry Integration:** The data is passed to the `AP_VisualOdom` frontend (AP_VisualOdom.cpp:235).
3. **EKF Fusion:** The MAVLink backend for visual odometry calls `AP_AHRS::writeExtNavVelData`. This pushes the external velocity directly into the EKF's state vector. This is particularly useful for damping and position-hold stability.

Data Fields

- `usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `x`: Global X speed (m/s).
- `y`: Global Y speed (m/s).
- `z`: Global Z speed (m/s).
- `covariance`: Linear velocity covariance matrix (m/s)^2. Upper right triangle (first three entries are the first ROW, next two entries are the second ROW, etc.).
- `reset_counter`: Estimate reset counter. This should be incremented when the estimate jumps in a discontinuous creation (e.g. at the start of a mission or when the system recovers from a tracking failure).

Practical Use Cases

1. **Velocity-Only SLAM:**
   - *Scenario:* An external camera system provides high-frequency velocity data but the position drifts too much for absolute navigation.
   - *Action:* The autopilot uses `VISION_SPEED_ESTIMATE` to "brake" and hold position effectively, while ignoring the drifting position data.
2. **Optical Flow Enhancement:**

    - *Scenario:* A high-end optical flow system calculates 3D velocity using a downward-facing camera and IMU.
    - *Action:* The system sends `VISION_SPEED_ESTIMATE` to ArduPilot, providing a more robust velocity reference than the standard `OPTICAL_FLOW` (100) message.
3. **Wind Speed Correction (Experimental):**
    - *Scenario:* A ground-based anemometer array tracks the drone and estimates its true ground speed.
    - *Action:* The ground system sends velocity updates to the drone via MAVLink to improve its flight controller's performance in turbulent conditions.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3964**: Message entry point.
- **libraries/AP_VisualOdom/AP_VisualOdom_MAV.cpp:62**: EKF integration logic for velocity estimates.

VICON_POSITION_ESTIMATE (ID 104)

SUPPORTED

Summary

The `VICON_POSITION_ESTIMATE` message is used to provide high-precision position and orientation data from an external Motion Capture (MoCap) system, such as Vicon or OptiTrack. This is the "Gold Standard" for indoor flight, providing millimeter-level accuracy that allows for aggressive autonomous maneuvers in environments where GPS is blocked or multipathed.

Status

Supported

Directionality

- **TX (Transmit):** None (Except SITL for simulation)
- **RX (Receive):** All Vehicles (Enables MoCap-based navigation)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_vicon_position_estimate` in libraries/GCS_MAVLink/GCS_Common.cpp:3868.

Data Processing

1. **Decoding:** The message is decoded into local X, Y, and Z position (m) and Roll, Pitch, and Yaw attitude (rad).
2. **Unification:** Like other vision messages, the data is forwarded to the `AP_VisualOdom` library.
3. **EKF Fusion:** The data is pushed to the EKF as "External Navigation" (ExtNav). Because MoCap data is extremely low-noise and has very low latency, the EKF can be tuned to "trust" this source almost exclusively, enabling extremely stable hover and precise path tracking.

Data Fields

- `usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `x`: Global X position (meters).
- `y`: Global Y position (meters).
- `z`: Global Z position (meters).
- `roll`: Roll angle (rad).
- `pitch`: Pitch angle (rad).
- `yaw`: Yaw angle (rad).
- `covariance`: Pose covariance matrix upper right triangle (first six entries are the first ROW, next five entries are the second ROW, etc.).

Practical Use Cases

1. **Indoor Research Labs:**
   - *Scenario:* A university team is testing a new obstacle avoidance algorithm using multiple drones.
   - *Action:* The lab's Vicon system tracks all drones and sends individual `VICON_POSITION_ESTIMATE` packets to each flight controller. This provides a "Ground Truth" for the drones to navigate safely within the lab.
2. **Cinematic Stage Flight:**

    - *Scenario:* A drone is required to fly a precise 3D path around actors on a film set inside a studio.
    - *Action:* An OptiTrack system provides the drone with its coordinates, ensuring the flight path is repeatable and safe to within a few centimeters.
3. **HIL Simulation Testing:**
    - *Scenario:* A developer wants to test how the drone reacts to high-frequency position updates.
    - *Action:* ArduPilot's SITL (Software In The Loop) simulator can be configured to generate "Fake Vicon" data, allowing the developer to test the ExtNav pipeline without a physical lab.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3868**: Message entry point.
- **libraries/SITL/SIM_Vicon.cpp**: SITL simulator for MoCap systems.

HIGHRES_IMU (ID 105)

SUPPORTED

Summary

The `HIGHRES_IMU` message is a high-fidelity, integrated sensor packet designed for performance-critical applications like Visual-Inertial Odometry (VIO) and high-speed logging. Unlike standard IMU messages that use 16-bit integers, `HIGHRES_IMU` uses single-precision floats for all sensor data and provides a unified view of the Accelerometer, Gyroscope, Magnetometer, and Barometer.

Status

Supported (Requires > 1MB Flash)

Directionality

- **TX (Transmit):** All Vehicles (High-fidelity sensor stream)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_highres_imu` within libraries/GCS_MAVLink/GCS_Common.cpp:2183.

Data Sourcing

This message aggregates data from across the sensor stack:

- **IMU:** Accelerometer (m/s^2) and Gyroscope (rad/s) data from the primary IMU.
- **Compass:** Magnetometer ($Gauss$) data.
- **Barometer:** Absolute pressure (hPa) and Pressure Altitude (m).
- **Airspeed:** Differential pressure (hPa) if an airspeed sensor is active.
- **Timestamp:** Uses a 64-bit microsecond timestamp (`time_usec`) for precise data alignment.

Efficiency

The "Highres" designation refers to the use of IEEE 754 floats, which allow for much greater dynamic range and precision compared to the scaled integers used in `RAW_IMU` or `SCALED_IMU`.

Data Fields

- **`time_usec`**: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- **`xacc`**: X acceleration (m/s^2).
- **`yacc`**: Y acceleration (m/s^2).
- **`zacc`**: Z acceleration (m/s^2).
- **`xgyro`**: Angular speed around X axis (rad/s).
- **`ygyro`**: Angular speed around Y axis (rad/s).
- **`zgyro`**: Angular speed around Z axis (rad/s).
- **`xmag`**: X Magnetic field (Gauss).
- **`ymag`**: Y Magnetic field (Gauss).
- **`zmag`**: Z Magnetic field (Gauss).
- **`abs_pressure`**: Absolute pressure (hectopascal).
- **`diff_pressure`**: Differential pressure (hectopascal).
- **`pressure_alt`**: Altitude calculated from pressure.

- `temperature`: Temperature (degrees celsius).
- `fields_updated`: Bitmap for fields that have updated since last message, bit 0 = xacc, bit 12: temperature.
- `id`: Id. Optional, default: 0.

Practical Use Cases

1. **VIO/SLAM Integration:**
    - *Scenario:* A drone is flying indoors using a companion computer (e.g., Raspberry Pi) running VIO (Visual Inertial Odometry).
    - *Action:* The companion computer subscribes to `HIGHRES_IMU` at 100Hz+. The high-precision float values and integrated microsecond timestamps are used to fuse camera data with IMU movement, providing stable position estimates in GPS-denied environments.
2. **External Kalman Filtering:**
    - *Scenario:* A researcher is developing a custom navigation filter on a ground computer.
    - *Action:* The researcher logs `HIGHRES_IMU` data. Because it includes pressure and magnetometer data in the same packet as the IMU, the researcher doesn't have to worry about inter-message jitter or alignment issues.
3. **High-Speed Vibration Monitoring:**
    - *Scenario:* Testing a new propulsion system with very high RPM motors.
    - *Action:* The developer uses `HIGHRES_IMU` to capture the full spectrum of high-frequency vibrations that might be clipped or aliased by lower-resolution integer messages.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2183**: Implementation of `send_highres_imu`.
- **libraries/GCS_MAVLink/GCS_config.h:131**: Conditional compilation logic based on board flash size.

SCALED_IMU2 (ID 116) SUPPORTED

Summary

The `SCALED_IMU2` message provides high-frequency acceleration and rotation data from the vehicle's **secondary** inertial sensor (IMU 1). ArduPilot uses this message to stream data from redundant IMUs, allowing Ground Control Stations to monitor the health and alignment of secondary sensors in multi-IMU systems.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Secondary IMU telemetry)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is implemented in a unified helper `GCS_MAVLINK::send_scaled_imu` within libraries/GCS_MAVLink/GCS_Common.cpp:2259.

Data Sourcing

- **Secondary Instance:** This message specifically transmits data from **IMU 1**.
- **Time Source:** Uses a millisecond timestamp (`AP_HAL::millis()`).
- **Scaling:** (Identical to `SCALED_IMU` ID 26)
    - **Accelerometer:** Scaled to milli-G (mG).
    - **Gyroscope:** Scaled to millirad/s (rad/s * 1000).
    - **Magnetometer:** Scaled to milli-Gauss (mGauss).

Scheduling

- Sent as part of the `MSG_SCALED_IMU2` stream.
- Triggered in `GCS_Common.cpp:6312` within the `try_send_message` loop.

Data Fields

- **`time_boot_ms`**: Timestamp (milliseconds since system boot).
- **`xacc`**: X acceleration (mg).
- **`yacc`**: Y acceleration (mg).
- **`zacc`**: Z acceleration (mg).
- **`xgyro`**: Angular speed around X axis (millirad /sec).
- **`ygyro`**: Angular speed around Y axis (millirad /sec).
- **`zgyro`**: Angular speed around Z axis (millirad /sec).
- **`xmag`**: X Magnetic field (milli tesla).
- **`ymag`**: Y Magnetic field (milli tesla).
- **`zmag`**: Z Magnetic field (milli tesla).

Practical Use Cases

1. **Redundancy Monitoring:**

    - *Scenario:* A high-end hexacopter has triple-redundant IMUs.
    - *Action:* The GCS graphs `SCALED_IMU` (IMU 0) and `SCALED_IMU2` (IMU 1) together. If one sensor shows significantly more vibration than the other, the pilot can identify a mechanical issue near that specific sensor mounting point.
2. **Vibration Isolation Testing:**
    - *Scenario:* A builder is testing a new silicone damping mount for the flight controller.
    - *Action:* By comparing the data from a hard-mounted IMU (reported via one message) against a dampened IMU (reported via another), the builder can quantify the effectiveness of the vibration isolation.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2259**: Core logic for scaling and sending IMU data.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6310**: Dispatcher for the secondary IMU stream.

GPS2_RAW (ID 124)

SUPPORTED

Summary

The `GPS2_RAW` message provides raw satellite positioning data from the vehicle's **secondary** GPS receiver (GPS 1). In multi-GPS or "Blended" setups, this message allows the Ground Control Station (GCS) to monitor the health and fix quality of the backup receiver independently of the primary or fused position estimate.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports secondary GPS data)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `AP_GPS::send_mavlink_gps2_raw` within libraries/AP_GPS/AP_GPS.cpp:1405.

Data Sourcing

- **Secondary Instance:** Unlike `GPS_RAW_INT` (24) which uses GPS instance 0, this message specifically pulls data from **GPS instance 1**.
- **Fields:** Includes the standard set of GPS metrics:
  - `lat`, `lon`: Sourced in $degrees \times 10^7$.
  - `alt`: MSL Altitude in millimeters.
  - `vel`, `cog`: Ground speed (cm/s) and Course Over Ground (centidegrees).
  - `satellites_visible`: Number of satellites used by the secondary receiver.
  - `dgps_numch`, `dgps_age`: Metadata for Differential GPS (DGPS) or RTK correction.

Scheduling

- Sent as part of the `MSG_GPS2_RAW` stream.
- Triggered in `GCS_Common.cpp:6213` within the `try_send_message` loop.

Data Fields

- **`time_usec`**: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- **`fix_type`**: GPS fix type (`GPS_FIX_TYPE`).
- **`lat`**: Latitude (WGS84, EGM96 ellipsoid), in degrees * 1E7.
- **`lon`**: Longitude (WGS84, EGM96 ellipsoid), in degrees * 1E7.
- **`alt`**: Altitude (MSL). Positive for up.
- **`eph`**: GPS HDOP horizontal dilution of precision in cm (m*100). If unknown, set to: UINT16_MAX.
- **`epv`**: GPS VDOP vertical dilution of precision in cm (m*100). If unknown, set to: UINT16_MAX.
- **`vel`**: GPS ground speed (m/s * 100). If unknown, set to: UINT16_MAX.
- **`cog`**: Course over ground (NOT heading, but direction of movement) in degrees * 100, 0.0..359.99 degrees. If unknown, set to: UINT16_MAX.
- **`satellites_visible`**: Number of satellites visible. If unknown, set to 255.
- **`dgps_numch`**: Number of DGPS satellites.
- **`dgps_age`**: Age of DGPS info.

Practical Use Cases

1. **Redundancy Verification:**
   - *Scenario:* A high-value cinematic drone is flying a mission.
   - *Action:* The pilot monitors `GPS_RAW_INT` and `GPS2_RAW` side-by-side. If the primary GPS loses lock (drops to 0 satellites), the pilot can see if the secondary GPS still has a healthy 3D fix before deciding to continue the flight.
2. **Blended GPS Analysis:**
   - *Scenario:* A developer is using `GPS_TYPE=1` (Blending) to merge data from two different brands of GPS receivers.
   - *Action:* The developer logs both messages to analyze which receiver provides better performance in high-multipath environments (e.g., near buildings).
3. **RTK Baseline Monitoring:**
   - *Scenario:* Using a "Moving Baseline" setup for GPS Yaw.
   - *Action:* The GCS uses `GPS2_RAW` to verify that the "Rover" GPS (the secondary unit) is receiving RTK corrections from the "Base" GPS.

Key Codebase Locations

- **libraries/AP_GPS/AP_GPS.cpp:1405**: Implementation of the MAVLink packet construction for GPS 1.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6213**: Scheduling logic.

GPS_RTK (ID 127)

SUPPORTED

Summary

The `GPS_RTK` message provides detailed status information about the Real-Time Kinematic (RTK) baseline solution from the primary GPS receiver. It includes the baseline vector (X, Y, Z in millimeters) and accuracy metrics. This message is primarily supported by specific high-precision GPS drivers (Septentrio SBF, Swift Navigation SBP, and Emlid ERB).

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports RTK baseline)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission is handled by `AP_GPS_Backend::send_mavlink_gps_rtk` within libraries/AP_GPS/GPS_Backend.cpp:176.

Drivers

Only specific GPS drivers implement the `supports_mavlink_gps_rtk_message()` flag to enable this message:

- **SBF:** Septentrio
- **SBP:** Swift Navigation (Piksi)
- **ERB:** Emlid Reach

Data Fields

- **`time_last_baseline_ms`**: Time since last baseline (ms) - *Currently sent as 0.*
- **`rtk_receiver_id`**: RTK receiver ID - *Currently sent as 0.*
- **`wn`**: GPS Week Number of last baseline.
- **`tow`**: GPS Time of Week of last baseline (ms).
- **`rtk_health`**: RTK health - *Currently sent as 0.*
- **`rtk_rate`**: RTK rate - *Currently sent as 0.*
- **`nsats`**: Number of satellites used for RTK.
- **`baseline_a_mm`**: RTK Baseline Coordinate A (mm) - (North or ECEF X depending on `baseline_coords_type`).
- **`baseline_b_mm`**: RTK Baseline Coordinate B (mm) - (East or ECEF Y depending on `baseline_coords_type`).
- **`baseline_c_mm`**: RTK Baseline Coordinate C (mm) - (Down or ECEF Z depending on `baseline_coords_type`).
- **`accuracy`**: RTK accuracy (mm).
- **`iar_num_hypotheses`**: Integer Ambiguity Resolution hypotheses.

Practical Use Cases

1. **Baseline Monitoring:**
    - *Scenario:* A user is setting up a dual-GPS yaw system using SBF receivers.

    - *Action:* The GCS monitors `baseline_a/b/c_mm` to visualize the vector between the two antennas, ensuring it matches the physical mounting distance.

Key Codebase Locations

- **libraries/AP_GPS/GPS_Backend.cpp:176**: Implementation of the sender.

GPS2_RTK (ID 128)

SUPPORTED

Summary

The `GPS2_RTK` message provides detailed status information about the Real-Time Kinematic (RTK) baseline solution from the **secondary** GPS receiver. It functions identically to `GPS_RTK` (127).

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports secondary RTK baseline)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission is handled by `AP_GPS_Backend::send_mavlink_gps_rtk` within libraries/AP_GPS/GPS_Backend.cpp:176.

Drivers

Only specific GPS drivers implement the `supports_mavlink_gps_rtk_message()` flag to enable this message:

- **SBF:** Septentrio
- **SBP:** Swift Navigation (Piksi)
- **ERB:** Emlid Reach

Data Fields

- `time_last_baseline_ms`: Time since last baseline (ms) - *Currently sent as 0*.
- `rtk_receiver_id`: RTK receiver ID - *Currently sent as 0*.
- `wn`: GPS Week Number of last baseline.
- `tow`: GPS Time of Week of last baseline (ms).
- `rtk_health`: RTK health - *Currently sent as 0*.
- `rtk_rate`: RTK rate - *Currently sent as 0*.
- `nsats`: Number of satellites used for RTK.
- `baseline_a_mm`: RTK Baseline Coordinate A (mm) - (North or ECEF X).
- `baseline_b_mm`: RTK Baseline Coordinate B (mm) - (East or ECEF Y).
- `baseline_c_mm`: RTK Baseline Coordinate C (mm) - (Down or ECEF Z).
- `accuracy`: RTK accuracy (mm).
- `iar_num_hypotheses`: Integer Ambiguity Resolution hypotheses.

Practical Use Cases

1. **Dual-Antenna GPS Yaw:**
   - *Scenario:* A rover uses two Swift Navigation Piksi Multi receivers for moving baseline yaw.
   - *Action:* The GCS monitors `GPS_RTK` (Primary) and `GPS2_RTK` (Secondary) to verify that both units are calculating baselines correctly relative to the base station or each other.

Key Codebase Locations

- **libraries/AP_GPS/GPS_Backend.cpp:176**: Implementation of the sender.

SCALED_IMU3 (ID 129)

SUPPORTED

Summary

The `SCALED_IMU3` message provides high-frequency acceleration and rotation data from the vehicle's **tertiary** inertial sensor (IMU 2). This message is primarily used on high-end flight controllers with three or more IMUs, providing the final layer of redundancy for state estimation.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Tertiary IMU telemetry)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is implemented in a unified helper `GCS_MAVLINK::send_scaled_imu` within libraries/GCS_MAVLink/GCS_Common.cpp:2259.

Data Sourcing

- **Tertiary Instance:** This message specifically transmits data from **IMU 2**.
- **Time Source:** Uses a millisecond timestamp (`AP_HAL::millis()`).
- **Scaling:** (Identical to `SCALED_IMU` ID 26)
  - **Accelerometer:** Scaled to milli-G (mG).
  - **Gyroscope:** Scaled to millirad/s (rad/s * 1000).
  - **Magnetometer:** Scaled to milli-Gauss (mGauss).

Scheduling

- Sent as part of the `MSG_SCALED_IMU3` stream.
- Triggered in `GCS_Common.cpp:6316` within the `try_send_message` loop.

Data Fields

- **`time_boot_ms`:** Timestamp (milliseconds since system boot).
- **`xacc`:** X acceleration (mg).
- **`yacc`:** Y acceleration (mg).
- **`zacc`:** Z acceleration (mg).
- **`xgyro`:** Angular speed around X axis (millirad /sec).
- **`ygyro`:** Angular speed around Y axis (millirad /sec).
- **`zgyro`:** Angular speed around Z axis (millirad /sec).
- **`xmag`:** X Magnetic field (milli tesla).
- **`ymag`:** Y Magnetic field (milli tesla).
- **`zmag`:** Z Magnetic field (milli tesla).

Practical Use Cases

1. **Triple Redundancy Voting:**
   - *Scenario:* A high-reliability flight controller uses a voting system between three IMUs.

    - *Action:* If two IMUs agree but `SCALED_IMU3` disagrees, the flight controller "votes out" the tertiary sensor. The GCS displays this disagreement using the streamed data from all three messages.
2. **Hardware Identification:**
    - *Scenario:* A developer is writing a driver for a new IMU chip mounted as the third sensor on a custom board.
    - *Action:* The developer monitors `SCALED_IMU3` to verify that the raw driver is correctly passing scaled values to the GCS.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2259**: Core logic for scaling and sending IMU data.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6315**: Dispatcher for the tertiary IMU stream.

DISTANCE_SENSOR (ID 132)

SUPPORTED

Summary

The `DISTANCE_SENSOR` message reports a single measurement from a distance sensor (LIDAR, Sonar, Radar). It provides the distance, sensor type, min/max range, and the sensor's orientation relative to the vehicle frame. This message is crucial for terrain following, precision landing, and obstacle avoidance.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports sensor data)
- **RX (Receive):** None (ArduPilot consumes `OBSTACLE_DISTANCE` or `OBSTACLE_DISTANCE_3D` for reception typically, though `DISTANCE_SENSOR` decoding exists in some limited proximity contexts)

Transmission (TX)

The transmission logic is in `GCS_MAVLINK::send_distance_sensor` within libraries/GCS_MAVLink/GCS_Common.cpp:411.

Data Sourcing

ArduPilot aggregates data from two libraries:

1. `AP_RangeFinder`: For dedicated 1D sensors (e.g., downward facing altimeters).
   - Iterates through all healthy backends.
   - Populates `type`, `orientation`, `min_distance`, `max_distance`, and `current_distance` directly from the driver.
2. `AP_Proximity`: For obstacle avoidance sensors (e.g., 360 LIDAR sectors).
   - Iterates through valid proximity sectors (0-7 for 8-way octomap).
   - Synthesizes a "Virtual Sensor" message for each active sector.
   - `id` starts at `PROXIMITY_SENSOR_ID_START`.

Data Fields

- `time_boot_ms`: Timestamp (milliseconds since system boot).
- `min_distance`: Minimum distance the sensor can measure (cm).
- `max_distance`: Maximum distance the sensor can measure (cm).
- `current_distance`: Current distance reading (cm).
- `type`: Type of distance sensor (`MAV_DISTANCE_SENSOR_LASER`, `ULTRASOUND`, etc.).
- `id`: Onboard ID of the sensor.
- `orientation`: Direction the sensor faces (`MAV_SENSOR_ORIENTATION`). 0=Forward, 24=Down.
- `covariance`: Measurement covariance (cm^2), 0 for unknown.
- `horizontal_fov`: Horizontal Field of View (radians).
- `vertical_fov`: Vertical Field of View (radians).
- `quaternion`: Quaternion of the sensor orientation (w, x, y, z).
- `signal_quality`: Signal quality (0 = unknown, 1 = invalid, 100 = perfect).

Practical Use Cases

1. **Terrain Following:**

    - *Scenario:* A plane is flying at low altitude over hilly terrain.
    - *Action:* The GCS monitors `DISTANCE_SENSOR` (Orientation: Down) to verify the plane is maintaining the target AGL (Above Ground Level) altitude.
2. **Obstacle Visualization:**
    - *Scenario:* A copter is flying near a wall.
    - *Action:* The GCS receives a stream of `DISTANCE_SENSOR` messages with Orientations 0 (Forward), 2 (Right), etc., and draws a "Radar View" showing the distance to obstacles in each quadrant.
3. **Sensor Health Check:**
    - *Scenario:* A pilot suspects a sonar sensor is faulty.
    - *Action:* Inspecting the message stream reveals that `current_distance` is stuck at `min_distance`, indicating a hardware fault or noise floor issue.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:411**: Implementation of the sender.
- **libraries/AP_RangeFinder/AP_RangeFinder.cpp**: Source of rangefinder data.

SCALED_PRESSURE2 (ID 137)

SUPPORTED

Summary

The `SCALED_PRESSURE2` message provides calibrated environmental data from the vehicle's **secondary** sensors. It reports absolute pressure (from Barometer 2) and differential pressure (from Airspeed Sensor 2). This allows Ground Control Stations to monitor sensor redundancy and health.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports secondary sensor data)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is implemented in `GCS_MAVLINK::send_scaled_pressure2` within libraries/GCS_MAVLink/GCS_Common.cpp:2359.

Data Sourcing

- **Absolute Pressure (`press_abs`):** Sourced from the secondary barometer (Index 1) via `AP_Baro`. Values are in Hectopascals (hPa).
- **Differential Pressure (`press_diff`):** Sourced from the secondary airspeed sensor (Index 1) via `AP_Airspeed`. Values are in hPa.
- **Temperature:** Includes the barometer's ambient temperature reading in centidegrees Celsius.
- **Temperature Differential:** Includes the airspeed sensor's temperature reading in centidegrees Celsius.

Data Fields

- `time_boot_ms`: Timestamp (milliseconds since system boot).
- `press_abs`: Absolute pressure (hectopascal).
- `press_diff`: Differential pressure 1 (hectopascal).
- `temperature`: Absolute pressure temperature (centidegrees Celsius).
- `temperature_press_diff`: Differential pressure temperature (centidegrees Celsius).

Practical Use Cases

1. **Redundancy Checks:**
   - *Scenario:* A pilot receives a "Bad Baro Health" warning.
   - *Action:* The GCS graphs `SCALED_PRESSURE` and `SCALED_PRESSURE2` side-by-side. If `SCALED_PRESSURE` is flatlining but `SCALED_PRESSURE2` is responding to altitude changes, it confirms the primary sensor has failed.
2. **Dual Airspeed Setup:**
   - *Scenario:* A VTOL plane has a pitot tube on each wing.
   - *Action:* The pilot monitors `press_diff` from both messages to ensure neither tube is blocked by ice or debris.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2359**: Message dispatcher.
- **libraries/GCS_MAVLink/GCS_Common.cpp:2308**: Shared logic for populating pressure instances.

ATT_POS_MOCAP (ID 138)

SUPPORTED

Summary

The `ATT_POS_MOCAP` message provides high-precision position and attitude data from an external Motion Capture (MoCap) system, such as Vicon or OptiTrack. It is functionally very similar to `VICON_POSITION_ESTIMATE` (104) and `VISION_POSITION_ESTIMATE` (102), serving as another entry point for external navigation data.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Enables MoCap-based navigation)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_att_pos_mocap` in libraries/GCS_MAVLink/GCS_Common.cpp:3948.

Data Processing

1. **Decoding:** The message is decoded into local X, Y, and Z position (m) and a Quaternion (w, x, y, z) for attitude.
2. **Unification:** The data is forwarded to the `AP_VisualOdom` library via `handle_pose_estimate`.
3. **EKF Fusion:** The EKF fuses this data as an External Navigation source, allowing the vehicle to fly autonomously without GPS.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `q`: Attitude quaternion (w, x, y, z order, zero-rotation is 1, 0, 0, 0).
- `x`: X position (meters).
- `y`: Y position (meters).
- `z`: Z position (meters).
- `covariance`: Pose covariance matrix upper right triangle.

Practical Use Cases

1. **Indoor University Labs:**
    - *Scenario:* A researcher uses an older MoCap system that natively outputs the `ATT_POS_MOCAP` packet format.
    - *Action:* The researcher connects the MoCap computer to the drone's telemetry port. ArduPilot treats the data identically to `VICON_POSITION_ESTIMATE`, enabling stable indoor loiter.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3948**: Message entry point.

- **libraries/AP_VisualOdom/AP_VisualOdom.cpp**: Core logic for handling pose estimates.

SCALED_PRESSURE3 (ID 143)

SUPPORTED

Summary

The `SCALED_PRESSURE3` message provides calibrated environmental data from the vehicle's **tertiary** sensors. It reports absolute pressure (from Barometer 3) and differential pressure (from Airspeed Sensor 3). This allows Ground Control Stations to monitor sensor redundancy and health.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports tertiary sensor data)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is implemented in `GCS_MAVLINK::send_scaled_pressure3` within libraries/GCS_MAVLink/GCS_Common.cpp:2364.

Data Sourcing

- **Absolute Pressure (`press_abs`):** Sourced from the tertiary barometer (Index 2) via `AP_Baro`. Values are in Hectopascals (hPa).
- **Differential Pressure (`press_diff`):** Sourced from the tertiary airspeed sensor (Index 2) via `AP_Airspeed`. Values are in hPa.
- **Temperature:** Includes the barometer's ambient temperature reading in centidegrees Celsius.
- **Temperature Differential:** Includes the airspeed sensor's temperature reading in centidegrees Celsius.

ArduSub Exception

In **ArduSub**, this message is repurposed to send water temperature data from the `AP_TemperatureSensor` library if enabled. The pressure fields are set to 0.

Data Fields

- `time_boot_ms`: Timestamp (milliseconds since system boot).
- `press_abs`: Absolute pressure (hectopascal).
- `press_diff`: Differential pressure 1 (hectopascal).
- `temperature`: Absolute pressure temperature (centidegrees Celsius).
- `temperature_press_diff`: Differential pressure temperature (centidegrees Celsius).

Practical Use Cases

1. **Redundancy Checks:**
    - *Scenario:* A pilot receives a "Bad Baro Health" warning.
    - *Action:* The GCS graphs `SCALED_PRESSURE` and `SCALED_PRESSURE3` side-by-side. If `SCALED_PRESSURE` is flatlining but `SCALED_PRESSURE3` is responding to altitude changes, it confirms the primary sensor has failed.
2. **Water Temperature Monitoring:**

    - *Scenario:* An ROV is diving in deep water.
    - *Action:* The operator monitors the `temperature` field of `SCALED_PRESSURE3` to track the external water temperature.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2364**: Message dispatcher.
- **libraries/GCS_MAVLink/GCS_Common.cpp:2308**: Shared logic for populating pressure instances.
- **ArduSub/GCS_Mavlink.cpp:105**: ArduSub override.

AP_ADC (ID 153)

SUPPORTED (LIMITED)

Summary

The `AP_ADC` message reports raw values from the Analog-to-Digital Converter (ADC). Currently, it is **only implemented in the ESP32 HAL** for debugging purposes.

Status

Supported (Limited)

Directionality

- **TX (Transmit):** ESP32 Boards Only
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is located in `AP_HAL_ESP32::AnalogIn` within libraries/AP_HAL_ESP32/AnalogIn.cpp:377.

Usage

- It sends the raw count of the first 6 ADC channels.
- This appears to be a board-specific debug feature and is not generally broadcast by STM32 or other platforms.

Data Fields

- `adc1`: ADC output 1.
- `adc2`: ADC output 2.
- `adc3`: ADC output 3.
- `adc4`: ADC output 4.
- `adc5`: ADC output 5.
- `adc6`: ADC output 6.

Practical Use Cases

1. **Hardware Debugging (ESP32):**
    - *Scenario:* A developer is porting ArduPilot to a new ESP32 flight controller.
    - *Action:* They monitor `AP_ADC` to verify that the analog pins (Current, Voltage, RSSI) are mapping correctly to the expected ADC channels.

Key Codebase Locations

- **libraries/AP_HAL_ESP32/AnalogIn.cpp:377**: Implementation of the sender.

VIBRATION (ID 241)

SUPPORTED

Summary

The `VIBRATION` message provides real-time metrics on the mechanical noise being experienced by the flight controller's accelerometers. It reports vibration levels in the X, Y, and Z axes and tracks "Clipping" events (when a vibration is so intense that it exceeds the sensor's maximum range). This message is the primary diagnostic tool for identifying motor imbalances, damaged propellers, or inadequate vibration isolation.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Mechanical health telemetry)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic resides in `GCS_MAVLINK::send_vibration` within libraries/GCS_MAVLink/GCS_Common.cpp:3005.

Data Sourcing

Vibration metrics are calculated in the `AP_InertialSensor` library using a multi-stage filtering process (AP_InertialSensor.cpp:2236):

1. **Isolation:** Raw acceleration is compared against a 5Hz low-pass filtered "floor" to isolate high-frequency noise from vehicle movement.
2. **Smoothing:** The square of this difference is smoothed via a 2Hz low-pass filter.
3. **Root:** The final vibration level is the square root of this smoothed value.
4. **Clipping:** The `clipping_0`, `clipping_1`, and `clipping_2` fields report the cumulative number of times each IMU has experienced an acceleration beyond its hard limit (e.g., 16G or 32G).

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `vibration_x`: Vibration levels on X-axis.
- `vibration_y`: Vibration levels on Y-axis.
- `vibration_z`: Vibration levels on Z-axis.
- `clipping_0`: first accelerometer clipping count.
- `clipping_1`: second accelerometer clipping count.
- `clipping_2`: third accelerometer clipping count.

Practical Use Cases

1. **Propeller Balancing:**
    - *Scenario:* A pilot installs a new set of carbon fiber propellers.
    - *Action:* The GCS graphs `vibration_z`. If the levels are significantly higher than with the previous propellers, the pilot knows the new set is unbalanced and needs mechanical correction.

2. **Post-Crash Inspection:**
    - *Scenario:* A drone survived a minor crash, but the pilot wants to ensure no internal damage occurred.
    - *Action:* By checking the `clipping` fields, the pilot can see if the motors are now producing extreme vibration spikes that weren't present before, indicating a bent motor shaft or a cracked frame arm.
3. **EKF Health Tuning:**
    - *Scenario:* A developer is seeing "EKF Lane Switches" in high-speed flight.
    - *Action:* By monitoring the `VIBRATION` levels, the developer can determine if the lane switches are being caused by mechanical noise "confusing" the state estimator.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3005**: Implementation of the MAVLink packet construction.
- **libraries/AP_InertialSensor/AP_InertialSensor.cpp:2236**: Core vibration calculation logic (`calc_vibration_and_clipping`).

ADSB_VEHICLE (ID 246) SUPPORTED (RX & TX)

Summary

Location and status of a nearby aircraft detected via ADS-B (Automatic Dependent Surveillance-Broadcast).

Status

Supported (RX & TX)

Directionality

- **TX (Transmit):** All Vehicles - Forwards traffic data to GCS.
- **RX (Receive):** All Vehicles - Receives traffic data from onboard ADSB receiver (e.g., PingRX).

Transmission (TX)

ArduPilot streams this message to the GCS if an onboard ADSB receiver detects a target. This allows the GCS to display traffic without its own ADSB receiver.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Reception (RX)

Handled by `AP_ADSB::handle_message`. The autopilot uses this data for "ADSB Avoidance" (fencing, climbing, or holding to avoid collision).

Source: libraries/AP_ADSB/AP_ADSB.cpp

Data Fields

- `ICAO_address`: Unique aircraft ID.
- `lat`: Latitude.
- `lon`: Longitude.
- `altitude`: Altitude.
- `heading`: Heading.
- `hor_velocity`: Horizontal speed.
- `ver_velocity`: Vertical speed.
- `callsign`: Aircraft callsign.
- `emitter_type`: Type (Light, Heavy, Heli, etc.).
- `tslc`: Time since last communication.
- `flags`: Status flags.
- `squawk`: Transponder code.

Practical Use Cases

1. **Collision Avoidance:**
   - *Scenario:* A Cessna flies near the drone.
   - *Action:* The onboard PingRX detects the ADS-B Out signal. ArduPilot receives `ADSB_VEHICLE`, calculates a collision risk, and automatically initiates a descent to clear the airspace.

Key Codebase Locations

- **libraries/AP_ADSB/AP_ADSB.cpp:820**: Handler.

WHEEL_DISTANCE (ID 9000) SUPPORTED (ROVER ONLY)

Summary

The `WHEEL_DISTANCE` message reports the cumulative distance traveled by each individual wheel, as measured by wheel encoders. This data is critical for odometry-based navigation in ground vehicles (Rover).

Status

Supported (Rover Only)

Directionality

- **TX (Transmit):** Autopilot (Reports wheel odometry to GCS/Companion)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the **ArduRover** firmware's `GCS_Mavlink` module. It is not currently implemented for Copter or Plane.

Core Logic

The implementation is in `Rover::send_wheel_encoder_distance` within Rover/GCS_Mavlink.cpp:406.

It iterates through the active wheel sensors (managed by `AP_WheelEncoder`) and populates the distance array.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `count`: Number of wheels reported.
- `distance`: Distance traveled per wheel (meters).

Practical Use Cases

1. **Dead Reckoning:**
    - *Scenario:* A rover enters a tunnel and loses GPS.
    - *Action:* The Companion Computer uses the `WHEEL_DISTANCE` updates (differential odometry) to estimate the vehicle's path and keep it centered in the lane.
2. **Slip Detection:**
    - *Scenario:* A rover is stuck in mud.
    - *Action:* The `distance` values increase rapidly while the IMU detects no acceleration. The autopilot detects the discrepancy and triggers a "Stuck" failsafe.

Key Codebase Locations

- **Rover/GCS_Mavlink.cpp:406**: Implementation of the sender.

WATER_DEPTH (ID 11038)

SUPPORTED (ROVER ONLY)

Summary

The `WATER_DEPTH` message reports the depth of the water column beneath a boat, along with the water temperature and the vehicle's geolocation/orientation. This is essential for hydrographic surveys and bathymetry.

Status

Supported (Rover Only)

Directionality

- **TX (Transmit):** Autopilot (Reports depth to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the **ArduRover** firmware's `GCS_Mavlink` module. It requires the `FRAME_CLASS` parameter to be set to "Boat" and a RangeFinder driver to be configured with orientation `PITCH_270` (Down).

Core Logic

The implementation is in `GCS_MAVLINK_Rover::send_water_depth` within Rover/GCS_Mavlink.cpp:189.

It iterates through all connected RangeFinders. If a sensor is facing down and has valid data, it populates the message.

Data Fields

- **`time_boot_ms`:** Timestamp (ms since boot).
- **`id`:** RangeFinder instance ID.
- **`healthy`:** Sensor health flag.
- **`lat` / `lng`:** Vehicle Latitude/Longitude (degE7).
- **`alt`:** Vehicle Altitude (m).
- **`roll` / `pitch` / `yaw`:** Vehicle attitude (radians). This allows post-processing software to correct for "slant range" errors if the boat is rolling in waves.
- **`distance`:** Depth in meters.
- **`temperature`:** Water temperature in degC.

Practical Use Cases

1. **Bathymetric Mapping:**
   - *Scenario:* A survey boat follows a grid pattern on a lake.
   - *Action:* The GCS records `WATER_DEPTH` messages. Post-processing software combines `lat`, `lng`, and `distance` (corrected by `roll`/`pitch`) to generate a 3D topographic map of the lakebed.

Key Codebase Locations

- **Rover/GCS_Mavlink.cpp:189**: Implementation of the sender.

HYGROMETER_SENSOR (ID 12920)

SUPPORTED (PLANE ONLY)

Summary

The `HYGROMETER_SENSOR` message reports the ambient temperature and relative humidity measured by an onboard sensor. In ArduPilot, this is typically sourced from high-precision airspeed sensors (like the Sensirion SDP3x series) that include integrated hygrometers.

Status

Supported (Plane Only)

Directionality

- **TX (Transmit):** Autopilot (Reports environmental data to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the **ArduPlane** firmware's `GCS_Mavlink` module. It iterates through the configured Airspeed sensors and checks if they provide hygrometer data.

Core Logic

The implementation is in `GCS_MAVLINK_Plane::send_hygrometer` within ArduPlane/GCS_Mavlink.cpp:473.

It checks for fresh data (`now - last_sample_ms < 2000`) before transmitting, ensuring old or stale readings are not broadcast.

Data Fields

- **`id`**: Sensor ID.
- **`temperature`**: Temperature in centi-degrees Celsius (degC * 100).
- **`humidity`**: Humidity in centi-percent (\% * 100).

Practical Use Cases

1. **Icing Detection:**
   - *Scenario:* A UAV is flying in cold, damp conditions.
   - *Action:* The GCS monitors `temperature` and `humidity`. High humidity near 0°C indicates a high risk of carburetor icing (for gas engines) or wing icing. The pilot can descend or activate anti-ice systems.
2. **Meteorological Survey:**
   - *Scenario:* Weather profiling.
   - *Action:* The drone performs a vertical ascent. The GCS logs `HYGROMETER_SENSOR` vs Altitude to generate a skew-T log-P diagram of the atmosphere.

Key Codebase Locations

- **ArduPlane/GCS_Mavlink.cpp:473**: Implementation of the sender.

CONTROL

RC_CHANNELS_RAW (ID 35)

LEGACY / SUPPORTED

Summary

The `RC_CHANNELS_RAW` message provides raw PWM values (in microseconds) for the first 8 channels of the vehicle's radio receiver. While fully supported for backward compatibility, it is largely considered a legacy message in ArduPilot, having been superseded by `RC_CHANNELS` (65) which supports up to 18 channels.

Status

Legacy / Supported (Superseded by ID 65)

Directionality

- **TX (Transmit):** All Vehicles (Reports RC input to GCS/OSD)
- **RX (Receive):** None (Use `RC_CHANNELS_OVERRIDE` for GCS-based RC input)

Transmission (TX)

The transmission logic is in `GCS_MAVLINK::send_rc_channels_raw` within libraries/GCS_MAVLink/GCS_Common.cpp:2117.

Data Sourcing

- **RC System:** Data is retrieved via `rc().get_radio_in(values, 8)`.
- **Format:** Raw PWM values (e.g., 1100 to 1900) for channels 1 through 8.
- **RSSI:** The message includes a `rssi` field (0-255), representing the signal strength of the RC link.

Constraints

- **Channel Limit:** This message is strictly limited to **8 channels**.
- **MAVLink Versioning:** ArduPilot primarily uses this message for MAVLink 1.0 compatibility. On modern MAVLink 2.0 links, `RC_CHANNELS` (65) is the preferred method for streaming all 16+ channels.

