Robotics and Autonomous Systems - Air (RAS-A) MAVLink Control Link Interoperability Profile (IOP)

Release Version 1.2

RAS-A MAVLink Control Link Interoperability Profile (IOP) for development of the Department of Defense (DOD) small unmanned aircraft systems (sUAS) Joint Reference Architecture (JRA)

19 MAY 2023

Distribution A: Approved for Public Release

Document

This document describes the interoperability profile for air vehicles and is approved for public release.

Standards Release Timeline

Phase	Dates	Standards Version
V 1 DRAFT	2021-07-30	Initial working group draft
V 1 Review	2021-08-02 - 2021-08-27	Initial draft review
V 1 Release	2021-09-28	1.0
V 1.1 DRAFT	2022-06-24	1.1 initial draft
V 1.1 Review	2022-06-24 - 2022-08-04	1.1 draft review
V 1.1 Release	2022-08-05	1.1
V 1.2 DRAFT	2023-03-24	1.2 initial draft
V 1.2	2023-05-19	1.2

Document Revision History

Version	Date	Changes	DoD Reviewer	Industry Editor
D0.1	2021-07-30	Initial draft	Capt Matthew Borowski, DIU	Lorenz Meier
D0.2	2021-09-10	Draft release v1.0	Maj Matthew Borowski, DIU	Lorenz Meier
1.0	2021-09-28	Final First Version	Maj Matthew Borowski, DIU	Lorenz Meier
1.1 DRAFT	2022-06-24	Draft release v1.1	Maj Matthew Borowski, DIU	Nuno Marques
1.1 DRAFT (reviewed)	2022-08-04	v1.1 draft (reviewed)	Maj Matthew Borowski, DIU	Nuno Marques
1.1	2022-08-05	Release v1.1 Changelog	Maj Matthew Borowski, DIU	Nuno Marques
1.2 DRAFT	2023-03-24	Draft release v1.2	Maj Matthew Borowski, DIU	RAS-A Consortium
1.2	2023-05-19	Release v1.2 Changelog	Maj Matthew Borowski, DIU	RAS-A Consortium

Document Approval

Version	Date	DoD Approver	DoD Approver Title	Dod Approver Signature
1.0	2021-09-28	Maj Matthew Borowski, DIU	Program Manager	« Signed »
1.1	2022-08-25	Maj Matthew Borowski, DIU	Program Manager	« Signed »
1.2	2023-06-02	Maj Matthew Borowski, DIU	Program Manager	BOROWSKI.MATTHEW.FRANCIS.1266351166 Digitally signed by BOROWSKI.MATTHEW.FRANCIS.1266351166 Date: 2023.06.02 12:51:46 -04'00'

Governance

The RAS-A MAVLink Control Link IOP is managed by the joint services and built on top of the MAVLink industry standard. The MAVLink standard is governed by the Dronecode Foundation.

- The RAS-A MAVLink Control Link IOP Release 1.0 is based on the MAVLink upstream message definitions from 30 JUL 21.
- Release 1.1 is based on MAVLink upstream *common* dialect and *ras_a* dialect message definitions from 21 JUL 2022.
- Release 1.2 is based on MAVLink upstream *common* dialect and *ras_a* dialect message definitions from 19 MAY 2023. The latest MAVLink technical message definitions as XML files for code generation of this interoperability profile is hosted at this link: XML definition.

This IOP document is self-contained and includes all documentation to implement the standard. It is managed by the Joint Program Office. External links are for reference only. Please refer to this PDF and for message sets and code generation to the released XML file.

Message Set Revision History

Version	Date	Technical message set identifier (GIT hash)
D0.2	2021-07-30	B030a760e8c350aa078542036bd9e8f39a494ac1 (link)
1.0	2021-07-30	B030a760e8c350aa078542036bd9e8f39a494ac1 (link)
1.1	2022-07-21	1a755ab85813215f8c002858ff3fe2f5ba652f99 (link)
1.2 DRAFT	2023-03-24	af284e17b62d07bc087ddab7ca35fb7586b01dba (link)
1.2	2023-05-19	67e4ebc74144246f292b8229798d7e4f7360ecd8 (link)

Contents

Scope

Purpose

The document provides a full specification to enable interoperability between ground control stations and different air vehicles, including models from different manufacturers and different types (rotary wing, fixed-wing, VTOL). The capabilities covered by this profile include manual control of the vehicle, autonomous missions (waypoints), payload control (cameras, cargo, etc), vehicle specific skills, radio link configuration and pairing, vehicle setup and configuration and maintenance.

As air vehicles operate in 3D space the underlying MAVLink protocol has been shown to also operate successfully in more constrained 2D environments like unmanned ground vehicles and unmanned surface vehicles.

Document Overview

This document provides detailed concepts, flow charts and descriptions for the packet layer, the MAVLink microservices concept and authentication / signing. The key services that are used throughout different profiles are the Command microservice, Parameter microservice and the Mission transfer protocol.

Higher-level microservices are built on top of these communication services. These include services like manual control through stick inputs or commands, camera controls and uploading missions and geofences.

MAVLink allows for customization / extension through so-called dialects, sets of messages that are supported in addition to the baseline message set. RAS-A has its own dialect built on top of the development dialect, which also extends the common set of messages. The use of the `ras_a` dialect allows to easily extend the IOP with the required messages that cannot be found in the common MAVLink set. Consequently, this will facilitate the transition of the RAS-A set to an upstream dialect, like `common`.

This document further covers the supported vehicle system modes and the arming sequence.

Source Documents

Government Documents

Table 5: Government Documents

ID	Version	Document
IOP- 4.0 JAUSProfilingRules-V4	4.0	RAS-G IOP JAUS Profiling Rules
IOP-OverarchingProfile- 4.0 V4	4.0	RAS-G IOP Overarching Profile
IOP- CommunicationsProfile-V4	4.0	RAS-G IOP Communications Profile

Non Government Documents

Table 6: Non Government Documents

ID	Version	Document
MAVLink standard	2.x	http://mavlink.io/en/

Joint Reference Architecture

The RAS-A MAVLink Control Link Interoperability Profile (IOP) shall be used to achieve interoperability between different vendors, leveraging the leading industry standard. The IOPs are separated, however, a minimum set is required for the typical operational scenario of a government asset. The required services are marked in the table below. All other services are supported by the common ground control station, but are not mandatory on the vehicle side. Options below can be either required (r), or optional (o).

These categories are defined as follows:

- If the system is implementing any similar functionality (for example has a camera), it has to provide the specified interface and version as the default option
- If flagged as required, then the interface has to be present at all times
- If flagged as not required, then the interface can be omitted if the vehicle does not have any such capability.

What is explicitly not permitted is to omit an optional interface despite the vehicle having such capability - e.g. if the vehicle has a camera, then the specified interface has to be provided. This is needed as the core objective of this standard is to enable the integration of different systems into a common control interface and hence a single approach to every category of functionality is required.

Table 7: Required Services on Vehicle Side for Reference Architecture

Task	MAVLink service	Purpose / usage	Req.
Establishing baseline communica-tion	Heartbeat/Connection protocol and initial handshaking (AUTOPILOT_VERSION and PROTOCOL_VERSION messages)	Discovery and identification of systems on the network, determine protocol version and vehicle capabilities	r
Datalink pairing	Pairing protocol (v1 accepted as per IOP v1.1. v2 to be supported and required in the IOP v2)	Connect and setup datalinks, and configure encryption.	o
Get telemetry and status information from the drone	Battery protocol and discriminated messages on the Telemetry section of this IOP	Obtain information of MAVLink components and/or vehicle status	r
Control of the drone by the operator	Manual Control protocol/microservice	Input coming from a physical interface to the operator (i.e. joystick). Direct manual control of the vehicle	r
Control of the drone by the operator	Command protocol/microservice	Provides interface to an operator (i.e. typically from a UI). Note that the list of required and optional commands are listed in this document	r
Control of a camera on the drone	Camera protocol/microservice	Interacting with camera such as starting / stopping video recording, change brightness or zoom, downloading images if the camera provides this functionality	o
Control of a gimbal on the drone	Gimbal (v2) protocol/microservice (v1.1 allowed as per IOP v1.1, but to be deprecated on IOP v1.2)	Orienting the camera towards specific angles or locations on the ground	o

Task	MAVLink service	Purpose / usage	Req.
Uploading and executing an autonomous mission	Mission protocol/microservice	Uploading waypoints and execute pre-planned or on-the-fly updated complex missions	r
Generic payload control	Command protocol/microservice	Control generic payloads as well as special mission payloads (like cargo). Note that the list of required and optional commands are listed in this document	o
Terrain following	Terrain Protocol	Enabling the vehicle to stay close to known terrain during a mission	o
File transfer (camera definitions or log files)	(MAVLink) File Transfer Protocol	Enables file transfer over MAVLink	o
Configure drone parameters	Parameter microservice/protocol	Exchange configuration settings between MAVLink components, including configuring the vehicle from the common GCS	r
Autonomous behaviors: Exploration	Exploration protocol/microservice	Controls autonomous exploration capability of a vehicle	o

Introduction to MAVLink

MAVLink is a very lightweight messaging protocol for communicating with drones (and between onboard drone components).