Data Fields

- `time_boot_ms`: Timestamp (milliseconds since system boot).
- `port`: Servo output port (set of 8 outputs = 1 port). Flight stacks running on Pixhawk should use: 0 = Main, 1 = Aux.
- `chan1_raw`: RC channel 1 value.
- `chan2_raw`: RC channel 2 value.
- `chan3_raw`: RC channel 3 value.
- `chan4_raw`: RC channel 4 value.
- `chan5_raw`: RC channel 5 value.
- `chan6_raw`: RC channel 6 value.
- `chan7_raw`: RC channel 7 value.
- `chan8_raw`: RC channel 8 value.
- `rssi`: Receive signal strength indicator in device-dependent units/scale. Values: [0-254], 255: invalid/unknown.

Practical Use Cases

1. **Radio Calibration:**
    - *Scenario:* A user is setting up a new transmitter and needs to define the "Min" and "Max" stick positions.
    - *Action:* Mission Planner monitors `RC_CHANNELS_RAW` while the user moves the sticks, recording the extreme PWM values for each channel.
2. **Legacy OSD Display:**
    - *Scenario:* A pilot is using an old "MinimOSD" that only supports MAVLink 1.0.
    - *Action:* The OSD reads `RC_CHANNELS_RAW` to show a stick-position overlay or RSSI indicator on the video feed.
3. **Link Health Monitoring:**
    - *Scenario:* A developer wants to verify that their ELRS or Crossfire receiver is talking to the flight controller correctly.
    - *Action:* By checking for fluctuating PWM values in `RC_CHANNELS_RAW` in the GCS "Status" tab, the developer can confirm the digital link is alive.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2117**: Implementation of the send function.
- **libraries/RC_Channel/RC_Channels.cpp**: Source of raw PWM input data.

SERVO_OUTPUT_RAW (ID 36) SUPPORTED

Summary

The `SERVO_OUTPUT_RAW` message provides the actual PWM values (in microseconds) being sent to the vehicle's actuators (motors and servos). This represents the output of the flight controller's mixer logic and is crucial for verifying that the autopilot is commanding the hardware as expected.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports actuator state to GCS)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is implemented in `GCS_MAVLINK::send_servo_output_raw` within libraries/GCS_MAVLink/GCS_Common.cpp:3292.

Data Sourcing

- **Hardware Read:** PWM values are read directly from the hardware abstraction layer via `hal.rcout->read()`.
- **Filtering:** ArduPilot uses `SRV_Channels::get_output_channel_mask(SRV_Channel::k_GPIO)` to identify and exclude any pins currently configured as GPIOs, ensuring only actuator signals are reported.
- **Extended Channel Support:** While a single MAVLink `SERVO_OUTPUT_RAW` packet supports 16 channels, ArduPilot supports up to 32 outputs. It handles this by sending two packets:
  - **Port 0:** Channels 1-16.
  - **Port 1:** Channels 17-32.

Scheduling

- Sent as part of the `MSG_SERVO_OUTPUT_RAW` stream.
- Triggered in `GCS_Common.cpp:6245` within the `try_send_message` loop.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `port`: Servo output port (set of 8 outputs = 1 port). Flight stacks running on Pixhawk should use: 0 = Main, 1 = Aux.
- `servo1_raw`: Servo output 1 value.
- `servo2_raw`: Servo output 2 value.
- `servo3_raw`: Servo output 3 value.
- `servo4_raw`: Servo output 4 value.
- `servo5_raw`: Servo output 5 value.
- `servo6_raw`: Servo output 6 value.
- `servo7_raw`: Servo output 7 value.
- `servo8_raw`: Servo output 8 value.
- `servo9_raw`: Servo output 9 value.

- `servo10_raw`: Servo output 10 value.
- `servo11_raw`: Servo output 11 value.
- `servo12_raw`: Servo output 12 value.
- `servo13_raw`: Servo output 13 value.
- `servo14_raw`: Servo output 14 value.
- `servo15_raw`: Servo output 15 value.
- `servo16_raw`: Servo output 16 value.

Practical Use Cases

1. **Mixer Verification:**
    - *Scenario:* A builder has configured a complex V-Tail plane and wants to ensure the tail servos move correctly in response to elevator and rudder inputs.
    - *Action:* The builder observes `SERVO_OUTPUT_RAW` in the GCS while moving the sticks to verify the mixer output without propellers attached.
2. **Motor Saturation Check:**
    - *Scenario:* A heavy-lift Octocopter is struggling to maintain altitude.
    - *Action:* The pilot checks the `servo_raw` values. If multiple motors are consistently at `2000` (Max PWM), it indicates the vehicle is underpowered or overloaded.
3. **Actuator Health Monitoring:**
    - *Scenario:* A servo starts drawing excessive current or becomes jittery.
    - *Action:* Mission Planner graphs the output values. If the output is steady but the vehicle is unstable, it points to a mechanical failure in the actuator itself.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3292**: Implementation of `send_servo_output_raw`.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6245**: Scheduler logic.

NAV_CONTROLLER_OUTPUT (ID 62) SUPPORTED

Summary

The `NAV_CONTROLLER_OUTPUT` message provides the output of the vehicle's navigation controllers and its progress toward the current waypoint. It is a critical telemetry packet for the Ground Control Station's (GCS) HUD, as it provides "Desired" versus "Actual" navigation data, often visualized as a "Flight Director" or heading bug.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports navigation state)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is vehicle-specific, as each vehicle type (Plane, Copter, Rover) has unique navigation requirements.

ArduPlane Logic

Implemented in ArduPlane/GCS_Mavlink.cpp:201.

- `nav_roll` / `nav_pitch` : Sourced from the plane's internal navigation target angles (`nav_roll_cd` , `nav_pitch_cd`).
- `wp_dist` : Calculated distance to the active waypoint in meters.
- `alt_error` : Vertical distance from the target altitude in meters.
- `aspd_error` : Difference between target airspeed and current filtered airspeed (multiplied by 100).

ArduCopter Logic

Implemented in ArduCopter/GCS_Mavlink.cpp:206.

- `nav_bearing` : The bearing required to reach the next waypoint.
- `target_bearing` : The heading the vehicle is currently trying to maintain.
- `xtrack_error` : The cross-track error (perpendicular distance from the path) in meters.

Data Fields

- `nav_roll` **:** Current desired roll in degrees.
- `nav_pitch` **:** Current desired pitch in degrees.
- `nav_bearing` **:** Current desired heading in degrees.
- `target_bearing` **:** Bearing to current waypoint/target in degrees.
- `wp_dist` **:** Distance to active waypoint in meters.
- `alt_error` **:** Current altitude error in meters.
- `aspd_error` **:** Current airspeed error in meters/second.
- `xtrack_error` **:** Current crosstrack error on x-y plane in meters.

Practical Use Cases

1. **Flight Director HUD:**
    - *Scenario:* A pilot is flying a fixed-wing plane in AUTO mode.
    - *Action:* The GCS HUD displays a small circle (the "Flight Director") indicating the `nav_roll` and `nav_pitch` commanded by the autopilot. The pilot can see exactly where the drone *wants* to go compared to where it is currently pointing.
2. **Navigation Health Monitoring:**
    - *Scenario:* A drone is fighting high crosswinds.
    - *Action:* The pilot monitors `xtrack_error`. If the error keeps increasing despite the drone leaning into the wind, it indicates the wind speed exceeds the drone's tilt limits.
3. **Approach Visualization:**
    - *Scenario:* A Rover is following a path through a series of narrow gates.
    - *Action:* The GCS uses `wp_dist` to show a countdown timer or distance bar to the next gate, helping the operator anticipate turns.

Key Codebase Locations

- **ArduPlane/GCS_Mavlink.cpp:201**: Plane-specific navigation output.
- **ArduCopter/GCS_Mavlink.cpp:206**: Copter-specific navigation output.
- **Rover/GCS_Mavlink.cpp:103**: Rover-specific navigation output.

RC_CHANNELS (ID 65)

SUPPORTED / RECOMMENDED

Summary

The `RC_CHANNELS` message is the modern standard for streaming the vehicle's radio control input to a Ground Control Station (GCS). It supports up to 18 channels of data, providing high-resolution PWM values and a link quality indicator (RSSI). This message is used by GCS software for radio calibration and real-time stick visualization.

Status

Supported / Recommended

Directionality

- **TX (Transmit):** All Vehicles (Reports RC input to GCS/OSD)
- **RX (Receive):** None (Use `RC_CHANNELS_OVERRIDE` or `RADIO_RC_CHANNELS` for input)

Transmission (TX)

The transmission logic is in `GCS_MAVLINK::send_rc_channels` within libraries/GCS_MAVLink/GCS_Common.cpp:2081.

Data Sourcing

- **RC System:** Values are retrieved from the `RC_Channels` singleton using `get_radio_in()`.
- **Channel Count:** While the MAVLink message supports 18 channels, ArduPilot typically supports up to **16 channels** (`NUM_RC_CHANNELS`), which are mapped to the first 16 fields of the message.
- **RSSI:** Sourced from the receiver driver (e.g., CRSF, ELRS, SBUS) and mapped to the $0\text{-}255$ range.
- **Timestamp:** Uses `AP_HAL::millis()` since boot.

Stream Configuration

- Sent as part of the `MSG_RC_CHANNELS` stream.
- This message is active on modern MAVLink 2.0 links and is the preferred way to monitor full-range radio inputs (e.g., auxiliary switches, camera dials).

Data Fields

- **`time_boot_ms`**: Timestamp (milliseconds since system boot).
- **`chancount`**: Total number of RC channels being received.
- **`chan1_raw`**: RC channel 1 value.
- **`chan2_raw`**: RC channel 2 value.
- **`chan3_raw`**: RC channel 3 value.
- **`chan4_raw`**: RC channel 4 value.
- **`chan5_raw`**: RC channel 5 value.
- **`chan6_raw`**: RC channel 6 value.
- **`chan7_raw`**: RC channel 7 value.
- **`chan8_raw`**: RC channel 8 value.
- **`chan9_raw`**: RC channel 9 value.
- **`chan10_raw`**: RC channel 10 value.
- **`chan11_raw`**: RC channel 11 value.
- **`chan12_raw`**: RC channel 12 value.

- `chan13_raw`: RC channel 13 value.
- `chan14_raw`: RC channel 14 value.
- `chan15_raw`: RC channel 15 value.
- `chan16_raw`: RC channel 16 value.
- `chan17_raw`: RC channel 17 value.
- `chan18_raw`: RC channel 18 value.
- `rssi`: Receive signal strength indicator in device-dependent units/scale. Values: [0-254], 255: invalid/unknown.

Practical Use Cases

1. **Full System Calibration:**
    - *Scenario:* A user is configuring a complex cinema drone with 12 auxiliary switches for camera tilt, zoom, and mode selection.
    - *Action:* Mission Planner uses `RC_CHANNELS` to show the movement of all 12 channels simultaneously in the "Mandatory Hardware" setup screen.
2. **Telemetry-based Stick Display:**
    - *Scenario:* A pilot wants to record their stick movements for a YouTube tutorial.
    - *Action:* An OSD or GCS records the `RC_CHANNELS` stream, allowing the pilot to overlay a "Stick Cam" on the final video using only telemetry logs.
3. **Radio Link Health (RSSI):**
    - *Scenario:* A pilot is flying a long-distance mission.
    - *Action:* The GCS HUD displays the `rssi` field as a percentage. If the value drops significantly, the pilot knows they are approaching the limit of their radio range.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2081**: Implementation of the send function.
- **libraries/RC_Channel/RC_Channel.h:14**: Defines `NUM_RC_CHANNELS`.

RC_CHANNELS_OVERRIDE (ID 70) SUPPORTED

Summary

The `RC_CHANNELS_OVERRIDE` message allows a Ground Control Station (GCS) to emulate a physical radio transmitter. By sending this message, a GCS can take direct control of the vehicle's sticks and switches, bypassing the physical RC receiver. This is the primary mechanism for flying drones using USB Joysticks, Gamepads, or automated ground-based control logic.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Enables GCS control)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_rc_channels_override` in libraries/GCS_MAVLink/GCS_Common.cpp:3994.

Application Logic

1. **Security:** ArduPilot only accepts overrides from the System ID configured as "My GCS" (`SYSID_MYGCS`).
2. **Channel Handling:**
    - **Value 1000-2000:** Sets the raw PWM value for that channel.
    - **Value 0 or 65535 (`UINT16_MAX`):** "Ignore". The current override for this channel is maintained, or it remains on physical RC.
    - **Value 65534:** "Release". Forces this specific channel to revert to the physical radio input.
3. **Persistence:** Overrides are applied to the `RC_Channels` library and remain active until a timeout occurs or they are explicitly released.

Timeout and Failsafe

ArduPilot implements a safety watchdog for overrides in `RC_Channel::has_override()` (libraries/RC_Channel/RC_Channel.cpp:510).

- **Heartbeat:** If no new `RC_CHANNELS_OVERRIDE` message is received within the time defined by the `RC_OVERRIDE_TIME` parameter (default 3 seconds), the autopilot automatically releases all overrides.
- **Failsafe:** If no physical RC receiver is present and the GCS override times out, the vehicle triggers an **RC Failsafe** (typically RTL or Land).

Data Fields

- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`chan1_raw`:** RC channel 1 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- **`chan2_raw`:** RC channel 2 value, in microseconds. A value of UINT16_MAX means to ignore this field.

- `chan3_raw`: RC channel 3 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan4_raw`: RC channel 4 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan5_raw`: RC channel 5 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan6_raw`: RC channel 6 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan7_raw`: RC channel 7 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan8_raw`: RC channel 8 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan9_raw`: RC channel 9 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan10_raw`: RC channel 10 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan11_raw`: RC channel 11 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan12_raw`: RC channel 12 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan13_raw`: RC channel 13 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan14_raw`: RC channel 14 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan15_raw`: RC channel 15 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan16_raw`: RC channel 16 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan17_raw`: RC channel 17 value, in microseconds. A value of UINT16_MAX means to ignore this field.
- `chan18_raw`: RC channel 18 value, in microseconds. A value of UINT16_MAX means to ignore this field.

Practical Use Cases

1. **Joystick Flying:**
   - *Scenario:* A pilot wants to fly a drone using an Xbox controller connected to their laptop.
   - *Action:* Mission Planner reads the controller inputs and streams `RC_CHANNELS_OVERRIDE` messages at 10Hz-25Hz to the drone.
2. **GCS-Triggered Actions:**
   - *Scenario:* A search-and-rescue team uses a button on their GCS dashboard to "Drop Payload".
   - *Action:* The GCS sends a single `RC_CHANNELS_OVERRIDE` packet for Channel 8 with a value of `2000` to trigger the gripper.
3. **Automated Landing Logic:**
   - *Scenario:* An external vision system is performing a precision landing on a moving platform.
   - *Action:* The vision system sends micro-adjustments to the roll/pitch channels via overrides to center the drone over the target.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3994**: Entry point for override handling.
- **libraries/RC_Channel/RC_Channel.cpp:510**: Implementation of the timeout/failsafe logic.
- **libraries/RC_Channel/RC_Channel.h**: Defines the override state storage.

COMMAND_LONG (ID 76)

SUPPORTED (CRITICAL)

Summary

The `COMMAND_LONG` message is the most versatile and critical control packet in the MAVLink protocol. It is used to trigger one-off actions or transition the vehicle into complex states. Commands range from basic operations like Arming and Disarming to advanced functions like initiating a Compass Calibration or triggering a camera shutter. ArduPilot processes these by mapping the message to an internal `MAV_CMD` dispatch system.

Status

Supported (Critical)

Directionality

- **TX (Transmit):** All Vehicles (Used to control external components like Gimbals/Cameras)
- **RX (Receive):** All Vehicles (Main entry point for GCS-initiated actions)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_command_long` in libraries/GCS_MAVLink/GCS_Common.cpp:5176.

Execution Flow

1. **Scripting Override:** If a Lua script has registered for a specific command (e.g., a custom payload trigger), ArduPilot's C++ handler yields to the script.
2. **Unification:** ArduPilot converts `COMMAND_LONG` (float params) into an internal `COMMAND_INT` (integer params) format via `try_command_long_as_command_int` (GCS_Common.cpp:5125) to ensure consistent processing across both message types.
3. **Dispatch:** The command is routed to `handle_command_int_packet`. Vehicle-specific logic (e.g., ArduCopter/GCS_Mavlink.cpp:758) first attempts to handle vehicle-specific commands (like `TAKEOFF` or `LAND`). If unhandled, it falls back to the common handler in `GCS_MAVLINK`.
4. **Acknowledgement:** Every `COMMAND_LONG` **must** be acknowledged. ArduPilot sends a `COMMAND_ACK` (77) back to the GCS with the result of the operation.

Transmission (TX)

ArduPilot acts as a controller for peripheral devices using `COMMAND_LONG`.

- **Gimbal Control:** ArduPilot sends `MAV_CMD_DO_MOUNT_CONTROL` to move a MAVLink-enabled gimbal.
- **Camera Integration:** It sends `MAV_CMD_DO_DIGICAM_CONTROL` to trigger photo capture on external MAVLink cameras.

Data Fields

- **`target_system`**: System which should execute the command.
- **`target_component`**: Component which should execute the command, 0 for all components.
- **`command`**: Command ID (`MAV_CMD`).
- **`confirmation`**: 0: First transmission of this command. 1-255: Confirmation transmissions (e.g. for kill command).
- **`param1`**: Parameter 1 (for the specific command).

- `param2`: Parameter 2 (for the specific command).
- `param3`: Parameter 3 (for the specific command).
- `param4`: Parameter 4 (for the specific command).
- `param5`: Parameter 5 (for the specific command).
- `param6`: Parameter 6 (for the specific command).
- `param7`: Parameter 7 (for the specific command).

Practical Use Cases

1. **Arming the Vehicle:**
   - *Scenario:* A pilot is ready for takeoff and clicks "Arm" in the GCS.
   - *Action:* The GCS sends `COMMAND_LONG` with `command = MAV_CMD_COMPONENT_ARM_DISARM (400)` and `param1 = 1`. ArduPilot verifies safety checks and arms the motors.
2. **Triggering a Reboot:**
   - *Scenario:* A user has updated a critical parameter that requires a power cycle.
   - *Action:* The GCS sends `COMMAND_LONG` with `command = MAV_CMD_PREFLIGHT_REBOOT_SHUTDOWN (246)`. ArduPilot saves logs and restarts the MCU.
3. **Emergency Return-to-Launch:**
   - *Scenario:* The pilot loses orientation and needs the drone to come back.
   - *Action:* Clicking the "RTL" button sends `COMMAND_LONG` with `command = MAV_CMD_NAV_RETURN_TO_LAUNCH (20)`.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5176**: Main command entry point.
- **libraries/GCS_MAVLink/GCS_Common.cpp:5415**: Common `MAV_CMD` dispatch switch.
- **ArduCopter/GCS_Mavlink.cpp:758**: Copter-specific command overrides.

COMMAND_ACK (ID 77)

SUPPORTED (CRITICAL)

Summary

The `COMMAND_ACK` message is the response to a `COMMAND_LONG` or `COMMAND_INT`. It informs the sender whether the requested command was accepted, denied, or if it failed during execution. For long-running tasks, it can also provide progress updates. This message is critical for any Ground Control Station to provide reliable feedback to the user after they click a button or send a command.

Status

Supported (Critical)

Directionality

- **TX (Transmit):** All Vehicles (Confirming results to GCS)
- **RX (Receive):** Specific Subsystems (Receiving ACKs from external components like Gimbals)

Transmission (TX)

Transmission happens automatically after a command is processed by `GCS_MAVLINK::handle_command_long` or `handle_command_int`.

Result Types

ArduPilot uses the following standard `MAV_RESULT` values:

- `MAV_RESULT_ACCEPTED` **(0):** Command was valid and has been executed (or started).
- `MAV_RESULT_TEMPORARILY_REJECTED` **(1):** Command is valid but cannot be executed now (e.g., trying to Arm while the vehicle is already armed).
- `MAV_RESULT_DENIED` **(2):** Command is invalid or the vehicle is in a state that prohibits it (e.g., Takeoff while disarmed).
- `MAV_RESULT_UNSUPPORTED` **(3):** The requested `MAV_CMD` is not implemented in ArduPilot.
- `MAV_RESULT_FAILED` **(4):** The command was accepted but failed to complete (e.g., a motor failed to spin up during an arming attempt).
- `MAV_RESULT_IN_PROGRESS` **(5):** The command is being processed. ArduPilot may send subsequent ACKs with a `progress` value (0-100\%).

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_command_ack` in libraries/GCS_MAVLink/GCS_Common.cpp:3978.

Usage

ArduPilot listens for ACKs when it acts as a controller for other MAVLink devices.

- **Accel Calibration:** Used in libraries/AP_AccelCal/AP_AccelCal.cpp to verify that external sensors have acknowledged calibration steps.
- **Gimbal Control:** ArduPilot verifies that an external MAVLink gimbal has accepted its orientation commands.

Data Fields

- `command` : Command ID (`MAV_CMD`).
- `result` : Result code (`MAV_RESULT`).
- `progress` : Progress (0 to 100, or 255 if not supported).
- `result_param2` : Additional parameter for result (optional).
- `target_system` : System ID of the target.
- `target_component` : Component ID of the target.

Practical Use Cases

1. **GCS UI Feedback:**
   - *Scenario:* A user clicks "Arm".
   - *Action:* The GCS displays "Arming..." and waits for the `COMMAND_ACK`. If it receives `ACCEPTED`, the UI turns Red (Armed). If it receives `TEMPORARILY_REJECTED`, the GCS shows a popup: "Pre-arm checks failed".
2. **Automated Scripts:**
   - *Scenario:* A Python script is performing an automated test of the camera trigger.
   - *Action:* The script sends `MAV_CMD_DO_DIGICAM_CONTROL` and blocks until it receives a `COMMAND_ACK`. This ensures the script doesn't proceed faster than the hardware can respond.
3. **Calibration Progress:**
   - *Scenario:* A user is performing a Compass Calibration.
   - *Action:* ArduPilot sends `COMMAND_ACK` with `MAV_RESULT_IN_PROGRESS` and the `progress` field updated (e.g., 10\%, 20\%...) as the user rotates the vehicle.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3978**: Reception handler.
- **libraries/GCS_MAVLink/GCS_Common.cpp:5200**: Typical location where `mavlink_msg_command_ack_send` is called after a command is processed.
- **libraries/AP_AccelCal/AP_AccelCal.cpp**: Example of RX handling.

ATTITUDE_TARGET (ID 83)

SUPPORTED

Summary

The `ATTITUDE_TARGET` message provides the "Desired" orientation of the vehicle, as commanded by the flight controller's navigation and position loops. By comparing this message against the actual orientation (reported in `ATTITUDE` or `ATTITUDE_QUATERNION`), Ground Control Stations can visualize how well the vehicle is tracking its commanded path and tuning parameters.

Status

Supported

Directionality

- **TX (Transmit):** Copter, Plane (QuadPlane only), Blimp (Reports desired targets)
- **RX (Receive):** None (Use `SET_ATTITUDE_TARGET` (82) for external input)

Transmission (TX)

The transmission logic is implemented in vehicle-specific GCS classes.

Copter Implementation

Located in ArduCopter/GCS_Mavlink.cpp:86.

- **Data Source:** The `AC_AttitudeControl` library provides the internal targets.
- **Format:**
  - `q`: The target orientation as a unit quaternion.
  - `body_roll_rate`, `body_pitch_rate`, `body_yaw_rate`: The target angular velocities in rad/s.
  - `thrust`: The normalized target thrust (0 to 1).

Plane Implementation

Located in ArduPlane/GCS_Mavlink.cpp:160.

- This message is typically only sent when the plane is in a VTOL (Vertical Take-Off and Landing) mode (QuadPlane), where the attitude controller's targets are relevant for hover stability.

Data Fields

- `time_boot_ms`: Timestamp (milliseconds since system boot).
- `type_mask`: Bitmap to indicate which dimensions should be ignored by the vehicle.
- `q`: Attitude quaternion (w, x, y, z order, zero-rotation is 1, 0, 0, 0).
- `body_roll_rate`: Body roll rate (rad/s).
- `body_pitch_rate`: Body pitch rate (rad/s).
- `body_yaw_rate`: Body yaw rate (rad/s).
- `thrust`: Collective thrust, normalized to 0 .. 1 (-1 .. 1 for vehicles capable of reverse trust).

Practical Use Cases

1. **PID Tuning Visualization:**

    - *Scenario:* A user is performing manual "AutoTune" and wants to see if the drone is responsive enough.
    - *Action:* The GCS graphs "Actual Roll" vs. "Target Roll" (from `ATTITUDE_TARGET`). If the actual roll lags significantly behind the target, the user knows to increase the P-gain.
2. **Navigation Health Monitoring:**
    - *Scenario:* A drone is flying in extremely high winds.
    - *Action:* The pilot notices the "Desired Tilt" (Target) is at 30 degrees, but the drone is only tilting 20 degrees. This indicates a physical limitation or motor saturation.
3. **Simulation Verification:**
    - *Scenario:* A developer is testing a new navigation algorithm in SITL.
    - *Action:* By logging `ATTITUDE_TARGET`, the developer can verify that the algorithm is generating smooth, mathematically sound setpoints before they are passed to the motors.

Key Codebase Locations

- **ArduCopter/GCS_Mavlink.cpp:86**: Copter transmission logic.
- **ArduPlane/GCS_Mavlink.cpp:160**: Plane transmission logic.
- **libraries/AC_AttitudeControl/AC_AttitudeControl.h**: Internal source of target orientation.

MISSION

MISSION_ITEM (ID 39)

LEGACY / SUPPORTED

Summary

The `MISSION_ITEM` message is the legacy mechanism for uploading and downloading autonomous mission waypoints using floating-point coordinates. While fully supported for backward compatibility, it is largely superseded by `MISSION_ITEM_INT` (73), which uses integer coordinates to avoid precision loss over long distances. ArduPilot unifies both formats internally to high-precision integer representation.

Status

Legacy / Supported (Superseded by ID 73)

Directionality

- **TX (Transmit):** All Vehicles (Downloading mission to GCS)
- **RX (Receive):** All Vehicles (Uploading mission from GCS)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_mission_item` in libraries/GCS_MAVLink/GCS_Common.cpp:915.

Conversion Logic

Upon receiving a `MISSION_ITEM` packet, ArduPilot immediately converts it to the integer format using `AP_Mission::convert_MISSION_ITEM_to_MISSION_ITEM_INT` in libraries/AP_Mission/AP_Mission.cpp:1498. This ensures that the mission is stored with the highest possible fidelity in the vehicle's EEPROM/Flash.

Transmission (TX)

During a mission download, ArduPilot will send `MISSION_ITEM` messages if the GCS requested them via a `MISSION_REQUEST` (40).

- **Logic:** The `MissionItemProtocol` state machine retrieves the command from `AP_Mission`, converts it to float-scale, and sends the packet.

Data Fields

- `target_system` : System ID.
- `target_component` : Component ID.
- `seq` : Sequence.
- `frame` : The coordinate system of the waypoint (`MAV_FRAME`).
- `command` : The scheduled action for the waypoint (`MAV_CMD`).
- `current` : false:0, true:1.
- `autocontinue` : autocontinue to next wp.
- `param1` : PARAM1, see MAV_CMD enum.
- `param2` : PARAM2, see MAV_CMD enum.
- `param3` : PARAM3, see MAV_CMD enum.
- `param4` : PARAM4, see MAV_CMD enum.
- `x` : PARAM5 / local: x position in meters * 1e4, global: latitude in degrees * 10^7.
- `y` : PARAM6 / local: y position in meters * 1e4, global: longitude in degrees * 10^7.
- `z` : PARAM7 / local: z position: altitude in meters (relative or absolute, depending on frame).

Practical Use Cases

1. **Legacy Script Support:**
   - *Scenario:* A researcher has a legacy DroneKit script that defines waypoints using floats.
   - *Action:* ArduPilot accepts the `MISSION_ITEM` messages and transparently converts them for the internal flight controller.
2. **Simple GCS Implementation:**
   - *Scenario:* A developer is building a minimal GCS and finds float math easier to implement than 1E7 integer scaling.
   - *Action:* The developer uses `MISSION_ITEM` to send simple waypoint coordinates.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:915**: Entry point for mission item handling.
- **libraries/AP_Mission/AP_Mission.cpp:1498**: Unification of Float and Integer coordinates.

MISSION_SET_CURRENT (ID 41)

SUPPORTED

Summary

The `MISSION_SET_CURRENT` message allows a Ground Control Station (GCS) to manually override the vehicle's progress through an autonomous mission. It can be used to jump ahead to a specific waypoint, restart the mission from the beginning, or skip a section of the flight path.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Manually sets mission index)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_mission_set_current` in libraries/GCS_MAVLink/GCS_Common.cpp:710.

Processing Logic

1. **Index Update:** It calls `AP_Mission::set_current_cmd(index)`.
2. **Execution Jump:**
   - If the mission is active, it calls `advance_current_nav_cmd(index)`, which forces the flight controller to stop its current navigation goal and immediately track toward the new waypoint.
   - If the mission is stopped/disarmed, it primes the mission engine to resume from that index upon the next mission start.
3. **Special Markers:** It supports jumping to the `MAV_CMD_DO_LAND_START` marker, which is commonly used during abort procedures to find the start of the landing sequence.
4. **Confirmation:** ArduPilot immediately responds with a `MISSION_CURRENT` (42) message echoing the new index.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `seq`: Sequence.

Practical Use Cases

1. **Waypoint Skipping:**
   - *Scenario:* A pilot is flying a survey mission and notices a localized cloud over waypoint 5.
   - *Action:* The pilot uses the "Set Current Waypoint" feature in the GCS to jump to waypoint 6, skipping the obscured area.
2. **Mission Restart:**
   - *Scenario:* A battery fail-safe triggered an early return. After swapping batteries, the pilot wants to start the mission from waypoint 1 again.
   - *Action:* The GCS sends `MISSION_SET_CURRENT` with index `0`.
3. **Emergency Landing Search:**
   - *Scenario:* A fixed-wing plane needs to land immediately due to high winds.

    - *Action:* The GCS triggers a jump to the "Land Start" index, allowing the plane to follow its pre-planned landing approach.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:710**: Entry point for setting the current item.
- **libraries/AP_Mission/AP_Mission.cpp:586**: Core mission state management for index changes.

MISSION_CURRENT (ID 42)

SUPPORTED

Summary

The `MISSION_CURRENT` message is the vehicle's report of its current progress through the active mission. It specifies the sequence number (index) of the waypoint or command the vehicle is currently executing.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports current mission index)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission logic is in `GCS_MAVLINK::send_mission_current` within libraries/GCS_MAVLink/GCS_Common.cpp:678.

Data Sourcing

- **Source:** The index is retrieved from the `AP_Mission` library via `get_current_nav_index()`.
- **Trigger:**
    1. **Periodic:** Sent as part of the `MSG_CURRENT_WAYPOINT` stream (typically 1Hz).
    2. **Event-Driven:** Sent immediately after a `MISSION_SET_CURRENT` (41) command is processed.
    3. **Autonomous:** Sent whenever the mission engine automatically advances to the next item in the list.

Data Fields

- `seq`: Sequence.

Practical Use Cases

1. **Mission Progress Visualization:**
    - *Scenario:* A pilot is watching the drone fly a complex path on a ground station.
    - *Action:* The GCS highlights the "active" waypoint on the map based on the index received in `MISSION_CURRENT`.
2. **Autonomous Event Triggers:**
    - *Scenario:* A photographer wants to be notified when the drone reaches waypoint 10 to start a manual video sequence.
    - *Action:* An app listens for `MISSION_CURRENT` and triggers a sound alert on the pilot's phone when `seq == 10`.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:678**: Implementation of `send_mission_current`.
- **libraries/AP_Mission/AP_Mission.h**: Provides access to the mission state.

MISSION_REQUEST_LIST (ID 43) SUPPORTED

Summary

The `MISSION_REQUEST_LIST` message is the standard way for a Ground Control Station (GCS) to initiate a mission download. Upon receiving this request, ArduPilot determines the number of stored mission items and responds with a `MISSION_COUNT` (44) message.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Triggers mission download handshake)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_mission_request_list` in libraries/GCS_MAVLink/GCS_Common.cpp:607.

Processing Logic

1. **Protocol Identification:** The message is routed to the appropriate `MissionItemProtocol` handler (e.g., Waypoints, Rally points, or Fences).
2. **Count Retrieval:** ArduPilot queries the `AP_Mission` library for the total number of items currently stored in the requested list.
3. **Response:** It immediately sends a `MISSION_COUNT` (44) packet back to the requester.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.

Practical Use Cases

1. **Post-Connect Synchronization:**
    - *Scenario:* A pilot connects a tablet to a drone that was previously flown.
    - *Action:* The tablet sends `MISSION_REQUEST_LIST` to see if there is an active mission still stored on the vehicle.
2. **Mission Verification:**
    - *Scenario:* After uploading a new route, the GCS wants to double-check the onboard state.
    - *Action:* The GCS requests the list to begin a verification download.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:607**: Entry point for list requests.
- **libraries/GCS_MAVLink/MissionItemProtocol.cpp**: Core state machine logic for responding to list requests.

MISSION_COUNT (ID 44)

SUPPORTED

Summary

The `MISSION_COUNT` message is a critical handshake packet used at the beginning of both mission uploads and downloads. It defines how many items are about to be transferred, allowing the receiver to allocate memory and prepare for the sequence of individual waypoint messages.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Starting a download)
- **RX (Receive):** All Vehicles (Starting an upload)

Transmission (TX)

ArduPilot sends `MISSION_COUNT` in response to a `MISSION_REQUEST_LIST` (43). This tells the GCS exactly how many waypoints are currently stored in the vehicle's memory.

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_mission_count` in libraries/GCS_MAVLink/GCS_Common.cpp:741.

Processing Logic

1. **Preparation:** The autopilot receives this message from a GCS wanting to upload a new mission.
2. **Clearance:** ArduPilot calls `truncate()` on the internal mission storage. This effectively clears existing items of that type to make room for the new list.
3. **Resource Allocation:** The mission protocol state machine is initialized to expect the specified number of items.
4. **Handshake Continuation:** ArduPilot responds by sending the first `MISSION_REQUEST` (40) for index 0.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `count`: Number of mission items in the sequence.

Practical Use Cases

1. **Uploading a Route:**
   - *Scenario:* A user draws a 10-point circle on a map in Mission Planner and clicks "Write WPs".
   - *Action:* Mission Planner sends `MISSION_COUNT` with value `10`. ArduPilot clears its mission memory and begins requesting the 10 points.
2. **Telemetry Resumption:**
   - *Scenario:* A link was temporarily lost during a mission download.
   - *Action:* The GCS re-sends the count request to re-verify the list size before continuing.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:741**: Entry point for mission count handling.
- **libraries/GCS_MAVLink/MissionItemProtocol.cpp**: Implements the `truncate` and `truncate` logic.

MISSION_CLEAR_ALL (ID 45)

SUPPORTED

Summary

The `MISSION_CLEAR_ALL` message allows a Ground Control Station (GCS) to wipe a specific set of autonomous instructions from the vehicle's memory. This is typically used to reset the drone to a clean state before uploading a new mission or to ensure no old geofence boundaries remain active.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Clears stored mission/fence/rally items)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_mission_clear_all` in libraries/GCS_MAVLink/GCS_Common.cpp:763.

Processing Logic

1. **Target Selection:** The message contains a `mission_type` field. ArduPilot uses this to determine whether to clear the Main Mission, the Geo-Fence, or the Rally points.
2. **Safety First:** Before clearing, ArduPilot cancels any active mission upload sessions to prevent the system from entering an inconsistent state.
3. **Scoped Wipe:** Only the requested list is cleared. The rest of the vehicle's configuration (parameters, calibration data) remains untouched in the EEPROM.
4. **Acknowledgement:** ArduPilot responds with a `MISSION_ACK` (47) message. If successful, it returns `MAV_MISSION_ACCEPTED`.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `mission_type`: Mission type (`MAV_MISSION_TYPE`).

Practical Use Cases

1. **New Mission Prep:**
   - *Scenario:* A user finishes a survey at Site A and drives to Site B.
   - *Action:* The GCS sends `MISSION_CLEAR_ALL` to remove the Site A waypoints before the user starts drawing a new flight plan.
2. **Fence Removal:**
   - *Scenario:* A pilot is flying in a restricted area with a temporary geofence. After the restriction is lifted, they want to fly freely.
   - *Action:* The GCS clears the fence type, allowing the drone to fly past the previous boundaries.
3. **Automation Cleanup:**
   - *Scenario:* A custom script is used to generate dynamic missions.
   - *Action:* At the end of each flight, the script clears the mission to ensure the drone doesn't accidentally restart the same route on the next takeoff.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:763**: Entry point for the clear command.
- **libraries/GCS_MAVLink/MissionItemProtocol.cpp:32**: Implements the clearing logic and ACK generation.

MISSION_ITEM_REACHED (ID 46)

SUPPORTED

Summary

The `MISSION_ITEM_REACHED` message is an event-driven notification sent by the autopilot to the Ground Control Station (GCS). It announces that a specific mission item (waypoint, takeoff, landing, or action) has been successfully completed. This is the primary signal used by GCS software to update progress bars and provide audio feedback to the pilot (e.g., "Waypoint 5 reached").

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports completion events)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is in `GCS::send_mission_item_reached_message` within libraries/GCS_MAVLink/GCS_Common.cpp:2717.

Trigger Logic

This message is not sent periodically. Instead, it is triggered by the mission engine:

1. **Verification:** The `AP_Mission` library calls a vehicle-specific `verify_command()` function (e.g., in ArduCopter/mode_auto.cpp) at high frequency (10Hz+).
2. **Completion:** When the vehicle determines it has fulfilled the criteria for the current item (e.g., it is within the waypoint radius and has finished any required loiter time), `verify_command` returns `true`.
3. **Dispatch:** Upon completion, the autopilot immediately broadcasts `MISSION_ITEM_REACHED` with the index of the finished item.

Scope

- **Navigation Commands:** Sent for waypoints, land, takeoff, and return-to-launch.
- **DO Commands:** Sent for non-movement actions (e.g., `MAV_CMD_DO_SET_SERVO`), which typically complete instantly upon execution.

Data Fields

- **`seq`:** Sequence.

Practical Use Cases

1. **GCS Audio Alerts:**
   - *Scenario:* A pilot is flying a survey mission and isn't looking at the screen.
   - *Action:* Mission Planner receives the message and uses Text-to-Speech to announce "Waypoint 10 reached. Returning home."
2. **Automated Data Logging:**
   - *Scenario:* A researcher wants to log exactly when a drone entered a specific study area.

    - *Action:* An external script listens for `MISSION_ITEM_REACHED`. When the index matches the boundary waypoint, it saves a timestamp and GPS coordinate.
3. **Companion Computer Actions:**
    - *Scenario:* A drone is delivering a package.
    - *Action:* When the companion computer sees the "Land" waypoint index reached, it triggers the gripper mechanism to release the payload.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2717**: Implementation of the broadcast function.
- **ArduCopter/mode_auto.cpp**: Typical trigger location for Copter missions.
- **ArduPlane/commands_logic.cpp**: Typical trigger location for Plane missions.

MISSION_ACK (ID 47)

SUPPORTED

Summary

The `MISSION_ACK` message is the terminal handshake packet in the MAVLink mission protocol. It signals the final result of a multi-message operation, such as uploading a new waypoint list or clearing the mission memory. Ground Control Stations (GCS) use this message to determine if a "Write" operation was successful or if an error occurred (e.g., the drone's memory is full).

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Confirming result to GCS)
- **RX (Receive):** None (Received but ignored by ArduPilot)

Transmission (TX)

The transmission logic is centered in `MissionItemProtocol::send_mission_ack` within libraries/GCS_MAVLink/MissionItemProtocol.cpp:334.

Response Types

ArduPilot uses the `type` field to communicate specific results:

- `MAV_MISSION_ACCEPTED` **(0):** The operation was successful (e.g., all waypoints received and saved).
- `MAV_MISSION_ERROR` **(1):** A generic failure occurred.
- `MAV_MISSION_UNSUPPORTED` **(3):** The requested mission type or command is not supported.
- `MAV_MISSION_NO_SPACE` **(4):** The vehicle has run out of EEPROM/Flash storage for mission items.
- `MAV_MISSION_INVALID` **(5):** One or more parameters in the uploaded mission items are invalid (e.g., unrealistic coordinates).
- `MAV_MISSION_INVALID_SEQUENCE` **(13):** Items were received out of order during an upload.

Reception (RX)

ArduPilot receives `MISSION_ACK` from the GCS after providing a mission for download.

- **Handling:** The message is decoded in `GCS_Common.cpp:4566`, but it is marked as "not used" and discarded. ArduPilot considers its job done once the last mission item is sent and does not require verification from the GCS.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `type`: Mission result (`MAV_MISSION_RESULT`).
- `mission_type`: Mission type (`MAV_MISSION_TYPE`).

Practical Use Cases

1. **Ensuring Reliable Uploads:**

    - *Scenario:* A user uploads 50 waypoints over a weak telemetry link.
    - *Action:* The GCS waits for the final `MISSION_ACK` with `type=0` before showing the "Upload Complete" dialog. If it receives an error, it prompts the user to "Retry".
2. **Storage Capacity Warnings:**
    - *Scenario:* A developer attempts to upload a 2000-point mission to an older flight controller.
    - *Action:* ArduPilot sends a `MISSION_ACK` with `MAV_MISSION_NO_SPACE`, allowing the GCS to explain why the full mission wasn't saved.
3. **Protocol Verification:**
    - *Scenario:* A companion computer clears the onboard fence.
    - *Action:* The computer listens for the ACK to verify the fence has indeed been deactivated before proceeding with a flight.

Key Codebase Locations

- **libraries/GCS_MAVLink/MissionItemProtocol.cpp:334**: Core logic for result reporting.
- **libraries/GCS_MAVLink/GCS_Common.cpp:4566**: Dispatcher that discards incoming ACKs.

MISSION_ITEM_INT (ID 73)

SUPPORTED / RECOMMENDED

Summary

The `MISSION_ITEM_INT` message is the modern standard for transmitting mission waypoints in MAVLink. It uses integer values for Latitude and Longitude (scaled by 1E7) to ensure high spatial precision, even on global scales. This is ArduPilot's preferred message for all mission planning activities.

Status

Supported / Recommended

Directionality

- **TX (Transmit):** All Vehicles (Downloading mission to GCS)
- **RX (Receive):** All Vehicles (Uploading mission from GCS)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_mission_item` in libraries/GCS_MAVLink/GCS_Common.cpp:915.

Internal Storage

- **Direct Mapping:** Since ArduPilot uses integer coordinates internally, `MISSION_ITEM_INT` messages are decoded directly into the internal mission representation without the precision loss associated with float conversions.
- **Storage:** Items are stored in the vehicle's non-volatile memory via the `AP_Mission` library.

Transmission (TX)

During a mission download, modern Ground Control Stations (like Mission Planner or QGroundControl) will request items using `MISSION_REQUEST_INT` (51). ArduPilot responds with `MISSION_ITEM_INT` packets.

Data Fields

- `target_system` : System ID.
- `target_component` : Component ID.
- `seq` : Sequence.
- `frame` : The coordinate system of the waypoint (`MAV_FRAME`).
- `command` : The scheduled action for the waypoint (`MAV_CMD`).
- `current` : false:0, true:1.
- `autocontinue` : autocontinue to next wp.
- `param1` : PARAM1, see MAV_CMD enum.
- `param2` : PARAM2, see MAV_CMD enum.
- `param3` : PARAM3, see MAV_CMD enum.
- `param4` : PARAM4, see MAV_CMD enum.
- `x` : PARAM5 / local: x position in meters * 1e4, global: latitude in degrees * 10^7.
- `y` : PARAM6 / local: y position in meters * 1e4, global: longitude in degrees * 10^7.
- `z` : PARAM7 / local: z position: altitude in meters (relative or absolute, depending on frame).
- `mission_type` : Mission type (`MAV_MISSION_TYPE`).

Practical Use Cases

1. **Professional Mission Planning:**
    - *Scenario:* A surveyor is planning an automated grid for a 1cm/pixel orthomosaic map.
    - *Action:* The GCS uses `MISSION_ITEM_INT` to ensure the waypoint coordinates are accurate to within 1.1cm on the Earth's surface.
2. **Long Range Navigation:**
    - *Scenario:* A fixed-wing drone is flying a 50km out-and-back mission.
    - *Action:* Integer coordinates prevent the "coordinate drift" that can occur in floating-point systems when coordinates have many digits before the decimal point.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:915**: Unified handler for float and integer mission items.
- **libraries/GCS_MAVLink/MissionItemProtocol_Waypoints.cpp**: Implements the waypoint-specific transfer logic.

TERRAIN_REQUEST (ID 133)

SUPPORTED

Summary

The `TERRAIN_REQUEST` message is part of the ArduPilot Terrain Protocol. It is sent by the vehicle to the Ground Control Station (GCS) to request digital elevation data (DEM) for a specific geographic area. The GCS responds by sending `TERRAIN_DATA` (134) messages containing the requested grid blocks.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Requests map data)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is in `AP_Terrain::request_missing` within libraries/AP_Terrain/TerrainGCS.cpp:72.

Protocol Logic

1. **Grid Management:** The `AP_Terrain` library divides the world into 4×4 grids of terrain heights.
2. **Bitmap Check:** It maintains a bitmap of which grids it has loaded. If the vehicle is flying towards an area where data is missing (and not on the SD card), it initiates a request.
3. **Masking:** The `mask` field tells the GCS exactly which 4×4 blocks within a larger tile are missing, optimizing bandwidth.

Data Fields

- `lat`: Latitude of grid (deg * 1E7).
- `lon`: Longitude of grid (deg * 1E7).
- `grid_spacing`: Grid spacing (in meters).
- `mask`: Bitmask of requested 4×4 blocks within the 8×7 grid.

Practical Use Cases

1. **Terrain Following:**
   - *Scenario:* A plane is flying a low-altitude mission over a mountain.
   - *Action:* As the plane approaches the mountain, `AP_Terrain` detects it is missing elevation data for the upcoming coordinates. It sends `TERRAIN_REQUEST` to the GCS. The GCS replies with the data, allowing the plane to climb autonomously to maintain safe ground clearance.

Key Codebase Locations

- **libraries/AP_Terrain/TerrainGCS.cpp:72**: Implementation of the sender.

TERRAIN_CHECK (ID 135)

SUPPORTED

Summary

The `TERRAIN_CHECK` message is a query used by the Ground Control Station (GCS) to verify if the vehicle has terrain data available for a specific location. This is often done during mission planning or pre-flight checks to ensure the drone can safely perform terrain-following operations.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Queries terrain database)

Reception (RX)

Reception is handled by `AP_Terrain::handle_terrain_check` within libraries/AP_Terrain/TerrainGCS.cpp:251.

Core Logic

1. **Query:** The GCS provides a latitude (`lat`) and longitude (`lon`).
2. **Lookup:** ArduPilot checks its internal `AP_Terrain` database (both RAM cache and SD card).
3. **Response:** It immediately sends back a `TERRAIN_REPORT` (136) message containing the terrain height at that location and the status of pending/loaded grid blocks.

Data Fields

- `lat`: Latitude (deg * 1E7).
- `lon`: Longitude (deg * 1E7).

Practical Use Cases

1. **Pre-Arming Safety:**
    - *Scenario:* A user uploads a mission that requires terrain following.
    - *Action:* The GCS sends `TERRAIN_CHECK` for each waypoint. If the drone replies with `TERRAIN_REPORT` indicating missing data, the GCS warns the user or initiates a data upload.

Key Codebase Locations

- **libraries/AP_Terrain/TerrainGCS.cpp:251**: Implementation of the handler.

TERRAIN_REPORT (ID 136)

SUPPORTED

Summary

The `TERRAIN_REPORT` message is sent by the vehicle to the Ground Control Station (GCS) to provide the terrain elevation at a specific coordinate. It is typically sent in response to a `TERRAIN_CHECK` (135) request or as part of the vehicle's periodic status stream to indicate terrain database health (loaded/pending blocks).

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports terrain data)
- **RX (Receive):** None (Ignored)

Transmission (TX)

The transmission logic is in `AP_Terrain::send_terrain_report` within libraries/AP_Terrain/TerrainGCS.cpp:221.

Protocol Logic

1. **Height Lookup:** It queries the internal terrain database for the elevation at the requested `lat`/`lon`.
2. **Statistics:** It calculates the number of `pending` (requested but not received) and `loaded` (valid) 4×4 grid blocks in the cache.
3. **Current Height:** It includes the vehicle's current height above terrain (`current_height`) if available.

Data Fields

- **`lat`**: Latitude (deg * 1E7).
- **`lon`**: Longitude (deg * 1E7).
- **`spacing`**: Grid spacing in meters (0 if no data available).
- **`terrain_height`**: Terrain height in meters AMSL.
- **`current_height`**: Current vehicle height above terrain in meters.
- **`pending`**: Number of 4×4 terrain blocks waiting to be loaded/received.
- **`loaded`**: Number of 4×4 terrain blocks currently in memory.

Practical Use Cases

1. **Database Visualization:**
   - *Scenario:* A pilot is preparing for a terrain-following mission.
   - *Action:* The GCS displays the `loaded` count. As the GCS pushes `TERRAIN_DATA` to the drone, this number increases, confirming the map is being saved to the SD card.
2. **Look-Ahead Safety:**
   - *Scenario:* The drone is flying towards a hill.
   - *Action:* The GCS (or onboard script) sends a `TERRAIN_CHECK` for a coordinate 500m ahead. The returned `TERRAIN_REPORT` shows the hill's altitude, allowing the system to verify clearance.

Key Codebase Locations

- **libraries/AP_Terrain/TerrainGCS.cpp:221**: Implementation of the sender.

FENCE_STATUS (ID 162)

SUPPORTED

Summary

The `FENCE_STATUS` message reports the current status of the vehicle's geofence, including whether a breach has occurred, the type of breach (altitude, boundary), and any active mitigation actions.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports fence status)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission is handled by `GCS_MAVLINK::send_fence_status` within libraries/GCS_MAVLink/GCS_Fence.cpp:66.

Logic

1. **Check:** If the fence is disabled, no message is sent.
2. **Translate Breaches:** The internal `AC_Fence` breach types are mapped to MAVLink enums (`FENCE_BREACH_MINALT`, `FENCE_BREACH_MAXALT`, `FENCE_BREACH_BOUNDARY`).
3. **Mitigation:** Checks `AC_Avoid` to see if velocity limiting (`FENCE_MITIGATE_VEL_LIMIT`) is active to prevent a breach.

Data Fields

- `breach_status`: 0 if currently inside fence, 1 if outside.
- `breach_count`: Number of fence breaches since arming.
- `breach_type`: Type of last breach (`FENCE_BREACH_NONE`, `MINALT`, `MAXALT`, `BOUNDARY`).
- `breach_time`: Time of last breach (ms since boot).
- `breach_mitigation`: Active mitigation action (`FENCE_MITIGATE_NONE`, `VEL_LIMIT`).

Practical Use Cases

1. **Pilot Awareness:**
   - *Scenario:* A pilot is flying near a geofence boundary.
   - *Action:* The GCS displays a warning icon when `breach_status` becomes 1, alerting the pilot that the vehicle has violated the safety zone.
2. **Safety Auditing:**
   - *Scenario:* Post-flight analysis.
   - *Action:* Reviewing the log to see where `breach_count` incremented helps identify if the mission plan was too close to restricted airspace.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Fence.cpp:66**: Implementation of the sender.

RALLY_POINT (ID 175)

SUPPORTED (RX & TX)

Summary

Represents a rally point (safe return location). This message is part of the legacy Rally Point protocol.

Status

Supported (RX & TX)

Directionality

- **TX (Transmit):** All Vehicles - Sends a rally point to the GCS (response to fetch).
- **RX (Receive):** All Vehicles - Receives a new rally point from the GCS (upload).

Description

This message allows a GCS to read or write rally points on the vehicle. While fully supported, ArduPilot 4.6+ issues a deprecation warning when this protocol is used, preferring the Mission Item protocol (using `MAV_CMD_NAV_RALLY_POINT`) instead.

Transmission (TX)

Sent in response to `RALLY_FETCH_POINT`.

Source: libraries/GCS_MAVLink/GCS_Rally.cpp

Reception (RX)

Handled by `GCS_MAVLINK::handle_rally_point`. Updates the rally point at the specified index in the storage.

Source: libraries/GCS_MAVLink/GCS_Rally.cpp

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `idx`: Index of the rally point (0-based).
- `count`: Total number of rally points.
- `lat`: Latitude (int32, deg * 1E7).
- `lng`: Longitude (int32, deg * 1E7).
- `alt`: Altitude (int16, meters).
- `break_alt`: Break altitude (int16, meters) - Alt to descend to before landing.
- `land_dir`: Landing direction (centidegrees).
- `flags`: Configuration flags (e.g., Favorable Wind).

Practical Use Cases

1. **Uploading Safe Points:**
   - *Scenario:* Uploading a set of safe landing zones before a mission.
   - *Action:* GCS sends `RALLY_POINT` messages to populate the list.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Rally.cpp**: Implementation.

PAYLOAD

CAMERA_INFORMATION (ID 259)

SUPPORTED

Summary

The `CAMERA_INFORMATION` message provides static metadata about an onboard camera to the Ground Control Station (GCS). It includes details like the manufacturer, model, focal length, sensor size, and resolution. This information is critical for survey planning software to calculate the "Ground Sample Distance" (GSD) and footprint of photos without requiring the user to manually enter camera specifications.

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Reports camera capabilities to GCS)
- **RX (Receive):** Specific Backends (Receives info from external MAVLink cameras)

Transmission (TX)

The transmission logic is centered in libraries/AP_Camera/AP_Camera.cpp:577, which delegates to individual backend implementations.

Data Sourcing

- **Metadata:** Sourced from the `AP_Camera` backend (AP_Camera_Backend.cpp:224).
- **Information Included:**
    - `vendor_name` / `model_name`: Identifies the hardware (e.g., "Sony", "Alpha 7").
    - `focal_length`: The lens focal length in millimeters.
    - `sensor_size_h` / `sensor_size_v`: Dimensions of the image sensor in millimeters.
    - `resolution_h` / `resolution_v`: Total pixel dimensions of the captured images.
- **Trigger:** Sent on request (e.g., via `MAV_CMD_REQUEST_MESSAGE`).

Reception (RX)

ArduPilot handles this message in the `MAVLinkCamV2` backend.

- **Scenario:** A specialized MAVLink-enabled camera (like a Gremsy or SIYI) is connected to the autopilot.
- **Action:** The camera sends its `CAMERA_INFORMATION` to the autopilot. ArduPilot's `AP_Camera_MAVLinkCamV2` backend parses this data and stores it, allowing the autopilot to proxy the information to the main Ground Control Station.

Data Fields

- `time_boot_ms`: Timestamp (time since system boot).
- `vendor_name`: Name of the camera vendor.
- `model_name`: Name of the camera model.
- `firmware_version`: Version of the camera firmware, encoded as: (Dev & 0xff) << 24 | (Patch & 0xff) << 16 | (Minor & 0xff) << 8 | (Major & 0xff).
- `focal_length`: Focal length.
- `sensor_size_h`: Image sensor size horizontal.
- `sensor_size_v`: Image sensor size vertical.

- `resolution_h`: Image resolution in pixels horizontal.
- `resolution_v`: Image resolution in pixels vertical.
- `lens_id`: Reserved for a lens ID.
- `flags`: Bitmap of camera capability flags.
- `cam_definition_version`: Camera definition version (iteration).
- `cam_definition_uri`: Camera definition URI (if any, otherwise only basic functions will be available). HTTP- (http://) or MAVLink FTP- (mavlinkftp://) formatted URI.

Practical Use Cases

1. **Automatic Survey Setup:**
   - *Scenario:* A user connects a new mapping camera to their drone.
   - *Action:* Mission Planner receives `CAMERA_INFORMATION`. It automatically updates the "Survey" grid calculator with the correct sensor size and focal length, ensuring the overlap and altitude calculations are mathematically correct for that specific camera.
2. **Remote Identity & Inventory:**
   - *Scenario:* A fleet manager is checking the status of 10 different drones.
   - *Action:* The management software queries `CAMERA_INFORMATION` to verify that each drone is equipped with the correct payload for the scheduled mission (e.g., Thermal vs. RGB).
3. **Lens Calibration verification:**
   - *Scenario:* A researcher is using a drone for high-precision scientific imaging.
   - *Action:* By checking the `focal_length` field, the researcher can verify that the lens hasn't been swapped or adjusted since the last calibration.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:577**: Common camera message dispatcher.
- **libraries/AP_Camera/AP_Camera_Backend.cpp:224**: Logic for populating the information packet.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6178**: Scheduler integration for camera telemetry.

CAMERA_SETTINGS (ID 260) SUPPORTED

Summary

The `CAMERA_SETTINGS` message reports the current dynamic state of an onboard camera, specifically its operating mode, zoom level, and focus level. Unlike `CAMERA_INFORMATION`, which describes static capabilities, `CAMERA_SETTINGS` provides real-time feedback to the Ground Control Station (GCS). This ensures that the pilot's UI accurately reflects the camera's physical state, especially during slow operations like zooming or focusing.

Status

Supported

Directionality

- **TX (Transmit):** Camera-enabled vehicles (Reports state to GCS)
- **RX (Receive):** None (ArduPilot does not parse this message from GCS)

Transmission (TX)

The transmission logic is abstracted through the `AP_Camera` and `AP_Mount` libraries, allowing various hardware drivers to report their status using this unified MAVLink message.

Core Logic

The primary entry point is `AP_Camera::send_camera_settings` in libraries/AP_Camera/AP_Camera.cpp:605. This function iterates through all initialized camera instances and calls their respective backends.

The message routing from `GCS_MAVLink` is handled in libraries/GCS_MAVLink/GCS_Common.cpp:6179, which delegates to `AP_Camera::send_mavlink_message` (libraries/AP_Camera/AP_Camera.cpp:452).

Supported Drivers (Backends)

Several advanced gimbal and camera drivers implement this message to provide feedback:

- **Siyi (`AP_Mount_Siyi`):**
  - Implemented in libraries/AP_Mount/AP_Mount_Siyi.cpp:1107.
  - Reports `zoomLevel` calculated from the current zoom multiplier (interpolated against the max zoom of 6x or 30x depending on hardware model).
  - Reports `mode_id` based on `_config_info.record_status` (Video vs Image).
- **Viewpro (`AP_Mount_Viewpro`):**
  - Implemented in libraries/AP_Mount/AP_Mount_Viewpro.cpp:936.
- **Topotek (`AP_Mount_Topotek`):**
  - Implemented in libraries/AP_Mount/AP_Mount_Topotek.cpp:562.
- **Xacti (`AP_Mount_Xacti`):**
  - Implemented in libraries/AP_Mount/AP_Mount_Xacti.cpp:390.

Data Fields

- **`time_boot_ms`**: Timestamp (time since system boot).
- **`mode_id`**: Camera mode (`CAMERA_MODE_IMAGE`, `CAMERA_MODE_VIDEO`, `CAMERA_MODE_IMAGE_SURVEY`).

- `zoomLevel`: Current zoom level as a percentage of the full range (0.0 to 100.0, NaN if not known).
- `focusLevel`: Current focus level as a percentage of the full range (0.0 to 100.0, NaN if not known).

Triggering

This message is typically sent in response to:

1. **Command Acknowledgement:** After receiving `MAV_CMD_DO_SET_CAM_ZOOM_FOCUS` or `MAV_CMD_REQUEST_CAMERA_SETTINGS`.
2. **State Change:** Some drivers may stream this message while the lens is moving (zooming) to provide smooth UI updates.

Practical Use Cases

1. **Zoom Slider Feedback:**
   - *Scenario:* A pilot uses a slider on the GCS to request 50\% zoom.
   - *Action:* The physical lens takes 1.5 seconds to move. The gimbal driver sends `CAMERA_SETTINGS` updates during this movement, allowing the GCS slider to move in sync with the actual lens, confirming the command is being executed.
2. **Focus Confirmation:**
   - *Scenario:* The user taps the screen to "Touch Focus".
   - *Action:* The camera executes the auto-focus routine. When complete, it sends `CAMERA_SETTINGS` with the new `focusLevel`, confirming to the user that the image is sharp.
3. **Mode Switching Verification:**
   - *Scenario:* The user switches from "Video" to "Photo" mode to take a high-res still.
   - *Action:* The camera takes a moment to stop recording and switch buffers. `CAMERA_SETTINGS` updates the `mode_id`, enabling the "Shutter" button on the GCS only when the camera is actually ready.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:605**: Main routing logic.
- **libraries/AP_Mount/AP_Mount_Siyi.cpp:1107**: Example of a driver implementing the message.
- **libraries/GCS_MAVLink/GCS_Common.cpp:6179**: Message definition and ID mapping.

CAMERA_CAPTURE_STATUS (ID 262) SUPPORTED

Summary

The `CAMERA_CAPTURE_STATUS` message reports the current status of the image and video capture subsystem. It allows the Ground Control Station (GCS) to know if the camera is currently taking photos (e.g., during a mapping mission) or recording video, along with the current image count and capture interval.

Status

Supported

Directionality

- **TX (Transmit):** Camera-enabled vehicles (Reports status to GCS)
- **RX (Receive):** None

Transmission (TX)

The transmission is handled by the `AP_Camera` library, specifically for cameras controlled directly by the autopilot's internal intervalometer (e.g., Servo, Relay, or basic MAVLink cameras).

Core Logic

The implementation resides in `AP_Camera_Backend::send_camera_capture_status` within libraries/AP_Camera/AP_Camera_Backend.cpp:346.

It is triggered by `AP_Camera::send_mavlink_message` in libraries/AP_Camera/AP_Camera.cpp:633.

Data Fields

- `time_boot_ms`: Timestamp (ms since boot).
- `image_status`:
  - Calculated from `time_interval_settings.num_remaining`.
  - **2** ("Interval set but idle") if `num_remaining > 0`.
  - **0** ("Idle") otherwise.
  - *Note:* The base implementation does not currently return **1** ("Capture in progress") or **3**, limiting its utility to showing *intent* rather than *active exposure*.
- `video_status`: Hardcoded to **0** (Idle) in the base implementation.
- `image_interval`: The configured interval in seconds, derived from `time_interval_settings.time_interval_ms`.
- `recording_time_ms`: Hardcoded to **0**.
- `available_capacity`: Hardcoded to **NaN** (Not Available).
- `image_index`: The total number of images captured in this session (`image_index`).

Limitations

- **Advanced Gimbals:** While `AP_Mount` has a virtual method for this message, most complex drivers (Siyi, Viewpro, Topotek) **do not** currently override it. This means they will not report their internal video recording status or SD card capacity via this message. They typically use `CAMERA_SETTINGS` (Mode) and `CAMERA_INFORMATION` instead.
- **Video Status:** The base `AP_Camera` implementation assumes it is only controlling a still shutter, so `video_status` is always 0.

Practical Use Cases

1. **Mapping Mission Progress:**
   - *Scenario:* A drone is flying a grid pattern for photogrammetry.
   - *Action:* The GCS monitors `image_index` to confirm that the camera is actually incrementing its photo count, ensuring the trigger cable hasn't failed.
2. **Intervalometer Confirmation:**
   - *Scenario:* The user commands "Take 1 photo every 2 seconds".
   - *Action:* The message reports `image_interval = 2.0` and `image_status = 2`, confirming the autopilot has accepted the command and is running the timer.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera_Backend.cpp:346**: Default implementation.
- **libraries/AP_Camera/AP_Camera.cpp:462**: Message handling.

VIDEO_STREAM_INFORMATION (ID 269) SUPPORTED

Summary

The `VIDEO_STREAM_INFORMATION` message provides the Ground Control Station (GCS) with the necessary connection details to display a live video feed. This includes the Stream URI (e.g., `rtsp://192.168.144.25:8554/main.264`), resolution, framerate, and encoding format.

Status

Supported

Directionality

- **TX (Transmit):** Camera-enabled vehicles (Reports stream config to GCS)
- **RX (Receive):** None

Transmission (TX)

The transmission logic is unique in ArduPilot: the core C++ engine handles the *sending* of the message, but the *content* is typically generated dynamically by a Lua script. This allows users to configure arbitrary IP cameras without modifying the firmware source code.

Core Logic

1. **Sending (C++):**
    - `AP_Camera_Backend::send_video_stream_information` in libraries/AP_Camera/AP_Camera_Backend.cpp:263 calls `mavlink_msg_video_stream_information_send_struct`.
    - It sends the data stored in the `_stream_info` struct.
2. **Populating (Lua):**
    - The `_stream_info` struct is populated by the `set_stream_information` binding.
    - The official script `libraries/AP_Scripting/applets/video-stream-information.lua` (Source) reads custom parameters (e.g., `VID1_CAMMODEL`, `VID1_IPADDR`) to construct the correct RTSP URI for supported cameras (Siyi, Herelink, Topotek, Viewpro) and pushes this data to the C++ backend.

Data Fields

- **`stream_id`**: Stream ID (1-based).
- **`count`**: Total number of streams.
- **`type`**: Protocol type (e.g., RTSP, RTPUDP, MPEG-TS).
- **`flags`**: Status flags (e.g., Thermal, Thermal Range).
- **`framerate`**: Frame rate in Hz.
- **`resolution_h`** / **`resolution_v`**: Resolution in pixels (e.g., 1920×1080).
- **`bitrate`**: Bitrate in bits/s.
- **`rotation`**: Rotation (0, 90, 180, 270).
- **`hfov`**: Horizontal Field of View (deg).
- **`name`**: Stream name (e.g., "Video").
- **`uri`**: The connection string (e.g., `rtsp://192.168.144.25:8554/main.264`).
- **`encoding`**: Encoding format (H.264, H.265).

Practical Use Cases

1. **Siyi/Herelink Integration:**
    - *Scenario:* A user connects a Siyi ZR10 gimbal.
    - *Action:* The Lua script detects `VID1_CAMMODEL = 2` (ZR10) and automatically populates the `uri` with `rtsp://192.168.144.25:8554/main.264`. QGroundControl receives this message and automatically starts playing the video feed without manual URL entry.
2. **Dual-Stream setups:**
    - *Scenario:* A drone has both a Visible and Thermal camera.
    - *Action:* The GCS receives two `VIDEO_STREAM_INFORMATION` messages (or one with `count=2`) and offers the pilot a dropdown to switch between "RGB" and "Thermal" views.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera_Backend.cpp:263**: C++ Sender.
- **libraries/AP_Scripting/applets/video-stream-information.lua**: Lua Logic for populating data.

CAMERA_FOV_STATUS (ID 271)

SUPPORTED

Summary

The `CAMERA_FOV_STATUS` message reports the geospatial projection of the camera's view. Specifically, it provides the 3D coordinates of the camera itself, the 3D coordinates of the "Point of Interest" (where the center of the image hits the ground), and the camera's absolute orientation in quaternion format. This is critical for applications like Augmented Reality (AR) overlays or "Click to Fly" map interfaces.

Status

Supported

Directionality

- **TX (Transmit):** Camera-enabled vehicles (Reports geospatial view)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_Camera` library, leveraging the `AP_Mount` library to perform the necessary 3D math (ray-tracing).

Core Logic

The implementation is in `AP_Camera_Backend::send_camera_fov_status` within libraries/AP_Camera/AP_Camera_Backend.cpp:295.

1. **POI Calculation:** It calls `mount->get_poi()` to calculate the intersection of the camera's optical axis with the terrain (or home altitude).
2. **Attitude:** It combines the gimbal's attitude (relative to the body) with the vehicle's AHRS yaw to produce an Earth-Frame Quaternion (`quat_ef`).
3. **Fallback:** If the POI cannot be calculated (e.g., the camera is looking above the horizon), it sends the camera's location but marks the POI coordinates as `INT32_MAX`.

Data Fields

- **`time_boot_ms`**: Timestamp (milliseconds since system boot).
- **`lat_camera`**: Latitude of camera (deg * 1E7).
- **`lon_camera`**: Longitude of camera (deg * 1E7).
- **`alt_camera`**: Altitude of camera (meters * 1000).
- **`lat_image`**: Latitude of center of image (deg * 1E7).
- **`lon_image`**: Longitude of center of image (deg * 1E7).
- **`alt_image`**: Altitude of center of image (meters * 1000).
- **`q`**: Quaternion of camera orientation (w, x, y, z).
- **`hfov`**: Horizontal field of view (deg).
- **`vfov`**: Vertical field of view (deg).

Practical Use Cases

1. **Map Projection:**
    - *Scenario:* A user is looking at the 2D map on the GCS.

    - *Action:* The GCS draws a "View Cone" polygon on the map, showing exactly what the drone's camera can see on the ground.
2. **Target Geolocation:**
    - *Scenario:* Search and Rescue. The pilot spots a missing person on the video feed.
    - *Action:* The GCS reads `lat_image` / `lon_image` to instantly provide the GPS coordinates of the person, without needing to be directly overhead.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera_Backend.cpp:295**: Implementation of the math and sending logic.

CAMERA_THERMAL_RANGE (ID 277)

SUPPORTED

Summary

The `CAMERA_THERMAL_RANGE` message reports the temperature statistics from a thermal camera. Specifically, it provides the maximum and minimum temperatures currently detected in the frame, along with the pixel coordinates of those hot/cold spots. This is essential for industrial inspections (e.g., finding hot spots on solar panels) or search and rescue (finding a heat signature in a cold environment).

Status

Supported

Directionality

- **TX (Transmit):** Thermal Camera-enabled vehicles (Reports temp stats to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is currently implemented for specific thermal gimbals via the `AP_Mount` library.

Core Logic

The implementation example can be found in `AP_Mount_Siyi::send_camera_thermal_range` within libraries/AP_Mount/AP_Mount_Siyi.cpp:1127.

1. **Driver Support:** Currently supported by the **Siyi ZT6** and **Siyi ZT30** gimbals.
2. **Data Retrieval:** The driver requests temperature data from the gimbal (via Siyi SDK) at 5Hz.
3. **Timeout:** If fresh data hasn't been received in 3 seconds (`AP_MOUNT_SIYI_THERM_TIMEOUT_MS`), it transmits `NaN` to indicate invalid data.

Data Fields

- **`time_boot_ms`:** Timestamp (milliseconds since system boot).
- **`stream_id`:** Video Stream ID (1 for first, 2 for second, etc).
- **`camera_id`:** Camera ID (1 for first, 2 for second, etc).
- **`max`:** Temperature max (degC).
- **`max_point_x`:** Temperature max point x (normalized 0..1).
- **`max_point_y`:** Temperature max point y (normalized 0..1).
- **`min`:** Temperature min (degC).
- **`min_point_x`:** Temperature min point x (normalized 0..1).
- **`min_point_y`:** Temperature min point y (normalized 0..1).

Practical Use Cases

1. **Solar Inspection:**
   - *Scenario:* A drone scans a solar farm.
   - *Action:* The GCS monitors the `max` field. If it spikes above 70°C, it flags the location and highlights the `max_point_x/y` on the video feed to show the defective cell.
2. **Firefighting:**
   - *Scenario:* Mapping a forest fire line.

    - *Action:* The `max` temperature confirms active combustion, while `min` helps calibrate the color palette of the display.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount_Siyi.cpp:1127**: Siyi implementation.

GIMBAL_MANAGER_INFORMATION (ID 280)

SUPPORTED

Summary

The `GIMBAL_MANAGER_INFORMATION` message is a cornerstone of the **MAVLink Gimbal Protocol v2**. It allows the Autopilot (acting as the Gimbal Manager) to advertise its high-level capabilities and configuration to the Ground Control Station (GCS). This includes supported axes, lock/follow modes, and physical angular limits.

Status

Supported

Directionality

- **TX (Transmit):** Gimbal-enabled vehicles (Reports capabilities to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is handled by the `AP_Mount` library, which aggregates the capabilities of the underlying driver.

Core Logic

The implementation is in `AP_Mount_Backend::send_gimbal_manager_information` within libraries/AP_Mount/AP_Mount_Backend.cpp:242.

It populates the `cap_flags` bitmask via `get_gimbal_manager_capability_flags()` (Line 208), which checks the configured `MNT_` parameters to see if Roll, Pitch, or Yaw control is enabled (`has_roll_control`, etc.).

Data Fields

- **`time_boot_ms`:** Timestamp (ms since boot).
- **`cap_flags`:** Bitmap of capabilities (`GIMBAL_MANAGER_CAP_FLAGS`).
  - **Modes:** `HAS_RETRACT`, `HAS_NEUTRAL`, `HAS_RC_INPUTS`.
  - **Pointing:** `CAN_POINT_LOCATION_LOCAL`, `CAN_POINT_LOCATION_GLOBAL`.
  - **Axes:** `HAS_ROLL_AXIS`, `HAS_PITCH_AXIS`, `HAS_YAW_AXIS`.
  - **Behaviors:** `HAS_ROLL_LOCK/FOLLOW`, `HAS_PITCH_LOCK/FOLLOW`, `HAS_YAW_LOCK/FOLLOW`.
- **`gimbal_device_id`:** Component ID of the gimbal device (usually 1).
- **`roll_min` / `roll_max`:** Physical limits in radians.
- **`pitch_min` / `pitch_max`:** Physical limits in radians.
- **`yaw_min` / `yaw_max`:** Physical limits in radians.

Practical Use Cases

1. **UI Configuration:**
   - *Scenario:* A GCS connects to a drone with a 2-axis gimbal (Pitch/Yaw only).
   - *Action:* The GCS reads this message, sees `HAS_ROLL_AXIS` is missing, and automatically hides the Roll control slider to declutter the interface.
2. **Safety Limits:**
   - *Scenario:* A user tries to command the gimbal to look straight up (+90 deg pitch).

    - *Action:* The GCS checks `pitch_max` (e.g., +45 deg) and clamps the command before sending it, preventing mechanical strain or frame collisions.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount_Backend.cpp:242**: Implementation of the sender.

GIMBAL_MANAGER_STATUS (ID 281) SUPPORTED

Summary

The `GIMBAL_MANAGER_STATUS` message is a key component of the **MAVLink Gimbal Protocol v2**. It reports the current status of the gimbal manager, specifically which high-level flags are active (e.g., Lock vs Follow) and, most importantly, **which MAVLink component currently has primary control** over the gimbal.

Status

Supported

Directionality

- **TX (Transmit):** Gimbal-enabled vehicles (Reports status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is handled by the `AP_Mount` library to reflect the internal state of the arbitration logic.

Core Logic

The implementation is in `AP_Mount_Backend::send_gimbal_manager_status` within libraries/AP_Mount/AP_Mount_Backend.cpp:257.

It provides two critical pieces of information:

1. **Arbitration:** It reports `primary_control_sysid` and `primary_control_compid`. This tells all listeners "Who is driving right now?" (e.g., The Mission Planner GCS, a Companion Computer, or the RC Controller).
2. **Flags:** It reports the active state of the gimbal axes (e.g., `GIMBAL_MANAGER_FLAGS_YAW_LOCK` vs Follow).

Data Fields

- **`time_boot_ms`:** Timestamp (milliseconds since system boot).
- **`flags`:** High level gimbal manager flags (bitmap).
- **`gimbal_device_id`:** Gimbal device ID.
- **`primary_control_sysid`:** Primary control system ID.
- **`primary_control_compid`:** Primary control component ID.
- **`secondary_control_sysid`:** Secondary control system ID.
- **`secondary_control_compid`:** Secondary control component ID.

Practical Use Cases

1. **Multi-Operator Handover:**
    - *Scenario:* A pilot is flying via RC (`[sysid](/field-manual/advanced-tuning/system-identification-mode.html)=1, compid=1`), but a Payload Operator wants to take control via a joystick on a second GCS (`sysid=255, compid=190`).
    - *Action:* The GCS sends a `MAV_CMD_DO_GIMBAL_MANAGER_CONFIGURE` to request control. The Autopilot accepts this and updates `GIMBAL_MANAGER_STATUS` to show `primary_control_sysid=255`. The Pilot's UI updates to show "Payload Operator has control".

2. **Mode Verification:**
    - *Scenario:* The user switches from "Follow Mode" to "Lock Mode".
    - *Action:* The GCS monitors `flags` to verify that `GIMBAL_MANAGER_FLAGS_YAW_LOCK` becomes active.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount_Backend.cpp:257**: Implementation of the sender.

GIMBAL_MANAGER_SET_ATTITUDE (ID 282)

SUPPORTED

Summary

The `GIMBAL_MANAGER_SET_ATTITUDE` message is the primary command for controlling a gimbal in the **MAVLink Gimbal Protocol v2**. It allows a Ground Control Station or Companion Computer to command the gimbal to a specific orientation (using Quaternions) or to a specific rotation rate (using Angular Velocity), while also managing high-level states like Retract and Neutral.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Controls the gimbal)

Reception (RX)

The message is handled by the `AP_Mount` library, which translates the MAVLink command into internal driver calls.

Core Logic

The implementation is in `AP_Mount::handle_gimbal_manager_set_attitude` within libraries/AP_Mount/AP_Mount.cpp:419.

1. **Target Selection:** Uses `gimbal_device_id` to select the correct gimbal instance.
2. **Mode Overrides:**
   - If `GIMBAL_MANAGER_FLAGS_RETRACT` is set, it calls `backend->[set_mode](/field-manual/mavlink-interface/set-mode.html)(MAV_MOUNT_MODE_RETRACT)`.
   - If `GIMBAL_MANAGER_FLAGS_NEUTRAL` is set, it calls `backend->set_mode(MAV_MOUNT_MODE_NEUTRAL)`.
3. **Mutual Exclusion:** It explicitly checks that you are not providing *both* a Quaternion `q` AND Angular Velocities `angular_velocity_x/y/z` simultaneously. If both are valid numbers (not NaN), the command is ignored.
4. **Attitude Control:**
   - Converts the Quaternion `q` into Euler angles (Roll, Pitch, Yaw).
   - Calls `set_angle_target` with the resulting degrees.
   - If `GIMBAL_MANAGER_FLAGS_YAW_LOCK` is set, the Yaw is interpreted as Earth-Frame (North = 0). Otherwise, it is Body-Frame.
5. **Rate Control:**
   - Converts `angular_velocity_x/y/z` (rad/s) into deg/s.
   - Calls `set_rate_target`.

Data Fields

- **`target_system`**: System ID.
- **`target_component`**: Component ID.
- **`flags`**: High level gimbal manager flags (bitmap).
- **`gimbal_device_id`**: Component ID of gimbal device to address.

- `q`: Quaternion components, w, x, y, z (1 0 0 0 is the null-rotation). Set fields to NaN if only angular velocity should be used.
- `angular_velocity_x`: X angular velocity (rad/s). NaN if only attitude should be used.
- `angular_velocity_y`: Y angular velocity (rad/s). NaN if only attitude should be used.
- `angular_velocity_z`: Z angular velocity (rad/s). NaN if only attitude should be used.

Practical Use Cases

1. **Look at Coordinate:**
   - *Scenario:* A companion computer wants the camera to look at a specific GPS location.
   - *Action:* The companion computer calculates the required Earth-Frame Quaternion to point at the target and sends this message with `GIMBAL_MANAGER_FLAGS_YAW_LOCK` set.
2. **Joystick Control:**
   - *Scenario:* A pilot is manually controlling the gimbal using a joystick knob.
   - *Action:* The GCS maps the joystick position to an angular velocity (e.g., +0.5 rad/s) and streams this message to smoothly pan the camera.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount.cpp:419**: Implementation of the handler.

GIMBAL_DEVICE_SET_ATTITUDE (ID 284) SUPPORTED

Summary

The `GIMBAL_DEVICE_SET_ATTITUDE` message is the low-level command used by the **Gimbal Manager** (Autopilot) to control the **Gimbal Device** (Physical Hardware). It is essentially the "driver-level" equivalent of `GIMBAL_MANAGER_SET_ATTITUDE`.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Commands the Gimbal Device)
- **RX (Receive):** None (ArduPilot does not obey this command from GCS; it is the sender)

Transmission (TX)

ArduPilot uses this message to drive MAVLink-capable gimbals, such as those from Gremsy.

Core Logic

The implementation is in `AP_Mount_Gremsy::send_gimbal_device_set_attitude` within libraries/AP_Mount/AP_Mount_Gremsy.cpp:311.

1. **Attitude Control:**
    - Converts internal target Euler angles (Roll/Pitch/Yaw) into a Quaternion `q`.
    - Sets `angular_velocity_x/y/z` to `NaN`.
    - Sets flags (e.g., `GIMBAL_DEVICE_FLAGS_YAW_LOCK`) to indicate if the target is Earth-Frame or Body-Frame.
2. **Rate Control:**
    - Sets `q` to `{NaN, NaN, NaN, NaN}`.
    - Populates `angular_velocity_x/y/z` with the target rates in rad/s.
3. **Retract:**
    - Sets `flags` to `GIMBAL_DEVICE_FLAGS_RETRACT`.
    - Sets all control values to `0` or `NaN`.

Data Fields

- **`target_system`**: System ID.
- **`target_component`**: Component ID.
- **`flags`**: Low level gimbal flags (bitmap).
- **`q`**: Quaternion components, w, x, y, z (1 0 0 0 is the null-rotation). Set fields to NaN if only angular velocity should be used.
- **`angular_velocity_x`**: X angular velocity (rad/s). NaN if only attitude should be used.
- **`angular_velocity_y`**: Y angular velocity (rad/s). NaN if only attitude should be used.
- **`angular_velocity_z`**: Z angular velocity (rad/s). NaN if only attitude should be used.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount_Gremsy.cpp:311**: Implementation of the sender.

GIMBAL_DEVICE_ATTITUDE_STATUS (ID 285)

SUPPORTED (TX & RX)

Summary

The `GIMBAL_DEVICE_ATTITUDE_STATUS` message is the primary telemetry packet for the **Gimbal Device** (physical hardware). It reports the low-level status of the gimbal, including its current orientation (quaternion), angular velocity, and hardware fault flags.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot → GCS (Reports the status of attached gimbals).
- **RX (Receive):** Gimbal Device → Autopilot (Reports hardware status to the manager).

Dual Role Implementation

ArduPilot supports this message in two distinct roles, acting as a bridge between the hardware and the GCS.

1. Reception (RX) - Consuming Hardware Data

The implementation is in `AP_Mount_Gremsy::handle_gimbal_device_attitude_status` within libraries/AP_Mount/AP_Mount_Gremsy.cpp:223.

- ArduPilot listens for this message from connected MAVLink gimbals (e.g., Gremsy).
- It caches the latest attitude quaternion `q` and `failure_flags`.
- This data is used to update the internal state of the `AP_Mount` backend.

2. Transmission (TX) - Reporting to GCS

The implementation is in `AP_Mount_Backend::send_gimbal_device_attitude_status` within libraries/AP_Mount/AP_Mount_Backend.cpp:175.

- ArduPilot acts as a "Virtual Gimbal Device" towards the GCS.
- It sends this message to the GCS, populated with either:
  - The cached data from the physical MAVLink gimbal (proxying).
  - The calculated attitude of a PWM/Servo gimbal (synthesized).
- This ensures the GCS receives a standard `GIMBAL_DEVICE_ATTITUDE_STATUS` regardless of the underlying hardware protocol.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `time_boot_ms`: Timestamp (milliseconds since system boot).
- `flags`: Current gimbal device flags (bitmap).
- `q`: Quaternion components, w, x, y, z (1 0 0 0 is the null-rotation).
- `angular_velocity_x`: X angular velocity (rad/s).
- `angular_velocity_y`: Y angular velocity (rad/s).
- `angular_velocity_z`: Z angular velocity (rad/s).
- `failure_flags`: Failure flags (bitmap).