MAVLink follows a modern hybrid publish-subscribe and point-to-point design pattern: Data streams are sent / published as **topics** while configuration sub-protocols such as the Mission Protocol or Parameter Protocol are point-to-point with retransmission.

Messages are defined within XML files. Each XML file defines the message set supported by a particular MAVLink system, also referred to as a “dialect”. The reference message set that is implemented by *most* ground control stations and autopilots is defined in `common.xml` (most dialects *build on top of* this definition, including the `ras_a.xml` dialect itself).

Code generators create software libraries for specific programming languages from these XML message definitions, which can then be used by drones, ground control stations, and other MAVLink systems to communicate. The generated libraries are typically MIT-licensed, and can therefore be *used* without limits in any closed-source application without publishing the source code of the closed-source application.

Key Features

- Very efficient. MAVLink 1 has just 8 bytes overhead per packet, including start sign and packet drop detection. MAVLink 2 has just 14 bytes of overhead (but is a much more secure and extensible protocol). Because MAVLink doesn't require any additional framing it is very well suited for applications with very limited communication bandwidth.
- Very reliable. MAVLink has been used since 2009 to communicate between many different vehicles, ground stations (and other nodes) over varied and challenging communication channels (high latency/noise). It provides methods for detecting packet drops, corruption, and for packet authentication.
- Many different programming languages can be used, running on numerous micro-controllers/operating systems (including ARM7, ATMega, dsPic, STM32 and Windows, Linux, MacOS, Android and iOS).
- Allows up to 255 concurrent systems on the network (vehicles, ground stations, etc.)
- Enables both off-board and onboard communications (e.g. between a GCS and drone, and between drone autopilot and MAVLink enabled drone camera).

Determining Protocol/Message Version

A library's MAVLink support can be determined in a number of ways:

- `AUTOPILOT_VERSION.capabilities` can be checked against the `MAV_PROTOCOL_CAPABILITY_MAVLINK2` flag to verify MAVLink 2 support.

- `PROTOCOL_VERSION.version` contains the MAVLink version number multiplied by 100: v1.0 is 100, v2.3 is 203 etc.

- `HEARTBEAT.mavlink_version` field contains the minor version number. This is the `<version>` field defined in the Message Definitions (version in common.xml for dialects that depend on the common message set).

- The major version can be determined from the packet start marker byte:
  - MAVLink 1: 0xFE
  - MAVLink 2: 0xFD
- A MAVLink library that does not support a protocol version will not recognize the protocol start marker; so no messages will even be detected (see Serialization).
- While messages do not contain version information, an extra CRC is used to ensure that a library will only process compatible messages (see Serialization > CRC_EXTRA).

Note: MAVLink version 2 is the required supported version.

Version Handshaking

Support for *MAVLink 2* is indicated in the `AUTOPILOT_VERSION` message by the `MAV_PROTOCOL_CAPABILITY_MAVLINK2` flag.

This is sufficient if the communication link between autopilot and GCS is completely transparent. However, most communication links are not completely transparent as they either include routing or in case of fixed-length wireless implementations on *packetization*. In order to also test the link, the *MAVLink 2* handshake protocol sends a *MAVLink 2* frame to test the complete communication chain.

To do so, the GCS sends a `COMMAND_LONG` or `COMMAND_INT` message with the command ID `MAV_CMD_REQUEST_PROTOCOL_VERSION`.

If the system supports *MAVLink 2* and the handshake it will respond with `PROTOCOL_VERSION` **encoded as MAVLink 2 packet**. If it does not support *MAVLink 2* it should NACK the command. The GCS should fall back to a timeout in case the command interface is not implemented properly.

The diagram below illustrates the complete sequence.

Note: Both AUTOPILOT_VERSION and PROTOCOL_VERSION are mandatory to be sent on request through MAV_CMD_REQUEST_MESSAGE.

Semi-Transparent Legacy Radios

Some popular legacy radios (e.g. the SiK radio series) operate in semi-transparent mode by injecting `RADIO_STATUS` messages into the MAVLink message stream. Per MAVLink spec these should actually emit a heartbeat with a different component ID than the autopilot to be discoverable. However, an additional heartbeat could be an issue for deployed systems. Therefore these radios can alternatively confirm their *MAVLink 2* compliance by emitting `RADIO_STATUS` in v2 message format after receiving the first MAVLink v2 frame.

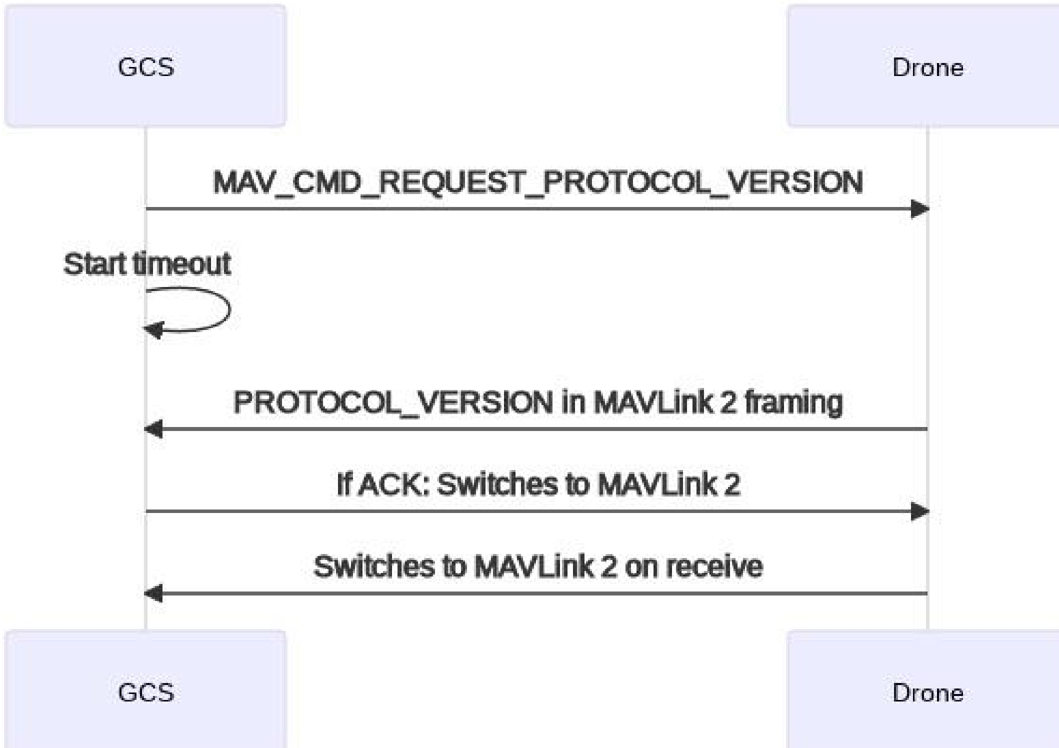

Figure 1: Mermaid sequence: Request protocol version

Capabilities

Vehicle capabilities should be reported through the `capabilities` field in the `AUTOPILOT_VERSION`, using the `MAV_PROTOCOL_CAPABILITY_MAVLINK2` flag. Reporting capabilities allows the GCS to adjust the configurations and UI according to what the vehicle is capable of executing.

The following capabilities are mandatory to exist and should be reported:

- `MAV_PROTOCOL_CAPABILITY_MISSION_FLOAT`
- `MAV_PROTOCOL_CAPABILITY_MISSION_INT`
- `MAV_PROTOCOL_CAPABILITY_COMMAND_INT`
- `MAV_PROTOCOL_CAPABILITY_PARAM_ENCODE_BYTEWISE`
- `MAV_PROTOCOL_CAPABILITY_FLIGHT_TERMINATION`
- `MAV_PROTOCOL_CAPABILITY_MAVLINK2`
- `MAV_PROTOCOL_CAPABILITY_PARAM_ENCODE_C_CAST`

Other capability flags made available in the RAS-A dialect are mandatory to be exposed when the vehicle/autopilot offers that same capability.

Note: Under this IOP, the current existing capabilities can be extended, and added under the `ras_a` dialect.

Versions and Signing

Packet signing is optional as encryption is already provided through pairing on the DoD-selected datalink. It can however provide an additional layer of operational safety.

Currently most MAVLink networks are configured to use unsigned MAVLink 2 messages.

Networking

RAS-A is expected to run on an IP network. RAS-A messages are encoded into UDP datagrams and sent over IP networks to their destination.

Specifically, a RAS-A capable device shall - Utilize Internet Protocol (IP) for routing and networking - Utilize a MAC address - Utilize an IPv4 IP address - Utilize ports - Utilize UDP to transport MAVLink v2 encoded messages

Routing

All bidirectional GCS to vehicle communication happens over the IP routing protocol. The system ID fields in the messages lose their meaning, as they are not used for routing. This does not make any assumptions about the physical layer.

Routing is done using the IP routing rules, all existing standard software and hardware components for IP, (routers, IP tables etc.) can be used to set up the network. The network has to be set up in a way, that participants that want to establish a communication, can reach the participants they want to establish a communication with by their IP. This is analogous to a server <> client model, where the client has to be able to reach the server by its IP address. In case the client sits behind a NAT, the server does not need to be able to reach the client, as the client initiates the communication.