- `delta_yaw`: Yaw angle delta (rad).
- `delta_yaw_velocity`: Yaw angular velocity delta (rad/s).
- `gimbal_device_id`: Gimbal device ID.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount_Backend.cpp:175**: TX Implementation.
- **libraries/AP_Mount/AP_Mount_Gremsy.cpp:223**: RX Implementation.

AUTOPILOT_STATE_FOR_GIMBAL_DEVICE (ID 286) SUPPORTED

Summary

The `AUTOPILOT_STATE_FOR_GIMBAL_DEVICE` message provides the vehicle's high-fidelity state estimation (Attitude, Velocity, Yaw Rate) directly to a connected **Gimbal Device**. This allows "Smart Gimbals" (like the Gremsy T3V3) to fuse the autopilot's robust AHRS solution with their own internal IMU data, resulting in superior horizon holding and drift correction compared to using the gimbal's IMU alone.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Broadcasts state to Gimbal)
- **RX (Receive):** None (Gimbal Device consumes this)

Transmission (TX)

The message is generated by the core `GCS_MAVLINK` library, drawing data from the AHRS (Attitude and Heading Reference System).

Core Logic

The implementation is in `GCS_MAVLINK::send_autopilot_state_for_gimbal_device` within libraries/GCS_MAVLink/GCS_Common.cpp:5989.

It is typically streamed at a high rate (e.g., 50Hz) to the gimbal over a private UART link.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `time_boot_us`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `q`: Quaternion components of autopilot attitude: w, x, y, z (1 0 0 0 is the null-rotation).
- `q_estimated_delay_us`: Estimated delay of the attitude data.
- `vx`: X Speed in NED (m/s).
- `vy`: Y Speed in NED (m/s).
- `vz`: Z Speed in NED (m/s).
- `v_estimated_delay_us`: Estimated delay of the speed data.
- `feed_forward_angular_velocity_z`: Feed forward Z angular velocity (rad/s).
- `estimator_status`: Bitmap indicating the status of the estimator (AHRS).
- `landed_state`: The landed state. Is set to MAV_LANDED_STATE_UNDEFINED if landed state is unknown.
- `angular_velocity_z`: Z component of angular velocity in NED (rad/s).

Practical Use Cases

1. **Horizon Drift Correction:**
   - *Scenario:* A drone is flying a long, straight mapping line.
   - *Action:* The gimbal uses the autopilot's `q` (which is corrected by GPS/Compass) to correct the slow drift of its own internal MEMS gyros, ensuring the camera remains perfectly level.

2. **Cornering Compensation:**
    - *Scenario:* The drone executes a sharp turn.
    - *Action:* The gimbal uses `angular_velocity_z` and `vx/vy` to anticipate the centripetal acceleration, pre-actuating the motors to keep the image stable.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5989**: Implementation of the sender.

GIMBAL_MANAGER_SET_PITCHYAW (ID 287) SUPPORTED

Summary

The `GIMBAL_MANAGER_SET_PITCHYAW` message is a high-level command for controlling a gimbal. Unlike `GIMBAL_MANAGER_SET_ATTITUDE`, which requires a full Quaternion, this message uses simple Euler angles (Pitch/Yaw) or angular rates. It is designed for typical "Point and Shoot" or "Pan/Tilt" operations where Roll control is not required (the gimbal is expected to keep the horizon level).

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Controls the gimbal)

Reception (RX)

The message is handled by the `AP_Mount` library.

Core Logic

The implementation is in `AP_Mount::handle_gimbal_manager_set_pitchyaw` within libraries/AP_Mount/AP_Mount.cpp:484.

1. **Target Selection:** Uses `gimbal_device_id` to select the gimbal instance.
2. **Mode Overrides:** Checks `GIMBAL_MANAGER_FLAGS_RETRACT` and `GIMBAL_MANAGER_FLAGS_NEUTRAL`.
3. **Mutual Exclusion:** It enforces that you cannot set *both* absolute angles (`pitch/yaw`) AND rates (`pitch_rate/yaw_rate`) in the same message.
4. **Control:**
   - **Angles:** If `pitch` and `yaw` are valid (not NaN), it converts them from Radians to Degrees and calls `set_angle_target`. Roll is hardcoded to 0.
   - **Rates:** If `pitch_rate` and `yaw_rate` are valid, it converts them from Rad/s to Deg/s and calls `set_rate_target`.
   - **Yaw Lock:** If `GIMBAL_MANAGER_FLAGS_YAW_LOCK` is set, Yaw is treated as Earth-Frame (North=0). Otherwise, it is Body-Frame.

Data Fields

- **`target_system`**: System ID.
- **`target_component`**: Component ID.
- **`flags`**: High level gimbal manager flags (bitmap).
- **`gimbal_device_id`**: Component ID of gimbal device to address.
- **`pitch`**: Pitch angle (rad).
- **`yaw`**: Yaw angle (rad).
- **`pitch_rate`**: Pitch angular rate (rad/s).
- **`yaw_rate`**: Yaw angular rate (rad/s).

Practical Use Cases

1. **Object Tracking:**

    - *Scenario:* A companion computer is running vision tracking.
    - *Action:* The computer calculates the required Pitch and Yaw error to center the target and sends this message with `pitch_rate` and `yaw_rate` to close the loop.
2. **Simple Pointing:**
    - *Scenario:* A user clicks "Look North" on the map.
    - *Action:* The GCS sends `pitch=0` and `yaw=0` (North) with `YAW_LOCK` set.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount.cpp:484**: Implementation of the handler.

WINCH_STATUS (ID 9005)

SUPPORTED

Summary

The `WINCH_STATUS` message reports the telemetry from an onboard winch system (e.g., used for delivery or retrieval). It allows the GCS to display line length, tension, and operational state.

Status

Supported

Directionality

- **TX (Transmit):** Autopilot (Reports winch status to GCS)
- **RX (Receive):** None

Transmission (TX)

The message is generated by the `AP_Winch` library. Support varies by backend.

Core Logic

The implementation example is in `AP_Winch_Daiwa::send_status` within libraries/AP_Winch/AP_Winch_Daiwa.cpp:77.

- **Daiwa Winch:** Reports full telemetry including tension, voltage, current, and clutch status.
- **PWM Winch:** Reports estimated line length and desired speed (as it lacks feedback).

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `line_length`: Length of the line released.
- `speed`: Speed of the line release.
- `tension`: Tension on the line.
- `voltage`: Voltage of the battery supplying the winch.
- `current`: Current draw from the winch.
- `temperature`: Temperature of the motor.
- `status`: Status flags.

Practical Use Cases

1. **Package Delivery:**
    - *Scenario:* A drone hovers to lower a package.
    - *Action:* The winch lowers the line. The GCS monitors `line_length`. When the package touches the ground, `tension` drops, allowing the GCS (or Lua script) to trigger the release mechanism.

Key Codebase Locations

- **libraries/AP_Winch/AP_Winch_Daiwa.cpp:77**: Implementation of the sender.

CAN_FRAME (ID 386) SUPPORTED (TX & RX)

Summary

The `CAN_FRAME` message encapsulates a raw Controller Area Network (CAN) frame. It is used to bridge the vehicle's onboard CAN buses over the MAVLink connection, effectively turning the Autopilot into an SLCAN adapter. This allows external tools (like the DroneCAN GUI) to interact with CAN peripherals via the autopilot's telemetry or USB link.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot → GCS (Forwards frames received on the physical CAN bus).
- **RX (Receive):** GCS → Autopilot (Injects frames onto the physical CAN bus).

Usage

The bridging functionality is controlled via the `MAV_CMD_CAN_FORWARD` command.

- **Enable:** Send `MAV_CMD_CAN_FORWARD` with `param1 = bus_id` to start receiving `CAN_FRAME` messages from that bus.
- **Disable:** Send `MAV_CMD_CAN_FORWARD` with `param1 = 0`.

Core Logic

The implementation is in `AP_CANManager` within libraries/AP_CANManager/AP_CANManager.cpp.

1. **RX (Injection):** `handle_can_frame` (Line 486) receives the message, buffers it, and writes it to the specified physical CAN bus.
2. **TX (Forwarding):** `can_frame_callback` (Line 665) listens to the physical CAN bus and forwards frames to MAVLink if forwarding is enabled for that channel.

Data Fields

- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`bus`:** Bus number.
- **`len`:** Frame length.
- **`id`:** Frame ID.
- **`data`:** Frame data.

Practical Use Cases

1. **DroneCAN Debugging:**
    - *Scenario:* A user wants to update the firmware on a CAN GPS or ESC.
    - *Action:* They connect the DroneCAN GUI to the Autopilot's USB port (MAVLink SLCAN). The Autopilot forwards all CAN traffic, allowing the GUI to see and update the peripheral as if it were directly connected.

CANFD_FRAME (ID 387)

SUPPORTED (TX & RX)

Summary

The `CANFD_FRAME` message encapsulates a raw CAN FD (Flexible Data-rate) frame. It serves the same purpose as `CAN_FRAME` (386) but supports the extended data length (up to 64 bytes) and faster bitrates of the CAN FD standard.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot → GCS (Forwards CAN FD frames).
- **RX (Receive):** GCS → Autopilot (Injects CAN FD frames).

Usage

The bridging functionality is controlled via the `MAV_CMD_CAN_FORWARD` command, just like `CAN_FRAME`.

Core Logic

The implementation is in `AP_CANManager` within libraries/AP_CANManager/AP_CANManager.cpp:706.

1. **Detection:** When a frame arrives on the bus, `can_frame_callback` checks `frame.isCanFDFrame()`.
2. **Routing:** If it is a CAN FD frame, it is wrapped in this message instead of the legacy `CAN_FRAME`.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `bus`: Bus number.
- `len`: Frame length.
- `id`: Frame ID.
- `data`: Frame data.

Practical Use Cases

1. **Modern Peripheral Updates:**
   - *Scenario:* Updating a newer DroneCAN servo that uses CAN FD for faster telemetry updates.
   - *Action:* The DroneCAN GUI sends firmware blocks using `CANFD_FRAME` messages, which the Autopilot injects onto the high-speed bus.

CAN_FILTER_MODIFY (ID 388)

SUPPORTED

Summary

The `CAN_FILTER_MODIFY` message allows the Ground Control Station (GCS) to dynamically configure the whitelist of CAN IDs that are forwarded by the Autopilot via the `CAN_FRAME` / `CANFD_FRAME` mechanism. This is essential for managing bandwidth on the telemetry link, allowing the GCS to subscribe only to specific DroneCAN messages of interest.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Configures internal CAN forwarding filters)

Usage

This message is typically sent by the DroneCAN GUI or a script after enabling forwarding with `MAV_CMD_CAN_FORWARD`.

Core Logic

The implementation is in `AP_CANManager::handle_can_filter_modify` within libraries/AP_CANManager/AP_CANManager.cpp:575.

It maintains a sorted list of whitelisted IDs (`can_forward.filter_ids`) and performs binary searches to filter outgoing frames efficiently.

Operations

The `operation` field controls how the provided list of IDs is applied:

- **0 (`CAN_FILTER_REPLACE`):** Replaces the entire current whitelist with the new list.
- **1 (`CAN_FILTER_ADD`):** Adds the new IDs to the existing whitelist (ignoring duplicates).
- **2 (`CAN_FILTER_REMOVE`):** Removes the specified IDs from the existing whitelist.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `bus`: Bus number.
- `operation`: What operation to perform on the filter list.
- `num_ids`: Number of IDs in ids list.
- `ids`: Filter IDs, length 16.

Practical Use Cases

1. **Selective Sniffing:**
   - *Scenario:* A developer wants to debug the GNSS output of a CAN GPS but doesn't want to flood the telemetry link with high-frequency ESC status messages.

- *Action:* The tool sends `CAN_FILTER_REPLACE` with only the DroneCAN GNSS Fix message ID. The Autopilot now only forwards those specific frames.

LOGGING

FILE_TRANSFER_PROTOCOL (ID 110)

SUPPORTED (RX & TX)

Summary

File transfer message. Acts as a transport layer for the MAVLink FTP protocol (an FTP-like protocol over MAVLink).

Status

Supported (RX & TX)

Directionality

- **TX (Transmit):** All Vehicles - Sends FTP replies (file data, directory listings).
- **RX (Receive):** All Vehicles - Receives FTP requests (read, list, write).

Reception (RX)

Handled by `GCS_MAVLINK::handle_file_transfer_protocol`.

Source: libraries/GCS_MAVLink/GCS_FTP.cpp

Protocol Logic

Implements a stateful FTP server.

- **OpCodes:** Open, Read, Write, Terminate, ListDirectory, CreateDirectory, RemoveFile, etc.
- **Transport:** Data is encapsulated in the `payload` field.

Data Fields

- **`target_network`:** Network ID (0 for default).
- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`payload`:** Variable length payload containing OpCode, Session ID, Offset, and Data.

Practical Use Cases

1. **Log Downloading:**
    - *Scenario:* Downloading Dataflash logs via MAVLink.
    - *Action:* Mission Planner uses MAVFTP to list files in `@SYS/logs` and download the `.bin` files.
2. **Script Upload:**
    - *Scenario:* Uploading a Lua script.
    - *Action:* User uploads `script.lua` to the `scripts/` directory on the SD card via MAVFTP.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_FTP.cpp:71**: Handler.

PARAMETERS

PARAMETER_REQUEST_READ (ID 20) SUPPORTED

Summary

The `PARAM_REQUEST_READ` message is used by a Ground Control Station (GCS) to request the current value of a specific onboard parameter. ArduPilot supports looking up parameters either by their index (offset in the list) or by their 16-character string ID.

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Requests a `PARAM_VALUE` reply)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_param_request_read` in libraries/GCS_MAVLink/GCS_Param.cpp:225.

Architecture: Asynchronous Lookup

ArduPilot does **not** look up the parameter immediately upon receipt to avoid blocking the main loop.

1. **Queueing:** The handler decodes the message and pushes a request into a `pending_param_request` queue.
2. **Processing:** A background IO timer (`GCS_MAVLINK::param_io_timer`) pops the request.
3. **Lookup:**
    - **By Index:** If `param_index` is not `-1`, it uses `AP_Param::find_by_index`.
    - **By ID:** If `param_index` is `-1`, it uses `AP_Param::find` with the `param_id` string.
4. **Response:** If found, the result is queued for transmission as a `PARAM_VALUE` message.

Data Fields

- **`target_system`**: System ID.
- **`target_component`**: Component ID.
- **`param_id`**: Onboard parameter id, terminated by NULL if the length is less than 16 human-readable chars and WITHOUT null termination (NULL) if the length is exactly 16 chars - applications have to provide 16+1 bytes storage if the ID is stored as string.
- **`param_index`**: Parameter index. Send -1 to use the param ID field as identifier (else the param id will be ignored).

Practical Use Cases

1. **Single Parameter Refresh:**
    - *Scenario:* A user changes a PID value and wants to verify it was written correctly.
    - *Action:* The GCS sends `PARAM_REQUEST_READ` for `ATC_RAT_RLL_P` to confirm the new value.
2. **Lazy Loading:**
    - *Scenario:* A mobile GCS wants to show a "Battery Settings" page but doesn't want to download all 1000+ parameters on connection.

    - *Action:* It requests only the specific parameters needed for that page (e.g., `BATT_MONITOR`, `BATT_CAPACITY`) on demand.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Param.cpp:225**: Contains `handle_param_request_read` and the async `param_io_timer`.

PARAM_REQUEST_LIST (ID 21)

SUPPORTED

Summary

The `PARAM_REQUEST_LIST` message triggers a bulk transfer of all onboard parameters. This is the primary "handshake" event when a Ground Control Station (GCS) connects to the vehicle, allowing it to download the full configuration state (typically 1000+ parameters).

Status

Supported

Directionality

- **TX (Transmit):** Specific Peripherals (ArduPilot acts as a GCS to a Gimbal)
- **RX (Receive):** All Vehicles (Triggers bulk download)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_param_request_list` in libraries/GCS_MAVLink/GCS_Param.cpp:206.

Streaming Architecture

ArduPilot handles the massive task of sending all parameters without blocking the main flight loop using a deferred state machine:

1. **Initialization:** The handler resets the internal cursor (`_queued_parameter`) to the first parameter index.
2. **Scheduling:** It does **not** loop immediately. Instead, it relies on the GCS scheduler (`try_send_message`).
3. **Iteration:** In subsequent main loop cycles, the scheduler calls `queued_param_send`.
4. **Rate Limiting:** `queued_param_send` is highly sophisticated:
    - It checks available bandwidth (`bw_in_bytes_per_second`).
    - It enforces a **1ms execution time limit** per burst to prevent CPU starvation.
    - It temporarily slows down other telemetry streams (by 4x) to prioritize the parameter download.

Transmission (TX)

Interestingly, ArduPilot can also act as a client for this message.

- **Solo Gimbal:** In libraries/AP_Mount/SoloGimbal_Parameters.cpp, ArduPilot sends `PARAM_REQUEST_LIST` to a connected Solo Gimbal to learn its configuration.

Data Fields

- **`target_system`:** System ID.
- **`target_component`:** Component ID.

Practical Use Cases

1. **Initial Connection:**

    - *Scenario:* A pilot connects Mission Planner to the drone via USB.
    - *Action:* Mission Planner sends `PARAM_REQUEST_LIST`. The green bar loads as ArduPilot streams ~1200 `PARAM_VALUE` messages back.
2. **Backup/Restore:**
    - *Scenario:* A user wants to save a "Known Good" config file.
    - *Action:* The GCS requests the list and saves the results to a `.param` file.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Param.cpp:206**: Contains `handle_param_request_list` and the `queued_param_send` state machine.

PARAM_VALUE (ID 22)

SUPPORTED

Summary

The `PARAM_VALUE` message is the fundamental unit of configuration data in MAVLink. It carries the name (`param_id`), value (as a float), and type of a single onboard parameter. It is primarily sent by the vehicle in response to read/list requests, but ArduPilot also receives it when acting as a GCS for smart peripherals (like Gimbals).

Status

Supported

Directionality

- **TX (Transmit):** All Vehicles (Streams configuration to GCS)
- **RX (Receive):** Specific Peripherals (Receives configuration from Gimbal)

Transmission (TX)

The transmission logic is centered in libraries/GCS_MAVLink/GCS_Param.cpp.

Encoding Logic

- **Storage vs. Protocol:** Internally, ArduPilot stores parameters as `int8`, `int16`, `int32`, or `float`. MAVLink 1.0/common only supports passing values as `float`.
- **Conversion:** `AP_Param::cast_to_float()` is used to convert integer parameters into the float field of the message. This works because a 32-bit float can exactly represent all 16-bit integers and most relevant 32-bit integers.
- **Type Hinting:** The message includes a `param_type` field (e.g., `MAV_PARAM_TYPE_INT32`). A smart GCS uses this to cast the float back to the correct integer type for display.

Senders

1. `queued_param_send`: In GCS_Param.cpp:41, the background stream used for `PARAM_REQUEST_LIST`.
2. `send_parameter_async_replies`: Used for `PARAM_REQUEST_READ` replies.
3. `handle_param_set`: In GCS_Param.cpp:263, immediate echo-back when a parameter is written.

Reception (RX)

ArduPilot handles this message in `GCS_MAVLINK::handle_param_value` within libraries/GCS_MAVLink/GCS_Common.cpp:821.

Usage

It delegates the message to the `AP_Mount` library.

- **Scenario:** A 3DR Solo Gimbal is connected.
- **Behavior:** The gimbal sends its own parameters to the autopilot. `SoloGimbal_Parameters::handle_param_value` parses these to populate its internal state, allowing the autopilot to "know" the gimbal's tuning.

Data Fields

- `param_id`: Onboard parameter id, terminated by NULL if the length is less than 16 human-readable chars and WITHOUT null termination (NULL) if the length is exactly 16 chars - applications have to provide 16+1 bytes storage if the ID is stored as string.
- `param_value`: Onboard parameter value.
- `param_type`: Onboard parameter type: see the MAV_PARAM_TYPE enum for supported data types.
- `param_count`: Total number of onboard parameters.
- `param_index`: Index of this onboard parameter.

Practical Use Cases

1. **Configuration Display:**
   - *Scenario:* A GCS receives `PARAM_VALUE` for `RTL_ALT` with value `1500.0`.
   - *Action:* It displays "Return Altitude: 15m" (1500cm).
2. **Verification:**
   - *Scenario:* A script sets a parameter.
   - *Action:* It waits for the `PARAM_VALUE` broadcast to confirm the autopilot accepted and applied the change.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Param.cpp:41**: Primary TX logic (streaming).
- **libraries/GCS_MAVLink/GCS_Common.cpp:821**: Entry point for received `PARAM_VALUE`.
- **libraries/AP_Mount/SoloGimbal_Parameters.cpp**: Client-side RX logic for gimbals.

PARAM_SET (ID 23)

SUPPORTED

Summary

The `PARAM_SET` message is the standard way to modify vehicle configuration over MAVLink. It allows a Ground Control Station (GCS) to write a new value to a specific parameter identified by its string ID. ArduPilot handles these requests by updating the internal parameter state and committing the change to non-volatile storage (EEPROM/Flash).

Status

Supported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Modifies configuration)

Reception (RX)

Reception is handled by `GCS_MAVLINK::handle_param_set` in libraries/GCS_MAVLink/GCS_Param.cpp:263.

Processing Logic

1. **Lookup:** The autopilot identifies the parameter by its 16-character name.
2. **Safety Check:** It verifies if the parameter allows modification via MAVLink using `allow_set_via_mavlink`.
    - If denied (e.g., a read-only hardware ID), ArduPilot sends a `PARAM_VALUE` message containing the *original* value as an implicit NACK.
3. **Validation:** It rejects `NaN` or `Inf` values.
4. **Update:** The internal value is updated via `set_float`.
5. **Persistence:** The change is saved to storage via `vp->save()`.
6. **Notification:** ArduPilot broadcasts a `PARAM_VALUE` message to all active channels to confirm the change. This is triggered by the `GCS_SEND_PARAM` macro in libraries/AP_Param/AP_Param.cpp.

Data Fields

- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`param_id`:** Onboard parameter id, terminated by NULL if the length is less than 16 human-readable chars and WITHOUT null termination (NULL) if the length is exactly 16 chars - applications have to provide 16+1 bytes storage if the ID is stored as string.
- **`param_value`:** Onboard parameter value.
- **`param_type`:** Onboard parameter type: see the MAV_PARAM_TYPE enum for supported data types.

Practical Use Cases

1. **Field Tuning:**
    - *Scenario:* A pilot notices the drone is oscillating in flight.
    - *Action:* The GCS sends `PARAM_SET` for `ATC_RAT_RLL_P` with a lower value. The drone applies it immediately, and the pilot observes the result.
2. **Mission Configuration:**
    - *Scenario:* Before a flight, the GCS sets `RTL_ALT` to 50m to avoid tall trees at the specific site.

    - *Action:* ArduPilot saves the value to Flash, ensuring it persists across reboots.
3. **Component Integration:**
    - *Scenario:* A companion computer enables a new feature (e.g., Obstacle Avoidance) by setting `OA_TYPE` to 1.
    - *Action:* ArduPilot re-initializes the Avoidance subsystem upon receiving the parameter change.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Param.cpp:263**: Main entry point for writing parameters.
- **libraries/AP_Param/AP_Param.cpp**: Implements the `GCS_SEND_PARAM` broadcast logic.

REMOTE-ID

OPEN_DRONE_ID_BASIC_ID (ID 12900)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_BASIC_ID` message provides the unique identification of the Unmanned Aircraft (UA), serving as the digital "license plate" for Remote ID compliance.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Broadcasts ID to GCS or Remote ID Module)
- **RX (Receive):** Autopilot (Receives config from GCS)

Usage

ArduPilot supports the full OpenDroneID protocol stack. It can act as the source of Remote ID data (TX) or as a configurator for external RID modules.

Core Logic

The implementation is in `AP_OpenDroneID::handle_msg` within libraries/AP_OpenDroneID/AP_OpenDroneID.cpp:768.

1. **RX:** If the GCS sends this message, ArduPilot updates its internal RID state (`pkt_basic_id`).
2. **TX:** ArduPilot periodically broadcasts this message to connected MAVLink peripherals (like a WiFi RID transmitter) to ensure they are broadcasting the correct ID.

Data Fields

- **`target_system`** / **`target_component`**: Target.
- **`id_type`**: Type of ID (`MAV_ODID_ID_TYPE`), e.g., Serial Number, CAA Registration ID.
- **`ua_type`**: Type of vehicle (`MAV_ODID_UA_TYPE`).
- **`uas_id`**: The unique ID string (max 20 bytes).

Practical Use Cases

1. **Regulatory Compliance:**
   - *Scenario:* Operating in FAA or EASA airspace.
   - *Action:* The drone broadcasts its ANSI/CTA-2063 Serial Number so that local authorities can identify the aircraft without physical access.
2. **Fleet Management:**
   - *Scenario:* A swarm show with 50 drones.
   - *Action:* The central computer verifies that every drone on the network is reporting the correct, pre-assigned `uas_id` before arming.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID.cpp:768**: Message handler and state management.

OPEN_DRONE_ID_LOCATION (ID 12901)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_LOCATION` message reports the vehicle's dynamic state (Position, Altitude, Velocity) for Remote ID compliance. This is the most frequent message in the ODID protocol (typically 1Hz).

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Broadcasts location to GCS/RID Module)
- **RX (Receive):** Autopilot (Receives updates if using external GPS)

Transmission (TX)

The message is generated by `AP_OpenDroneID::send_location_message`.

Core Logic

The implementation is in libraries/AP_OpenDroneID/AP_OpenDroneID.cpp.

It pulls data from the AHRS and GPS:

- `latitude` / `longitude`: Current position.
- `altitude_barometric`: Barometric altitude relative to takeoff.
- `altitude_geodetic`: GPS altitude (WGS84).
- `height`: Height above ground/takeoff.
- `speed_horizontal`: Ground speed.
- `direction`: Course over ground.

Data Fields

- `target_system` / `target_component`: Target.
- `status`: Status (`MAV_ODID_STATUS`) e.g., Airborne, Ground, Emergency.
- `direction`: Direction (0-360 deg).
- `speed_horizontal`: Speed (cm/s).
- `speed_vertical`: Vertical speed (cm/s).
- `latitude` / `longitude`: Position (degE7).
- `altitude_barometric`: Baro Alt (m).
- `altitude_geodetic`: GPS Alt (m).
- `height_reference`: Reference datum.
- `height`: Height (m).
- `horizontal_accuracy` / `vertical_accuracy` / `barometer_accuracy` / `speed_accuracy`: Accuracy metrics.
- `timestamp`: Time since boot.
- `timestamp_accuracy`: Time accuracy.

Practical Use Cases

1. **Airspace Safety:**
    - *Scenario:* A medical helicopter is flying low near a drone operation.

    - *Action:* The helicopter's ADS-B In or traffic awareness system receives the drone's `OPEN_DRONE_ID_LOCATION` broadcast, alerting the pilot to the drone's precise location and altitude.
2. **Flight Recorder:**
    - *Scenario:* Post-flight path analysis.
    - *Action:* The GCS logs these messages to reconstruct the 3D flight path with high-fidelity velocity vectors, useful for verifying flight boundaries were respected.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID.cpp**: Implementation of `send_location_message`.

OPEN_DRONE_ID_AUTHENTICATION (ID 12902)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_AUTHENTICATION` message provides authentication data for the Unmanned Aircraft. This allows observers to verify that the broadcast Remote ID data is authentic and has not been spoofed.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Broadcasts auth data)
- **RX (Receive):** Autopilot (Receives auth config)

Transmission (TX)

The message is handled by `AP_OpenDroneID`.

Core Logic

It transmits an authentication signature or page of authentication data. Since auth data can be large, it may be split across multiple messages (pages).

Data Fields

- **`target_system` / `target_component`:** Target.
- **`authentication_type`:** Type of authentication (`MAV_ODID_AUTH_TYPE`).
- **`data_page`:** Page number.
- **`last_page_index`:** Last page index.
- **`length`:** Length of data in this page.
- **`timestamp`:** Timestamp.
- **`authentication_data`:** Raw data buffer (up to 23 bytes).

Practical Use Cases

1. **Anti-Spoofing:**
    - *Scenario:* A malicious actor tries to broadcast fake drone locations to disrupt an airport.
    - *Action:* Security systems check the digital signature provided in this message against a trusted registry. The fake broadcasts fail validation and are flagged as spoofed.
2. **Secure Access:**
    - *Scenario:* A delivery drone enters a secure facility.
    - *Action:* The facility's receiver validates the authentication token to confirm the drone is an authorized delivery vehicle before opening the landing bay.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID.cpp**: Handling of authentication pages.

OPEN_DRONE_ID_SELF_ID (ID 12903)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_SELF_ID` message allows the operator to provide a free-text description of the flight or vehicle. This is often used for "Mission Description" or "Emergency Text" in the Remote ID broadcast.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Broadcasts self-ID)
- **RX (Receive):** Autopilot (Receives config)

Transmission (TX)

The message is handled by `AP_OpenDroneID`.

Core Logic

It broadcasts the configured description string.

Data Fields

- **`target_system` / `target_component`:** Target.
- **`description_type`:** Type of description (`MAV_ODID_DESC_TYPE`).
- **`description`:** Description string (up to 23 bytes).

Practical Use Cases

1. **Emergency Communication:**
    - *Scenario:* A drone is performing an emergency medical delivery.
    - *Action:* The operator sets the description to "EMERGENCY MEDICAL BLOOD". Anyone receiving the RID signal sees this text and understands the priority nature of the flight.
2. **Mission Identification:**
    - *Scenario:* Multiple survey teams are working in the same area.
    - *Action:* Team A sets their description to "Survey Team A", allowing them to distinguish their drones from Team B on the monitoring app.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID.cpp**: Text handling logic.

OPEN_DRONE_ID_SYSTEM (ID 12904)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_SYSTEM` message provides operator location and system metadata. This includes the pilot's location (Takeoff or Live GCS location) and the altitude reference.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Broadcasts system data)
- **RX (Receive):** Autopilot (Receives GCS/Operator location)

Transmission (TX)

The message is handled by `AP_OpenDroneID`.

Core Logic

It pulls the Operator Location either from the GCS (via `OPEN_DRONE_ID_SYSTEM_UPDATE`) or uses the vehicle's home location if GCS location is unavailable.

Data Fields

- **`target_system` / `target_component`:** Target.
- **`flags`:** Flags.
- **`operator_latitude`:** Operator Lat (degE7).
- **`operator_longitude`:** Operator Lon (degE7).
- **`area_count`:** Count of area points.
- **`area_radius`:** Radius of operation.
- **`area_ceiling`:** Ceiling height.
- **`area_floor`:** Floor height.
- **`category_eu`:** EU Category.
- **`class_eu`:** EU Class.
- **`operator_altitude_geo`:** Operator Altitude (Geodetic).
- **`timestamp`:** Timestamp.

Practical Use Cases

1. **Pilot Accountability:**
    - *Scenario:* A drone is flying dangerously near a crowd.
    - *Action:* Law enforcement checks the Remote ID broadcast. The `operator_latitude`/`longitude` points them to the pilot's location, allowing them to intervene directly.
2. **Home Point Verification:**
    - *Scenario:* A user sets up for a long-range flight.
    - *Action:* The GCS sends its GPS location to the drone. The drone broadcasts this as the Operator Location, ensuring compliance with regulations that require the pilot's location to be known.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID.cpp**: Operator location handling.

OPEN_DRONE_ID_OPERATOR_ID (ID 12905)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_OPERATOR_ID` message broadcasts the Operator ID (e.g., FAA Registration Number or CAA Operator ID) required for Remote ID compliance.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Broadcasts Operator ID)
- **RX (Receive):** Autopilot (Receives Operator ID config)

Transmission (TX)

The message is handled by `AP_OpenDroneID`.

Core Logic

It broadcasts the configured Operator ID string.

Data Fields

- **`target_system`** / **`target_component`**: Target.
- **`operator_id_type`**: Type of ID (`MAV_ODID_OPERATOR_ID_TYPE`).
- **`operator_id`**: Operator ID string (up to 20 bytes).

Practical Use Cases

1. **Compliance Audits:**
    - *Scenario:* An aviation authority audits drone operations at a commercial site.
    - *Action:* They monitor the RID broadcast and compare the `operator_id` against their database of registered commercial operators to ensure the company is compliant.
2. **Incident Reporting:**
    - *Scenario:* A drone crashes on private property.
    - *Action:* The homeowner uses a RID scanner app to read the `operator_id`, which can be provided to authorities to identify the owner for insurance claims.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID.cpp**: ID string storage.

OPEN_DRONE_ID_MESSAGE_PACK (ID 12915)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_MESSAGE_PACK` allows packing multiple OpenDroneID messages into a single MAVLink payload. This is efficient for bandwidth-constrained links or for ensuring atomic updates of RID data.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Broadcasts packed data)
- **RX (Receive):** Autopilot (Parses packed data)

Transmission (TX)

ArduPilot can parse this message if received from a peripheral, but typically transmits individual ODID messages for compatibility.

Data Fields

- **`target_system` / `target_component`:** Target.
- **`single_message_size`:** Size of each message in the pack.
- **`msg_pack_size`:** Total size.
- **`messages`:** Buffer containing concatenated ODID messages.

Practical Use Cases

1. **Bandwidth Optimization:**
    - *Scenario:* A telemetry link has very limited bandwidth (e.g., LoRa).
    - *Action:* Instead of sending 5 separate MAVLink headers for Basic ID, Location, System, etc., the system packs them into one `MESSAGE_PACK`, reducing overhead and ensuring all data arrives together.
2. **Atomic Updates:**
    - *Scenario:* Updating the Remote ID state on a companion computer.
    - *Action:* The Autopilot sends a pack containing both the new Location and the new Vector. The companion computer processes them simultaneously, avoiding a race condition where the location updates before the vector.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID.cpp**: Message parsing logic.

OPEN_DRONE_ID_ARM_STATUS (ID 12918)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_ARM_STATUS` message reports the health and arming status of the Remote ID system. It allows the Remote ID module (e.g., a DroneCAN device) to block the vehicle from arming if the RID system is not healthy or compliant, and provides a text error message explaining why.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Forwards RID status to GCS)
- **RX (Receive):** Autopilot (Receives status from DroneCAN RID module)

Usage

ArduPilot acts as a bridge for this message.

Core Logic

The implementation is in `handle_arm_status` within libraries/AP_OpenDroneID/AP_OpenDroneID_DroneCAN.cpp:215.

1. **Reception:** The Autopilot receives a `dronecan.[remoteid](/field-manual/remote-id/core-concepts-and-regulations.html).ArmStatus` message from the CAN bus.
2. **State Update:** It updates the internal `AP_OpenDroneID` state, which the Arming Checks (`AP_Arming`) monitor. If the status is not "Good", the autopilot will refuse to arm.
3. **Forwarding:** It immediately forwards the status to the GCS via MAVLink so the pilot can see the error message (e.g., "RID: System Failure").

Data Fields

- **`status`**: Status (`MAV_ODID_ARM_STATUS`) e.g., `GOOD_TO_ARM`, `PRE_FLIGHT_CHECKS_FAIL`.
- **`error`**: Text error message (up to 50 bytes).

Practical Use Cases

1. **Pre-Flight Safety:**
   - *Scenario:* A user tries to arm the drone, but the Remote ID module has not yet acquired a GPS lock.
   - *Action:* The module sends `OPEN_DRONE_ID_ARM_STATUS` with status `FAIL` and error "Wait for GPS". The autopilot blocks arming, and the GCS displays "Wait for GPS" to the pilot.
2. **Tamper Detection:**
   - *Scenario:* The Remote ID antenna is disconnected.
   - *Action:* The module detects the hardware fault and reports "Antenna Fail". The drone prevents takeoff, ensuring regulatory compliance.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID_DroneCAN.cpp:215**: Handler for DroneCAN to MAVLink bridging.

OPEN_DRONE_ID_SYSTEM_UPDATE (ID 12919)

SUPPORTED (TX & RX)

Summary

The `OPEN_DRONE_ID_SYSTEM_UPDATE` message allows the Ground Control Station (GCS) to send real-time updates about the Operator's location to the drone. This is crucial for satisfying the Remote ID requirement to broadcast the pilot's location, especially if the pilot is mobile.

Status

Supported (TX & RX)

Directionality

- **TX (Transmit):** Autopilot (Forwards update to RID module)
- **RX (Receive):** Autopilot (Receives Operator Location from GCS)

Usage

The GCS typically sends this message at 1Hz if the GCS has a GPS source (e.g., tablet GPS).

Core Logic

The implementation is in `AP_OpenDroneID::handle_msg` within libraries/AP_OpenDroneID/AP_OpenDroneID.cpp:777.

1. **Reception:** The Autopilot receives the message from the GCS.
2. **Update:** It updates the internal `pkt_system` structure with the new `operator_latitude`, `operator_longitude`, and `operator_altitude_geo`.
3. **Transmission:** The Autopilot then uses these updated values when constructing the `OPEN_DRONE_ID_SYSTEM` message broadcast to the Remote ID module.

Data Fields

- **`target_system` / `target_component`:** Target.
- **`operator_latitude`:** Operator Lat (degE7).
- **`operator_longitude`:** Operator Lon (degE7).
- **`operator_altitude_geo`:** Operator Altitude (Geodetic).
- **`timestamp`:** Timestamp.

Practical Use Cases

1. **Mobile Command Center:**
    - *Scenario:* A pilot is operating from a moving vehicle.
    - *Action:* The GCS sends `OPEN_DRONE_ID_SYSTEM_UPDATE` continuously. The drone updates its broadcast to reflect the pilot's changing position, ensuring compliance with "Mobile GCS" regulations.
2. **Dynamic Home Point:**
    - *Scenario:* The drone takes off from a boat.
    - *Action:* As the boat moves, the GCS updates the operator location. If the drone needs to RTL (Return to Launch), it can use this updated operator location (depending on configuration) or at least correctly report where the pilot *is* currently located.

Key Codebase Locations

- **libraries/AP_OpenDroneID/AP_OpenDroneID.cpp:777**: Message handler.

SIMULATION

SIMSTATE (ID 164)

SUPPORTED (SITL ONLY)

Summary

The `SIMSTATE` message provides the "Ground Truth" state of the vehicle from the Software In The Loop (SITL) simulator. It allows developers to compare the autopilot's estimated state (AHRS/EKF) against the perfect physical reality simulated by the physics engine.

Status

Supported (SITL Only)

Directionality

- **TX (Transmit):** SITL Vehicles (Reports perfect simulation state)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission is handled by `SIM::simstate_send` within libraries/SITL/SITL.cpp:1514.

Data Source

All fields are populated directly from the internal `SITL::state` structure, which holds the physics engine's calculations for position, attitude, and dynamics.

Data Fields

- **`roll`**: True roll angle (radians).
- **`pitch`**: True pitch angle (radians).
- **`yaw`**: True yaw angle (radians), normalized to +/- PI.
- **`xacc`**: True X acceleration (m/s/s) in body frame.
- **`yacc`**: True Y acceleration (m/s/s) in body frame.
- **`zacc`**: True Z acceleration (m/s/s) in body frame.
- **`xgyro`**: True roll rate (rad/s) in body frame.
- **`ygyro`**: True pitch rate (rad/s) in body frame.
- **`zgyro`**: True yaw rate (rad/s) in body frame.
- **`lat`**: True latitude (deg * 1E7).
- **`lng`**: True longitude (deg * 1E7).

Practical Use Cases

1. **Estimator Tuning:**
   - *Scenario:* A developer is tuning the EKF.
   - *Action:* By plotting `SIMSTATE.roll` vs `ATTITUDE.roll`, they can see exactly how much error the estimator has and how much lag is introduced by filtering.
2. **Vibration Testing:**
   - *Scenario:* Testing how the EKF handles high vibration.
   - *Action:* The simulator adds noise to the IMU data (`RAW_IMU`). Comparing the noisy `RAW_IMU` against the clean `SIMSTATE` and the filtered `ATTITUDE` helps verify the vibration rejection logic.

Key Codebase Locations

- **libraries/SITL/SITL.cpp:1514**: Implementation of the sender.

SYSTEM

AUTOPILOT_VERSION_REQUEST (ID 183)

SUPPORTED (RX ONLY)

Summary

Request the `AUTOPILOT_VERSION` message from the vehicle.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives request and responds with version info.

Reception (RX)

Handled by `GCS_MAVLINK::handle_send_autopilot_version`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

1. GCS sends `AUTOPILOT_VERSION_REQUEST` (optionally specifying target system/component).
2. Vehicle checks if the request is for itself.
3. Vehicle responds with `AUTOPILOT_VERSION` (148).

Data Fields

- **`target_system`**: System ID.
- **`target_component`**: Component ID.

Practical Use Cases

1. **Connecting to GCS:**
   - *Scenario:* Mission Planner connects to the drone.
   - *Action:* It sends this request to determine firmware version, board capabilities, and custom version tags (Git hash).

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4244**: Handler.

NAMED_VALUE_INT (ID 252)

SUPPORTED (RX ONLY)

Summary

Key-value pair of a string name and an integer value.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Copter (Toy Mode) - Receives control events.

Reception (RX)

Handled by `ToyMode::handle_message` in ArduCopter. Used for interactions with specific "Toy" controllers (SkyViper).

Source: ArduCopter/GCS_Mavlink.cpp

Data Fields

- `time_boot_ms`: Timestamp.
- `name`: Name string (10 chars max).
- `value`: Int value.

Practical Use Cases

1. **Toy Controller Buttons:**
    - *Scenario:* A SkyViper controller button is pressed.
    - *Action:* It sends a `NAMED_VALUE_INT` (e.g., "BTN_A", 1). ArduPilot receives this and triggers a flip or mode change.

Key Codebase Locations

- **ArduCopter/GCS_Mavlink.cpp:1511**: Handler.

SETUP_SIGNING (ID 256)

SUPPORTED (RX ONLY)

Summary

Setup a secret key for MAVLink packet signing.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives new signing key.

Reception (RX)

Handled by `GCS_MAVLink::handle_setup_signing`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Sets the 32-byte secret key and the initial timestamp. This enables cryptographic authentication of MAVLink packets.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `secret_key`: 32-byte secret key.
- `initial_timestamp`: Initial timestamp.

Practical Use Cases

1. **Securing the Link:**
    - *Scenario:* Preventing hijacking.
    - *Action:* GCS generates a key and sends `SETUP_SIGNING`. The drone now ignores commands that aren't signed with this key.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4181**: Handler.

TELEMETRY

RADIO_STATUS (ID 109)

SUPPORTED (RX ONLY)

Summary

Status reports from a 3DR/SiK-compatible telemetry radio.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Processed for link quality and flow control.

Reception (RX)

Handled by `GCS_MAVLINK::handle_radio_status`. This message is functionally identical to `RADIO` (166) in ArduPilot's handling logic.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

See `RADIO` (166). The handler updates RSSI and TX Buffer stats to perform adaptive flow control.

Data Fields

- `rssi`: Local signal strength.
- `remrssi`: Remote signal strength.
- `txbuf`: Transmit buffer remaining \%.
- `noise`: Background noise.
- `remnoise`: Remote background noise.
- `rxerrors`: Receive errors.
- `fixed`: Corrected packets.

Practical Use Cases

1. **Telemetry Health:**
    - *Scenario:* Monitoring link quality.
    - *Action:* GCS uses `remrssi` to show the signal bars for the drone's radio.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:862**: Handler.

RADIO (ID 166)

SUPPORTED (RX ONLY)

Summary

Status reports from a 3DR/SiK-compatible telemetry radio. These messages are typically generated locally by the telemetry radio firmware and injected into the autopilot's serial stream to report link quality and buffer status.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - ArduPilot parses this message to monitor link quality and perform adaptive flow control.

Reception (RX)

ArduPilot processes this message in `GCS_MAVLINK::handle_radio_status`. The primary use is **Adaptive Flow Control**. The system monitors the radio's transmit buffer (`txbuf`) to dynamically adjust the rate at which MAVLink streams are sent.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

1. **RSSI Tracking:** Updates `last_radio_status.rssi` and `last_radio_status.remrssi` (Remote RSSI).
2. **Flow Control:** Checks `packet.txbuf` (Remaining Free Space \%):
   - **< 20\%:** Radio buffer full. Increases `stream_slowdown_ms` significantly (slows down telemetry).
   - **< 50\%:** Radio buffer getting full. Increases `stream_slowdown_ms` slightly.
   - **> 95\%:** Buffer empty. Decreases `stream_slowdown_ms` significantly (speeds up telemetry).
   - **> 90\%:** Buffer mostly empty. Decreases `stream_slowdown_ms` slightly.
3. **Logging:** Writes a `RAD` (Radio) log entry to the onboard SD card dataflash log.

Data Fields

- **`rssi`:** Local signal strength (0-255). Often scaled to percentage.
- **`remrssi`:** Remote signal strength (0-255).
- **`txbuf`:** Remaining free space in the radio's transmit buffer (0-100\%).
- **`noise`:** Background noise level.
- **`remnoise`:** Remote background noise level.
- **`rxerrors`:** Count of receive errors.
- **`fixed`:** Count of corrected packets.

Practical Use Cases

1. **Telemetry Link Optimization:**
   - *Scenario:* User is flying at long range with a SiK telemetry radio.

    - *Action:* As the radio link degrades or the bandwidth saturates, the radio reports low `txbuf`. ArduPilot automatically throttles the stream rate (Hz) of attitude/GPS updates to prevent packet loss and latency buildup.

2. **Link Quality Monitoring:**
    - *Scenario:* Post-flight analysis of a mission.
    - *Action:* The user reviews the `RAD` log messages to see a graph of RSSI vs Distance, helping to verify antenna placement and noise floor issues.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:862**: `handle_radio_status` implementation.

DATA96 (ID 172)

SUPPORTED (RX ONLY)

Summary

Generic 96-byte data packet. In ArduPilot, this is used to receive firmware update data and test commands for the `AP_Radio` system from a Ground Control Station.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - ArduPilot receives this from the GCS and forwards it to the `AP_Radio` driver.

Reception (RX)

ArduPilot processes this message in `GCS_MAVLINK::handle_data_packet`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

The handler inspects the `type` field:

- **Type 42:** Firmware Upload. Passed to `AP_Radio::handle_data_packet`.
- **Type 43:** Play Tune. Passed to `AP_Radio`.

Data Fields

- **`type`**: Data type ID (42=FW Upload, 43=Play Tune).
- **`len`**: Data length.
- **`data`**: Raw data (96 bytes).

Practical Use Cases

1. **Updating Radio Firmware:**
   - *Scenario:* User uploads new firmware to the onboard telemetry radio via the GCS.
   - *Action:* The GCS sends `DATA96` packets containing the binary. ArduPilot receives them and flushes them to the radio hardware.
2. **Radio Testing:**
   - *Scenario:* Debugging radio speakers/buzzers.
   - *Action:* GCS sends a "Play Tune" command (Type 43) via `DATA96`.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3818**: `handle_data_packet` implementation.

LED_CONTROL (ID 186)

SUPPORTED (RX ONLY)

Summary

Control the color and pattern of onboard RGB LEDs (e.g., NeoPixel, Toshiba LEDs).

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives commands to set LED state.

Reception (RX)

Handled by `AP_Notify::handle_led_control`.

Source: libraries/AP_Notify/AP_Notify.cpp

Protocol Logic

Passed to the `AP_Notify` library, which overrides the standard status LED patterns.

- **Patterns:** Solid, Blink, Flash.
- **Colors:** RGB bytes.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `instance`: LED instance (0 for all).
- `pattern`: Pattern ID (0=Solid, 1=Custom, etc.).
- `custom_len`: Custom pattern length.
- `custom_bytes`: Custom pattern data.

Practical Use Cases

1. **Light Shows / Swarming:**
    - *Scenario:* A swarm of drones performs a night show.
    - *Action:* The central computer sends `LED_CONTROL` messages to change the color of each drone in sync with the music.

Key Codebase Locations

- **libraries/AP_Notify/AP_Notify.cpp:475**: Handler.

PLAY_TUNE (ID 258)

SUPPORTED (RX ONLY)

Summary

Command the vehicle to play a musical tune on its buzzer and/or ESCs.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives tune commands.

Reception (RX)

Handled by `AP_Notify::handle_play_tune`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Parses the tune string (formatted in the "Play string" format, e.g., "A B C") and queues it for playback.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `tune`: Tune string (30 chars max).
- `tune2`: Extension string (200 chars).

Practical Use Cases

1. **Lost Model Finder:**
    - *Scenario:* Drone lands in tall grass.
    - *Action:* GCS sends `PLAY_TUNE` with a loud, repetitive beep sequence to help the pilot locate it.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4346**: Handler.

SENSORS

GPS_INJECT_DATA (ID 123) RX ONLY

Summary

The `GPS_INJECT_DATA` message carries RTCM correction data sent from a ground-based RTK Base Station (via the GCS) to the vehicle. This data is injected into the onboard GPS receiver to enable Real-Time Kinematic (RTK) precision (centimeter-level accuracy).

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Receives RTCM corrections)

Reception (RX)

Reception is handled by `AP_GPS::handle_gps_inject` within libraries/AP_GPS/AP_GPS.cpp:1236.

Protocol Logic

1. **Extraction:** The message payload (`data`) and length (`len`) are extracted.
2. **Routing:** The data is passed to `handle_gps_rtcm_fragment`, which routes the raw RTCM bytes to the configured GPS instances.
3. **Forwarding:** Drivers like `AP_GPS_UBLOX` write these bytes directly to the GPS module's UART port.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `len`: Data length (bytes).
- `data`: Raw data (up to 110 bytes).

Practical Use Cases

1. **RTK Positioning:**
   - *Scenario:* A survey drone uses a Here3+ GPS.
   - *Action:* Mission Planner connects to a local Base Station (via USB or NTRIP). It packages the RTCM stream into `GPS_INJECT_DATA` messages and sends them to the drone over the telemetry link. The drone achieves "RTK Fixed" status.

Key Codebase Locations

- **libraries/AP_GPS/AP_GPS.cpp:1236**: Implementation of the handler.

LANDING_TARGET (ID 149)

RX ONLY

Summary

The `LANDING_TARGET` message is sent by a companion computer or smart camera to the autopilot. It contains the location of a landing target (like an IR beacon or AprilTag) relative to the vehicle. This data drives the **Precision Landing** feature.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Enables Precision Landing)

Reception (RX)

Reception is handled by `AC_PrecLand_Companion::handle_msg` within libraries/AC_PrecLand/AC_PrecLand_Companion.cpp:22.

Protocol Logic

1. **Frame Check:** ArduPilot expects `frame` to be `MAV_FRAME_BODY_FRD` if `position_valid` is set.
2. **Position Parsing:**
   - **If `position_valid=1`:** Uses `x`, `y`, `z` (meters) to calculate a Line-of-Sight (LOS) vector.
   - **If `position_valid=0`:** Uses `angle_x` and `angle_y` (radians) to compute the LOS vector.
3. **Timestamp:** The `time_usec` is jitter-corrected to match the autopilot's time base.
4. **Usage:** The calculated LOS vector is fed into the Precision Landing EKF to estimate the target's position and velocity relative to the vehicle.

Data Fields

- **`time_usec`:** Timestamp (micros).
- **`target_num`:** Target ID.
- **`frame`:** Coordinate frame (`MAV_FRAME`).
- **`angle_x`:** X-axis angular offset (radians).
- **`angle_y`:** Y-axis angular offset (radians).
- **`distance`:** Distance to the target (meters).
- **`size_x`:** Size of target along x-axis (radians).
- **`size_y`:** Size of target along y-axis (radians).
- **`x`:** X position (meters).
- **`y`:** Y position (meters).
- **`z`:** Z position (meters).
- **`q`:** Quaternion of landing target orientation.
- **`type`:** Type of landing target (`LANDING_TARGET_TYPE`).
- **`position_valid`:** Boolean indicating validity of x/y/z.

Practical Use Cases

1. **IR Lock:**
   - *Scenario:* A copter has an IR-Lock camera.

    - *Action:* The IR-Lock driver (onboard or external) detects a beacon and streams `LANDING_TARGET` messages. The copter adjusts its descent to land precisely on the beacon.
2. **Vision-Based Landing:**
    - *Scenario:* A Raspberry Pi runs OpenCV to track an AprilTag.
    - *Action:* The Pi sends `LANDING_TARGET` messages with the tag's relative position. ArduPilot guides the vehicle onto the tag.

Key Codebase Locations

- **libraries/AC_PrecLand/AC_PrecLand_Companion.cpp:22**: Implementation of the handler.

GPS_INPUT (ID 232)

SUPPORTED (RX ONLY)

Summary

Inject raw GPS data into the autopilot. This allows a companion computer or external system to act as a virtual GPS sensor.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives external GPS data.

Reception (RX)

Handled by `AP_GPS::handle_msg`. The data is fed into the `AP_GPS_MAV` backend.

Source: libraries/AP_GPS/AP_GPS_MAV.cpp

Protocol Logic

The autopilot treats this message as a reading from a physical GPS connected via MAVLink.

- **Time:** Must be synced or corrected.
- **Flags:** Indicate valid fields.

Data Fields

- `time_usec`: Timestamp.
- `gps_id`: ID of the GPS sensor (0-3).
- `ignore_flags`: Flags for ignored fields.
- `lat`: Latitude.
- `lon`: Longitude.
- `alt`: Altitude.
- `hdop`: HDOP.
- `vdop`: VDOP.
- `speed_accuracy`: Speed accuracy.
- `horiz_accuracy`: Horizontal accuracy.
- `vert_accuracy`: Vertical accuracy.
- `satellites_visible`: Sat count.

Practical Use Cases

1. **Visual Odometry Bridge:**
    - *Scenario:* A companion computer runs VIO (Visual Inertial Odometry).
    - *Action:* It converts VIO poses into `GPS_INPUT` messages so the flight controller can fly in "Loiter" mode without a real GPS.

Key Codebase Locations

- **libraries/AP_GPS/AP_GPS_MAV.cpp:46**: Handler.

GPS_RTCM_DATA (ID 233)

SUPPORTED (RX ONLY)

Summary

RTCM Real-Time Kinematic (RTK) corrections.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives RTK corrections.

Reception (RX)

Handled by `AP_GPS::handle_msg`.

Source: libraries/AP_GPS/AP_GPS.cpp

Protocol Logic

The autopilot receives these fragments of RTCMv3 data from the GCS (typically from an NTRIP caster or a local base station) and forwards them to the onboard GPS unit via I2C or Serial to enable RTK Fixed/Float modes.

- **fragmentation:** Max 180 bytes per message. Large RTCM packets are split across multiple messages.

Data Fields

- `flags`: Fragmentation flags.
- `len`: Data length.
- `data`: RTCM data bytes.

Practical Use Cases

1. **Centimeter-Level Positioning:**
    - *Scenario:* Surveying mission.
    - *Action:* GCS connects to a CORS network and streams `GPS_RTCM_DATA` to the drone. The drone's GPS uses this to achieve 2cm accuracy.

Key Codebase Locations

- **libraries/AP_GPS/AP_GPS.cpp:1253**: Handler.

OBSTACLE_DISTANCE (ID 330)

RX ONLY

Summary

The `OBSTACLE_DISTANCE` message carries 2D radial distance measurements from a 360-degree sensor (typically a spinning LIDAR or a ring of sonar/ToF sensors). It reports obstacles as an array of distances relative to the vehicle's heading.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Consumes sensor data for Obstacle Avoidance)

Reception (RX)

ArduPilot parses this message to populate its internal **Proximity** database, which in turn drives the **Avoidance** library (Simple Avoidance, BendyRuler, etc.).

Core Logic

The handler is implemented in `AP_Proximity_MAV::handle_obstacle_distance_msg` within libraries/AP_Proximity/AP_Proximity_MAV.cpp:120.

1. **Parsing:** It reads the `distances[]` array and the `increment` (angular width of each sector).
2. **Filtering:** It validates each reading against `min_distance` and `max_distance`.
3. **Fusion:** The valid points are pushed into the **3D Proximity Boundary** (`frontend.boundary`) and the **Object Avoidance Database** (`database_push`).
4. **Correction:** It applies any user-configured `PRX_YAW_CORR` or `PRX_ORIENT` offsets to align the sensor data with the vehicle frame.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `sensor_type`: Class id of the distance sensor type.
- `distances`: Distance of obstacles around the UAV with index 0 corresponding to local North + angle_offset, unless otherwise specified in the frame field.
- `increment`: Angular width in degrees of each array element.
- `min_distance`: Minimum distance the sensor can measure in centimeters.
- `max_distance`: Maximum distance the sensor can measure in centimeters.
- `increment_f`: Angular width in degrees of each array element.
- `angle_offset`: Relative angle offset of the 0-index element in the array.
- `frame`: Coordinate frame of reference for the yaw rotation and offset of the sensor data.

Practical Use Cases

1. **360 LIDAR Avoidance:**
    - *Scenario:* A drone is equipped with an RP-LIDAR A2.
    - *Action:* A companion computer (Raspberry Pi running ROS) reads the LIDAR, converts the point cloud into an `OBSTACLE_DISTANCE` array, and sends it to the flight controller. ArduPilot

uses this to stop the drone before it hits a wall ("Simple Avoidance").

Key Codebase Locations

- **libraries/AP_Proximity/AP_Proximity_MAV.cpp:120**: Implementation of the handler.

ODOMETRY (ID 331)

RX ONLY

Summary

The `ODOMETRY` message is a high-bandwidth packet designed to communicate visual odometry or VIO (Visual Inertial Odometry) data to the Autopilot. It is primarily used to provide external navigation data (Position, Velocity, Attitude) to the EKF3 when GPS is unavailable (e.g., indoor flight).

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Consumes VIO data for EKF fusion)

Reception (RX)

ArduPilot acts as a consumer of this message, typically from a Companion Computer running ROS or a smart camera (like Realsense T265 or ModalAI VOXL).

Core Logic

The handler is implemented in `GCS_MAVLINK::handle_odometry` within libraries/GCS_MAVLink/GCS_Common.cpp:3882.

Strict Frame Requirements

ArduPilot is very strict about the coordinate frames used in this message. If these fields do not match exactly, the message is **silently ignored**:

- `frame_id` MUST be `MAV_FRAME_LOCAL_FRD` (20).
- `child_frame_id` MUST be `MAV_FRAME_BODY_FRD` (12).

Data Fields

- **`time_usec`**: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- **`frame_id`**: Coordinate frame of reference for the pose data.
- **`child_frame_id`**: Coordinate frame of reference for the velocity data.
- **`x`**: X Position (meters).
- **`y`**: Y Position (meters).
- **`z`**: Z Position (meters).
- **`q`**: Quaternion components, w, x, y, z (1 0 0 0 is the null-rotation).
- **`vx`**: X Linear velocity (m/s).
- **`vy`**: Y Linear velocity (m/s).
- **`vz`**: Z Linear velocity (m/s).
- **`rollspeed`**: Roll angular speed (rad/s).
- **`pitchspeed`**: Pitch angular speed (rad/s).
- **`yawspeed`**: Yaw angular speed (rad/s).
- **`pose_covariance`**: Pose covariance matrix upper right triangle.
- **`velocity_covariance`**: Velocity covariance matrix upper right triangle.
- **`reset_counter`**: Estimate reset counter.
- **`estimator_type`**: Type of estimator that is providing the odometry.

- `quality`: Optional odometry quality metric as a percentage.

Practical Use Cases

1. **Indoor Non-GPS Flight:**
   - *Scenario:* A drone flying inside a warehouse using a Realsense T265.
   - *Action:* The companion computer reads the camera, converts the pose to MAVLink `ODOMETRY` (ensuring FRD frames), and sends it to the flight controller. The EKF3 fuses this data to hold position without GPS.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3882**: Implementation of the handler.

VISION_POSITION_DELTA (ID 11011) RX ONLY

Summary

The `VISION_POSITION_DELTA` message reports the change in position (delta) and change in angle (delta) of the vehicle frame since the last update. This is an alternative to providing absolute `ODOMETRY` and is often easier for optical flow sensors or visual SLAM systems to calculate.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Consumes VIO data for EKF fusion)

Reception (RX)

ArduPilot consumes this message via the `AP_VisualOdom` library.

Core Logic

The handler is implemented in `AP_VisualOdom_Backend::handle_vision_position_delta_msg` within libraries/AP_VisualOdom/AP_VisualOdom_Backend.cpp:41.

1. **Rotation:** It rotates the incoming delta vectors based on the `VISO_ORIENT` parameter (e.g., if the camera is facing down or backward).
2. **Fusion:** It calls `AP::[ahrs](/field-manual/mavlink-interface/ahrs.html)().writeBodyFrameOdom` to send the delta measurements to the EKF3.

Data Fields

- **`time_usec`**: Timestamp (us since UNIX epoch).
- **`time_delta_usec`**: Time since the last reported delta.
- **`angle_delta`**: Change in angular position (roll, pitch, yaw) in radians.
- **`position_delta`**: Change in position (x, y, z) in meters.
- **`confidence`**: Confidence level (0-100\%).

Practical Use Cases

1. **Optical Flow:**
   - *Scenario:* A specialized optical flow sensor calculates the distance moved (delta) between frames rather than an absolute position.
   - *Action:* The sensor sends `VISION_POSITION_DELTA` updates. The EKF integrates these deltas to estimate velocity and hold position.

Key Codebase Locations

- **libraries/AP_VisualOdom/AP_VisualOdom_Backend.cpp:41**: Implementation of the handler.

OBSTACLE_DISTANCE_3D (ID 11037)

RX ONLY

Summary

The `OBSTACLE_DISTANCE_3D` message provides the location of an obstacle as a 3D vector relative to the vehicle's body frame. Unlike the 2D `OBSTACLE_DISTANCE` array, this message reports individual points in 3D space, making it suitable for depth cameras (Realsense, OAK-D) that can detect obstacles above or below the vehicle.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** Autopilot (Consumes sensor data for Avoidance)

Reception (RX)

ArduPilot acts as a consumer of this message, feeding the **Proximity** library.

Core Logic

The handler is implemented in `AP_Proximity_MAV::handle_obstacle_distance_3d_msg` within libraries/AP_Proximity/AP_Proximity_MAV.cpp:215.

1. **Frame Check:** It strictly requires `MAV_FRAME_BODY_FRD`.
2. **Batching:** It accumulates points into a temporary boundary buffer. When the `time_boot_ms` timestamp changes (indicating a new frame of data), the accumulated points are pushed to the main 3D boundary.
3. **Mapping:** It calculates the pitch and yaw of the obstacle vector to assign it to the correct sector in the 3D spherical buffer.

Data Fields

- **`time_boot_ms`:** Timestamp (ms since boot). All points belonging to the same "frame" (e.g., depth image) should share the same timestamp.
- **`sensor_type`:** Class id of the distance sensor type.
- **`frame`:** Coordinate frame (Must be `MAV_FRAME_BODY_FRD`).
- **`obstacle_id`:** Unique ID of the obstacle (unused by ArduPilot).
- **`x`, `y`, `z`:** Position of the obstacle in meters (FRD).
- **`min_distance`:** Minimum distance the sensor can measure.
- **`max_distance`:** Maximum distance the sensor can measure.

Practical Use Cases

1. **Depth Camera Avoidance:**
   - *Scenario:* A drone with a forward-facing Realsense D435.
   - *Action:* The companion computer processes the depth map. For every "close" pixel cluster, it sends an `OBSTACLE_DISTANCE_3D` message. ArduPilot builds a local 3D map around the vehicle and prevents the pilot from flying forward into the wall.

Key Codebase Locations

- **libraries/AP_Proximity/AP_Proximity_MAV.cpp:215**: Implementation of the handler.

CONTROL

SET_GPS_GLOBAL_ORIGIN (ID 48)

SUPPORTED (RX ONLY)

Summary

Sets the GPS coordinates of the vehicle's local origin (0,0,0).

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives EKF origin data.

Reception (RX)

Handled by `GCS_MAVLink::handle_set_gps_global_origin`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Used to manually set the EKF origin. This is crucial for non-GPS navigation (e.g., Optical Flow, Vicon) where the vehicle needs a global reference frame to align with the real world or to "fake" a GPS lock at a specific location.

- **Sets:** `AP::[ahrs](/field-manual/mavlink-interface/ahrs.html)().set_origin()`.

Data Fields

- `target_system`: System ID.
- `latitude`: Latitude (deg * 1E7).
- `longitude`: Longitude (deg * 1E7).
- `altitude`: Altitude (mm).
- `time_usec`: Timestamp.

Practical Use Cases

1. **Indoor-Outdoor Transition:**
    - *Scenario:* A drone takes off indoors (Vicon) and flies outdoors (GPS).
    - *Action:* The GCS or companion computer sets the global origin to the building's entrance coordinates so the indoor local coordinates map correctly to global lat/lon.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4192**: Handler.

REQUEST_DATA_STREAM (ID 66)

SUPPORTED (RX ONLY)

Summary

Request a data stream (e.g., "All", "Raw Sensors", "RC Channels").

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives stream requests.

Description

This message is **Deprecated** in favor of `MAV_CMD_SET_MESSAGE_INTERVAL` but remains widely supported for legacy GCS compatibility.

Reception (RX)

Handled by `GCS_MAVLINK::handle_request_data_stream`.

Source: libraries/GCS_MAVLink/GCS_Param.cpp

Protocol Logic

The GCS requests a "Stream ID" (e.g., `MAV_DATA_STREAM_EXTENDED_STATUS`) and a rate (Hz). ArduPilot maps this ID to a set of actual MAVLink messages and sets their transmission interval.

- **SRx_EXTRA1:** Attitude, etc.
- **SRx_POSITION:** GPS, Global Position.
- **SRx_RAW_SENSORS:** Raw IMU.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `req_stream_id`: ID of requested stream (0=All).
- `req_message_rate`: Rate in Hz.
- `start_stop`: 1 to start, 0 to stop.

Practical Use Cases

1. **Connecting GCS:**
    - *Scenario:* Mission Planner connects.
    - *Action:* It sends `REQUEST_DATA_STREAM` for "ALL" streams at defined rates to start the telemetry flow.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Param.cpp:129**: Handler.

MANUAL_CONTROL (ID 69) SUPPORTED (RX ONLY)

Summary

Raw manual control inputs (Joystick) from the GCS.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives joystick input.

Reception (RX)

Handled by `GCS_MAVLINK::handle_manual_control`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Used for "Joystick" or "Virtual RC" control where the GCS sends control axes directly, rather than the RC receiver.

- **Mapping:** The X/Y/Z/R axes are mapped to Roll/Pitch/Throttle/Yaw (or similar) based on vehicle type and mode.
- **Buttons:** Button bits are decoded to trigger auxiliary functions (like arming, mode switching).

Data Fields

- `target`: Target system.
- `x`: X-axis (Pitch) -1000..1000.
- `y`: Y-axis (Roll) -1000..1000.
- `z`: Z-axis (Throttle) 0..1000.
- `r`: R-axis (Yaw) -1000..1000.
- `buttons`: 16-bit button mask.
- `buttons2`: Extension for more buttons.

Practical Use Cases

1. **Flying via Tablet:**
   - *Scenario:* User flies a drone using on-screen virtual joysticks in QGroundControl.
   - *Action:* QGC sends `MANUAL_CONTROL` messages. ArduPilot treats these as pilot input, overriding (or replacing) the physical RC receiver.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:7038**: Handler.

COMMAND_INT (ID 75)

SUPPORTED (RX ONLY)

Summary

Send a command with up to seven parameters to the MAV. This is the preferred method for sending commands that involve location data (Latitude/Longitude), as it uses integers for precision.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives commands.

Reception (RX)

Handled by `GCS_MAVLINK::handle_command_int`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

- **Decoding:** Extracts the command ID (`MAV_CMD`) and parameters.
- **Routing:** Dispatches to the appropriate handler (e.g., `handle_MAV_CMD_DO_SET_ROI`).
- **Frame:** Explicitly handles the `frame` field (e.g., `MAV_FRAME_GLOBAL_RELATIVE_ALT`), which `COMMAND_LONG` lacks.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `frame`: Coordinate frame (MAV_FRAME).
- `command`: Command ID (MAV_CMD).
- `current`: (Not used).
- `autocontinue`: (Not used).
- `param1`: Parameter 1 (float).
- `param2`: Parameter 2 (float).
- `param3`: Parameter 3 (float).
- `param4`: Parameter 4 (float).
- `x`: Latitude/X (int32).
- `y`: Longitude/Y (int32).
- `z`: Altitude/Z (float).

Practical Use Cases

1. **Region of Interest (ROI):**
   - *Scenario:* User points the camera at a specific GPS coordinate.
   - *Action:* GCS sends `COMMAND_INT` (CMD: `MAV_CMD_DO_SET_ROI`) with the lat/lon in `x`/`y`.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5643**: Handler.

SET_ATTITUDE_TARGET (ID 82)

SUPPORTED (RX ONLY)

Summary

Sets the vehicle's attitude (roll, pitch, yaw) and throttle target. Primary method for offboard control in "Guided" mode.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives offboard control targets.

Reception (RX)

Handled by `GCS_MAVLINK::handle_set_attitude_target`.

Source: ArduCopter/GCS_Mavlink.cpp

Protocol Logic

- **Bitmask:** The `type_mask` field determines which fields are ignored. ArduPilot supports various combinations (e.g., Attitude only, Rates only, or mixed).
- **Input:** Quaternions for attitude, rad/s for rates, 0..1 for thrust.
- **Mode:** Requires vehicle to be in `GUIDED` (Copter) or similar offboard mode.

Data Fields

- `time_boot_ms`: Timestamp.
- `target_system`: System ID.
- `target_component`: Component ID.
- `type_mask`: Bitmap to ignore fields.
- `q`: Attitude quaternion (w, x, y, z).
- `body_roll_rate`: Roll rate (rad/s).
- `body_pitch_rate`: Pitch rate (rad/s).
- `body_yaw_rate`: Yaw rate (rad/s).
- `thrust`: Collective thrust (0.0 to 1.0).

Practical Use Cases

1. **Companion Computer Control:**
   - *Scenario:* An onboard NVIDIA Jetson is flying the drone using a neural network.
   - *Action:* The Jetson sends `SET_ATTITUDE_TARGET` at 50Hz to control the drone's orientation and thrust directly.

Key Codebase Locations

- **ArduCopter/GCS_Mavlink.cpp:1205**: Handler.

SET_POSITION_TARGET_LOCAL_NED (ID 84)

SUPPORTED (RX ONLY)

Summary

Sets the vehicle's position, velocity, and/or acceleration target in a local North-East-Down (NED) frame.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives local offboard control targets.

Reception (RX)

Handled by `GCS_MAVLINK::handle_set_position_target_local_ned`.

Source: ArduCopter/GCS_Mavlink.cpp

Protocol Logic

- **Frame:** Supports `MAV_FRAME_LOCAL_NED`, `MAV_FRAME_LOCAL_OFFSET_NED`, `MAV_FRAME_BODY_NED`, `MAV_FRAME_BODY_OFFSET_NED`.
- **Mask:** `type_mask` ignores unused fields.
- **Control:** Can control Position (XYZ), Velocity (XYZ), Acceleration (XYZ), and Yaw/YawRate.

Data Fields

- `time_boot_ms`: Timestamp.
- `target_system`: System ID.
- `target_component`: Component ID.
- `coordinate_frame`: Valid MAV_FRAME.
- `type_mask`: Ignore flags.
- `x`: X Position (m).
- `y`: Y Position (m).
- `z`: Z Position (m).
- `vx`: X Velocity (m/s).
- `vy`: Y Velocity (m/s).
- `vz`: Z Velocity (m/s).
- `afx`: X Acceleration (m/s^2).
- `afy`: Y Acceleration (m/s^2).
- `afz`: Z Acceleration (m/s^2).
- `yaw`: Yaw angle (rad).
- `yaw_rate`: Yaw rate (rad/s).

Practical Use Cases

1. **Vision-Based Following:**
   - *Scenario:* A drone tracks a person using a camera.
   - *Action:* The onboard computer calculates the relative vector to the person and sends velocity commands (`vx`, `vy`) via `SET_POSITION_TARGET_LOCAL_NED` (Frame: `MAV_FRAME_BODY_NED`)

to keep the person centered.

Key Codebase Locations

- **ArduCopter/GCS_Mavlink.cpp:1280**: Handler.

SET_POSITION_TARGET_GLOBAL_INT (ID 86)

SUPPORTED (RX ONLY)

Summary

Sets the vehicle's position, velocity, and/or acceleration target in the Global frame (Latitude/Longitude).

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives global offboard control targets.

Reception (RX)

Handled by `GCS_MAVLINK::handle_set_position_target_global_int`.

Source: ArduCopter/GCS_Mavlink.cpp

Protocol Logic

- **Frame:** Supports `MAV_FRAME_GLOBAL_INT`, `MAV_FRAME_GLOBAL_RELATIVE_ALT_INT`, `MAV_FRAME_GLOBAL_TERRAIN_ALT_INT`.
- **Precision:** Uses integers (deg * 1E7) for Lat/Lon to prevent floating point errors.

Data Fields

- `time_boot_ms`: Timestamp.
- `target_system`: System ID.
- `target_component`: Component ID.
- `coordinate_frame`: MAV_FRAME.
- `type_mask`: Ignore flags.
- `lat_int`: Latitude.
- `lon_int`: Longitude.
- `alt`: Altitude (m).
- `vx`: Velocity X.
- `vy`: Velocity Y.
- `vz`: Velocity Z.
- `afx`: Accel X.
- `afy`: Accel Y.
- `afz`: Accel Z.
- `yaw`: Yaw.
- `yaw_rate`: Yaw Rate.

Practical Use Cases

1. **Dynamic Re-tasking:**
    - *Scenario:* Search and Rescue drone.
    - *Action:* Operator clicks a point on the map. The GCS sends a `SET_POSITION_TARGET_GLOBAL_INT` to fly the drone to that specific coordinate in Guided mode.

Key Codebase Locations

- **ArduCopter/GCS_Mavlink.cpp:1386**: Handler.

FOLLOW_TARGET (ID 144)

RX ONLY

Summary

The `FOLLOW_TARGET` message reports the kinematic state (position, velocity, attitude) of a target vehicle or object that the drone should follow. This message is typically sent by a Ground Control Station (GCS) or a companion computer acting as a "virtual beacon."

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Updates Follow Me target)

Reception (RX)

Reception is handled by `AP_Follow::handle_follow_target_message` within libraries/AP_Follow/AP_Follow.cpp:374.

Protocol Logic

1. **Validation:** The message is ignored if the Lat/Lon are zero or if the `est_capabilities` bitmask doesn't indicate valid position data.
2. **Target Update:**
   - **Position:** Lat, Lon, and Alt (AMSL) are updated.
   - **Velocity:** If provided (`est_capabilities` bit 1), the target's NED velocity is updated.
   - **Heading:** If attitude quaternion is provided (`est_capabilities` bit 3), the target's heading is extracted.
3. **Timestamping:** The `timestamp` field is jitter-corrected to align the external time base with the autopilot's boot time.

Data Fields

- `timestamp` **:** Timestamp (ms since boot).
- `est_capabilities` **:** Bitmask of valid fields (1: Pos, 2: Vel, 4: Accel, 8: Att).
- `lat` **:** Latitude (deg * 1E7).
- `lon` **:** Longitude (deg * 1E7).
- `alt` **:** Altitude (meters AMSL).
- `vel` **:** Velocity (m/s) in NED frame.
- `acc` **:** Acceleration (m/s^2).
- `attitude_q` **:** Attitude quaternion (w, x, y, z).
- `rates` **:** Angular rates (rad/s).
- `position_cov` **:** Position covariance.
- `custom_state` **:** Custom state field.

Practical Use Cases

1. **Computer Vision Tracking:**
   - *Scenario:* A companion computer tracks a car using a camera.

    - *Action:* It estimates the car's GPS coordinates and velocity, then streams `FOLLOW_TARGET` to the flight controller. The drone flies in "Follow" mode, smoothly tracking the car.
2. **Swarm Formation:**
    - *Scenario:* Drone B follows Drone A.
    - *Action:* Drone A broadcasts its position (via `GLOBAL_POSITION_INT` or similar). The GCS or an onboard script reformats this into `FOLLOW_TARGET` and sends it to Drone B.

Key Codebase Locations

- **libraries/AP_Follow/AP_Follow.cpp:374**: Implementation of the handler.

MISSION

MISSION_WRITE_PARTIAL_LIST (ID 38)

SUPPORTED (RX ONLY)

Summary

Initiate a partial write transaction for mission items.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives partial upload request.

Reception (RX)

Handled by `GCS_MAVLINK::handle_mission_write_partial_list`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Allows the GCS to update a specific range of waypoints (e.g., changing points 10-15 of a 100-point mission) without re-uploading the entire list.

1. GCS sends `MISSION_WRITE_PARTIAL_LIST`.
2. Vehicle responds with `MISSION_REQUEST` for the first item in the range.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `start_index`: Start index.
- `end_index`: End index.

Practical Use Cases

1. **In-Flight Mission Update:**
   - *Scenario:* Pilot wants to move a few waypoints mid-mission.
   - *Action:* GCS uploads only the modified points using this message, saving bandwidth.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4513**: Handler.

MISSION_REQUEST (ID 40)

SUPPORTED (RX ONLY)

Summary

Request the information of a mission item.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives request for item.

Reception (RX)

Handled by `GCS_MAVLINK::handle_mission_request`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

This message is **deprecated** in favor of `MISSION_REQUEST_INT` (51) but is still supported for backward compatibility.

- ArduPilot converts it internally to a `MISSION_REQUEST_INT` and processes it.
- A warning is sent to the GCS: "got MISSION_REQUEST; use MISSION_REQUEST_INT!".

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `seq`: Sequence number of the mission item.

Practical Use Cases

1. **Legacy GCS Support:**
   - *Scenario:* Connecting an old version of Mission Planner.
   - *Action:* It uses this message to download waypoints.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4529**: Handler.

MISSION_REQUEST_INT (ID 51)

SUPPORTED (RX ONLY)

Summary

Request the information of a mission item with the sequence number `seq`.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives request for a mission item.

Reception (RX)

Handled by `GCS_MAVLINK::handle_mission_request_int`.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

1. GCS sends `MISSION_REQUEST_INT` with a sequence number.
2. ArduPilot retrieves the waypoint from storage.
3. ArduPilot responds with `MISSION_ITEM_INT`.

Data Fields

- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`seq`:** Sequence number of the mission item to fetch.
- **`mission_type`:** Mission type (Mission, Fence, Rally).

Practical Use Cases

1. **Downloading Mission:**
   - *Scenario:* User clicks "Read" in Mission Planner.
   - *Action:* Mission Planner iterates through all waypoints, sending a `MISSION_REQUEST_INT` for each one to download the plan from the drone.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:629**: Handler.

TERRAIN_DATA (ID 134)

RX ONLY

Summary

The `TERRAIN_DATA` message carries a 4×4 grid of terrain height measurements. It is sent by the Ground Control Station (GCS) to the vehicle in response to a `TERRAIN_REQUEST`.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Updates terrain database)

Reception (RX)

Reception is handled by `AP_Terrain::handle_terrain_data` within libraries/AP_Terrain/TerrainGCS.cpp:264.

Core Logic

1. **Validation:** It verifies that the incoming packet corresponds to a pending request in the internal grid cache.
2. **Population:** It maps the 4×4 data block into the larger 8×7 internal grid structure.
3. **Persistence:** Once a grid is fully populated or sufficiently updated, it is marked as `GRID_CACHE_DIRTY`, triggering a write to the SD card database.

Data Fields

- **`lat`**: Latitude of grid (deg * 1E7).
- **`lon`**: Longitude of grid (deg * 1E7).
- **`grid_spacing`**: Grid spacing (in meters).
- **`gridbit`**: Index of the 4×4 block within the larger grid.
- **`data`**: Array of 16 int16_t values representing terrain height in meters.

Practical Use Cases

1. **Offline Terrain Following:**
   - *Scenario:* A user uploads a mission with `TERRAIN_FRAME` waypoints but the drone is not yet flying.
   - *Action:* The GCS proactively pushes `TERRAIN_DATA` for the mission area to the drone. The drone stores this on its SD card. During flight, even if telemetry is lost, the drone has the data needed to fly safely close to the ground.

Key Codebase Locations

- **libraries/AP_Terrain/TerrainGCS.cpp:264**: Implementation of the handler.

RALLY_FETCH_POINT (ID 176)

SUPPORTED (RX ONLY)

Summary

Request to fetch a specific rally point from the vehicle.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives request from GCS.

Reception (RX)

Handled by `GCS_MAVLINK::handle_rally_fetch_point`. The vehicle responds by sending a `RALLY_POINT` message for the requested index.

Source: libraries/GCS_MAVLink/GCS_Rally.cpp

Protocol Logic

1. GCS sends `RALLY_FETCH_POINT` with `idx`.
2. Vehicle retrieves point at `idx`.
3. Vehicle replies with `RALLY_POINT`.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `idx`: Index of the rally point to fetch.

Practical Use Cases

1. **Downloading Safe Points:**
   - *Scenario:* GCS connects to a drone and wants to display existing safe zones.
   - *Action:* GCS iterates through indices, sending `RALLY_FETCH_POINT` for each.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Rally.cpp:73**: `handle_rally_fetch_point` implementation.

PAYLOAD

DIGICAM_CONTROL (ID 155)

RX ONLY

Summary

The `DIGICAM_CONTROL` message is a legacy method for controlling onboard cameras (taking photos, zooming, etc.). It is largely deprecated in favor of the `MAV_CMD_DO_DIGICAM_CONTROL` command, but ArduPilot still supports receiving this message for backward compatibility.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Triggers camera action)

Reception (RX)

Reception is handled by `AP_Camera::handle_message` within libraries/AP_Camera/AP_Camera.cpp:261.

Protocol Logic

1. **Decoding:** The message is decoded into a `mavlink_digicam_control_t` packet.
2. **Action:** The `control()` function is called with the unpacked parameters.
3. **Triggering:** If `shot` is 1, the camera shutter is triggered via the configured backend (Servo, Relay, or MAVLink).

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `session`: Session control (e.g. show/hide lens).
- `zoom_pos`: Zoom's absolute position.
- `zoom_step`: Zooming step value to offset zoom from the current position.
- `focus_lock`: Focus Locking, Unlocking or Re-locking.
- `shot`: Shooting Command (1 to take a picture).
- `command_id`: Command Identity.
- `extra_param`: Extra parameter.
- `extra_value`: Extra value.

Practical Use Cases

1. **Legacy GCS Support:**
   - *Scenario:* An older Ground Control Station only supports this message for camera triggering.
   - *Action:* When the user clicks "Trigger Camera", the GCS sends `DIGICAM_CONTROL`. ArduPilot receives it and fires the relay to take a picture.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera.cpp:261**: Implementation of the handler.

GIMBAL_REPORT (ID 200)

SUPPORTED (RX ONLY)

Summary

Status report from a 3DR Solo Gimbal (or compatible device).

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives status from the gimbal.

Reception (RX)

Handled by `GCS_MAVLink` and passed to `AP_Mount`.

Source: libraries/AP_Mount/AP_Mount.cpp

Protocol Logic

The `AP_Mount` library decodes the message to update its internal state of the gimbal's actual angles.