In RAS-A, the concept a broadcast message (e.g. a telemetry) message is supposed to be sent to all active connections that a system have. This is in contrast to MAVLink, where the message is supposed to be sent as broadcast on the local network segment. In that sense, RAS-A does not use broadcast, but multi-unicast.

In RAS-A, each message that arrives at a certain participant is intended for that participant. No RAS-A participant shall route messages to other participants. This falls into the sole responsibility of the IP routing protocol.

Connections

A "RAS-A connection" is defined as two network participants who send each other `HEARTBEAT` (message ID 0) messages. `HEARTBEAT` is expected to be sent at a rate of 1Hz. Should one of the participant not receive 5 consecutive heartbeat messages, the connection is considered broken and needs to be re-established.

The RAS-A connection gets established after the RAS-A pairing process. The pairing process is the lower level network configuration process that assigns IP addresses and network configuration to the nodes. The RAS-A connection gets established, as soon as the lower level pairing protocol notifies the presence of devices.

A connection can be established from either the GCS or the vehicle. To establish a connection, the participant starts sending the heartbeat messages to well known port 14550 of the connection target. Each participant is expected to host a listening socket at well-known port 14550 and accept incoming heartbeat messages on that port as connection establishment.

Identification

A participant is reachable by its IP address and a port. Packets get routed by IP routing rules to the correct participant. All communication follows a client <> server model. The client establishes the communication by sending data to a well-known port of the server. The client may use any ephemeral port for this outgoing communication. The server will respond to this same port. In the case there is a NAT active in the network, the [IP, Port] tuple may only be a local identifier for the participant, but not a global identifier.

The RAS-A header is not modified, the system ID field is still present, filled out and transmitted. However, its value is no longer considered to hold any meaning. System ID collisions are expected to happen and are ignored.

Example: A GCS is connected to two drones at the same time. Both drones report system id 1, the GCS reports system id 254. From GCS perspective, the two drones have different (IP port) pairs by which the GCS identifies them. The fact that they have the same system id is ignored.

Packet Serialization

This topic provides detailed information about about MAVLink packet serialization, including the over-the-wire formats for MAVLink v1 and v2 packets, the ordering of fields in the message payload, and the CRC_EXTRA used for ensuring that the sender and receiver share a compatible message definition.

It is primarily intended for developers who are creating/maintaining a MAVLink generator

MAVLink users do not typically need to understand the serialization format, as encoding/decoding is handled by the MAVLink libraries.

Packet Format

This section shows the serialized message format of MAVLink packets (the format is inspired by the CAN and SAE AS-4 standards).

MAVLink 2 Packet Format

Below is the over-the-wire format for a MAVLink 2 packet (the in-memory representation might differ).

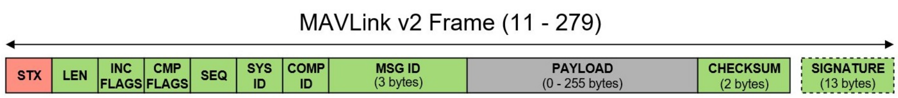

Figure 2: Over-the-wire MAVLink 2 Format Frame

Byte Index	C version	Content	Value	Explanation
0	uint8_t magic	Packet start marker	0xFD	Protocol-specific start-of-text (STX) marker used to indicate the beginning of a new packet. Any system that does not understand the protocol version will skip the packet.
1	uint8_t len	Payload length	0 - 255	Indicates length of the following payload section. This may be affected by payload truncation.
2	uint8_t incompat_flags	Incompatibility Flags		Flags that must be understood for MAVLink compatibility (implementation discards packet if it does not understand flag).

Byte Index	C version	Content	Value	Explanation
3	uint8_t compat_ flags	Compatibility Flags		Flags that can be ignored if not understood (implementation can still handle packet even if it does not understand flag).
4	uint8_t seq	Packet sequence number	0 - 255	Used to detect packet loss. Components increment value for each message sent.
5	uint8_t sysid	System ID (sender)	1 - 255	*Deprecated - Byte does not hold any meaning*
6	uint8_t compid	Component ID (sender)	1 - 255	ID of *component* sending the message. Unqiue values within a system used for routing and topological purposes. Historically used to differentiate *components* in a *system* (e.g. autopilot and a camera) using appropriate values in `MAV_COMPONENT`. Note that the broadcast address `MAV_COMP_ID_ALL` may not be used in this field as it is an invalid *source* address.
7 to 9	uint32_t msgid:24	Message ID (low, middle, high bytes)	0 - 16777215	ID of *message type* in payload. Used to decode data back into message object.
n-byte payload	`uint8_t payload [max 255]`	Payload		Message data. Depends on message type (i.e. Message ID) and contents. Byte index: n=0: NA, n=1: 10, n>=2: 10 to (9+n)

Byte Index	C version	Content	Value	Explanation
(n+10) to (n+11)	`uint16_t checksum`	Checksum (low byte, high byte)		CRC-16/MCRF4XX for message (excluding magic byte). Includes CRC_EXTRA byte.
(n+12) to (n+25)	`uint8_t signature[13]`	Signature		(Optional) Signature to ensure the link is tamper-proof.

Table 6: Over-the-wire MAVLink 2 Format

The minimum packet length is 12 bytes for acknowledgement packets without payload. And the maximum 280 bytes for a signed message that uses the whole payload.

Incompatibility Flags (MAVLink 2)

Incompatibility flags are used to indicate features that a MAVLink library must support in order to be able to handle the packet. This includes any feature that affects the packet format/ordering.

A MAVLink implementation must discard a packet if it does not understand any flag in the `incompat_flags` field.

Supported incompatibility flags include (at time of writing):

Table 9: Supported Incompatibility Flags

Flag	C flag	Feature
0x01	`MAVLINK_IFLAG_SIGNED`	The packet is signed (a signature has been appended to the packet).

Compatibility Flags (MAVLink 2)

Compatibility flags are used to indicate features that won't prevent a MAVLink library from handling the packet (even if the feature is not understood). This might include, for example, a flag to indicate that a packet should be treated as "high priority" (such a message could be handled by any MAVLink implementation because packet format and structure is not affected).

A MAVLink implementation can safely ignore flags it doesn't understand in the `compat_flags` field.

Payload Format

MAVLink does not include information about the message structure in the payload itself (in order to reduce overhead)! Instead the sender and receiver must share a common understanding of the meaning, order and size of message fields in the over-the-wire format.

Messages are encoded within the MAVLink packet:

- The `msgid` (message id) field identifies the specific message encoded in the packet.
- The payload field contains the message data.
  - MAVLink reorders the message fields in the payload for over-the-wire transmission (from the order in the original XML Message Definitions).
  - MAVLink 2 truncates any zero-filled bytes at the end of the payload before the message is sent and sets the packet len field appropriately (MAVLink 1 always sends all bytes).
- The len field contains the length of the payload data.
- A `CRC_EXTRA` byte is added to the message checksum. A receiver can use this to confirm that it is compatible with the payload message format/definition. A MAVLink library should notify a bad CRC during decoding if a message specification is incompatible (e.g. the C library mavlink_parse_char() gives a status `MAVLINK_FRAMING_BAD_CRC`).

Field Reordering

Message payload fields are reordered for transmission as follows:

- Fields are sorted according to their native data size:
  - (u)int64_t, double (8 bytes)
  - (u)int32_t, float (4)
  - (u)int16_t (2)
  - (u)int8_t, char (1)
- If two fields have the same length, their order is preserved as it was present before the data field size ordering
- Arrays are handled based on the data type they use, not based on the total array size
- The over-the-air order is the same as for the `struct` and thus represents the reordered fields
- The `CRC_EXTRA` field is calculated *after* the reordering, to ensure that a mistake during field reordering will be caught by a faulty CRC. The provided

Python, C and C# reference implementations are tested to have the correct field reordering, this is only a concern for custom implementations.

The only exception to the above reordering is for MAVLink 2 extension fields. Extension fields are sent in XML-declaration order and are not included in the `CRC_EXTRA` calculation. This allows new extension fields to be appended to the end of a message without breaking binary compatibility.

This ordering is unique and can be easily implemented in a protocol generator by using a stable sorting algorithm. The alternative to using sorting would be either to use inefficient alignment, which is bad for the target architectures for typical MAVLink applications, or to have function calls in the order of the variable size instead of the application context. This would lead to very confusing function signatures of serialization functions.

Empty-Byte Payload Truncation (MAVLink 2)

MAVLink 2 truncates any empty (zero-filled) bytes at the end of the serialized payload before it is sent. This contrasts with *MAVLink 1*, where bytes were sent for all fields regardless of content.

The actual fields affected/bytes saved depends on the message and its content (MAVLink field reordering means that all we can say is that any truncated fields will typically be those with the smallest data size, or extension fields).

The first byte of the payload is never truncated, even if the payload consists entirely of zeros.

The protocol only truncates empty bytes at the end of the serialized message payload; any null bytes/empty fields within the body of the payload are not affected.