- **Note:** This is specific to the "Solo Gimbal" protocol, which is distinct from the newer MAVLink Gimbal Protocol v2 (`GIMBAL_DEVICE_...`).

Data Fields

- **`target_system`**: System ID.
- **`target_component`**: Component ID.
- **`delta_time`**: Time since last update.
- **`delta_angle_x`**: Delta angle X.
- **`delta_angle_y`**: Delta angle Y.
- **`delta_angle_z`**: Delta angle Z.
- **`delta_velocity_x`**: Delta velocity X.
- **`delta_velocity_y`**: Delta velocity Y.
- **`delta_velocity_z`**: Delta velocity Z.
- **`joint_roll`**: Joint angle roll.
- **`joint_el`**: Joint angle elevation.
- **`joint_az`**: Joint angle azimuth.

Practical Use Cases

1. **Solo Gimbal Integration:**
    - *Scenario:* Using a legacy 3DR Solo Gimbal.
    - *Action:* The gimbal reports its joint angles, allowing the autopilot to stabilize it or display its orientation.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount.cpp:996**: Handler.

GOPRO_HEARTBEAT (ID 215)

SUPPORTED (RX ONLY)

Summary

Heartbeat from a GoPro camera connected via a Solo Gimbal.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Detects GoPro presence.

Reception (RX)

Handled by `GCS_MAVLink` and used for routing logic.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Used to determine if a GoPro is connected and active on the gimbal link.

Data Fields

- `status`: Status flags.
- `capture_mode`: Current capture mode.
- `flags`: Additional flags.

Practical Use Cases

1. **Gimbal Integration:**
    - *Scenario:* User turns on Solo drone.
    - *Action:* Autopilot receives `GOPRO_HEARTBEAT` and enables GoPro control widgets in the GCS.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4227**: Handler.

GIMBAL_DEVICE_INFORMATION (ID 283)

RX ONLY

Summary

The `GIMBAL_DEVICE_INFORMATION` message provides static configuration data about a specific gimbal device, such as its vendor name, model name, firmware version, and physical angular limits. In the MAVLink Gimbal Protocol v2 architecture, this message is typically emitted by the **Gimbal Device** (the physical hardware) and consumed by the **Gimbal Manager** (the Autopilot).

Status

RX Only

Directionality

- **TX (Transmit):** None (ArduPilot typically does not generate this message).
- **RX (Receive):** Autopilot (Receives from MAVLink Gimbals like Gremsy).

Reception (RX)

ArduPilot listens for this message to auto-configure its internal gimbal limits.

Core Logic

The handler is implemented in `AP_Mount_Gremsy::handle_gimbal_device_information` within libraries/AP_Mount/AP_Mount_Gremsy.cpp:181.

1. **Discovery:** When a MAVLink gimbal connects, ArduPilot requests this message.
2. **Configuration:** It extracts `roll_min`, `roll_max`, `pitch_min`, `pitch_max`, etc., and updates the internal `AP_Mount` parameters.
3. **Propagation:** These limits are then re-packaged and sent to the GCS via the `GIMBAL_MANAGER_INFORMATION` message.

Data Fields

- **`time_boot_ms`**: Timestamp (milliseconds since system boot).
- **`vendor_name`**: Name of the gimbal vendor.
- **`model_name`**: Name of the gimbal model.
- **`custom_name`**: Custom name of the gimbal given to it by the user.
- **`firmware_version`**: Version of the gimbal firmware, encoded as: (Dev & 0xff) << 24 | (Patch & 0xff) << 16 | (Minor & 0xff) << 8 | (Major & 0xff).
- **`hardware_version`**: Version of the gimbal hardware, encoded as: (Dev & 0xff) << 24 | (Patch & 0xff) << 16 | (Minor & 0xff) << 8 | (Major & 0xff).
- **`uid`**: UID of gimbal device (or 0 if not known).
- **`cap_flags`**: Bitmap of gimbal capability flags.
- **`custom_cap_flags`**: Bitmap of custom gimbal capability flags.
- **`roll_min`**: Minimum hardware roll angle (positive: rolling to the right, negative: rolling to the left).
- **`roll_max`**: Maximum hardware roll angle (positive: rolling to the right, negative: rolling to the left).
- **`pitch_min`**: Minimum hardware pitch angle (positive: up, negative: down).
- **`pitch_max`**: Maximum hardware pitch angle (positive: up, negative: down).
- **`yaw_min`**: Minimum hardware yaw angle (positive: to the right, negative: to the left).
- **`yaw_max`**: Maximum hardware yaw angle (positive: to the right, negative: to the left).

Practical Use Cases

1. **Plug-and-Play Configuration:**
   - *Scenario:* A user swaps a Gremsy T3 for a Gremsy Mio.
   - *Action:* The Autopilot receives `GIMBAL_DEVICE_INFORMATION` from the new gimbal, sees different physical limits, and automatically updates its internal safety constraints without user intervention.

Key Codebase Locations

- **libraries/AP_Mount/AP_Mount_Gremsy.cpp:181**: Implementation of the handler.

LOGGING

LOG_REQUEST_LIST (ID 117)

RX ONLY

Summary

The `LOG_REQUEST_LIST` message is sent by a Ground Control Station (GCS) to request a list of available logs on the vehicle's DataFlash storage. The vehicle responds by streaming `LOG_ENTRY` messages for the requested range.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Starts log listing)

Reception (RX)

Reception is handled by `AP_Logger::handle_log_request_list` within libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:68.

Protocol Logic

1. **State Check:** It checks if a log download is already in progress (`_log_sending_link ≠ nullptr`). If so, it rejects the request.
2. **Range Setup:** It parses the `start` and `end` log indices.
3. **Sanitization:** It clamps the range against the actual number of logs present (`_log_num_logs`).
4. **Activation:** It sets the `transfer_activity` state to `TransferActivity::LISTING` and assigns the requesting link as the active channel. The scheduler then iteratively sends `LOG_ENTRY` messages.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `start`: First log ID to list.
- `end`: Last log ID to list.

Practical Use Cases

1. **Log Browser:**
    - *Scenario:* A user connects to the drone via USB and opens the "Download Logs" screen in Mission Planner.
    - *Action:* The GCS sends `LOG_REQUEST_LIST` (start=0, end=0xFFFF) to get a full directory of flight logs.

Key Codebase Locations

- **libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:68**: Implementation of the handler.

LOG_REQUEST_DATA (ID 119)

RX ONLY

Summary

The `LOG_REQUEST_DATA` message is sent by a Ground Control Station (GCS) to request a specific chunk of data from a log file. It initiates or continues the download of a DataFlash log.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Starts/Continues log download)

Reception (RX)

Reception is handled by `AP_Logger::handle_log_request_data` within libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:107.

Protocol Logic

1. **State Check:** Verifies if another download is active on a different channel.
2. **Initialization:** If this is a new request or a different log ID, it looks up the log's physical location on storage (page/offset boundaries).
3. **Range Setup:** It sets the internal read pointer (`_log_data_offset`) to the requested offset (`ofs`) and calculates the bytes remaining to send (`count`).
4. **Activation:** It transitions to the `TransferActivity::SENDING` state, which causes the scheduler to pump `LOG_DATA` packets.

Data Fields

- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`id`:** Log ID (from `LOG_ENTRY`).
- **`ofs`:** Offset into the log file (bytes).
- **`count`:** Number of bytes to send.

Practical Use Cases

1. **Downloading a Log:**
   - *Scenario:* A user downloads a 10MB log.
   - *Action:* The GCS sends a sequence of `LOG_REQUEST_DATA` messages.
     1. `id=5, ofs=0, count=90` → Receives `LOG_DATA`.
     2. `id=5, ofs=90, count=90` → Receives `LOG_DATA`.
     3. ...until EOF.

Key Codebase Locations

- **libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:107**: Implementation of the handler.

LOG_ERASE (ID 121)

RX ONLY

Summary

The `LOG_ERASE` message is sent by a Ground Control Station (GCS) to command the vehicle to erase all stored DataFlash logs.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Erases all logs)

Reception (RX)

Reception is handled by `AP_Logger::handle_log_request_erase` within libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:165.

Protocol Logic

1. **Decoding:** The message is decoded (though it has no fields to use).
2. **Action:** The `EraseAll()` function is called on the logger backend. This typically reformats the SD card or erases the flash chip, removing *all* logs.

Data Fields

- `target_system`**:** System ID.
- `target_component`**:** Component ID.

Practical Use Cases

1. **Maintenance:**
    - *Scenario:* A user wants to clear space on the SD card before a new mission.
    - *Action:* Clicking "Erase Logs" in the GCS sends this message.

Key Codebase Locations

- **libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:165**: Implementation of the handler.

LOG_REQUEST_END (ID 122)

RX ONLY

Summary

The `LOG_REQUEST_END` message is sent by a Ground Control Station (GCS) to terminate a log transfer session. This tells the vehicle to stop sending `LOG_DATA` or `LOG_ENTRY` messages and release any resources associated with the transfer.

Status

RX Only

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles (Stops log transfer)

Reception (RX)

Reception is handled by `AP_Logger::handle_log_request_end` within libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:174.

Protocol Logic

1. **State Reset:** It sets `transfer_activity` to `IDLE`.
2. **Resource Release:** It clears the `_log_sending_link` pointer, allowing other MAVLink channels to initiate log transfers.
3. **Backend Notification:** It calls `end_log_transfer()` on the active logger backend (e.g., to close the file handle).

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.

Practical Use Cases

1. **Cancel Download:**
   - *Scenario:* A user realizes they selected the wrong log and clicks "Cancel" in the GCS.
   - *Action:* The GCS sends `LOG_REQUEST_END`, and the stream of `LOG_DATA` packets stops immediately.
2. **Download Complete:**
   - *Scenario:* The GCS successfully receives the last byte of the log file.
   - *Action:* It sends `LOG_REQUEST_END` to politely close the session.

Key Codebase Locations

- **libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:174**: Implementation of the handler.

REMOTE_LOG_BLOCK_STATUS (ID 185)

SUPPORTED (RX ONLY)

Summary

Status/Acknowledgment message for the Remote Logging protocol. Sent by the listener (e.g., companion computer) to the autopilot.

Status

Supported (RX Only)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** All Vehicles - Receives acks for sent log blocks.

Reception (RX)

Handled by `AP_Logger::handle_remote_log_block_status`.

Source: libraries/AP_Logger/AP_Logger.cpp

Protocol Logic

The autopilot maintains a buffer of sent blocks. It waits for this status message to confirm which blocks have been safely received by the remote node.

- **Gap Filling:** If the status indicates missing blocks, the autopilot re-sends them.

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `seqno`: Latest received sequence number.
- `status`: Boolean/Enum status (1=OK).

Practical Use Cases

1. **Reliable Log Streaming:**
    - *Scenario:* Companion computer misses a packet due to CPU load.
    - *Action:* It sends `REMOTE_LOG_BLOCK_STATUS` with the last valid sequence number. ArduPilot resends the missing data.

Key Codebase Locations

- **libraries/AP_Logger/AP_Logger.cpp:850**: Handler.

TRANSMIT ONLY

TELEMETRY

DATA16 (ID 169)

SUPPORTED (TX ONLY)

Summary

Generic 16-byte data packet. In ArduPilot, this is used exclusively by the `AP_Radio` library to facilitate firmware updates for attached telemetry radios (like the Cypress or CC2500 based radios).

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** Specific Driver (AP_Radio) - Sends firmware data chunks to the radio hardware.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message when performing a firmware upload to a connected telemetry radio.

Source: libraries/AP_Radio/AP_Radio_cc2500.cpp

Protocol Logic

The `AP_Radio` driver chunks the firmware binary and sends it using type `42`.

- **Type:** 42 (Firmware Update).
- **Len:** Length of data in bytes.
- **Data:** Raw bytes.

Data Fields

- `type`: Data type ID (42 for firmware upload).
- `len`: Data length.
- `data`: Raw data (16 bytes).

Practical Use Cases

1. **Radio Firmware Update:**
    - *Scenario:* A user initiates a radio firmware update via Mission Planner.
    - *Action:* ArduPilot acts as a bridge, receiving the firmware via MAVLink (likely in `DATA96`) and writing it to the radio chip using `DATA16` (or direct SPI/UART depending on the exact hardware path, but `DATA16` is the MAVLink encapsulation for this specific radio driver). *Correction:* The search results show `mavlink_msg_data16_send` being called *by the radio driver*. This suggests the autopilot might be sending data *back* to the GCS or to another node? Actually, looking at the code `mavlink_msg_data16_send(fwupload.chan...` suggests it's sending it *out* via MAVLink. If `AP_Radio` is the *radio driver* on the autopilot, it might be reporting status or echoing data.

    Re-reading code: `mavlink_msg_data16_send(fwupload.chan, 42, 4, data16);` It sends it to `fwupload.chan`. This is likely an ack or status back to the GCS during the update process.

Key Codebase Locations

- **libraries/AP_Radio/AP_Radio_cc2500.cpp:1493**: Sending implementation.

AHRS2 (ID 178)

SUPPORTED (TX ONLY)

Summary

Status of the secondary Attitude and Heading Reference System (AHRS) or EKF core.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends secondary EKF/AHRS state.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to report the solution from the secondary estimation backend (e.g., EKF3 Core 1 if Core 0 is primary, or DCM if EKF is primary).

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

The system calls `AP::ahrs().get_secondary_attitude()` and `get_secondary_position()`. If valid, it populates the message.

- **Role:** Allows the GCS to monitor the health and divergence of the backup estimator.

Data Fields

- `roll`: Roll angle (rad).
- `pitch`: Pitch angle (rad).
- `yaw`: Yaw angle (rad).
- `altitude`: Altitude (meters).
- `lat`: Latitude (deg * 1E7).
- `lng`: Longitude (deg * 1E7).

Practical Use Cases

1. **EKF Health Monitoring:**
   - *Scenario:* A developer wants to see if the secondary EKF core is agreeing with the primary core during a flight with magnetic interference.
   - *Action:* The GCS plots `AHRS2.roll` vs `ATTITUDE.roll` to check for divergence.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:587**: Sending implementation.

BATTERY2 (ID 181)

SUPPORTED (TX ONLY)

Summary

Voltage and current report for the secondary battery.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends secondary battery status.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to report the status of the battery monitor at index 1 (the second battery).

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Queries `AP::battery().voltage(1)` and `current_amps(1)`.

- **Note:** This message is considered legacy/deprecated in favor of `BATTERY_STATUS` (which supports an ID field), but it is still actively sent for compatibility with older GCS implementations that expect a dedicated "Battery 2" message.

Data Fields

- `voltage`: Voltage (millivolts).
- `current_battery`: Current (centiamps).

Practical Use Cases

1. **Dual Battery Setup:**
   - *Scenario:* A large drone has two independent LiPo batteries for redundancy.
   - *Action:* The GCS displays "Batt 1" (from `SYS_STATUS`) and "Batt 2" (from `BATTERY2`) separately.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:2810**: Sending implementation.

MAG_CAL_PROGRESS (ID 191)

SUPPORTED (TX ONLY)

Summary

Reports the progress of the onboard compass calibration process.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends calibration status.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message periodically while `AP_Compass` is in calibration mode.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

- **Progress:** 0-100\%.
- **Direction Mask:** Bitmask of which orientations (North, South, Up, Down, etc.) have been successfully sampled.
- **ID:** Compass ID being calibrated.

Data Fields

- `compass_id`: Compass instance/ID.
- `cal_mask`: Bitmask of required orientations.
- `cal_status`: Status flags.
- `attempt`: Attempt number.
- `completion_pct`: Completion percentage.
- `completion_mask`: Bitmask of completed orientations.
- `direction_x/y/z`: Current direction vector.

Practical Use Cases

1. **Onboard Calibration:**
    - *Scenario:* User initiates compass calibration via GCS.
    - *Action:* GCS displays a "dance" guide, highlighting which sides of the vehicle still need to be presented to the sky/ground based on `completion_mask`.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Streaming entry.

MAG_CAL_REPORT (ID 192)

SUPPORTED (TX ONLY)

Summary

Reports the final results of the onboard compass calibration.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends final calibration offsets.
- **RX (Receive):** None

Transmission (TX)

Sent once when calibration completes (success or failure).

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Contains the calculated hard-iron offsets, diagonals, and off-diagonals (soft iron), plus the "fitness" (error) score.

Data Fields

- `compass_id`: Compass instance/ID.
- `cal_mask`: Calibration mask used.
- `cal_status`: Final status (Success/Failed).
- `autosaved`: True if parameters were saved.
- `fitness`: Error score (lower is better).
- `ofs_x/y/z`: Hard iron offsets.
- `diag_x/y/z`: Soft iron diagonal scaling.
- `offdiag_x/y/z`: Soft iron off-diagonal scaling.

Practical Use Cases

1. **Calibration Verification:**
   - *Scenario:* Calibration finishes.
   - *Action:* GCS displays the `fitness` score. If it's too high (e.g., > 10), the user knows the calibration was poor (likely magnetic interference nearby) and should retry.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Streaming entry.

EKF_STATUS_REPORT (ID 193)

SUPPORTED (TX ONLY)

Summary

Detailed health and status report of the Extended Kalman Filter (EKF).

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends EKF variance metrics.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to report the "health" of the EKF fusion. This data drives the "EKF Status" / "Vibe" HUD elements in Mission Planner.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

The system queries the active EKF core for the ratio of innovation to gate size (test ratio) for various sensors.

- **Values < 1.0:** Healthy (Consistency within expected noise).
- **Values > 1.0:** Inconsistent (Sensor data conflicts with prediction).

Data Fields

- `flags`: Flags indicating which sensors are active/fused.
- `velocity_variance`: Velocity innovation test ratio.
- `pos_horiz_variance`: Horizontal position innovation test ratio.
- `pos_vert_variance`: Vertical position innovation test ratio.
- `compass_variance`: Compass innovation test ratio.
- `terrain_alt_variance`: Terrain altitude innovation test ratio.
- `airspeed_variance`: Airspeed innovation test ratio.

Practical Use Cases

1. **Pre-Arm Check:**
   - *Scenario:* User tries to arm but gets "EKF Variance" error.
   - *Action:* GCS checks `compass_variance`. If high, it indicates magnetic interference or bad calibration.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:1101**: Streaming entry.

PID_TUNING (ID 194)

SUPPORTED (TX ONLY)

Summary

Real-time PID controller data for tuning analysis.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends PID components.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message when `GCS_PID_MASK` is set to monitor a specific axis (e.g., Roll, Pitch, Yaw).

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Captures the internal state of the AC_PID controllers.

- **axis:** enum (ROLL=1, PITCH=2, YAW=3, etc).
- **desired:** Target rate/angle.
- **achieved:** Actual rate/angle.
- **P/I/D/FF:** Contribution of each term.

Data Fields

- `axis`: Axis ID.
- `desired`: Desired value.
- `achieved`: Achieved value.
- `FF`: Feed-Forward component.
- `P`: Proportional component.
- `I`: Integral component.
- `D`: Derivative component.
- `SRate`: Slew rate (optional).
- `PDmod`: P/D modulation (optional).

Practical Use Cases

1. **Tuning Optimization:**
    - *Scenario:* User is manually tuning a racing drone.
    - *Action:* They enable PID logging for the Roll axis. The GCS graphs `desired` vs `achieved` and the `P/I/D` terms in real-time, helping the user adjust gains to minimize overshoot.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:1102**: Streaming entry.

DEEPSTALL (ID 195)

SUPPORTED (TX ONLY)

Summary

Status of the Deep Stall landing controller (used by fixed-wing aircraft).

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** Plane - Sends landing status.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message when a deep stall landing is in progress.

Source: libraries/AP_Landing/AP_Landing_Deepstall.cpp

Protocol Logic

Reports the progress of the stall maneuver.

- **Stage:** Estimate/Wait/Descend/Land.
- **Cross-track error:** Distance from landing line.

Data Fields

- `landing_lat`: Target latitude.
- `landing_lon`: Target longitude.
- `path_lat`: Path latitude.
- `path_lon`: Path longitude.
- `arc_entry_lat`: Arc entry latitude.
- `arc_entry_lon`: Arc entry longitude.
- `altitude`: Current altitude.
- `expected_travel_distance`: Estimated distance to touch down.
- `cross_track_error`: Deviation from path.
- `stage`: Landing stage (FlyToArc, Arc, Approach, Land).

Practical Use Cases

1. **Autonomous Landing:**
   - *Scenario:* A plane performs a vertical deep-stall landing in a tight space.
   - *Action:* The GCS monitors the `stage` and `cross_track_error` to ensure the vehicle is committed to the correct landing spot.

Key Codebase Locations

- **libraries/AP_Landing/AP_Landing_Deepstall.cpp:439**: Sending implementation.

HIGH_LATENCY2 (ID 235)

SUPPORTED (TX ONLY)

Summary

optimized telemetry message designed for high-latency, low-bandwidth links (e.g., Iridium SBD, RockBlock, LoRa).

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends condensed telemetry.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message periodically (default 5s) on links where `MSG_HIGH_LATENCY2` is enabled.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Packs essential flight data (Mode, Battery, GPS, Attitude, Airspeed, Failsafes) into a single compact message to minimize data usage and cost on satellite links.

Data Fields

- `timestamp`: Time since boot.
- `type`: Vehicle type.
- `autopilot`: Autopilot type.
- `custom_mode`: Flight mode.
- `latitude`: Latitude.
- `longitude`: Longitude.
- `altitude`: Altitude.
- `target_altitude`: Target altitude.
- `heading`: Heading.
- `target_heading`: Target heading.
- `target_distance`: Distance to WP.
- `throttle`: Throttle \%.
- `airspeed`: Airspeed.
- `airspeed_sp`: Airspeed setpoint.
- `groundspeed`: Groundspeed.
- `windspeed`: Windspeed.
- `wind_heading`: Wind heading.
- `eph`: GPS horizontal accuracy.
- `epv`: GPS vertical accuracy.
- `temperature_air`: Air temp.
- `climb_rate`: Climb rate.
- `battery`: Battery \%.
- `custom0/1/2`: Custom debug/user fields.
- `failure_flags`: Bitmask of system failures.

Practical Use Cases

1. **Beyond Visual Line of Sight (BVLOS):**
   - *Scenario:* A glider flies 50km away, losing direct radio contact.
   - *Action:* It switches to Iridium satellite telemetry, sending one `HIGH_LATENCY2` packet every 10 seconds to keep the operator informed of its position and battery without incurring high data costs.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:1139**: Streaming entry.

EXTENDED_SYS_STATE (ID 245)

SUPPORTED (TX ONLY)

Summary

Extended system state, primarily used for VTOL aircraft to report their flight state (transitioning, hovering, flying fixed-wing) and landing status.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles (especially QuadPlane/VTOL) - Sends VTOL state.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to update the GCS on the specific state of a VTOL vehicle.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

- **vtol_state:** MAV_VTOL_STATE (Undefined, Transitioning, MC, FW).
- **landed_state:** MAV_LANDED_STATE (On Ground, In Air, Taking Off, Landing).

Data Fields

- `vtol_state`: VTOL flight state.
- `landed_state`: Landed state.

Practical Use Cases

1. **VTOL Transition Monitoring:**
   - *Scenario:* A QuadPlane takes off and transitions to forward flight.
   - *Action:* The GCS uses `vtol_state` to change the HUD symbology (e.g., from Copter style to Plane style) during the transition.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:1121**: Streaming entry.

SENSORS

GPS_GLOBAL_ORIGIN (ID 49)

SUPPORTED (TX ONLY)

Summary

Publishes the GPS coordinates of the vehicle's local origin (0,0,0).

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends current EKF origin.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to confirm the EKF origin has been set or changed.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Queries `AP::[ahrs](/field-manual/mavlink-interface/ahrs.html)().get_origin()` and broadcasts it.

- **Trigger:** Sent after `SET_GPS_GLOBAL_ORIGIN` is received or when the EKF initializes its origin (e.g., getting a 3D GPS lock).

Data Fields

- `latitude`: Latitude (deg * 1E7).
- `longitude`: Longitude (deg * 1E7).
- `altitude`: Altitude (mm).
- `time_usec`: Timestamp.

Practical Use Cases

1. **Map Alignment:**
   - *Scenario:* Drone is flying in "Guided NoGPS" mode.
   - *Action:* The GCS receives `GPS_GLOBAL_ORIGIN` and uses it to render the drone's position on the map, even though the drone itself is only navigating using local meters relative to startup.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:3064**: Sending implementation.

WIND (ID 168)

SUPPORTED (TX ONLY)

Summary

Reports the estimated wind speed and direction. This data is derived from the EKF (Extended Kalman Filter) or a dedicated wind vane sensor.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** Copter, Plane, Blimp - Sends estimated wind vector.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to report the wind estimate calculated by the AHRS/EKF or measured by `AP_WindVane`.

Source: ArduCopter/GCS_Mavlink.cpp

Protocol Logic

The system retrieves the wind vector (x, y, z) from `AP::ahrs().wind_estimate()` and converts it to polar coordinates (Direction, Speed) for transmission.

- **Direction:** `degrees(atan2f(-wind.y, -wind.x))` (Direction the wind is coming *from*).
- **Speed:** `wind.length()`.
- **Vertical Speed:** `wind.z`.

Data Fields

- **`direction`**: Wind direction (0..360 degrees). 0 = North, 90 = East. Direction wind is coming from.
- **`speed`**: Wind speed in m/s.
- **`speed_z`**: Vertical wind speed in m/s.

Practical Use Cases

1. **Situational Awareness:**
    - *Scenario:* Pilot flying a Copter in loiter mode sees the vehicle leaning into the wind.
    - *Action:* The GCS displays the `WIND` message data, confirming a 15m/s headwind, helping the pilot estimate battery consumption (fighting wind consumes more power).
2. **Glider Operations:**
    - *Scenario:* A glider pilot looks for thermals.
    - *Action:* The vertical wind speed component (`speed_z`) can indicate rising or sinking air masses.

Key Codebase Locations

- **ArduCopter/GCS_Mavlink.cpp:1583**: Copter sending implementation.
- **ArduPlane/GCS_Mavlink.cpp:317**: Plane sending implementation.
- **libraries/AP_WindVane/AP_WindVane.cpp:432**: Library implementation for vehicles with physical wind sensors.

RANGEFINDER (ID 173)

SUPPORTED (TX ONLY)

Summary

Raw output from the primary downward-facing rangefinder (lidar/sonar).

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** Rover, Copter, Plane - Sends raw sensor data.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to report the distance measured by the primary rangefinder.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

The system queries `AP::rangefinder()` for the primary instance's distance and voltage.

- **Distance:** Converted to meters.
- **Voltage:** Raw voltage from analog sensors (if applicable).

Data Fields

- `distance`: Distance in meters.
- `voltage`: Voltage in volts (for analog sensors).

Practical Use Cases

1. **Precision Landing:**
    - *Scenario:* Monitoring altitude during the final phase of a landing.
    - *Action:* The GCS displays the raw rangefinder data to verify the terrain height.
2. **Obstacle Avoidance Tuning:**
    - *Scenario:* Calibrating a forward-facing ranger.
    - *Action:* User checks if the reported distance matches the physical distance to an obstacle.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:507**: Sending implementation.

AIRSPEED_AUTOCAL (ID 174)

SUPPORTED (TX ONLY)

Summary

Reports the status of the in-flight airspeed calibration algorithm.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** Plane (and others with airspeed sensors) - Sends calibration metrics.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message when `AP_AIRSPEED_AUTOCAL_ENABLE` is true and an airspeed sensor is calibrating.

Source: libraries/AP_Airspeed/Airspeed_Calibration.cpp

Protocol Logic

The `AP_Airspeed` library calculates the ratio between GPS ground speed and airspeed to estimate the ratio error.

- **Ratio:** The estimated ratio.
- **Diff/Scale:** Error metrics (difference in X/Y/Z acceleration vs expected drag).

Data Fields

- `vx`: Velocity X (m/s).
- `vy`: Velocity Y (m/s).
- `vz`: Velocity Z (m/s).
- `diff_pressure`: Differential pressure (Pa).
- `EAS2TAS`: Estimated True Airspeed ratio.
- `ratio`: The calculated airspeed ratio.
- `state_x`: Kalman filter state X.
- `state_y`: Kalman filter state Y.
- `state_z`: Kalman filter state Z.
- `pax`: Positive acceleration X.
- `pby`: Positive acceleration Y.
- `pcz`: Positive acceleration Z.

Practical Use Cases

1. **Calibration Verification:**
   - *Scenario:* Pilot performs loiter circles to calibrate the airspeed sensor.
   - *Action:* The GCS plots `ratio` to see if it converges to a stable value.

Key Codebase Locations

- **libraries/AP_Airspeed/Airspeed_Calibration.cpp:207**: Sending implementation.

COMPASSMOT_STATUS (ID 177)

SUPPORTED (TX ONLY)

Summary

Status of the compass motor interference calibration.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** Copter - Sends calibration progress.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message during the CompassMot calibration process.

Source: ArduCopter/compassmot.cpp

Protocol Logic

As the user raises the throttle, the system measures magnetic interference.

- **throttle:** Current throttle level.
- **current:** Current draw (Amps).
- **interference:** Magnitude of interference.

Data Fields

- `throttle`: Throttle value (0-1000).
- `current`: Current (Amps).
- `interference`: Interference level (0-1000). 1000 = 100\% of max acceptable.
- `compensation_x`: Compensation vector X.
- `compensation_y`: Compensation vector Y.
- `compensation_z`: Compensation vector Z.

Practical Use Cases

1. **Compass/Motor Calibration:**
    - *Scenario:* User performs the "CompassMot" procedure.
    - *Action:* GCS displays a live graph of interference vs throttle using this message.

Key Codebase Locations

- **ArduCopter/compassmot.cpp:222**: Sending implementation.

RPM (ID 226)

SUPPORTED (TX ONLY)

Summary

RPM (Revolutions Per Minute) from up to two sensors.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends RPM data.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to report data from the `AP_RPM` library.

Source: libraries/GCS_MAVLink/GCS_Common.cpp

Protocol Logic

Queries `AP::rpm()` for the speed of the first two configured RPM sensors.

Data Fields

- **`rpm1`**: Speed of sensor 1 (RPM).
- **`rpm2`**: Speed of sensor 2 (RPM).

Practical Use Cases

1. **Heli Rotor Speed:**
    - *Scenario:* Helicopter pilot monitors head speed.
    - *Action:* GCS displays `rpm1` (Main Rotor) and `rpm2` (Tail Rotor).

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:1105**: Streaming entry.

COLLISION (ID 247)

SUPPORTED (TX ONLY)

Summary

Information about a potential collision with an object (or aircraft).

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends collision warning.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message when the `AP_Avoidance` library detects an impending collision (via ADSB or other sensors).

Source: libraries/AP_Avoidance/AP_Avoidance.cpp

Protocol Logic

- **src:** Source of threat (ADSB, MavLink, etc).
- **id:** ID of threat.
- **action:** Action taken (None, Report, Climb, Descend).
- **threat_level:** Severity.
- **time_to_minimum_delta:** Time until closest approach.
- **altitude_minimum_delta:** Closest vertical distance.
- **horizontal_minimum_delta:** Closest horizontal distance.

Data Fields

- `src`: Source ID.
- `id`: Threat ID.
- `action`: Avoidance action.
- `threat_level`: Threat level.
- `time_to_minimum_delta`: Time to impact (seconds).
- `altitude_minimum_delta`: Vertical miss distance (meters).
- `horizontal_minimum_delta`: Horizontal miss distance (meters).

Practical Use Cases

1. **Pilot Alert:**
   - *Scenario:* Automatic avoidance triggers.
   - *Action:* GCS flashes a big red "COLLISION ALERT" warning based on this message, informing the pilot why the drone just suddenly dove 10 meters.

Key Codebase Locations

- **libraries/AP_Avoidance/AP_Avoidance.cpp:411**: Sending implementation.

CONTROL

RC_CHANNELS_SCALED (ID 34)

SUPPORTED (TX ONLY)

Summary

The scaled values of the RC channels.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** Rover (and possibly others) - Sends scaled servo/motor outputs.
- **RX (Receive):** None

Transmission (TX)

ArduRover sends this message to report the status of its motor outputs and control surfaces, scaled to -10000 to +10000.

Source: Rover/GCS_Mavlink.cpp

Protocol Logic

- **Scaling:** -100\% (-10000), 0\% (0), +100\% (10000).
- **Fields:** Channels 1-8. Inactive channels set to UINT16_MAX.

Data Fields

- `time_boot_ms`: Timestamp.
- `port`: Servo output port (0 for primary).
- `chan1_scaled`: Channel 1 scaled value.
- `chan2_scaled`: Channel 2 scaled value.
- `chan3_scaled`: Channel 3 scaled value.
- `chan4_scaled`: Channel 4 scaled value.
- `chan5_scaled`: Channel 5 scaled value.
- `chan6_scaled`: Channel 6 scaled value.
- `chan7_scaled`: Channel 7 scaled value.
- `chan8_scaled`: Channel 8 scaled value.
- `rssi`: Received Signal Strength Indicator (0-255).

Practical Use Cases

1. **Rover Diagnostics:**
    - *Scenario:* Debugging skid-steering mixing.
    - *Action:* GCS displays `chan1_scaled` (Left Throttle) and `chan3_scaled` (Right Throttle) to verify the mixing logic.

Key Codebase Locations

- **Rover/GCS_Mavlink.cpp:133**: Sending implementation.

POSITION_TARGET_LOCAL_NED (ID 85)

SUPPORTED (TX ONLY)

Summary

Reports the current commanded vehicle position, velocity, and acceleration in the local NED frame.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** Copter, Plane, Rover - Sends current target state.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to report what the autopilot is *trying* to achieve (the setpoint), which might differ from the actual vehicle state (`LOCAL_POSITION_NED`).

Source: ArduCopter/GCS_Mavlink.cpp

Protocol Logic

Queries the active flight mode controller (e.g., `pos_control`) for its target position, velocity, and acceleration.

Data Fields

- `time_boot_ms`: Timestamp.
- `coordinate_frame`: MAV_FRAME_LOCAL_NED.
- `type_mask`: Bitmap of valid fields.
- `x`: Target X (m).
- `y`: Target Y (m).
- `z`: Target Z (m).
- `vx`: Target VX (m/s).
- `vy`: Target VY (m/s).
- `vz`: Target VZ (m/s).
- `afx`: Target Accel X.
- `afy`: Target Accel Y.
- `afz`: Target Accel Z.
- `yaw`: Target Yaw.
- `yaw_rate`: Target Yaw Rate.

Practical Use Cases

1. **Control Loop Analysis:**
    - *Scenario:* Investigating "toilet bowling" (oscillation).
    - *Action:* GCS plots `POSITION_TARGET_LOCAL_NED.x` vs `LOCAL_POSITION_NED.x`. If the target leads the actual position correctly, the navigator is fine. If the target oscillates wildly, the navigation tuning is bad.

Key Codebase Locations

- **ArduCopter/GCS_Mavlink.cpp:142**: Sending implementation.

POSITION_TARGET_GLOBAL_INT (ID 87)

SUPPORTED (TX ONLY)

Summary

Reports the current commanded vehicle position, velocity, and acceleration in the Global frame.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends current target state.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to report the current navigation target in global coordinates.

Source: ArduCopter/GCS_Mavlink.cpp

Data Fields

- `time_boot_ms`: Timestamp.
- `coordinate_frame`: MAV_FRAME.
- `type_mask`: Bitmap of valid fields.
- `lat_int`: Target Latitude.
- `lon_int`: Target Longitude.
- `alt`: Target Altitude (m).
- `vx`: Target VX.
- `vy`: Target VY.
- `vz`: Target VZ.
- `afx`: Target Accel X.
- `afy`: Target Accel Y.
- `afz`: Target Accel Z.
- `yaw`: Target Yaw.
- `yaw_rate`: Target Yaw Rate.

Practical Use Cases

1. **Target Visualization:**
   - *Scenario:* Autonomous mission.
   - *Action:* GCS draws a "Ghost Drone" on the map representing the `POSITION_TARGET_GLOBAL_INT`. This shows where the drone *wants* to be vs where it is.

Key Codebase Locations

- **ArduCopter/GCS_Mavlink.cpp:109**: Sending implementation.

BUTTON_CHANGE (ID 257)

SUPPORTED (TX ONLY)

Summary

Report a change in the state of a physical button connected to the autopilot.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles - Sends button events.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message when the `AP_Button` library detects a state change (press/release) on a configured button pin.

Source: libraries/AP_Button/AP_Button.cpp

Data Fields

- **`time_boot_ms`:** Timestamp.
- **`last_change_ms`:** Time of last change.
- **`state`:** New state (0=Released, 1=Pressed).

Practical Use Cases

1. **Safety Switch Monitoring:**
    - *Scenario:* User presses the safety button on the GPS mast.
    - *Action:* The GCS receives `BUTTON_CHANGE`, confirming the action.

Key Codebase Locations

- **libraries/AP_Button/AP_Button.cpp:355**: Sending implementation.

PAYLOAD

CAMERA_FEEDBACK (ID 180)

SUPPORTED (TX ONLY)

Summary

precise location and attitude information for the moment a photo was taken. Crucial for aerial mapping and photogrammetry.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles (with Camera enabled) - Sends trigger confirmation.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message whenever a camera trigger event is successfully logged (either by software command or hardware hot-shoe feedback).

Source: libraries/AP_Camera/AP_Camera_Backend.cpp

Protocol Logic

When a photo is taken, `AP_Camera` records the vehicle's position (`lat`, `lng`, `alt`) and attitude (`roll`, `pitch`, `yaw`) at that exact timestamp. This data is then streamed to the GCS.

- **Hot Shoe:** If hardware feedback is enabled, this message corresponds to the exact shutter close time.
- **Open Loop:** If no feedback, it corresponds to the trigger command time.

Data Fields

- `time_usec`: Image timestamp (microseconds since boot).
- `target_system`: System ID.
- `cam_idx`: Camera ID.
- `img_idx`: Image index (counter).
- `lat`: Latitude (deg * 1E7).
- `lng`: Longitude (deg * 1E7).
- `alt_msl`: Altitude MSL (meters).
- `alt_rel`: Relative Altitude (meters).
- `roll`: Roll (degrees).
- `pitch`: Pitch (degrees).
- `yaw`: Yaw (degrees).
- `foc_len`: Focal length (mm).
- `flags`: Feedback flags (e.g., `CAMERA_FEEDBACK_PHOTO`).
- `completed_captures`: Count of completed captures.

Practical Use Cases

1. **Photogrammetry / Mapping:**
   - *Scenario:* A drone flies a survey grid taking photos every 20 meters.

    - *Action:* The GCS records `CAMERA_FEEDBACK` messages to a log. Post-flight, this log is used to geotag the images (Write EXIF data) with high precision, allowing for accurate 3D model reconstruction.

Key Codebase Locations

- **libraries/AP_Camera/AP_Camera_Backend.cpp:205**: Sending implementation.

GIMBAL_CONTROL (ID 201)

SUPPORTED (TX ONLY)

Summary

Control message for the 3DR Solo Gimbal.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles (if Solo Gimbal configured) - Sends control demands.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message to command the gimbal's rate and orientation.

Source: libraries/AP_Mount/SoloGimbal.cpp

Protocol Logic

The `AP_Mount_SoloGimbal` driver converts user or autopilot demands (e.g., "Look Down") into rate commands for the gimbal motors.

Data Fields

- **`target_system`**: System ID.
- **`target_component`**: Component ID.
- **`demand`**: Demand vector (rate or angle).

Practical Use Cases

1. **Gimbal Pointing:**
    - *Scenario:* Pilot uses a slider to tilt the camera.
    - *Action:* ArduPilot sends `GIMBAL_CONTROL` commands to the gimbal to execute the movement.

Key Codebase Locations

- **libraries/AP_Mount/SoloGimbal.cpp:118**: Sending implementation.

LOG_ENTRY (ID 118)

TX ONLY

Summary

The `LOG_ENTRY` message is a reply to a `LOG_REQUEST_LIST` (117) command. It provides metadata about a single DataFlash log file on the vehicle, including its ID, size, and timestamp. The vehicle sends a stream of these messages to list all available logs.

Status

TX Only

Directionality

- **TX (Transmit):** All Vehicles (Log listing response)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission is handled by `AP_Logger::handle_log_send_listing` within libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:250.

Protocol Logic

1. **Iteration:** The logger iterates from `start` to `end` as requested by `LOG_REQUEST_LIST`.
2. **Metadata Retrieval:** For each log index, it calls `get_log_info` to retrieve the file size (bytes) and UTC timestamp.
3. **Completion:** When the last entry is sent, the transfer activity state is reset to `IDLE`.

Data Fields

- `id`: Log id.
- `num_logs`: Total number of logs available.
- `last_log_num`: High log number.
- `time_utc`: UTC timestamp of log creation (seconds since 1970).
- `size`: Log size in bytes.

Practical Use Cases

1. **Log Browser:**
    - *Scenario:* A user clicks "Download Logs" in Mission Planner.
    - *Action:* Mission Planner receives a stream of `LOG_ENTRY` messages and populates a table showing Log ID, Date, and Size, allowing the user to select specific flights for download.

Key Codebase Locations

- **libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:250**: Implementation of the sender.

`LOG_DATA` (ID 120)

TX ONLY

Summary

The `LOG_DATA` message carries a fixed-size chunk of binary data from a specific DataFlash log file. It is the response to a `LOG_REQUEST_DATA` (119) message. A sequence of these messages constitutes a file download.

Status

TX Only

Directionality

- **TX (Transmit):** All Vehicles (Log content transfer)
- **RX (Receive):** None (Ignored)

Transmission (TX)

Transmission is handled by `AP_Logger::handle_log_send_data` within libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:302.

Protocol Logic

1. **Read:** It reads `len` bytes (max 90) from the storage backend (File or Block) at the current `_log_data_offset`.
2. **Pack:** It packs the data into the `data` array of the message.
3. **Pad:** If the read bytes are fewer than 90, the remaining bytes are zeroed out.
4. **Send:** The message is broadcast on the requesting channel.
5. **Advance:** The `_log_data_offset` is incremented. If the requested byte count is met or the end of the log is reached, transmission stops (`IDLE`).

Data Fields

- **`id`**: Log ID (1 through N).
- **`ofs`**: Offset into the log file (bytes).
- **`count`**: Number of bytes in this packet (typically 90).
- **`data`**: Raw log data (90 bytes).

Practical Use Cases

1. **Downloading a Log:**
   - *Scenario:* Mission Planner is downloading a log.
   - *Action:* ArduPilot sends `LOG_DATA` packets rapidly. Mission Planner concatenates the `data` fields to reconstruct the `.BIN` file on the user's disk.

Key Codebase Locations

- **libraries/AP_Logger/AP_Logger_MAVLinkLogTransfer.cpp:302**: Implementation of the sender.

REMOTE_LOG_DATA_BLOCK (ID 184)

SUPPORTED (TX ONLY)

Summary

A block of dataflash log data sent to a remote listener. This is part of the "Remote Logging" feature where logs are streamed to a companion computer or GCS in real-time.

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** All Vehicles (if configured) - Sends log blocks.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message via `AP_Logger_MAVLink` when remote logging is enabled.

Source: libraries/AP_Logger/AP_Logger_MAVLink.cpp

Protocol Logic

The logger buffers log data and sends it in 200-byte chunks (blocks) to the MAVLink stream.

- **Reliability:** The system expects `REMOTE_LOG_BLOCK_STATUS` acks to ensure data integrity.

Data Fields

- **`target_system`:** System ID of listener.
- **`target_component`:** Component ID of listener.
- **`seqno`:** Sequence number (high/low or 32-bit index).
- **`data`:** Log data block (200 bytes).

Practical Use Cases

1. **Redundant Logging:**
    - *Scenario:* A high-value drone operation requires logs to be saved on a companion computer (Raspberry Pi) in case the flight controller is destroyed.
    - *Action:* The FC streams `REMOTE_LOG_DATA_BLOCK` messages to the Pi, which writes them to disk.

Key Codebase Locations

- **libraries/AP_Logger/AP_Logger_MAVLink.cpp**: Implementation.

SIMULATION

SIM_STATE (ID 108)

SUPPORTED (TX ONLY)

Summary

Status of the simulation environment (SITL).

Status

Supported (TX Only)

Directionality

- **TX (Transmit):** SITL - Sends simulation ground truth.
- **RX (Receive):** None

Transmission (TX)

ArduPilot sends this message when running in Software In The Loop (SITL) mode to report the "true" physical state of the vehicle, bypassing sensor noise and estimation errors.

Source: libraries/SITL/SITL.cpp

Protocol Logic

- **q1-q4:** True attitude quaternion.
- **roll/pitch/yaw/etc:** True rates and positions.

Data Fields

- `q1` : Quaternion q1.
- `q2` : Quaternion q2.
- `q3` : Quaternion q3.
- `q4` : Quaternion q4.
- `roll` : Roll angle (rad).
- `pitch` : Pitch angle (rad).
- `yaw` : Yaw angle (rad).
- `xacc` : X acceleration (m/s^2).
- `yacc` : Y acceleration (m/s^2).
- `zacc` : Z acceleration (m/s^2).
- `xgyro` : X angular speed (rad/s).
- `ygyro` : Y angular speed (rad/s).
- `zgyro` : Z angular speed (rad/s).
- `lat` : Latitude (deg).
- `lon` : Longitude (deg).
- `alt` : Altitude (m).
- `std_dev_horz` : Horizontal position standard deviation.
- `std_dev_vert` : Vertical position standard deviation.
- `vn` : Velocity North (m/s).
- `ve` : Velocity East (m/s).
- `vd` : Velocity Down (m/s).

Practical Use Cases

1. **Estimation Analysis:**
    - *Scenario:* Developing a new EKF.
    - *Action:* Developer compares `SIM_STATE.roll` (Truth) vs `ATTITUDE.roll` (Estimated) to quantify the filter's performance.

Key Codebase Locations

- **libraries/SITL/SITL.cpp:1557**: Sending implementation.

SYSTEM

RESOURCE_REQUEST (ID 142)

UNSUPPORTED

Summary

The `RESOURCE_REQUEST` message allows a component to request a specific resource (like a file, image, or binary blob) from another component by URI.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot uses specific protocols for resource management:

1. **FTP (`FILE_TRANSFER_PROTOCOL`):** For file system access (logs, params, terrain).
2. **Mission Protocol:** For waypoints.
3. **Parameter Protocol:** For settings.

Intended Data Fields (Standard)

- `request_id`: Request ID.
- `uri_type`: The type of requested URI.
- `uri`: The URI of the resource requested.
- `transfer_type`: The transfer type.
- `storage`: The storage ID.

LIMITS_STATUS (ID 167)

UNSUPPORTED

Summary

Status of the legacy `AP_Limits` system. This module was the predecessor to the modern Geofence system.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

This message is deprecated and no longer used in the ArduPilot codebase. It was previously used to report the status of the `AP_Limits` module (safety limits), which has been superseded by the `AC_Fence` library and messages like `FENCE_STATUS`.

Data Fields

- **`limits_state`:** State of AP_Limits.
- **`last_trigger`:** Time of last breach.
- **`last_action`:** Action taken.
- **`last_recovery`:** Recovery action.
- **`last_clear`:** Time of last clear.
- **`breach_count`:** Number of fence breaches.
- **`mods_enabled`:** Bitmask of enabled modules.
- **`mods_required`:** Bitmask of required modules.
- **`mods_triggered`:** Bitmask of triggered modules.

Theoretical Use Cases

1. **Legacy Fence Monitoring:**
   - *Scenario:* Older versions of ArduPilot used this to indicate if the vehicle had breached a geofence or altitude limit.
   - *Alternative:* Use `FENCE_STATUS` (162).

MEMORY_VECT (ID 249) UNSUPPORTED

Summary

Send raw memory content.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Debugging.

DEBUG_VECT (ID 250) UNSUPPORTED

Summary

Named vector debug data.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It prefers `NAMED_VALUE_FLOAT`.

Theoretical Use Cases

Debugging.

DEBUG (ID 254)

UNSUPPORTED

Summary

Generic debug message.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Debugging.

PROTOCOL_VERSION (ID 300)

UNSUPPORTED

Summary

The `PROTOCOL_VERSION` message is designed to allow a MAVLink node to report the range of protocol versions it supports (e.g., MAVLink 1.0 vs 2.0). **ArduPilot does not currently implement or transmit this message.**

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

ArduPilot handles protocol versioning implicitly at the communication layer.

- **Discovery:** Ground Control Stations detect the MAVLink version by inspecting the start-of-frame byte ($0xFE$ for v1.0, $0xFD$ for v2.0).
- **Capabilities:** Comprehensive capability and version reporting are handled by the `AUTOPILOT_VERSION` (148) message, which ArduPilot fully supports.
- **Absence:** There is no handler or sender for ID 300 in the `GCS_MAVLink` libraries.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Lacks a mapping for ID 300.

DEBUG_FLOAT_ARRAY (ID 350)

UNSUPPORTED

Summary

The `DEBUG_FLOAT_ARRAY` message allows sending a large array of floating-point values for debugging purposes. It is useful for dumping internal state vectors or matrices.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

For debugging, ArduPilot typically uses:

- `NAMED_VALUE_FLOAT` **(251):** For key-value pairs.
- `STATUSTEXT` **(253):** For text logs.
- `DEBUG_VECT` **(250):** For 3D vectors.

Intended Data Fields (Standard)

- `time_usec`**:** Timestamp.
- `name`**:** Name ID.
- `array_id`**:** Unique ID for the array.
- `data`**:** Array of floats.

Theoretical Use Cases

1. **Visualizing Internal State:**
   - *Scenario:* Tuning a new neural network controller.
   - *Action:* The autopilot outputs the entire 64-float activation layer of the neural net via `DEBUG_FLOAT_ARRAY` at 50Hz. A visualizer tool plots this heatmap in real-time, helping the developer understand what features the network is activating on.
2. **Spectral Analysis:**
   - *Scenario:* In-flight vibration analysis.
   - *Action:* The autopilot performs an FFT on the gyro data and downlinks the frequency bins as a float array. The engineer sees a spike at 80Hz and knows the props need balancing.

TELEMETRY

ATTITUDE_QUATERNION_COV (ID 61) UNSUPPORTED

Summary

The attitude in the aeronautical frame, expressed as quaternion, with covariance.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It uses `ATTITUDE_QUATERNION` (31) or `ATTITUDE` (30) for attitude reporting.

Theoretical Use Cases

Attitude reporting with uncertainty.

GLOBAL_POSITION_INT_COV (ID 63) UNSUPPORTED

Summary

The filtered global position with covariance.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It uses `GLOBAL_POSITION_INT` (33) for position reporting.

Theoretical Use Cases

Position reporting with uncertainty.

LOCAL_POSITION_NED_COV (ID 64) UNSUPPORTED

Summary

The filtered local position with covariance.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It uses `LOCAL_POSITION_NED` (32) for local position reporting.

Theoretical Use Cases

Local position reporting with uncertainty.

DATA_STREAM (ID 67) UNSUPPORTED

Summary

Data stream status information.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message (neither sending nor receiving).

Theoretical Use Cases

Reporting the status of a data stream.

LOCAL_POSITION_NED_SYSTEM_GLOBAL_OFFSET (ID 89)

UNSUPPORTED

Summary

The `LOCAL_POSITION_NED_SYSTEM_GLOBAL_OFFSET` message is defined in MAVLink to provide the offset between a local North-East-Down (NED) coordinate system and the global Latitude/Longitude frame. **ArduPilot does not implement or transmit this message.**

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

ArduPilot handles the relationship between local and global coordinates internally within the EKF (Extended Kalman Filter) and the `AP_AHRS` library. The origin of the local coordinate system is established relative to the home position or a fixed global coordinate, and this mapping is exposed through other messages like `HOME_POSITION` (242) and `GLOBAL_POSITION_INT` (33).

- **Absence:** There is no handler for ID 89 in the `GCS_MAVLink` libraries.
- **Alternative:** To find the relationship between local NED and global coordinates, a GCS should compare `LOCAL_POSITION_NED` (32) against `GLOBAL_POSITION_INT` (33) or query the `HOME_POSITION` (242).

Data Fields (Standard)

- **`time_boot_ms`:** Timestamp (milliseconds since system boot).
- **`x`:** X Position (meters).
- **`y`:** Y Position (meters).
- **`z`:** Z Position (meters).
- **`roll`:** Roll (rad).
- **`pitch`:** Pitch (rad).
- **`yaw`:** Yaw (rad).

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Lacks a handler for ID 89.

ALTITUDE (ID 141)

UNSUPPORTED

Summary

The `ALTITUDE` message provides a dedicated high-precision altitude report, separating monotonic (continuous), AMSL (Above Mean Sea Level), local, relative, and terrain-relative altitudes. This message is designed to resolve ambiguity between different altitude frames.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** currently broadcast this message.

Instead, ArduPilot reports altitude via:

1. **`GLOBAL_POSITION_INT` (33):** Contains `alt` (AMSL) and `relative_alt` (Above Home).
2. **`VFR_HUD` (74):** Contains `alt` (AMSL) for HUD displays.
3. **`TERRAIN_REPORT` (136):** Contains terrain height and current height above terrain.

Intended Data Fields (Standard)

- **`time_usec`:** Timestamp (micros).
- **`altitude_monotonic`:** Monotonic altitude (never resets).
- **`altitude_amsl`:** Altitude above mean sea level.
- **`altitude_local`:** Local altitude (relative to 0,0,0 origin).
- **`altitude_relative`:** Relative altitude (above home).
- **`altitude_terrain`:** Altitude above terrain.
- **`bottom_clearance`:** Distance to bottom surface (ground/water).

CONTROL_SYSTEM_STATE (ID 146)

UNSUPPORTED

Summary

The `CONTROL_SYSTEM_STATE` message is defined in MAVLink to provide a unified, high-frequency "State Vector" (Position, Velocity, Acceleration, Attitude, and Rates) to external controllers or companion computers. **ArduPilot does not implement or transmit this message.**

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

ArduPilot prefers to send its state estimation data through individual, specialized messages that are better integrated with its EKF (Extended Kalman Filter) architecture:

- **Orientation:** `ATTITUDE` (30) or `ATTITUDE_QUATERNION` (31).
- **Position:** `LOCAL_POSITION_NED` (32) or `GLOBAL_POSITION_INT` (33).
- **High-Frequency Data:** `HIGHRES_IMU` (105).
- **Filter Health:** `EKF_STATUS_REPORT` (ArduPilot Custom).

By using these discrete messages, ArduPilot ensures compatibility with a wider range of Ground Control Stations and reduces the bandwidth required for users who only need a subset of the state data.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `x_acc`: X acceleration in body frame.
- `y_acc`: Y acceleration in body frame.
- `z_acc`: Z acceleration in body frame.
- `x_vel`: X velocity in body frame.
- `y_vel`: Y velocity in body frame.
- `z_vel`: Z velocity in body frame.
- `x_pos`: X position in local frame.
- `y_pos`: Y position in local frame.
- `z_pos`: Z position in local frame.
- `airspeed`: Airspeed.
- `vel_variance`: Variance of velocity.
- `pos_variance`: Variance of position.
- `q`: The attitude, represented as Quaternion.
- `roll_rate`: Angular rate in roll axis.
- `pitch_rate`: Angular rate in pitch axis.
- `yaw_rate`: Angular rate in yaw axis.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Lacks a handler or mapping for ID 146.

DATA32 (ID 170)

UNSUPPORTED

Summary

Generic 32-byte data packet.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Generic data transmission.

DATA64 (ID 171)

UNSUPPORTED

Summary

Generic 64-byte data packet.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Generic data transmission.

AHRS3 (ID 182)

UNSUPPORTED

Summary

Status of a third AHRS/EKF core.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. Use `AHRS2` for secondary status or standard `ATTITUDE` / `GLOBAL_POSITION_INT` for primary status.

Theoretical Use Cases

Debugging a tertiary estimator.

ESTIMATOR_STATUS (ID 230) UNSUPPORTED

Summary

Estimator status report.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It uses `EKF_STATUS_REPORT` (193) for detailed estimator health.

Theoretical Use Cases

General estimator status.

HIGH_LATENCY (ID 234) UNSUPPORTED

Summary

Legacy high latency telemetry message.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot supports `HIGH_LATENCY2` (235) instead.

Theoretical Use Cases

Satellite telemetry.

V2_EXTENSION (ID 248)

UNSUPPORTED

Summary

MAVLink v2 extension message.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message directly.

Theoretical Use Cases

Protocol extension wrapper.

FLIGHT_INFORMATION (ID 264) UNSUPPORTED

Summary

The `FLIGHT_INFORMATION` message is designed to provide high-level metadata about the current flight session. Its primary purpose is to provide a unique identifier (UUID) for the flight, along with precise UTC timestamps for the arming and takeoff events. This facilitates the automatic organization and synchronization of flight logs across different systems (e.g., Drone vs. GCS vs. Cloud).

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot currently **does not** implement this message.

Flight UUIDs and arming timestamps are typically handled internally by the logging system (`DataFlash` / `.BIN` logs) and are not broadcast over MAVLink in this specific format. Ground Control Stations generally infer "New Flight" events by monitoring `HEARTBEAT` mode changes (e.g., Disarmed → Armed) or `GLOBAL_POSITION_INT` data.

Intended Data Fields (Standard)

- **`time_boot_ms`**: Timestamp (ms since boot).
- **`arming_time_utc`**: Timestamp at arming (us since UNIX epoch).
- **`takeoff_time_utc`**: Timestamp at takeoff (us since UNIX epoch).
- **`flight_uuid`**: Universally Unique Identifier (UUID) for the flight. This should ideally correspond to the filename of the log file.

Theoretical Use Cases

1. **Cloud Logging Sync:**
   - *Scenario:* A drone uploads telemetry to a cloud service.
   - *Action:* The `flight_uuid` allows the cloud service to automatically group telemetry packets into discrete "Flight" objects, even if the connection drops and reconnects.
2. **Legal Compliance:**
   - *Scenario:* Automated flight logging for commercial operations.
   - *Action:* The `takeoff_time_utc` provides a definitive, tamper-evident start time for the flight log, useful for pilot logbooks.

WIFI_CONFIG_AP (ID 299)

UNSUPPORTED

Summary

The `WIFI_CONFIG_AP` message is designed to configure the WiFi credentials (SSID and Password) of a MAVLink-enabled WiFi Access Point or Station. It is intended for setting up telemetry radios or companion computers without needing a physical USB connection.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

While ArduPilot supports networking via `AP_Networking` and PPP/Ethernet interfaces, the configuration of WiFi credentials on external bridges (like ESP8266/ESP32 MAVLink bridges) is typically handled via their own web interfaces or specific setup tools, not via this MAVLink message processed by the Flight Controller.

Intended Data Fields (Standard)

- **`ssid`:** Name of the WiFi network (up to 32 chars).
- **`password`:** Password (up to 64 chars).
- **`mode`:** `WIFI_CONFIG_AP_MODE` (AP, Station, Disabled).
- **`response`:** `WIFI_CONFIG_AP_RESPONSE` (Accepted, Rejected, Error).

Theoretical Use Cases

1. **Headless Provisioning:**
    - *Scenario:* A user buys a new telemetry module.
    - *Action:* Instead of connecting a USB cable, they connect to the module's default hotspot. The GCS sends `WIFI_CONFIG_AP` to instruct the module to join the user's home WiFi network (Station Mode).
2. **Field Reconfiguration:**
    - *Scenario:* A drone swarm needs to change WiFi channels to avoid interference.
    - *Action:* The GCS broadcasts a new configuration to all drones simultaneously via MAVLink, switching their onboard WiFi radios to a new SSID or frequency.

ISBD_LINK_STATUS (ID 335)

UNSUPPORTED

Summary

The `ISBD_LINK_STATUS` message reports the status of an Iridium SBD (Short Burst Data) satellite link. This includes signal quality, session status, and ring call alerts.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

While ArduPilot supports Iridium SBD modems (like the RockBlock) via the `AP_IridiumSBD` library, it does not currently expose the link status via this specific MAVLink message. Status information is typically logged internally or reported via `STATUSTEXT` messages.

Intended Data Fields (Standard)

- **`timestamp`:** Timestamp.
- **`last_heartbeat`:** Timestamp of last heartbeat.
- **`failed_sessions`:** Number of failed sessions.
- **`successful_sessions`:** Number of successful sessions.
- **`signal_quality`:** Signal quality (0-5).
- **`ring_pending`:** Ring alert pending.
- **`tx_session_pending`:** TX session pending.
- **`rx_session_pending`:** RX session pending.

Theoretical Use Cases

1. **Link Troubleshooting:**
    - *Scenario:* A BVLOS drone over the ocean stops reporting position.
    - *Action:* The last received `ISBD_LINK_STATUS` showed a signal quality of 0. This confirms the issue was antenna obstruction or satellite coverage, rather than a system crash.
2. **Cost Optimization:**
    - *Scenario:* Satellite data is expensive per byte.
    - *Action:* The GCS monitors `tx_session_pending` and `signal_quality`. It only queues non-critical telemetry uploads when `signal_quality` is 5/5, minimizing the chance of failed (but billed) transmission attempts.

UTM_GLOBAL_POSITION (ID 340)

UNSUPPORTED

Summary

The `UTM_GLOBAL_POSITION` message is used to report the vehicle's position to a UTM (Unmanned Traffic Management) system. It includes secure authentication tokens (UAS ID) and flight state information.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

For Remote ID and traffic management compliance, ArduPilot uses the **OpenDroneID** protocol family:

- `OPEN_DRONE_ID_BASIC_ID` (12900)
- `OPEN_DRONE_ID_LOCATION` (12901)
- `OPEN_DRONE_ID_AUTHENTICATION` (12902)
- `OPEN_DRONE_ID_SELF_ID` (12903)
- `OPEN_DRONE_ID_SYSTEM` (12904)
- `OPEN_DRONE_ID_OPERATOR_ID` (12905)

Intended Data Fields (Standard)

- **`time`**: Timestamp (us).
- **`uas_id`**: Unique Aerial System ID (18 chars).
- **`lat`** / **`lon`** / **`alt`**: Position (degE7, mm).
- **`relative_alt`**: Altitude above home (mm).
- **`vx`** / **`vy`** / **`vz`**: Velocity (cm/s).
- **`h_acc`** / **`v_acc`** / **`vel_acc`**: Uncertainty (mm).
- **`next_lat`** / **`next_lon`** / **`next_alt`**: Next waypoint.
- **`update_rate`**: Update rate (cs).
- **`flight_state`**: Flight state.
- **`flags`**: Flags.

Theoretical Use Cases

1. **Air Traffic Integration:**
   - *Scenario:* A delivery drone enters controlled airspace.
   - *Action:* The drone transmits `UTM_GLOBAL_POSITION` to a networked UTM Service Provider (USS). This provider forwards the position to Air Traffic Control, allowing them to see the drone on their radar screens alongside manned aircraft.

SMART_BATTERY_INFO (ID 370)

UNSUPPORTED

Summary

The `SMART_BATTERY_INFO` message provides static configuration and health info for a smart battery, such as its capacity, chemistry, and cycle count.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot supports Smart Batteries (SMBus, DroneCAN) but maps their data into the standard **`BATTERY_STATUS` (147)** message, which includes fields for voltages, current, temperature, and cell voltages.

Intended Data Fields (Standard)

- **`id`:** Battery ID.
- **`capacity_full_specification`:** Capacity when new.
- **`capacity_full`:** Current full capacity.
- **`cycle_count`:** Number of discharge cycles.
- **`weight`:** Weight.
- **`discharge_minimum_voltage`:** Minimum voltage.
- **`charging_minimum_voltage`:** Minimum charging voltage.
- **`resting_minimum_voltage`:** Resting voltage.
- **`charging_maximum_voltage`:** Max charging voltage.
- **`cells_in_series`:** Serial cell count.
- **`discharge_maximum_current`:** Max discharge current.
- **`charging_maximum_current`:** Max charging current.
- **`manufacture_date`:** Manufacture date.
- **`serial_number`:** Serial number.
- **`name`:** Name string.

Theoretical Use Cases

1. **Inventory Management:**
   - *Scenario:* A fleet of delivery drones swaps batteries 50 times a day.
   - *Action:* The GCS reads `serial_number` and `cycle_count` via `SMART_BATTERY_INFO`. It logs this data to a database to track exactly how many cycles each specific battery pack has undergone, scheduling retirement before performance degrades.
2. **Chemistry Verification:**
   - *Scenario:* A charger connects to the vehicle.

   - *Action:* The charger reads the `chemistry` field (not shown above but implied by standard) to automatically select the correct LiPo/Li-Ion/LiFePO4 charging profile, preventing fire hazards.

TUNNEL (ID 385)

UNSUPPORTED

Summary

The `TUNNEL` message is designed to tunnel arbitrary data (payloads) between MAVLink components. It allows for custom protocols or vendor-specific data to be transported transparently over the MAVLink network.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot supports **Serial Tunneling** over the **DroneCAN** bus (using `uavcan.tunnel.Targetted` messages) to transparently pass serial data (like GPS configuration or firmware updates) to CAN peripherals. However, it does not currently support the MAVLink `TUNNEL` message for this purpose.

Intended Data Fields (Standard)

- `target_system` / `target_component`: Targeted component.
- `payload_type`: Type of payload (enum `MAV_TUNNEL_PAYLOAD_TYPE`).
- `payload_length`: Length of data (up to 128 bytes).
- `payload`: Raw data buffer.

Theoretical Use Cases

1. **Proprietary Payload Control:**
   - *Scenario:* A user has a custom sensor payload that speaks a proprietary binary protocol (not MAVLink).
   - *Action:* Instead of writing a new MAVLink parser, the GCS encapsulates the binary blob in a `TUNNEL` message. The payload on the drone (connected to the Mavlink stream) unwraps the blob and consumes it directly.
2. **Debug Shell:**
   - *Scenario:* A developer wants to access a Linux shell on a Companion Computer via the telemetry radio.
   - *Action:* SSH traffic is chunked and wrapped into `TUNNEL` messages, creating a virtual terminal connection over the telemetry link.

ADAP_TUNING (ID 11010)

UNSUPPORTED

Summary

The `ADAP_TUNING` message is designed to report internal variables of an adaptive controller or auto-tuning process.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

For tuning telemetry, ArduPilot typically uses:

- `PID_TUNING` **(194):** Real-time PID data.
- `MAV_CMD_DO_AUTOTUNE_ENABLE`: To start AutoTune.
- `STATUSTEXT`: For AutoTune progress updates.

Intended Data Fields (Standard)

- `axis`: Axis being tuned (Roll, Pitch, Yaw).
- `desired`: Desired rate/angle.
- `achieved`: Achieved rate/angle.
- `error`: Error value.
- `theta`: Adaptive gain.
- `omega`: Adaptive frequency.
- `sigma`: Adaptive sigma.
- `theta_dot`: Derivative of gain.
- `omega_dot`: Derivative of frequency.
- `sigma_dot`: Derivative of sigma.
- `f`: Feed-forward value.
- `f_dot`: Derivative of feed-forward.
- `u`: Output.

Theoretical Use Cases

1. **Real-Time MRAC Debugging:**
   - *Scenario:* A developer is testing a Model Reference Adaptive Controller (MRAC).
   - *Action:* The controller streams `ADAP_TUNING` to report how the adaptive gains (`theta`) are evolving in real-time response to disturbances. The developer plots these gains to ensure they are converging and not oscillating.
2. **System Identification:**
   - *Scenario:* Identifying the moment of inertia of a new airframe.

    - *Action:* An adaptive estimator varies the control inputs and measures the response. `ADAP_TUNING` reports the estimated physical parameters (`omega`, `sigma`) as they settle.

SENSORS

GPS_STATUS (ID 25)

UNSUPPORTED

Summary

The `GPS_STATUS` message is defined in the MAVLink protocol to transmit detailed per-satellite information, including the PRN (satellite ID), Signal-to-Noise Ratio (SNR), Elevation, and Azimuth for up to 20 satellites. **ArduPilot does not implement or transmit this message.**

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

While many GPS modules (u-blox, MTK) provide the satellite constellation data required for this message, ArduPilot does not currently aggregate or stream this data via MAVLink. Basic satellite count and fix quality are instead provided by `GPS_RAW_INT` (24).

- **Absence:** There is no `send_gps_status` function in the GCS libraries.
- **Alternative:** Advanced users often rely on u-blox "pass-through" (Serial over MAVLink) or DataFlash logs to view detailed satellite geometry during post-flight analysis.

Data Fields (Standard)

- **`satellites_visible`:** Number of satellites visible.
- **`satellite_prn`:** Global satellite ID.
- **`satellite_used`:** 0: Satellite not used, 1: used for localization.
- **`satellite_elevation`:** Elevation (0: right on top of receiver, 90: on the horizon) of satellite.
- **`satellite_azimuth`:** Direction of satellite, 0: 0 deg, 255: 360 deg.
- **`satellite_snr`:** Signal to noise ratio of satellite.

Key Codebase Locations

- **libraries/GCS_MAVLink/ap_message.h**: Message ID 25 is missing from the supported message enum.
- **libraries/GCS_MAVLink/GCS_Common.cpp**: Lacks a handler for ID 25 in both TX and RX loops.

RAW_PRESSURE (ID 28)

UNSUPPORTED

Summary

The `RAW_PRESSURE` message is defined in the MAVLink protocol to transmit uncalibrated pressure readings. **ArduPilot does not implement or transmit this message.** Instead, it uses `SCALED_PRESSURE` (29) to report physics-ready absolute and differential pressure data.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

ArduPilot's philosophy is to perform sensor calibration and scaling onboard. Consequently, it bypasses "Raw" messages (which often imply raw ADC counts or uncompensated values) for subsystems like Barometers.

- **Replacement:** Users should look for `SCALED_PRESSURE` (29) for the primary barometer.
- **Secondary Sensors:** `SCALED_PRESSURE2` (137) and `SCALED_PRESSURE3` (143) are used for redundant sensors.

Data Fields (Standard)

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `press_abs`: Absolute pressure (raw).
- `press_diff1`: Differential pressure 1 (raw, 0 if nonexistant).
- `press_diff2`: Differential pressure 2 (raw, 0 if nonexistant).
- `temperature`: Raw Temperature measurement (raw).

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:1030**: Lacks a mapping for `MAVLINK_MSG_ID_RAW_PRESSURE` (28), while mapping ID 29 to `MSG_SCALED_PRESSURE`.

OPTICAL_FLOW_RAD (ID 106)

UNSUPPORTED

Summary

The `OPTICAL_FLOW_RAD` message is an advanced alternative to the standard `OPTICAL_FLOW` (100) message. It transmits flow data as integrated angular speeds (radians) and includes integrated gyroscope data for better rotation compensation. **ArduPilot does not currently implement a handler or sender for this message.**

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

While many modern flow sensors (like the PX4Flow) output data in radians, ArduPilot's MAVLink backend for optical flow (`AP_OpticalFlow_MAV`) is currently hardcoded to expect the pixel-based `OPTICAL_FLOW` (100) message.

- **Absence:** There is no mapping for `MAVLINK_MSG_ID_OPTICAL_FLOW_RAD` in `GCS_Common.cpp` or `AP_OpticalFlow`.
- **Recommendation:** Developers using external MAVLink flow sensors should ensure they are configured to output the standard pixel-based `OPTICAL_FLOW` (100) packet.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `sensor_id`: Sensor ID.
- `integration_time_us`: Integration time.
- `integrated_x`: Integrated optical flow around X-axis (rad).
- `integrated_y`: Integrated optical flow around Y-axis (rad).
- `integrated_xgyro`: Integrated gyro around X-axis (rad).
- `integrated_ygyro`: Integrated gyro around Y-axis (rad).
- `integrated_zgyro`: Integrated gyro around Z-axis (rad).
- `temperature`: Temperature (centidegrees).
- `quality`: Optical flow quality / confidence. 0: bad, 255: maximum quality.
- `time_delta_distance_us`: Time since the distance was sampled.
- `distance`: Distance to the center of the flow field. Positive value: distance known. Negative value: Unknown distance.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:4047**: Shows that only `OPTICAL_FLOW` (100) is handled.
- **libraries/AP_OpticalFlow/AP_OpticalFlow_MAV.cpp:92**: Decodes only the pixel-based message.

GPS2_STATUS (ID 125)

UNSUPPORTED

Summary

The `GPS2_STATUS` message is defined in the MAVLink protocol to provide detailed satellite-level information for a secondary GPS receiver, including PRN, SNR, Elevation, and Azimuth for each visible satellite. **ArduPilot does not implement or transmit this message.**

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

Consistent with the lack of support for the primary `GPS_STATUS` (25) message, ArduPilot does not stream satellite constellations for secondary receivers via MAVLink. General health, fix type, and satellite counts for the second GPS are instead provided by `GPS2_RAW` (124).

- **Absence:** There is no handler for `MAVLINK_MSG_ID_GPS2_STATUS` in the `GCS_MAVLink` libraries.
- **Alternative:** Users needing detailed satellite info typically use "Pass-Through" logging or vendor-specific tools.

Intended Data Fields (Standard)

- `satellites_visible`: Number of satellites visible.
- `satellite_prn`: Global satellite ID.
- `satellite_used`: 0: Satellite not used, 1: used for localization.
- `satellite_elevation`: Elevation (0: right on top of receiver, 90: on the horizon) of satellite.
- `satellite_azimuth`: Direction of satellite, 0: 0 deg, 255: 360 deg.
- `satellite_snr`: Signal to noise ratio of satellite.

Theoretical Use Cases

1. **Deep Multipath Analysis:**
   - *Scenario:* A rover operating in an urban canyon with two GPS receivers.
   - *Action:* By analyzing the SNR and Elevation of individual satellites from both receivers, a researcher could determine which unit is seeing more obstructed sky and tune the blending weights accordingly.
2. **Constellation Verification:**
   - *Scenario:* Verifying a "Dual Frequency" (L1/L5) receiver.
   - *Action:* Checking the PRN codes to confirm the secondary receiver is actually tracking the specific satellites (e.g., Galileo E5a) claimed by the manufacturer.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Lacks a handler or sender for ID 125.

SENSOR_OFFSETS (ID 150)

UNSUPPORTED

Summary

The `SENSOR_OFFSETS` message was historically used to report raw calibration offsets for the IMU and Magnetometer.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message. It is deprecated.

Calibration offsets are now handled via:

1. **Parameters:** `INS_ACCOFFS_*`, `COMPASS_OFS_*`.
2. `MAV_CMD_PREFLIGHT_SET_SENSOR_OFFSETS` **(242):** For setting offsets (though this is also largely superseded by onboard calibration routines).

Data Fields

- `mag_ofs_x`: Magnetometer X offset.
- `mag_ofs_y`: Magnetometer Y offset.
- `mag_ofs_z`: Magnetometer Z offset.
- `mag_declination`: Magnetic declination (radians).
- `raw_press`: Raw pressure (Pa).
- `raw_temp`: Raw temperature (degC).
- `gyro_cal_x`: Gyro X calibration.
- `gyro_cal_y`: Gyro Y calibration.
- `gyro_cal_z`: Gyro Z calibration.
- `accel_cal_x`: Accel X calibration.
- `accel_cal_y`: Accel Y calibration.
- `accel_cal_z`: Accel Z calibration.

WIND_COV (ID 231) UNSUPPORTED

Summary

Wind covariance report.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It uses `WIND` (168) for wind estimation.

Theoretical Use Cases

Wind estimation with uncertainty metrics.

RAW_RPM (ID 339)

UNSUPPORTED

Summary

The `RAW_RPM` message is designed to report the unscaled frequency/RPM from a sensor, often before applying scaling factors like pole count or gear ratio.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot supports RPM reporting via:

1. `RPM` **(226):** For dedicated RPM sensors (Hall effect, Optical).
2. `ESC_TELEMETRY_1_TO_4` **(11030+):** For RPM data coming directly from ESCs (BLHeli, DroneCAN).

The `RAW_RPM` message is not currently used in the ArduPilot MAVLink stream.

Intended Data Fields (Standard)

- `index`**:** RPM Sensor index.
- `frequency`**:** Frequency in Hz.

Theoretical Use Cases

1. **Sensor Debugging:**
    - *Scenario:* A user is setting up a new RPM sensor but doesn't know the magnet layout on the flywheel.
    - *Action:* The sensor reports `RAW_RPM` (raw interrupts per second). The user revs the engine to a known speed (audible check) and compares it to the raw frequency to derive the correct scaling factor (Poles).
2. **Vibration Analysis:**
    - *Scenario:* Detecting prop imbalance.
    - *Action:* High-frequency raw data might reveal micro-fluctuations in rotation speed within a single rotation (if sampled fast enough), indicating a damaged blade.

CONTROL

MANUAL_SETPOINT (ID 81)

UNSUPPORTED

Summary

Manual setpoint message.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Manual control setpoints.

SET_ACTUATOR_CONTROL_TARGET (ID 139)

UNSUPPORTED

Summary

The `SET_ACTUATOR_CONTROL_TARGET` message is designed to set the "Actuator Control Target," which allows for direct control of the vehicle's actuators (motors and servos) normalized from -1.0 to 1.0. This bypasses the higher-level flight modes and rate controllers.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

For direct actuator control, ArduPilot typically uses:

1. **`RC_CHANNELS_OVERRIDE` (70):** To override radio inputs.
2. **`MAV_CMD_DO_SET_SERVO`:** To set a specific servo to a PWM value.
3. **`MAV_CMD_DO_SET_ACTUATOR`:** A newer command to control actuators.

Data Fields

- **`time_usec`:** Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- **`group_mlx`:** Actuator group. The "_mlx" indicates this is a multi-instance message and a MAVLink parser should use this field to difference between instances.
- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`controls`:** Actuator controls. Normed -1..+1 where 0 is neutral or de-throttled. Each action system number 0 is also a special value and means no action. The mapping of controls to actuators is vehicle specific.

ACTUATOR_CONTROL_TARGET (ID 140)

UNSUPPORTED

Summary

The `ACTUATOR_CONTROL_TARGET` message reports the current "Target" state of the actuators (motors and servos), normalized from -1.0 to 1.0. This allows a Ground Control Station (GCS) to monitor what the mixer is trying to achieve.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

Instead, ArduPilot uses:

1. `SERVO_OUTPUT_RAW` **(36):** To report the actual PWM output sent to the motors.
2. `NAV_CONTROLLER_OUTPUT` **(62):** To report high-level navigation targets (Roll, Pitch, Bearing).

Data Fields

- `time_usec`**:** Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `group_mlx`**:** Actuator group. The "_mlx" indicates this is a multi-instance message and a MAVLink parser should use this field to difference between instances.
- `controls`**:** Actuator controls. Normed -1..+1 where 0 is neutral or de-throttled. Each action system number 0 is also a special value and means no action. The mapping of controls to actuators is vehicle specific.

SET_MAG_OFFSETS (ID 151)

UNSUPPORTED

Summary

The `SET_MAG_OFFSETS` message was historically used to set the magnetometer calibration offsets.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message. It is deprecated.

Modern systems use:

1. **Onboard Compass Calibration:** Invoked via `MAV_CMD_DO_START_MAG_CAL` (42424) or `MAG_CAL_REPORT` messages.
2. **Parameters:** Setting `COMPASS_OFS_X/Y/Z` directly.

Data Fields

- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`mag_ofs_x`:** Magnetometer X offset.
- **`mag_ofs_y`:** Magnetometer Y offset.
- **`mag_ofs_z`:** Magnetometer Z offset.

SET_HOME_POSITION (ID 243)

UNSUPPORTED

Summary

The `SET_HOME_POSITION` message is defined in the MAVLink standard to allow Ground Control Stations to update the vehicle's home location. However, **ArduPilot does not implement a handler for this specific message ID.** Instead, ArduPilot uses the command-based `MAV_CMD_DO_SET_HOME` (via `COMMAND_LONG` or `COMMAND_INT`) to manage home position updates.

Status

Unsupported (Use `MAV_CMD_DO_SET_HOME` instead)

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

ArduPilot prefers the command protocol (`MAV_CMD`) for state changes because it provides a standardized acknowledgement mechanism (`COMMAND_ACK`). Standalone messages like `SET_HOME_POSITION` often lack built-in ACKs, making them less reliable for critical safety operations.

- **Alternative:** Ground Control Stations should send a `COMMAND_LONG` with `command = 179 (MAV_CMD_DO_SET_HOME)`.
- **Behavior:** When received via the command protocol, ArduPilot verifies that the vehicle is disarmed (in most configurations) before committing the new home coordinates to the AHRS and storage.

Data Fields (Standard)

- `target_system`: System ID.
- `latitude`: Latitude (WGS84), in degrees * 1E7.
- `longitude`: Longitude (WGS84, in degrees * 1E7.
- `altitude`: Altitude (MSL), in meters * 1000 (positive for up).
- `x`: Local X position of this position in the local coordinate frame.
- `y`: Local Y position of this position in the local coordinate frame.
- `z`: Local Z position of this position in the local coordinate frame.
- `q`: World to surface normal and heading transformation of the takeoff position. Used to indicate the heading and slope of the ground.
- `approach_x`: Local X position of the end of the approach vector. Multicopters should set this position based on their takeoff path. Grass-landing fixed wing aircraft should set it the same way as multicopters. Runway-landing fixed wing aircraft should set it to the opposite direction of the takeoff, assuming the takeoff happened from the threshold / touchdown zone.
- `approach_y`: Local Y position of the end of the approach vector. Multicopters should set this position based on their takeoff path. Grass-landing fixed wing aircraft should set it the same way as multicopters. Runway-landing fixed wing aircraft should set it to the opposite direction of the takeoff, assuming the takeoff happened from the threshold / touchdown zone.
- `approach_z`: Local Z position of the end of the approach vector. Multicopters should set this position based on their takeoff path. Grass-landing fixed wing aircraft should set it the same way as multicopters. Runway-landing fixed wing aircraft should set it to the opposite direction of the takeoff, assuming the takeoff happened from the threshold / touchdown zone.
- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp:5529**: Handling of the `MAV_CMD_DO_SET_HOME` equivalent.

ACTUATOR_OUTPUT_STATUS (ID 375)

UNSUPPORTED

Summary

The `ACTUATOR_OUTPUT_STATUS` message is designed to report the status of actuators (servos, motors) in a more flexible array format (up to 32 actuators) compared to the older `SERVO_OUTPUT_RAW` (limited to 16).

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot relies on `SERVO_OUTPUT_RAW` **(36)** to report the PWM output values of its servo channels (1-16). For higher channel counts or specific actuator feedback, other mechanisms or multiple messages would be required, but `ACTUATOR_OUTPUT_STATUS` is not currently used.

Intended Data Fields (Standard)

- **`time_usec`:** Timestamp (us since UNIX epoch).
- **`active`:** Number of active actuators.
- **`actuator`:** Array of 32 float values (output).

Theoretical Use Cases

1. **Complex Robotics:**
    - *Scenario:* A hexapod robot with 18 servos (3 per leg).
    - *Action:* `SERVO_OUTPUT_RAW` can only report the first 16. `ACTUATOR_OUTPUT_STATUS` allows reporting all 18 joint angles in a single atomic message frame, simplifying the log analysis for walking gaits.
2. **Normalized Feedback:**
    - *Scenario:* Analyzing mixer performance.
    - *Action:* The message reports values as normalized floats (0.0 to 1.0 or -1.0 to 1.0) rather than raw PWM microseconds. This makes it easier to debug the mixer logic independent of the specific servo calibration endpoints.