CRC_EXTRA Calculation

The CRC_EXTRA CRC is used to verify that the sender and receiver have a shared understanding of the over-the-wire format of a particular message.

Changes in message specifications that might make the over-the-wire format incompatible include: new/removed fields, or changes to field name, data type, order, or array length.

When the MAVLink code generator runs, it takes a checksum of the XML structure for each message and creates an array defining `MAVLINK_MESSAGE_CRCS`. This is used to initialize the `mavlink_message_crcs[ ]` array in the C/C++ implementation, and is similarly used in the Python (or any other, such as the C# and JavaScript) implementation.

When the sender calculates the checksum for a message it adds the `CRC_EXTRA` byte onto the end of the data that the checksum is calculated over. The recipient calculates a checksum for the received message and adds its own `CRC_EXTRA` for the particular message id. If the `CRC_EXTRA` for the sender and receiver are different the checksums will not match.

This approach ensures that only messages where the sender and recipient are using the same message structure will be decoded (or at least it makes a mistake much more unlikely, as for any checksum application).

If you are doing your own implementation of MAVLink you can get this checksum in one of two ways: you can include the generated headers and use `MAVLINK_MESSAGE_CRCS` to get the right seed for each message type, or you can re-implement the code that calculates the seed.

As MAVLink internally reorders the message fields according to their size to prevent word / half-word alignment issues (see data structure alignment (Wikipedia) for further reference), and a wrongly implemented reordering potentially can cause inconsistencies as well, the `CRC_EXTRA` is calculated based on the over-the-air message layout rather than the XML order.

MAVLink 2 extension fields are not included in the CRC_EXTRA calculation.

This is the Python code that calculates the CRC_EXTRA seed:

```python
def message_checksum(msg):
    '''calculate a 8-bit checksum of the key fields of a
       message, so we can detect incompatible XML changes'''
    from .mavcrc import x25crc
    crc = x25crc()
    crc.accumulate_str(msg.name + ' ')
    # in order to allow for extensions the crc does not include
    # any field extensions
    crc_end = msg.base_fields()
    for i in range(crc_end):
        f = msg.ordered_fields[i]
        crc.accumulate_str(f.type + ' ')
        crc.accumulate_str(f.name + ' ')
        if f.array_length:
            crc.accumulate([f.array_length])
    return (crc.crc&0xFF) ^ (crc.crc>>8)
```

Graph 4: Python Code for `CRC_EXTRA` Calculation

This uses the same CRC-16/MCRF4XX checksum that is used at runtime. It calculates a CRC over the message name (such as “RAW_IMU”) followed by the type and name of each field, space separated. The order of the fields is the order they are sent over the wire. For arrays, the array length is also added.

Checksum

The packet format includes a 2-byte CRC-16/MCRF4XX to allow detection of message corruption. See the MAVLink source code for the documented C-implementation.

The CRC covers the whole message, excluding magic byte and the signature (if present). The CRC includes the `CRC_EXTRA` byte, which is used to ensure that the sending and receiving systems share a common understanding of the message definition.

Signing / Authentication

MAVLink 2 adds support for message signing, which allows a MAVLink system to verify that messages originate from a trusted source. However, some DoD datalinks are already complying with the pairing protocol that already includes and enables AES256 encryption, so signing and authentication is optional under this IOP.

This topic provides a general overview of message signing, which will be useful both for developers using existing MAVLink libraries and for writers of new MAVLink code generators. It explains how a system can determine if a message is signed and whether the signature is valid, how to allow unsigned messages to be accepted, and how to create and share the *secret* used to create the signature.

More detailed information for developers using existing MAVLink libraries can be found here:

- C Message Signing (mavgen)
- Pymavlink Message Signing (mavgen)

Frame Format

For a signed packet the **0x01** bit of the incompatibility flag field is set true and an additional 13 bytes of "signature" data appended to the packet. The signed packet format is shown below.

Figure 3: MAVLink 2 Signed Packet Format

The incompatibility flags in the packet header are used to indicate that the MAVLink library must reject the packet if it does not understand or cannot handle the flag. In other words, a MAVLink library that does not support signing must drop signed packets. The C library uses `MAVLINK_IFLAG_SIGNED` to represent the “supports message signing” bit.

The 13 bytes of the signature are:

Table 10: 13 Bytes Signature Description

Data	Description
linkID (8 bits)	ID of link on which packet is sent. Normally this is the same as the *channel*.
timestamp (48 bits)	Timestamp in 10 microsecond units since 1st January 2015 GMT time. This *must* monotonically increase for every message on a particular link. Note that means the timestamp may get ahead of the actual time if the packet rate averages more than 100,000 packets per second.
signature (48 bits)	A 48 bit signature for the packet, based on the complete packet, timestamp, and secret key.

See below for more information about the fields.

Link IDs

The 8 bit link ID is provided to ensure that the signature system is robust for multi-link MAVLink systems. Each implementation should assign a link ID to each of the MAVLink communication channels it has enabled and should put this ID in the link ID field. The link ID is especially important where there may be a significant latency difference between different links (such as WiFi combined with a telemetry radio).

The monotonically increasing timestamp rule is applied separately for each logical stream, where a stream is defined by the tuple:

(SystemID,ComponentID,LinkID)

For more information see C Message Signing > Handling Link IDs.

Signature

The 48 bit (6 byte) signature is the first 48 bits of a SHA-256 hash of the complete packet (without the signature, but including the timestamp) appended to the secret key. The secret key is 32 bytes of binary data stored on both ends of a MAVLink channel (i.e. an autopilot and a ground station or MAVLink API).

This is shown below, where + represents concatenation and sha256_48() is a sha256 implementation which returns the first 48 bits of the normal sha256 output:

```
signature = sha256_48(secret_key + header + payload + CRC + link-ID + timestamp)
```

Timestamp Handling

The timestamp is a 48 bit number with units of 10 microseconds since 1st January 2015 GMT. For systems where the time since 1/1/1970 is available (the unix epoch) you can use an offset in seconds of 1420070400.

This is a loose definition, as the various update mechanisms detailed below may result in the timestamp being significantly different from actual GMT time.

All timestamps generated must be at least 1 more than the previous timestamp sent in the same session for the same link/(SystemID, ComponentID, LinkID) tuple. The timestamp may get ahead of GMT time if there is a burst of packets at a rate of more than 100 thousand packets per second.

A MAVLink-enabled device may not know the current GMT time, for example if it does not have a reliable time source, or if it has just booted and not yet obtained the time from GPS or some other system.

Systems should implement the following rules to obtain a reliable timestamp:

- The current timestamp should be stored regularly in persistent storage (ideally at least once a minute)
- The timestamp used on startup should be the maximum of the timestamp implied by the system clock and the stored timestamp
- If the system does not have an RTC mechanism then it should update its timestamp when GPS lock is achieved. The maximum of the timestamp from the GPS and the stored timestamp should be used.
- The timestamp should be incremented by one on each message sent from a particular link.
- When a correctly signed message is decoded the timestamp should be replaced by the timestamp of the incoming message if that timestamp is greater than the current timestamp. The link timestamp must never be updated with the timestamp from an incorrectly signed packet (even if these are being accepted).
- The timestamp on incoming signed messages should be checked against the previous timestamp for the incoming (linkID,srcSystem,SrcComponent) tuple and the message rejected if it is smaller.

- If there is no previous message with the given (linkID,srcSystem,SrcComponent) then the timestamp should be accepted if it is not more than 6 million (one minute) behind the current timestamp.

For devices that store the timestamp in persistent storage, implementations can prevent race conditions by storing two timestamp values. On Write the smaller of the two values should be updated. On read the larger of the two values should be used.

Accepting Signed Packets

When a signed packet arrives it should be discarded if the:

- Timestamp is older than the previous packet from the same logical stream - where a logical stream is defined as the sequence of MAVLink packets with the same (SystemID, ComponentID, LinkID) tuple.
- Computed 48 bit signature does not match the signature included in the packet.
- The timestamp is more than 1 minute (6,000,000) behind the local system's timestamp.

Accepting Unsigned Packets

MAVLink libraries should provide a mechanism that allows a system to conditionally accept *unsigned* packets.

The rules for accepting these packets will be implementation specific, but could be based on a combination of a parameter setting, transport type, message type, (in)compatibility flags etc.

All packets that do not meet the system-specific unsigned packet acceptance rules must be rejected (otherwise there is no benefit gained from signing/authentication).

Some suggestions for when to accept unsigned packets:

- Accept all unsigned packets based on a system-specific parameter.
- Accept all unsigned packets if the connection is over a "secure channel" (e.g. local USB cable or local wired Ethernet cable).
- `RADIO_STATUS` packets are always accepted without signing (to make life easier for telemetry radios).
- Accept all unsigned packets when in an "unsigned mode" (perhaps triggered by a hardware button pressed on boot).
- Accept all unsigned packets until a signed packet is received (unconditionally), then move to the more restricted signing rules above.

Accepting Incorrectly Signed Packets

MAVLink libraries should provide a mechanism that allows a system to conditionally accept incorrectly signed packets.