MISSION_REQUEST_PARTIAL_LIST (ID 37)

UNSUPPORTED

Summary

Request a partial list of mission items from the system.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It relies on the standard `MISSION_REQUEST_LIST` and individual `MISSION_REQUEST` messages to download missions.

Theoretical Use Cases

Efficiently downloading a small range of waypoints from a large mission.

SAFETY_SET_ALLOWED_AREA (ID 54)

UNSUPPORTED

Summary

Set a safety zone (volume) for the vehicle.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It uses the `AC_Fence` library and the Fence Protocol (items with `MAV_CMD_NAV_FENCE_...`) for geofencing.

Theoretical Use Cases

Defining a safety box.

SAFETY_ALLOWED_AREA (ID 55)

UNSUPPORTED

Summary

Read out the safety zone the MAV currently assumes.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Reading geofence boundaries.

TRAJECTORY_REPRESENTATION_WAYPOINTS (ID 332) UNSUPPORTED

Summary

The `TRAJECTORY_REPRESENTATION_WAYPOINTS` message is used to describe a trajectory as a series of waypoints. It allows for more complex path planning descriptions than standard mission items, often used in offboard control scenarios.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot typically handles trajectories via:

1. **Mission Protocol:** Uploading a list of `MISSION_ITEM_INT` waypoints.
2. **Guided Mode:** Sending individual `SET_POSITION_TARGET_LOCAL_NED` or `SET_POSITION_TARGET_GLOBAL_INT` commands.

Intended Data Fields (Standard)

- **`time_usec`:** Timestamp.
- **`valid_points`:** Number of valid points (up to 5).
- **`pos_x` / `pos_y` / `pos_z`:** Arrays of X, Y, Z positions.
- **`vel_x` / `vel_y` / `vel_z`:** Arrays of X, Y, Z velocities.
- **`acc_x` / `acc_y` / `acc_z`:** Arrays of X, Y, Z accelerations.
- **`pos_yaw`:** Array of yaw angles.
- **`vel_yaw`:** Array of yaw rates.
- **`command`:** Array of MAV_CMD commands associated with waypoints.

Theoretical Use Cases

1. **Lookahead Planning:**
   - *Scenario:* A companion computer is planning a path through a dynamic environment.
   - *Action:* It sends a packet of 5 waypoints representing the immediate future trajectory (0s, 1s, 2s, 3s, 4s) to the flight controller. This provides the controller with "intent" data, allowing for smoother feed-forward control than sending a single setpoint at 50Hz.
2. **Swarm Choreography:**
   - *Scenario:* Drones flying in formation.
   - *Action:* The central computer broadcasts the next 5 seconds of the "dance" to all drones. If the radio link drops for a second, the drones continue on the pre-buffered path without jittering.

TRAJECTORY_REPRESENTATION_BEZIER (ID 333) UNSUPPORTED

Summary

The `TRAJECTORY_REPRESENTATION_BEZIER` message describes a trajectory using Bezier curves. This allows for extremely smooth path definitions with continuous velocity and acceleration profiles, often used in advanced motion planning.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

Internally, ArduPilot uses S-Curve navigation (SCurve) for smooth path generation between waypoints, but it does not currently accept external Bezier curve definitions via MAVLink.

Intended Data Fields (Standard)

- `time_usec`: Timestamp.
- `valid_points`: Number of valid points.
- `pos_x` / `pos_y` / `pos_z`: Arrays of control points.
- `delta`: Array of time deltas for each segment.
- `pos_yaw`: Array of yaw angles.

Theoretical Use Cases

1. **Cinematic Videography:**
   - *Scenario:* Filming a car chase.
   - *Action:* A director designs a complex, sweeping camera move in 3D software. The software exports the path as a series of Bezier curves, which are uploaded to the drone. The drone executes the curve perfectly, ensuring smooth acceleration that doesn't jerk the gimbal.
2. **Obstacle Avoidance:**
   - *Scenario:* High-speed flight through a forest.
   - *Action:* A path planner generates a curved tube that avoids trees. Sending Bezier control points allows the drone to follow this curved tube much more accurately than connecting straight line waypoints.

PAYLOAD

`CAMERA_TRIGGER` (ID 112) UNSUPPORTED

Summary

The `CAMERA_TRIGGER` message is defined in MAVLink to notify external systems that a camera shutter has been activated. **ArduPilot does not implement this message.** Instead, it uses the more modern and detailed `CAMERA_FEEDBACK` (180) and `CAMERA_IMAGE_CAPTURED` (263) messages to report camera events.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

ArduPilot's camera subsystem (`AP_Camera`) focuses on providing rich feedback about the exact moment and location of a photo capture, which is essential for photogrammetry. The simple `CAMERA_TRIGGER` message lacks the precision and metadata (like roll/pitch/yaw and GPS coordinates) required for professional mapping, so it is bypassed in favor of other messages.

- **Replacement:** Users and GCS developers should listen for `CAMERA_FEEDBACK` (180) to receive the exact position and attitude data associated with a shutter event.

Data Fields

- **`time_usec`:** Timestamp for the image frame in microseconds.
- **`seq`:** Image sequence number.

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Lacks a handler or sender for ID 112.

DATA_TRANSMISSION_HANDSHAKE (ID 130) UNSUPPORTED

Summary

The `DATA_TRANSMISSION_HANDSHAKE` message is part of the legacy MAVLink Image Transmission Protocol. It is used to initiate a data transfer session (typically for images or other large binary blobs) between a component and a Ground Control Station.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

This protocol has largely been superseded by:

1. **MAVLink FTP (`FILE_TRANSFER_PROTOCOL`):** For transferring logs, parameters, and mission files.
2. **RTSP/UDP Streaming:** For live video (see `VIDEO_STREAM_INFORMATION`).
3. `CAMERA_IMAGE_CAPTURED`**:** For notifying GCS of new photos, often with HTTP download links.

Data Fields

- `type`**:** Type of requested/acknowledged data.
- `size`**:** Total data size in bytes.
- `width`**:** Width of a matrix or image.
- `height`**:** Height of a matrix or image.
- `packets`**:** Number of packets being sent.
- `payload`**:** Payload size per packet.
- `jpg_quality`**:** JPEG quality.

Theoretical Use Cases

1. **Low-Bandwidth Image Transfer:**
    - *Scenario:* A satellite-connected drone takes a surveillance photo.
    - *Action:* Since the link is too slow for RTSP, the drone initiates a handshake to send the image as a series of small, acknowledged MAVLink packets (`ENCAPSULATED_DATA`), allowing the GCS to reconstruct the image over several minutes.

ENCAPSULATED_DATA (ID 131) UNSUPPORTED

Summary

The `ENCAPSULATED_DATA` message carries the actual binary payload (chunks of an image or file) during a transfer initiated by `DATA_TRANSMISSION_HANDSHAKE`.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message. It is part of the legacy Image Transmission Protocol.

Data Fields

- `seqnr`: Sequence number (starting at 0).
- `data`: Raw data (up to 253 bytes).

Theoretical Use Cases

1. **Chunked File Transfer:**
    - *Scenario:* Transmitting a 10KB thumbnail image.
    - *Action:* The image is split into 40 packets. Each packet is sent as an `ENCAPSULATED_DATA` message. The receiver uses the `seqnr` to reassemble the binary blob in the correct order.

DIGICAM_CONFIGURE (ID 154)

LEGACY / UNSUPPORTED

Summary

The `DIGICAM_CONFIGURE` message was historically used to configure onboard camera settings like aperture, shutter speed, and ISO.

Status

Legacy / Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message directly.

Instead, camera configuration is handled via the Mission Command protocol using **`MAV_CMD_DO_DIGICAM_CONFIGURE` (202)**.

- **Mission Items:** Users add `DO_DIGICAM_CONFIGURE` commands to missions.
- **Command Handling:** `AP_Camera` processes these commands to adjust settings on supported camera backends (e.g., MAVLink cameras, Relay triggers).

Data Fields

- **`target_system`:** System ID.
- **`target_component`:** Component ID.
- **`mode`:** Shooting mode (A, S, M, Auto, etc.).
- **`shutter_speed`:** Shutter speed (1/value).
- **`aperture`:** Aperture (F-stop * 10).
- **`iso`:** ISO sensitivity.
- **`exposure_type`:** Exposure type.
- **`command_id`:** Command Identity.
- **`engine_cut_off`:** Main engine cut-off time (seconds/10).
- **`extra_param`:** Extra parameter.
- **`extra_value`:** Extra value.

MOUNT_CONFIGURE (ID 156)

LEGACY / UNSUPPORTED

Summary

The `MOUNT_CONFIGURE` message was historically used to configure the operation mode (retract, neutral, mavlink targeting, rc targeting, gps point) of a gimbal/mount.

Status

Legacy / Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (ArduPilot explicitly warns of deprecation if received)

Description

ArduPilot **deprecated** this message in version 4.5. Receiving it may trigger a `send_received_message_deprecation_warning`.

Users should instead use the Mission Command:

- `MAV_CMD_DO_MOUNT_CONFIGURE` **(204)**

Data Fields

- `target_system`**:** System ID.
- `target_component`**:** Component ID.
- `mount_mode`**:** Mount operation mode (`MAV_MOUNT_MODE` enum).
- `stab_roll`**:** (1 = yes, 0 = no).
- `stab_pitch`**:** (1 = yes, 0 = no).
- `stab_yaw`**:** (1 = yes, 0 = no).

MOUNT_CONTROL (ID 157)

LEGACY / UNSUPPORTED

Summary

The `MOUNT_CONTROL` message was historically used to control the pitch, roll, and yaw angles of a gimbal/mount.

Status

Legacy / Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (ArduPilot explicitly warns of deprecation if received)

Description

ArduPilot **deprecated** this message in version 4.5. Receiving it may trigger a `send_received_message_deprecation_warning`.

Users should instead use the Mission Command:

- `MAV_CMD_DO_MOUNT_CONTROL` **(205)**

Data Fields

- `target_system`: System ID.
- `target_component`: Component ID.
- `input_a`: Pitch (centi-degrees) or Lat (deg * 1E7) depending on mode.
- `input_b`: Roll (centi-degrees) or Lon (deg * 1E7) depending on mode.
- `input_c`: Yaw (centi-degrees) or Alt (cm) depending on mode.
- `save_position`: Save current position (1 = yes).

CAMERA_STATUS (ID 179)

UNSUPPORTED

Summary

Legacy camera status message.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

This message is not supported by ArduPilot. Use `CAMERA_FEEDBACK` (180) or `CAMERA_IMAGE_CAPTURED` (263) instead.

Theoretical Use Cases

Legacy camera reporting.

GIMBAL_TORQUE_CMD_REPORT (ID 214)

UNSUPPORTED

Summary

Report torque commands for gimbal.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Debugging gimbal motor torques.

GOPRO_GET_REQUEST (ID 216)

UNSUPPORTED

Summary

Request a setting from a GoPro.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not process this message. It is likely routed directly between the GCS and the Gimbal/Camera if used.

Theoretical Use Cases

Retrieving camera settings.

GOPRO_GET_RESPONSE (ID 217)

UNSUPPORTED

Summary

Response to a GoPro setting request.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not process this message.

Theoretical Use Cases

Returning camera settings.

GOPRO_SET_REQUEST (ID 218)

UNSUPPORTED

Summary

Request to set a GoPro setting.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not process this message.

Theoretical Use Cases

Changing camera settings.

GOPRO_SET_RESPONSE (ID 219)

UNSUPPORTED

Summary

Response to a GoPro set request.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not process this message.

Theoretical Use Cases

Confirming camera setting changes.

STORAGE_INFORMATION (ID 261)

UNSUPPORTED

Summary

The STORAGE_INFORMATION message is designed to report the status, capacity, and usage of onboard storage media (e.g., SD cards, internal flash, SSDs). It supports reporting for multiple storage devices via a storage_id index. This message allows a Ground Control Station to warn the pilot if storage is low or if a drive is failing before starting a mission or recording session.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot currently does not implement this message. While ArduPilot supports MEMINFO for RAM usage and SYS_STATUS for general system health, it does not have a standardized mechanism for reporting detailed file system statistics via this specific MAVLink message.

If implemented in the future, it would likely be used by the AP_Logger or AP_Filesystem libraries to report the status of the microSD card.

Intended Data Fields (Standard)

- **time_boot_ms**: Timestamp (ms since boot).
- **storage_id**: ID of the storage device (1-based index).
- **storage_count**: Total number of storage devices.
- **status**: Status flags (STORAGE_STATUS enum) indicating if the device is ready, unformatted, or has errors.
- **total_capacity**: Total size in MiB.
- **used_capacity**: Used space in MiB.
- **available_capacity**: Free space in MiB.
- **read_speed**: Read throughput in MiB/s.
- **write_speed**: Write throughput in MiB/s.

Theoretical Use Cases

1. **Pre-Flight Check:**
   - *Scenario:* A photographer is about to launch a mission.
   - *Action:* The GCS checks available_capacity. If it's less than 1GB (insufficient for the planned 4K video recording), it prevents arming and prompts the user to format the card.
2. **Health Monitoring:**
   - *Scenario:* A microSD card is degrading.
   - *Action:* The autopilot detects slow write speeds and reports a low write_speed. The GCS warns the user that high-rate logging might be compromised.

`CAMERA_IMAGE_CAPTURED` (ID 263)

UNSUPPORTED

Summary

The `CAMERA_IMAGE_CAPTURED` message is intended to be emitted every time a camera captures an image. It includes detailed metadata such as the precise UTC timestamp, geolocation (lat/lon/alt), orientation (quaternion), and the file URL. It is effectively a richer, modern replacement for the older `CAMERA_FEEDBACK` message.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

Instead, ArduPilot uses the legacy `CAMERA_FEEDBACK` **(180)** message to report successful image captures. This usually occurs when the camera's hot-shoe signal triggers the flight controller's feedback pin.

Intended Data Fields (Standard)

- `time_boot_ms`: Timestamp (ms since boot).
- `time_utc`: Capture time (us since UNIX epoch).
- `camera_id`: Deprecated/unused.
- `lat`, `lon`, `alt`: Location where image was taken.
- `relative_alt`: Altitude above ground.
- `q`: Quaternion of camera orientation.
- `image_index`: Zero-based index of this image.
- `capture_result`: Success (1) or Failure (0).
- `file_url`: URL of the image (local path or HTTP).

Alternative

For image capture feedback in ArduPilot, refer to `CAMERA_FEEDBACK` **(180)**.

Theoretical Use Cases

1. **Instant Geotagging:**
   - *Scenario:* A companion computer is building a map in real-time.
   - *Action:* It receives `CAMERA_IMAGE_CAPTURED` containing the exact pose (`q`) and location (`lat/lon`) of the image just taken, allowing it to project the image onto the map without needing to interpolate log data later.
2. **File Management:**
   - *Scenario:* A WiFi-connected camera.

    - *Action:* The message includes `file_url`, allowing the GCS to immediately download a thumbnail of the image via HTTP.

MOUNT_ORIENTATION (ID 265)

UNSUPPORTED

Summary

The `MOUNT_ORIENTATION` message is a modern replacement for the older `MOUNT_STATUS` message. It is designed to report the precise orientation of a gimbal or camera mount in the global frame (Roll/Pitch/Yaw relative to Earth) and the body frame (Yaw relative to vehicle).

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot currently **does not** implement this message.

Instead, ArduPilot uses the legacy `MOUNT_STATUS` **(158)** message to report gimbal angles. `MOUNT_STATUS` provides the pitch, roll, and yaw angles in centi-degrees, typically relative to the body frame (depending on the mode).

Intended Data Fields (Standard)

- **`time_boot_ms`:** Timestamp (ms since boot).
- **`roll`:** Roll in global frame (deg).
- **`pitch`:** Pitch in global frame (deg).
- **`yaw`:** Yaw relative to the vehicle (deg).
- **`yaw_absolute`:** Yaw in global frame (relative to North) (deg).

Alternative

For gimbal feedback in ArduPilot, refer to `MOUNT_STATUS` **(158)**.

Theoretical Use Cases

1. **Augmented Reality (AR):**
    - *Scenario:* A GCS overlays street names on the live video feed.
    - *Action:* The GCS uses `yaw_absolute` and `pitch` from `MOUNT_ORIENTATION` to accurately project the 3D map data onto the 2D video plane, independent of the drone's own heading.

VIDEO_STREAM_STATUS (ID 270)

UNSUPPORTED

Summary

The `VIDEO_STREAM_STATUS` message is designed to report the real-time health and telemetry of a video stream, such as current bitrate, framerate, and resolution. This differs from `VIDEO_STREAM_INFORMATION`, which provides static configuration data (like the URI).

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

While ArduPilot supports sending static stream configuration via `VIDEO_STREAM_INFORMATION` (269), it does not currently monitor or report the real-time performance statistics (bandwidth usage, dropped frames) of the video link.

Intended Data Fields (Standard)

- **`stream_id`:** Video Stream ID (1-based).
- **`flags`:** Status flags (`VIDEO_STREAM_STATUS_FLAGS`) indicating if the stream is running, thermal, etc.
- **`framerate`:** Frame rate (Hz).
- **`resolution_h`:** Horizontal resolution (pix).
- **`resolution_v`:** Vertical resolution (pix).
- **`bitrate`:** Bit rate (bits/s).
- **`rotation`:** Video image rotation (deg).
- **`hfov`:** Horizontal Field of View (deg).

Theoretical Use Cases

1. **Adaptive Streaming:**
   - *Scenario:* The radio link quality drops.
   - *Action:* The camera reports a reduced `bitrate` via `VIDEO_STREAM_STATUS`. The GCS sees this and automatically downgrades the video player quality to match, preventing buffering or lag.
2. **Diagnostics:**
   - *Scenario:* The video feed is black.
   - *Action:* The GCS checks `VIDEO_STREAM_STATUS`. If `framerate` is 0, it knows the camera sensor has failed. If `framerate` is 30 but the image is black, it might be a lens cap or exposure issue.

CAMERA_TRACKING_IMAGE_STATUS (ID 275) UNSUPPORTED

Summary

The `CAMERA_TRACKING_IMAGE_STATUS` message is designed to report the status of an on-camera object tracking system, specifically using image coordinates (pixel space). It reports the location of the tracked point or rectangle within the video frame.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

While ArduPilot supports sending tracking commands *to* a camera (e.g., `MAV_CMD_CAMERA_TRACK_POINT`), it does not currently have a mechanism to relay the high-frequency *status* of that tracking (the bounding box location) back to the GCS via this MAVLink message.

Intended Data Fields (Standard)

- **`tracking_status`:** Current status (`CAMERA_TRACKING_STATUS_FLAGS`).
- **`tracking_mode`:** Current mode (`CAMERA_TRACKING_MODE`).
- **`target_data`:** Generic data.
- **`point_x`, `point_y`:** Point location in image (0..1).
- **`radius`:** Radius of tracked object.
- **`rec_top_x`, `rec_top_y`:** Bounding box top-left.
- **`rec_bottom_x`, `rec_bottom_y`:** Bounding box bottom-right.

Theoretical Use Cases

1. **Visual Confirmation:**
   - *Scenario:* A user taps a car on the screen to track it.
   - *Action:* The camera reports the `rec_top_x` / `rec_bottom_y` of the object it has locked onto. The GCS draws a green rectangle around the car, confirming to the user that the "lock" command was successful and the camera is tracking the correct object.
2. **Target Loss Warning:**
   - *Scenario:* The tracked object moves behind a tree.
   - *Action:* The message stops arriving or reports a "Lost" status. The GCS alerts the pilot to manually intervene.

CAMERA_TRACKING_GEO_STATUS (ID 276)

UNSUPPORTED

Summary

The `CAMERA_TRACKING_GEO_STATUS` message is designed to report the status of an on-camera object tracking system, specifically using geospatial coordinates (Latitude/Longitude). Unlike `CAMERA_FOV_STATUS`, which reports where the *center* of the image is looking, this message reports the location of the *specific object* being tracked (e.g., a car or person).

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

While ArduPilot can calculate the camera's center-of-view intersection with the ground (`CAMERA_FOV_STATUS`), it does not currently support relaying specific object-tracking telemetry from smart cameras back to the GCS via this MAVLink message.

Intended Data Fields (Standard)

- **`tracking_status`:** Current status (`CAMERA_TRACKING_STATUS_FLAGS`).
- **`tracking_mode`:** Current mode (`CAMERA_TRACKING_MODE`).
- **`lat`, `lon`, `alt`:** Location of the tracked object.
- **`h_acc`, `v_acc`:** Accuracy/Uncertainty of the location.
- **`vel_n`, `vel_e`, `vel_d`:** Velocity of the tracked object.
- **`vel_acc`:** Velocity accuracy.
- **`dist`:** Distance to the tracked object.
- **`hdg`:** Heading of the tracked object.

Theoretical Use Cases

1. **Target Speed Estimation:**
    - *Scenario:* Following a suspect vehicle.
    - *Action:* The camera uses optical flow and rangefinding to calculate the vehicle's speed (`vel_n`, `vel_e`). The GCS displays this speed to the operator.
2. **Coordinate Handover:**
    - *Scenario:* A surveillance drone spots a target.
    - *Action:* It broadcasts `CAMERA_TRACKING_GEO_STATUS` with the target's `lat/lon`. A second "interceptor" drone receives this location and automatically navigates to intercept the target.

GIMBAL_MANAGER_SET_MANUAL_CONTROL (ID 288) UNSUPPORTED

Summary

The `GIMBAL_MANAGER_SET_MANUAL_CONTROL` message provides a mechanism for direct "stick input" control of a gimbal. It accepts normalized values (-1.0 to 1.0) for pitch, yaw, and zoom, mimicking the raw inputs from an RC controller or joystick.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

For manual gimbal control via MAVLink, ArduPilot typically uses:

1. **`GIMBAL_MANAGER_SET_PITCHYAW` (287):** sending angular rates (rad/s) which can be mapped from joystick axes.
2. **`RC_CHANNELS_OVERRIDE` (70):** overriding the specific RC channels mapped to the gimbal functions.

Intended Data Fields (Standard)

- **`target_system` / `target_component`:** Targeted component.
- **`flags`:** Bitmap (`GIMBAL_MANAGER_FLAGS`).
- **`gimbal_device_id`:** Component ID of the gimbal.
- **`pitch`:** Normalized pitch input (-1..1).
- **`yaw`:** Normalized yaw input (-1..1).
- **`pitch_rate`:** Normalized pitch rate input (-1..1).
- **`yaw_rate`:** Normalized yaw rate input (-1..1).
- **`zoom`:** Normalized zoom input (-1..1).

Theoretical Use Cases

1. **Standardized Joystick Mapping:**
   - *Scenario:* A GCS wants to support a generic USB gamepad for gimbal control.
   - *Action:* Instead of needing to know specific RC channel mappings (e.g., "Channel 6 is tilt"), the GCS simply sends `GIMBAL_MANAGER_SET_MANUAL_CONTROL` with `pitch = joystick.y`. This abstracts the radio configuration from the user interface.
2. **Smooth Zooming:**
   - *Scenario:* A variable-speed zoom rocker on a smart controller.
   - *Action:* The controller sends `zoom = 0.5` (half speed in) or `zoom = -1.0` (full speed out), allowing precise lens control without needing complex rate calculations.

UAVCAN_NODE_STATUS (ID 310)

UNSUPPORTED

Summary

The `UAVCAN_NODE_STATUS` message is designed to bridge the status of devices on the UAVCAN (now DroneCAN) bus onto the MAVLink network. It reports the health, uptime, and mode of individual CAN nodes.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

While ArduPilot has robust support for DroneCAN (the protocol formerly known as UAVCAN v0) and manages many peripherals (ESCs, GPS, Compass) via CAN, it does not act as a transparent bridge reflecting the raw status of every CAN node onto MAVLink using this specific message.

Instead, ArduPilot abstracts the peripherals. For example, a CAN GPS is reported via `GPS_RAW_INT` and `GPS_STATUS`, and a CAN ESC is reported via `ESC_TELEMETRY`, rather than as a generic "UAVCAN Node."

Intended Data Fields (Standard)

- **`time_boot_ms`:** Timestamp (ms since boot).
- **`time_usec_1`:** Timestamp (us since UNIX epoch).
- **`uptime_sec`:** Time since the node started.
- **`health`:** Node health (OK, Warning, Error, Critical).
- **`mode`:** Node mode (Operational, Initialization, Maintenance, etc.).
- **`sub_mode`:** Sub-mode.
- **`vendor_specific_status_code`:** Vendor specific status.

Theoretical Use Cases

1. **Unified Health Monitor:**
   - *Scenario:* A vehicle has 20 CAN peripherals (lights, servos, sensors).
   - *Action:* The GCS displays a "CAN Bus Health" table listing every node ID, its uptime, and its health status. This allows a technician to instantly spot a device that is rebooting frequently or in an error state, even if it doesn't have a dedicated MAVLink message.
2. **Network Topology Mapping:**
   - *Scenario:* Diagnosing a bus wiring fault.
   - *Action:* By monitoring which nodes fall offline (stop sending `UAVCAN_NODE_STATUS`) when a specific cable is wiggled, ground crew can isolate physical connection issues.

UAVCAN_NODE_INFO (ID 311)

UNSUPPORTED

Summary

The `UAVCAN_NODE_INFO` message provides detailed static information about a UAVCAN (DroneCAN) node, such as its hardware version, software version, and unique ID.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

Similar to `UAVCAN_NODE_STATUS` (310), ArduPilot does not transparently bridge raw DroneCAN node information onto the MAVLink network using this message.

Intended Data Fields (Standard)

- `time_boot_ms`: Timestamp (ms since boot).
- `uptime_sec`: Time since the node started.
- `name`: Node name string.
- `hw_version_major` / `minor`: Hardware version.
- `sw_version_major` / `minor`: Software version.
- `sw_vcs_commit`: Version Control System commit ID.

Theoretical Use Cases

1. **Firmware Auditing:**
   - *Scenario:* A fleet manager needs to ensure all ESCs are running the latest safety-critical firmware.
   - *Action:* The GCS queries `UAVCAN_NODE_INFO` for all nodes. It compares the `sw_version` against a manifest and flags any ESCs running outdated code.
2. **Asset Tracking:**
   - *Scenario:* Tracking individual components for warranty purposes.
   - *Action:* The message contains the Unique ID (UUID) of the node. The GCS logs this UUID, allowing the operator to track the flight hours of a specific servo or GPS unit across multiple airframes.

LOGGING

LOGGING_DATA (ID 266)

UNSUPPORTED

Summary

The `LOGGING_DATA` message is part of a protocol for streaming log data from the vehicle to a Ground Control Station (GCS) in real-time or as a robust download mechanism. It works in conjunction with `LOGGING_DATA_ACKED` (267) to ensure data integrity.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

For downloading DataFlash logs (`.BIN` files), ArduPilot uses the standard MAVLink log download protocol involving:

- **`LOG_REQUEST_LIST` (117)**
- **`LOG_ENTRY` (118)**
- **`LOG_REQUEST_DATA` (119)**
- **`LOG_DATA` (120)**

The `LOGGING_DATA` (266) message appears to be an alternative or streaming-focused mechanism that is not currently part of the ArduPilot codebase.

Intended Data Fields (Standard)

- **`target_system`:** System ID of the target.
- **`target_component`:** Component ID of the target.
- **`sequence`:** Sequence number (can wrap).
- **`length`:** Data length (bytes).
- **`first_message_offset`:** Offset into data where the first message starts.
- **`data`:** Logged data (buffer).

Alternative

For log downloads in ArduPilot, refer to **`LOG_DATA` (120)**.

Theoretical Use Cases

1. **Black Box Streaming:**
   - *Scenario:* A high-value experimental aircraft.
   - *Action:* The vehicle streams critical log data in real-time via `LOGGING_DATA`. If the vehicle is lost or destroyed, the last seconds of telemetry are preserved on the GCS.
2. **Reliable Download:**
   - *Scenario:* Downloading logs over a lossy telemetry link.

    - *Action:* The protocol uses the sequence numbers and ACKs to re-transmit only the lost packets, ensuring a bit-perfect copy of the log file eventually arrives.

LOGGING_DATA_ACKED (ID 267)

UNSUPPORTED

Summary

The `LOGGING_DATA_ACKED` message is the acknowledged variant of the `LOGGING_DATA` (266) streaming protocol. It sends log data that requires an explicit `LOGGING_ACK` (268) from the receiver, ensuring critical data is not lost during transmission.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

It is part of the same unsupported log streaming protocol as `LOGGING_DATA` (266). ArduPilot relies on the standard `LOG_DATA` (120) message for log downloads.

Intended Data Fields (Standard)

- `target_system`: System ID of the target.
- `target_component`: Component ID of the target.
- `sequence`: Sequence number.
- `length`: Data length.
- `first_message_offset`: Offset to start of first message.
- `data`: Logged data.

Alternative

For log downloads in ArduPilot, refer to `LOG_DATA` **(120)**.

Theoretical Use Cases

1. **Guaranteed Delivery:**
    - *Scenario:* Sending a critical "Crash Report" packet.
    - *Action:* The vehicle sends the crash dump via `LOGGING_DATA_ACKED` and re-transmits it until the GCS confirms receipt with `LOGGING_ACK`, ensuring the data is not lost in radio noise.

`LOGGING_ACK` (ID 268)

UNSUPPORTED

Summary

The `LOGGING_ACK` message is used to acknowledge the receipt of a `LOGGING_DATA_ACKED` (267) packet. This closes the loop for the reliable log streaming protocol.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

It is part of the same unsupported log streaming protocol as `LOGGING_DATA` (266) and `LOGGING_DATA_ACKED` (267).

Intended Data Fields (Standard)

- **`target_system`:** System ID of the target.
- **`target_component`:** Component ID of the target.
- **`sequence`:** Sequence number (must match the one in `LOGGING_DATA_ACKED`).

Theoretical Use Cases

1. **Flow Control:**
    - *Scenario:* The GCS is overwhelmed by incoming log data.
    - *Action:* The GCS delays sending `LOGGING_ACK`, implicitly telling the vehicle to pause transmission until the buffer clears.

PARAMETERS

PARAM_MAP_RC (ID 50)

UNSUPPORTED

Summary

Bind a parameter to an RC channel.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. Parameter tuning via RC channels is handled internally via the `TUNE` parameter and `RCx_OPTION` settings, not via this MAVLink message.

Theoretical Use Cases

Dynamic parameter tuning via RC.

PARAM_EXT_REQUEST_READ (ID 320)

UNSUPPORTED

Summary

The `PARAM_EXT_REQUEST_READ` message is part of the **MAVLink Extended Parameter Protocol**. This protocol was designed to overcome limitations of the original parameter protocol (e.g., small index size, lack of type safety for strings/arrays).

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot relies on the robust and widely supported standard parameter protocol (`PARAM_REQUEST_READ`, `PARAM_VALUE`). For large datasets or complex configuration, it utilizes the MAVLink FTP (`FILE_TRANSFER_PROTOCOL`) to transfer parameter files or logs, rather than the intermediate Extended Parameter Protocol.

Intended Data Fields (Standard)

- **`target_system` / `target_component`:** Targeted component.
- **`param_id`:** Parameter id, terminated by NULL if the length is less than 16 human-readable chars.
- **`param_index`:** Parameter index. Set to -1 to use the `param_id`.

Theoretical Use Cases

1. **Exceeding 65k Parameters:**
   - *Scenario:* A massive system with more than 65,535 parameters (the limit of the standard protocol's `int16` index).
   - *Action:* The Extended Protocol allows for larger indices, enabling access to the full parameter set.
2. **Long String Parameters:**
   - *Scenario:* Storing a long URL or API key as a parameter.
   - *Action:* The standard protocol is limited to 4 bytes (float/int32). The Extended Protocol supports variable-length types, allowing full strings to be read natively.

PARAM_EXT_REQUEST_LIST (ID 321) UNSUPPORTED

Summary

The `PARAM_EXT_REQUEST_LIST` message is used to request a full list of parameters from a component using the **MAVLink Extended Parameter Protocol**.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot uses `PARAM_REQUEST_LIST` (21) for listing parameters.

Intended Data Fields (Standard)

- `target_system` **/** `target_component`**:** Targeted component.

Theoretical Use Cases

1. **High-Latency Links:**
    - *Scenario:* Syncing parameters over a very slow satellite link.
    - *Action:* The Extended Protocol can be more efficient for specific data types, potentially reducing the overhead compared to the standard protocol's float conversions.
2. **Type Safety:**
    - *Scenario:* A GCS wants to ensure it never misinterprets a bitmask as a float.
    - *Action:* The Extended Protocol explicitly carries type information for every parameter, eliminating ambiguity during the sync process.

PARAM_EXT_VALUE (ID 322)

UNSUPPORTED

Summary

The `PARAM_EXT_VALUE` message is the response to a `PARAM_EXT_REQUEST_READ` or `PARAM_EXT_REQUEST_LIST` request. It contains the value of a parameter in the Extended Parameter Protocol format.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot uses `PARAM_VALUE` (22) for parameter values.

Intended Data Fields (Standard)

- **`param_id`:** Parameter id.
- **`param_value`:** Parameter value (up to 128 bytes).
- **`param_type`:** Parameter type (`MAV_PARAM_EXT_TYPE`).
- **`param_count`:** Total number of parameters.
- **`param_index`:** Index of this parameter.

Theoretical Use Cases

1. **Transferring Structs:**
   - *Scenario:* Configuring a complex fence boundary defined by a struct.
   - *Action:* The 128-byte payload of `PARAM_EXT_VALUE` allows transmitting small structures or arrays as a single atomic parameter update, rather than breaking them into 32 separate float parameters.
2. **64-bit Integers:**
   - *Scenario:* Storing a precise 64-bit timestamp or large integer counter.
   - *Action:* The standard protocol truncates to 32-bit floats (lossy). `PARAM_EXT_VALUE` supports `uint64` natively.

PARAMETER `PARAM_EXT_SET` (ID 323) UNSUPPORTED

Summary

The `PARAM_EXT_SET` message is used to set the value of a parameter using the **MAVLink Extended Parameter Protocol**.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot uses `PARAM_SET` (23) for setting parameter values.

Intended Data Fields (Standard)

- **`target_system` / `target_component`:** Targeted component.
- **`param_id`:** Parameter id.
- **`param_value`:** Parameter value (up to 128 bytes).
- **`param_type`:** Parameter type (`MAV_PARAM_EXT_TYPE`).

Theoretical Use Cases

1. **Atomic Configuration:**
   - *Scenario:* Setting an IP address.
   - *Action:* Instead of setting 4 separate octet parameters, the GCS sends one `PARAM_EXT_SET` with the full string "192.168.1.50", ensuring the IP doesn't become invalid in the middle of the update process.
2. **64-bit Precision:**
   - *Scenario:* Setting a precise UTC time offset.
   - *Action:* Sending a `int64` value directly ensures no precision is lost to floating-point representation.

PARAMET_EXT_ACK (ID 324) UNSUPPORTED

Summary

The `PARAM_EXT_ACK` message is a response to a `PARAM_EXT_SET` command in the **MAVLink Extended Parameter Protocol**, indicating success or failure.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot **does not** implement this message.

ArduPilot uses the standard `PARAM_VALUE` broadcast as an implicit acknowledgement of a `PARAM_SET`.

Intended Data Fields (Standard)

- **`param_id`:** Parameter id.
- **`param_value`:** Parameter value.
- **`param_type`:** Parameter type.
- **`param_result`:** Result code (`PARAM_ACK_ACCEPTED`, `PARAM_ACK_VALUE_UNSUPPORTED`, etc.).

Theoretical Use Cases

1. **Explicit Error Handling:**
   - *Scenario:* A user tries to set a parameter to an invalid value.
   - *Action:* The standard protocol just ignores it (or maybe sends a `STATUSTEXT`). `PARAM_EXT_ACK` allows the drone to reply with `PARAM_ACK_VALUE_UNSUPPORTED`, giving the GCS immediate, programmatic feedback that the write failed.
2. **Transaction Confirmation:**
   - *Scenario:* Automated configuration script.
   - *Action:* The script waits for `PARAM_EXT_ACK` with `PARAM_ACK_ACCEPTED` before proceeding to the next step, ensuring robust configuration sequence.

SIMULATION

HIL_STATE (ID 90)

UNSUPPORTED

Summary

Hardware in the loop state.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not support this message. It uses `HIL_STATE_QUATERNION` (115) for HIL simulation to avoid gimbal lock singularities associated with Euler angles.

Theoretical Use Cases

Legacy HIL simulation.

HIL_CONTROLS (ID 91)

UNSUPPORTED

Summary

Hardware in the loop control outputs.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Sending control surface outputs to a simulator.

HIL_RC_INPUTS_RAW (ID 92) UNSUPPORTED

Summary

Raw RC inputs for HIL.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Injecting RC input into a simulation.

HIL_ACTUATOR_CONTROLS (ID 93) UNSUPPORTED

Summary

HIL actuator controls.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message.

Theoretical Use Cases

Sending actuator commands to a simulator.

HIL_SENSOR (ID 107)

DEPRECATED / UNSUPPORTED

Summary

The `HIL_SENSOR` message was designed to allow external simulators to provide raw sensor data (IMU, Barometer, Magnetometer) to the flight controller during Hardware-In-The-Loop (HIL) simulation. **ArduPilot no longer supports this message.** It has been deprecated and removed from the core libraries in favor of the more robust and higher-performance Software-In-The-Loop (SITL) architecture.

Status

Deprecated / Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

ArduPilot has transitioned away from MAVLink-based HIL for sensors. Historically, messages like `HIL_SENSOR` were used with simulators like X-Plane or older versions of Gazebo. Modern ArduPilot workflows use SITL, where sensor data is injected via specialized backend classes (e.g., `AP_InertialSensor_SITL`) that communicate with simulators using high-bandwidth JSON/UDP protocols or shared memory, rather than the bandwidth-limited MAVLink stream.

- **Absence:** There is no mapping for `MAVLINK_MSG_ID_HIL_SENSOR` (107) in the `GCS_MAVLink` libraries.
- **Parameters:** Legacy parameters like `HIL_MODE` are now marked as `unused` in the firmware source code.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `xacc`: X acceleration (m/s^2).
- `yacc`: Y acceleration (m/s^2).
- `zacc`: Z acceleration (m/s^2).
- `xgyro`: Angular speed around X axis (rad/s).
- `ygyro`: Angular speed around Y axis (rad/s).
- `zgyro`: Angular speed around Z axis (rad/s).
- `xmag`: X Magnetic field (Gauss).
- `ymag`: Y Magnetic field (Gauss).
- `zmag`: Z Magnetic field (Gauss).
- `abs_pressure`: Absolute pressure (hectopascal).
- `diff_pressure`: Differential pressure (hectopascal).
- `pressure_alt`: Altitude calculated from pressure.
- `temperature`: Temperature (degrees celsius).
- `fields_updated`: Bitmap for fields that have updated since last message, bit 0 = xacc, bit 12: temperature.

Recommendation

Developers looking to perform simulations should use the **SITL** system. If external sensor injection is required on physical hardware, consider using the `VISION_POSITION_ESTIMATE` (102) or `GPS_INPUT` (232) pipelines, which remain active and supported.

Key Codebase Locations

- **ArduPlane/Parameters.h**: Shows `HIL_MODE` is unused.
- **libraries/AP_HAL/AP_HAL_Boards.h**: Explicitly marks HIL sensor constants as `HAL_INS_HIL_UNUSED`.

HIL_OPTICAL_FLOW (ID 114)

UNSUPPORTED

Summary

Simulated optical flow sensor data.

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None

Description

ArduPilot does not implement this message. It likely uses standard `OPTICAL_FLOW` (100) or direct sensor injection for simulation.

Theoretical Use Cases

Simulating optical flow.

HIL_STATE_QUATERNION (ID 115) UNSUPPORTED

Summary

The `HIL_STATE_QUATERNION` message is defined in MAVLink to provide the full ground-truth state of a vehicle (position, attitude, velocity, and acceleration) from a simulator to the flight controller. **ArduPilot does not implement this message.**

Status

Unsupported

Directionality

- **TX (Transmit):** None
- **RX (Receive):** None (Ignored)

Analysis

Like `HIL_SENSOR` (107), this message is part of the legacy MAVLink-based Hardware-In-The-Loop (HIL) architecture. ArduPilot has migrated its simulation strategy to the Software-In-The-Loop (SITL) system, which uses higher-performance, non-MAVLink protocols for state injection.

- **Absence:** There is no handler for `MAVLINK_MSG_ID_HIL_STATE_QUATERNION` (115) in any of ArduPilot's GCS or state estimation libraries.
- **Alternative:** Developers using ArduPilot's SITL should use the built-in simulator drivers which handle state synchronization automatically without using this MAVLink packet.

Data Fields

- `time_usec`: Timestamp (microseconds since UNIX epoch or microseconds since system boot).
- `attitude_quaternion`: Vehicle attitude expressed as normalized quaternion (w, x, y, z).
- `rollspeed`: Body frame roll / phi angular speed (rad/s).
- `pitchspeed`: Body frame pitch / theta angular speed (rad/s).
- `yawspeed`: Body frame yaw / psi angular speed (rad/s).
- `lat`: Latitude (degE7).
- `lon`: Longitude (degE7).
- `alt`: Altitude (meters).
- `vx`: Ground X Speed (Latitude) (cm/s).
- `vy`: Ground Y Speed (Longitude) (cm/s).
- `vz`: Ground Z Speed (Altitude) (cm/s).
- `ind_airspeed`: Indicated airspeed (cm/s).
- `true_airspeed`: True airspeed (cm/s).
- `xacc`: X acceleration (mG).
- `yacc`: Y acceleration (mG).
- `zacc`: Z acceleration (mG).

Key Codebase Locations

- **libraries/GCS_MAVLink/GCS_Common.cpp**: Lacks a handler for ID 115.