This feature might be useful for finding a lost vehicle with a corrupted secret key (the GCS could choose to still display position information, albeit ideally with a different "untrusted" icon).

A system that is accepting incorrectly signed packets should provide a highly conspicuous indication that the connection is unsafe/insecure. Malformed signed packets indicate a bad configuration, transport failure, protocol failure, or hostile manipulation.

Secret Key Management

A secret key is 32 bytes of binary data that are used to create message signatures that can be verified by other holders of the key. The key should be created on one system in the network (often a GCS) and shared to other trusted devices via secure channels. Systems must have a shared key in order to be able to communicate.

The `mavgen` C and Python libraries support only one key per link. This is a choice of the library and not a limit/requirement of the protocol. An implementation might instead store a pool of keys, and/or manage keys on a per-connection basis.

The secret key should be stored in persistent storage, and must not be exposed via any publicly accessible communication protocol. In particular, the key must not be exposed in MAVLink parameters, MAVLink log files or *dataflash* log files that may be used for public log analysis.

The method of generating the secret key is implementation dependent. For example, it could be generated by:

- A user-entered string that is then run through SHA-256.
- A random key generator.

The secret key may be shared to other devices using the `SETUP_SIGNING` message. The message should only ever be sent over a secure link (e.g. USB or wired Ethernet) as a direct message to each connected system_id/component_id. The receiving system must be set up to process the message and store the received secret key to the appropriate permanent storage.

The same secure method can be used to both *set* and *reset* a system's key (resetting a key does not have to be "more secure" than setting it in the first place).

The SETUP_SIGNING message should never be broadcast, and received SETUP_SIGNING messages must never be automatically forwarded to other active MAVLink devices/streams/channels. This is to avoid the case where a key received over a secure link (e.g. USB) is automatically forwarded to another system over an insecure link (e.g. Wifi).

Autopilots that don't offer MAVLink over USB might create a module that can set the secret key from a command line interface (e.g. the NSH Shell).

We recommend that GCS implementations should generate the secret key and share this with connected systems over a secure link (e.g. USB). The receiving system may be configured to ignore message signatures on the secure channel (i.e. accept all signed, unsigned or incorrectly signed packets), so that it is possible to reset a key that has been lost or corrupted.

Logging

In order to avoid leaking the secret key used for signing, systems should omit `SETUP_SIGNING` messages from logs (or replace the secret with 32 0xFF bytes in the logged message).

Similarly, signed packets should have the signature incompatibility bit cleared and the signature block removed before being put into telemetry log files. This makes it harder for potential attackers to collect large amounts of signature data with which to attack the system.

Packaging and Streaming Video and Metadata

MAVLink Camera Protocol microservice is used to configure camera video streaming which may optionally include metadata streaming. Video and metadata streams shall comply with MISB ST 804.4 - Real-Time Protocol for Motion Imagery and Metadata with the following exceptions: - MPEG-2 video compression shall not be used (MPEG-2 transport streams may still be optionally used)

Overview

Supported network protocols: - RTP - SRTP - RTCP - RTSP

Supported transport: - Native carriage - MPEG-2 Transport Stream (MPEG-2 TS)

Supported video compression standards: - h.264/AVC

Support for optional KLV-encoded metadata streams.

Supported standards related to packaging and streaming video and metadata are shown below.

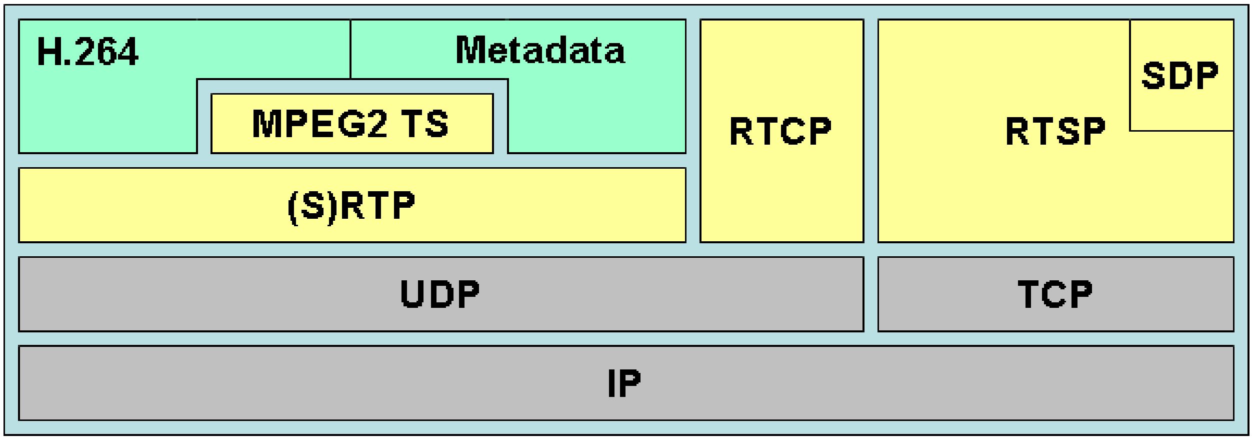

Figure 4: MISB ST 804.4 - Overview

Metadata

When KLV metadata is included with video it shall be in accordance with MISB ST 0601.18 - UAS Datalink Local Set. No fields are required for servers to include and servers are not limited to only using fields specified (although additional fields will not be required to be parsed). All MISB ST 0601.18 fields may be parsed by clients and clients may parse additional fields as well.

Security

Cryptographically secure media streams are not required (encryption is required at the data link layer). Secure RTP (SRTP) is acceptable under MISB ST 804.4 and should to be implemented as follows when utilized: - Implement both encryption and authentication - Use 256-bit key lengths

Microservices

The MAVLink microservices define higher-level protocols that MAVLink systems should adopt in order to better inter-operate.

The microservices are used to exchange many types of data, including: parameters, missions, trajectories, images, other files. Microservices that support data to be delivered that would surpass MAVLink message size limits should define how to disassemble, reassemble, and provide a loss-less delivery mechanism. Other services provide command acknowledgment and/or error reporting.

Most services use the client-server pattern, such that the GCS (client) initiates a request and the vehicle (server) responds with data.

The microservices are listed below:

- Datalink Pairing Protocol
- Heartbeat/Connection Protocol
- Generic Payload Attribute Protocol
- Telemetry
  - General requirements
  - Battery Protocol
- Manual Control Protocol
- Mission Protocol
- Parameter Protocol
- Extended Parameter Protocol
- Command Protocol
- Camera Protocol
  - Camera Definition
- Gimbal Protocol v2
  - Gimbal Protocol v1 (superseded)
- File Transfer Protocol (FTP)
- Payload Protocols
- Terrain Protocol
- Exploration Protocol

Datalink Pairing Protocol

Introduction

The pairing process is a secure way to establish a connection between a vehicle and a ground station device (GCS). It also covers mutual discovery, meaning that the two devices do not need prior knowledge about the other device. Once the pairing process is successful, a paired ground station will be able to communicate with the vehicle and exchange information such as flight information (telemetry), video from the vehicle and commands.

Once initiated the pairing process configures the radio so that the connection is reserved to a specific ground station/vehicle pair and won't interfere with other ground stations or vehicles.

The pairing process that is described in this section does not prescribe use of a particular hardware layer (often called the radio). The modular architecture allows extension to additional hardware with the simple implementation of a driver.

The following datalinks have been implemented and verified to work with the pairing protocol. Note this is not an exclusive list of supported hardware.

- Microhard pMDDL and pDDL series (tested on the pMDDL2450 and pDDL1800)
- Doodle Labs Smart Radios
- Silvus Technologies Radios
- Persistent Systems Wave Relay / MPU5
- Trellisware Ghost (TW-870 / TW-875)

Compatible datalinks are required to support AES-256 encryption by default.

There can be different approaches to pair a ground station to a vehicle:

- *In-Band Pairing (IB Pairing)*: Pairing is performed over the same radio link that is being configured. For example, to pair a GCS and a vehicle that have Microhard radios, the radios must initially be configured with some predefined settings so that the two sides can start communicating. This initial connection allows them to exchange the connection settings that will be applied on both sides to establish the final connection. This final connection is then used to exchange telemetry and video between the GCS and vehicle.

- *Out-Of-Band Pairing (OOB Pairing)*: Pairing is performed using a mechanism or channel that is separate to the radio link that is being paired. There are many possible out-of-band mechanisms for exchanging the radio connection settings between the GCS and vehicle, including using a USB cable to transfer the data, or reading the connection settings from a QR code using a camera (this option is discussed in more detail later in this section).

A design assumption for the in-band pairing approach is that the pairing procedure has to be performed in a safe area where radio communication cannot be overheard due to the fact that this procedure is not cryptographically secure.

In-Band Pairing Flow

A GCS that implements this protocol is expected to support UDP unicast as a transport layer to transfer the JSON files over the network between the GCS and the vehicle.

The following diagram and steps show the flow of events from boot until the GCS is connected to the vehicle.

1. Initiate pairing on the vehicle by issuing the MAVLink command MAV_CMD_START_RX_PAIR UX is left to the implementer. One example could be a specific button the user has to press. The vehicle goes into pairing mode and configures its radio into a pairing mode.
2. The user initiates pairing on the GCS. The GCS goes into pairing mode and configures its radio for pairing. The GCS and the vehicle are now able to communicate over the radio link in pairing mode.

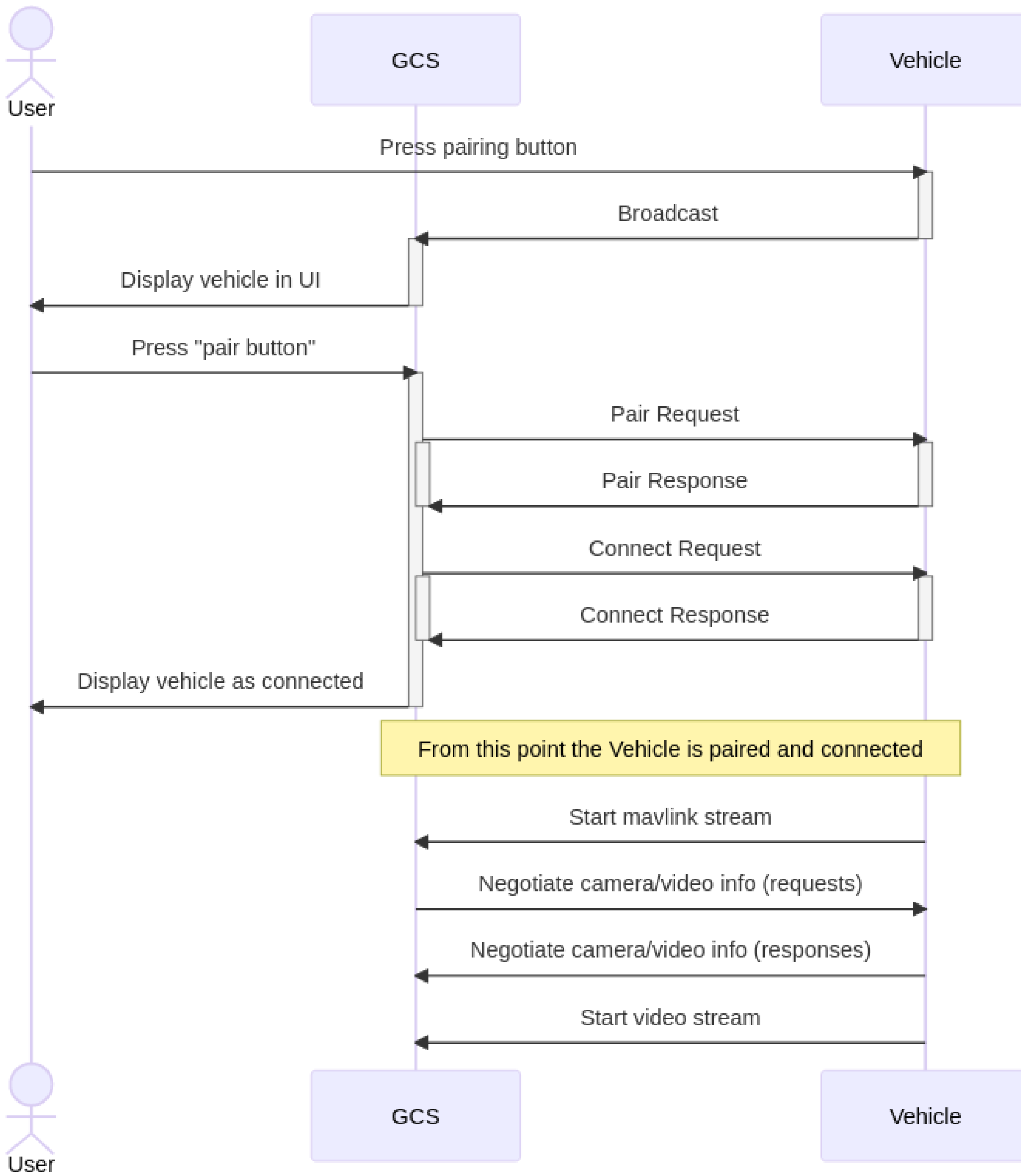

Figure 5: Full pairing protocol

3. The vehicle will start broadcasting the information required for discovery (See discovery in the API section)
4. The GCS receives the discovery message and now knows the vehicle's address
5. At this point the GCS has discovered the vehicle. When this happens most GCS show the vehicle in the list of discovered vehicles, and the user can then click on a pair button in the UI to start exchanging the configuration data that will be used once the systems are paired.
6. After the user indicates that they want to pair with the vehicle, the GCS will start sending periodic pair requests until the vehicle responds with an acknowledgement or there is a timeout. This timeout should be set to 10 seconds. The requests should be sent out at least once every 3 seconds. If the ack is received by the GCS, it will assume that the the vehicle accepted the pairing request and will now change the radio settings to the values defined in the pair request. The GCS should change the settings to the same values and move to the next step.
7. The GCS will start sending periodic connect requests until the vehicle responds with an acknowledgement or there is a timeout. This timeout should be set to 10 seconds. The requests should be sent out at least once every 3 seconds. If the ack is received by the GCS, it will consider the vehicle to have accepted the proposed connection settings and be paired.
8. The systems can now connect and exchange MAVLink messages. Usually the GCS will bind to a local port (either hardcoded or randomly assigned) that was previously sent to the vehicle in the connect message.
9. The GCS and the vehicle can store the the pairing settings locally.

In the case of multiple GCS connecting the same vehicle the individual pairing between a specific ground station and the vehicle is exactly the same as described above: a sequence of events between the ground station and the vehicle. The hand-off of pairing (and subsequently control) is expected to be covered by future revisions of RAS-A.

The protocol uses a status message to determine if the other end of the communication is still connected. The message should be transmitted at 1Hz frequency from both sides of the communication, and timeouts should be provided. The standard does not enforce the length of the status timeout because this could vary with different radios and different types of systems. The status message/connection state information can be used for a few different purposes:

- with some radios we might want to change some radio settings when we lose connection with the other side
- we can use this information to communicate to other software components both on the vehicle and on the GCS if we are still connected or not
- the pairing software on both sides does not need to monitor various

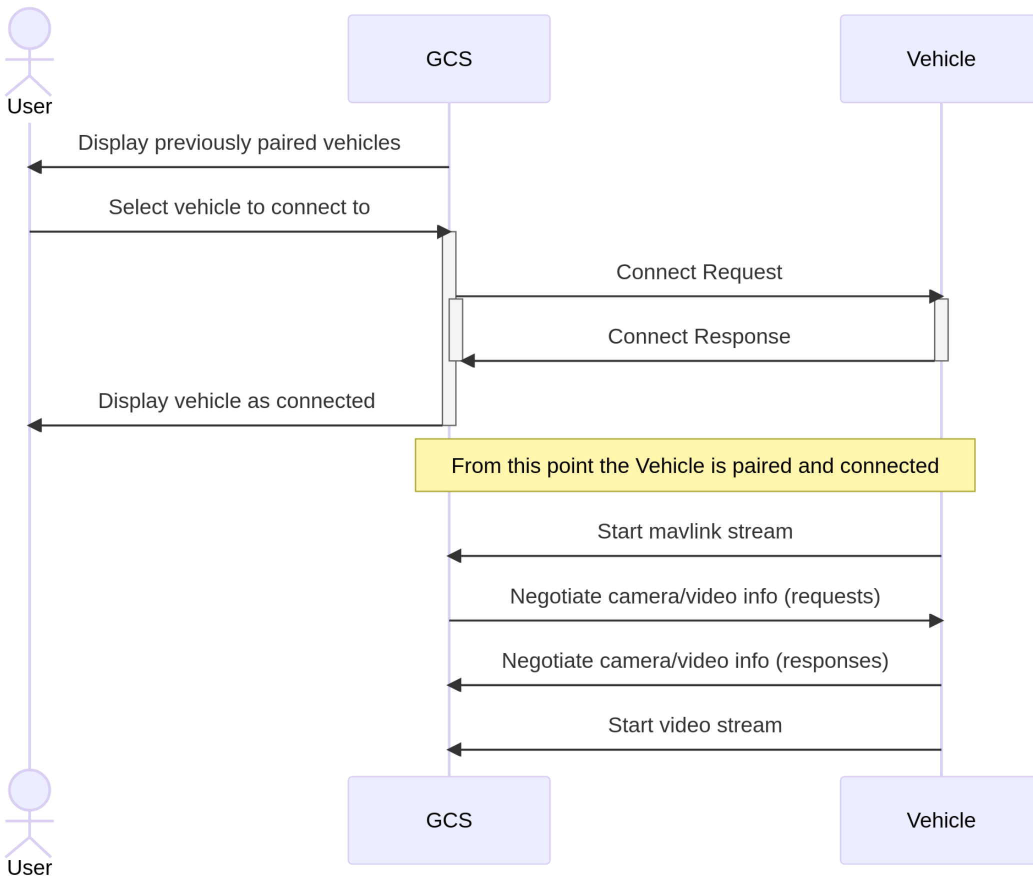

Figure 6: Connect protocol when already paired

MAVLink `HEARTBEAT` messages other components. This could become quite convoluted when multi GCS or multi vehicle will be supported in the near future.

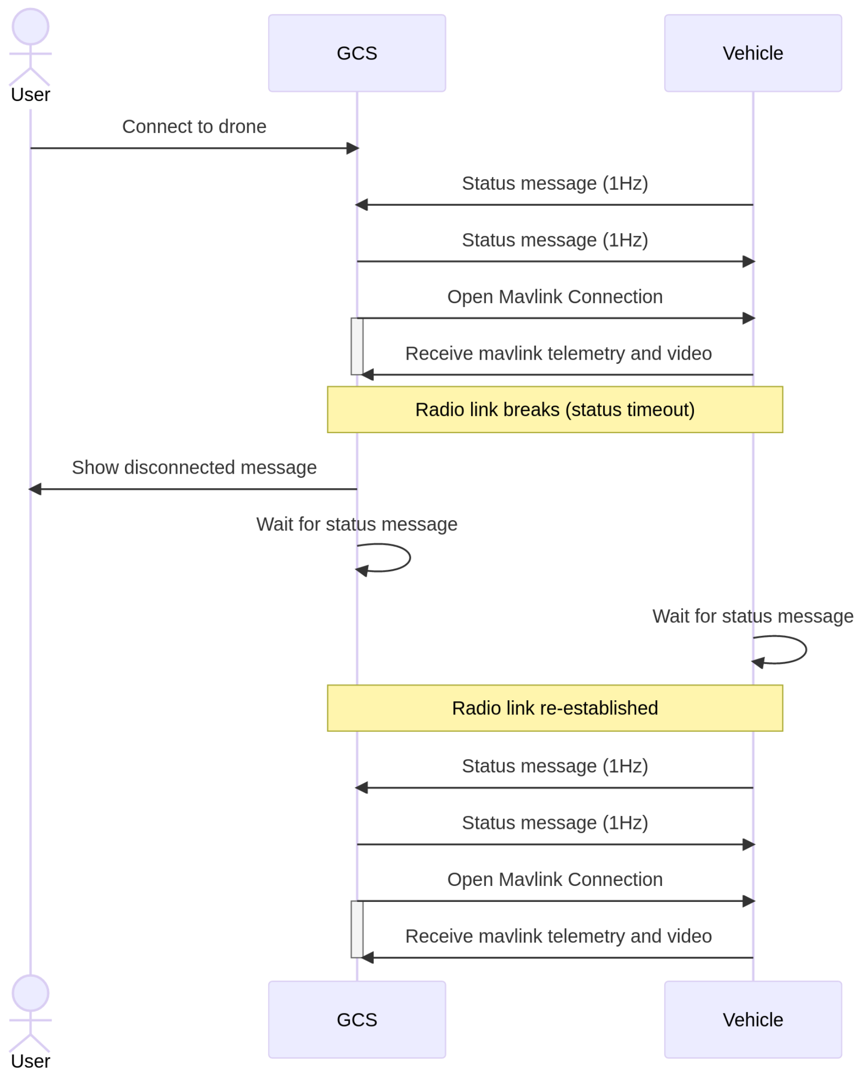

Figure 7: Connection loss and reconnection protocol

Default Radio Settings When performing In Band Pairing over the same radio link that will then be used to connect to the vehicle, a set of default radio settings has to be defined and applied on both sides (GCS and Vehicle) such that the two can start communicating. At this point the two sides can exchange the various pairing messages and start the connection for control, telemetry and video streaming.

Frequency All the radios used as part of this ecosystem support 1 or more of the following 6 bands: 1.6, 1.8, 2.0, 2.2, 2.3, 2.4 GHz. If supported the default band for pairing should be the 1.8 GHZ one. If not supported then any other band is accepted as long as one of the following frequencies is used for each band:

Band (GHz)	Default Pairing Frequency (MHz)
1.6	1723
1.8	1823
2.0	2023
2.2	2223
2.3	2323
2.4	2423

Other Settings The following are the other default settings other than the frequency mentioned above. The security key is the only one that can be defined by the vendor. All other settings need to match the ones in the table below.

Setting	Default Value
Bandwidth	<lowest-bandwidth>
Network ID	MH
Security	AES-256
Security Key	<vendor-specific>
TX power	<minimum-power>
Drone Radio Topology	access-point

Default Network Settings The following are the default network settings used by the pairing protocol. By default all GCSs should send UDP packets to a specific port on the vehicle side.

Setting	Default Value
Network ID	UDP
Pairing Port	29350

Variable Settings The following table shows the settings that can be modified from radio defaults during the “pair” phase of the In Band pairing or using the “reconfigure” message while already paired to the vehicle. If settings are not changed with one of the two methods, the assumption is that the radio defaults mentioned above will be used.

Setting	Allowed Values
Network ID	<any-string-that-meets-radio-requirements>
Channel	Hz
Bandwidth	Hz
Radio Topology	access-point, station, mesh, relay
IP Address	<valid-ip-address>
Port	<port-used-to-send-pairing-messages>
Protocol	UDP, TCP

Not all options in the radio topology setting are supported on every radio. The user should first figure out if the radio supports it and only then try to set the valid options for the given radio.

Pairing Manager API Except for the discovery message, all other message need to be sent as a UDP datagram containing the serialized JSON definition and sent to the source IP that sent the discovery message.

> **Note:** Every message contains a list of drivers that represents the radio types that are available to pair with on the vehicle and on the GCS side. These are sent irrespective of the network interface. This is to give the master side the flexibility of pairing over multiple data links even if they are not all available during pairing time.

Discovery Advertise the vehicle on the network so that GCS devices can discover the vehicle and show it to the user. This message needs to be sent out at least every 3 seconds. The user can then select which vehicle to pair to and send the connect request. The message needs to be a UDP datagram containing the serialized JSON definition and sent either via broadcast or multicast. In the future the standard could be extended with more integrated mechanisms such as MDNS and Zeroconf. For in band pairing during this phase the radios have the default (known) settings set so that the two sides can communicate and start exchanging information. For out of band pairing this step might not be necessary depending on the type of out of band pairing.

```
{
  "drivers" :
  [
    {
      "ip" : "192.168.168.20",
      "name" : "Microhard",
      "port" : 29360,
      "remote_ip" : "192.168.168.165"
```

```
    }
  ],
  "machine_name" : "vehicle-name",
  "request" : "broadcast"
}
```


Attribute	Type	Required	Description
drivers	array	yes	List of drivers (radio types) that are available to pair with on the vehicle.
ip	string	yes	Radio IP for the unit connect to the vehicle (slave).
name	string	yes	Type of driver. Usually this refers to the radio model.
port	int	yes	Port used on the vehicle side (slave).
remote_ip	string	yes	Vehicle IP for the network interface used by this driver.
machine_name	string	yes	The vehicle name that will be shown to the user when this message is received by the ground side (master).
request	string	yes	Set to “broadcast” when sending this message.

Pair Send pairing request from the ground station to the vehicle. If the vehicle responds with a success then the ground station will add it to the list of paired vehicles and can then connect without exchanging information next time. During this phase the radios have the default (known) settings set so that the two sides can communicate and start exchanging information.

Request


```
{
  "drivers" :
  [
    {
      "instance" : "MH2450",
      "ip" : "192.168.168.3",
```

```
      "name" : "Microhard",
      "port" : 29350,
      "remote_ip" : "192.168.168.241"
    }
  ],
  "machine_name" : "QGCGov",
  "request" : "pair",
  "bandwidth": 4,
  "frequency": 1863,
  "tx_power": 30,
  "network_id": "StringToBeUsedAsMicrohardNetworkId",
  "encryption_password": "StringToBeUsedasMicrohardAES256EncryptionPassword"
}
```


Attribute	Type	Required	Description
drivers	array	yes	List of drivers (radio types) that are available to pair with on the vehicle.
instance	string	yes	This field contains the driver instance identifier. This is important when we have multiple radio of the same type connected to the GCS.
ip	string	yes	Radio IP for the unit connect to the GCS (master).
name	string	yes	Type of driver. Usually this refers to the radio model.
port	int	yes	Port used on the GCS side (master).
remote_ip	string	yes	GCS IP for the network interface used by this driver.
machine_name	string	yes	The GCS name that will be used by the vehicle side.
request	string	yes	Set to “pair” when sending this message.

Response

```
{
  "drivers" :
  [
    {
      "instance" : "MH2450",
      "ip" : "192.168.168.3",
      "name" : "Microhard",
      "port" : 29350,
      "remote_ip" : "192.168.168.241"
    }
  ],
  "machine_name" : "QGCGov",
  "request" : "pair",
  "accepted": true,
  "bandwidth": 4,
  "frequency": 1863,
  "tx_power": 30,
  "network_id": "StringToBeUsedAsMicrohardNetworkId",
  "encryption_password": "StringToBeUsedasMicrohardAES256EncryptionPassword"
}
```


Attribute	**Type**	**Required**	**Description**
drivers	array	yes	List of drivers (radio types) that are available to pair with on the vehicle.
instance	string	yes	This field contains the driver instance identifier. This is important when we have multiple radio of the same type connected to the GCS.
name	string	yes	Type of driver. Usually this refers to the radio model.
port	int	yes	Port used on the vehicle side (slave).
remote_ip	string	yes	Vehicle IP for the network interface used by this driver.
machine_name	string	yes	The vehicle name that will be used by the GCS side.

Attribute	Type	Required	Description
response	string	yes	Set to “pair” when sending this message.

Connect Send connect request from the ground station to the vehicle. If the vehicle responds with a success then the ground station will add it to the list of connected vehicles and will open ports to receive telemetry and video from the vehicle. After the vehicle has responded with an accepted ack to the pair message, both sides will reconfigure radios to the newly agreed upon RF seetings and the connect message shall occur on the new radio settings.

Request

```
{
  "drivers" :
  [
    {
      "instance" : "MH2450",
      "mavlink_port" : 15667,
      "name" : "Microhard",
      "remote_ip" : "192.168.168.241"
    }
  ],
  "machine_name" : "QGCGov",
  "request" : "connect"
}
```

Attribute	Type	Required	Description
drivers	array	yes	List of drivers (radio types) that are available to pair with on the vehicle.
instance	string	yes	This field contains the driver instance identifier. This is important when we have multiple radio of the same type connected to the GCS.
mavlink_port	int	yes	Port on the GCS side that the vehicle should start streaming telemetry to at the end of the connect phase.

Attribute	Type	Required	Description
name	string	yes	Type of driver. Usually this refers to the radio model.
remote_ip	string	yes	GCS IP for the network interface used by this driver.
machine_name	string	yes	The GCS name that will be used by the vehicle side.
request	string	yes	Set to “connect” when sending this message.

Response

```
{
  "drivers" :
  [
    {
      "instance" : "MH2450",
      "mavlink_port" : 15667,
      "name" : "Microhard",
      "remote_ip" : "192.168.168.165"
    }
  ],
  "machine_name" : "vehicle-name",
  "request" : "connect",
  "accepted" : "y"  # y|n
}
```

Attribute	Type	Required	Description
drivers	array	yes	List of drivers (radio types) that are available to pair with on the vehicle.
instance	string	yes	This field contains the driver instance identifier. This is important when we have multiple radio of the same type connected to the GCS.

Attribute	Type	Required	Description
mavlink_port	int	yes	Port on the GCS side that the vehicle should start streaming telemetry to at the end of the connect phase.
name	string	yes	Type of driver. Usually this refers to the radio model.
remote_ip	string	yes	Vehicle IP for the network interface used by this driver.
machine_name	string	yes	The vehicle name that will be used by the vehicle side.
response	string	yes	Set to "connect" when sending this message.

Disconnect Send the disconnect request from the ground station to the vehicle. Since we don't know the state of the radios after we send this command, we don't expect a response for this message. The GCS will assume it is disconnect unless the status message continues coming in. In this case it will show the vehicle as still connected and the user will need to repeat the action.

Request

```
{
  "machine_name" : "QGCGov",
  "request" : "disconnect",
}
```

Attribute	Type	Required	Description
machine_name	string	yes	The GCS name that will be used by the vehicle side.
request	string	yes	Set to "disconnect" when sending this message.

Status Send the status message from the vehicle to the ground and then the ground will respond by taking the same message and replacing the vehicle's driver information with its own. This message has a 6 seconds timeout on both sides to detect when the other side disconnects.

Request


```
{
  "drivers" :
  [
    {
      "ip" : "192.168.168.20",
      "name" : "Microhard",
      "port" : 29360,
      "remote_ip" : "192.168.168.165"
    }
  ],
  "machine_name" : "vehicle-name",
  "request" : "status",
  "seq" : 17,
  "timestamp" : 1655992758756
}
```


Attribute	**Type**	**Required**	**Description**
drivers	array	yes	List of drivers (radio types) that are available to pair with on the vehicle.
ip	string	yes	Radio IP for the unit connect to the vehicle (slave).
name	string	yes	Type of driver. Usually this refers to the radio model.
port	int	yes	Port used on the vehicle side (slave).
remote_ip	string	yes	Vehicle IP for the network interface used by this driver.
machine_name	string	yes	The vehicle name that will be used by the GCS side.
request	string	yes	Set to “status” when sending this message.
seq	int	yes	Sequence number for the status message.
timestamp	int	yes	Timestamp from the vehicle side when sending this message.

Response


```
{
  "instance" : "MH2450",
  "machine_name" : "QGCGov",
  "mavlink_port" : 15667,
  "remote_ip" : "192.168.168.241",
  "response" : "status",
  "seq" : 17,
  "timestamp" : 1655992758756
}
```


Attribute	Type	Required	Description
instance	string	yes	This field contains the driver instance identifier. This is important when we have multiple radio of the same type connected to the GCS.
machine_name	string	yes	The vehicle name that will be used by the GCS side.
mavlink_port	int	yes	Port on the GCS side that the vehicle should start streaming telemetry to at the end of the connect phase.
remote_ip	string	yes	Vehicle IP for the network interface used by this driver.
machine_name	string	yes	The vehicle name that will be used by the GCS side.
response	string	yes	Set to “status” when sending this message.
seq	int	yes	Sequence number for the status message.
timestamp	int	yes	Timestamp from the vehicle side when sending this message.

Reconfigure Send the reconfigure request from the ground station to the vehicle. This will change the modem parameters on both sides while we are connected to a vehicle. No response is expected as the connection usually breaks after this command. If the remote doesn't respond to status message in timeout time the settings are reset back to the previous valid values.

Request


```
{
  "drivers" :
  [
    {
      "channel" : "52",
      "instance" : "MH2450",
      "name" : "Microhard"
    }
  ],
  "machine_name" : "QGCGov",
  "request" : "reconfigure"
}
```


Attribute	Type	Required	Description
drivers	array	yes	List of drivers (radio types) that are available to pair with on the vehicle.
channel	string	yes	Channel to set on both the master and slave sides. This could be replaced by other settings (eg. bandwidth).
instance	string	yes	This field contains the driver instance identifier. This is important when we have multiple radio of the same type connected to the GCS.
name	string	yes	Type of driver. Usually this refers to the radio model.
machine_name	string	yes	The vehicle name that will be used by the GCS side.
request	string	yes	Set to "reconfigure" when sending this message.

MAVLink API For systems that use MAVLink communications onboard, the following command is recommended to trigger pairing:

MAVLink API	Description
`MAV_CMD_START_RX_PAIR`	MAVLink command used to set the pairing manager to pairing mode so it can be discovered by the GCS. When this command is received the microhard is set to pairing settings and the pairing manager will start to broadcast its IP to be discovered by the GCS.
`RADIO_STATUS`	MAVLink stream to send out the radio status to other components. The message contains the most common radio indicators, such as signal strength and noise. The most common use cases for this message are to send it to the GCS to display for the user or to send to the logging system onboard so that this data can be logged.

Out-of-Band Pairing Flow

There are many possible ways to implement out-of-band pairing. This specification focuses mainly on using a QR code or a USB cable. Additional mechanisms may be added in future.

The connect message Send connect request from the ground station to the vehicle. If the vehicle responds with a success then the ground station will add it to the list of connected vehicles and will open ports to receive telemetry and video from the vehicle. During this phase the radios will change from the default pairing settings to the connect settings that are specified in this message. The message needs to be a UDP datagram containing the serialized YAML definition and sent to the source IP that sent the discovery message.

Request

```yaml
request: connect
hostname: gcs-0001

drivers:
  - type: microhard
```

```yaml
    # radio configuration
    frequency: 2450000000 #Hz
    bandwidth: 4000000 #Hz
    topology: mesh
    network_id: network_mh_0001
    security_key: 0000000011111111222222223333333344444444555555556666666aaaaaaaa
    tx_power: 100 #mW

    # device configuration
    device_ip: 192.168.168.3

    # MAVLink service: this is only required for systems that expose a MAVLink port so
    mavlink_protocol: udp
    mavlink_port: 15667

  - type: psmpu5

    # radio configuration
    frequency: 2450000000
    bandwidth: 10000000
    topology: mesh
    network_id: network_mpu5_0001
    security_key: 0000000011111111222222223333333344444444555555556666666aaaaaaaa

    # device configuration
    device_ip: 172.20.100.103

    # MAVLink service: this is only required for systems that expose a MAVLink port so
    mavlink_protocol: udp
    mavlink_port: 15667
```

General Settings

Attribute	Type	Required	Description
request	string	yes	Set to “connect” when sending this message.
hostname	string	yes	GCS hostname. Mainly used for caching purposes on the vehicle side.

Attribute	Type	Required	Description
drivers	string	yes	A list that will contain all the available radios in the system. Each element in the list will contain all the settings for the specific radio.

Radio Specific Settings

Attribute	Type	Required	Description
frequency	int	yes	Frequency to be set on the vehicle radio.
bandwidth	int	yes	Bandwidth to be set on the vehicle radio.
topology	string	yes	Radio network topology. Possible options: ACCESS_POINT, STATION, MESH, RELAY
network_id	string	no	Network id to specify in the radio. Not all radios support this so this field is optional.
security_key	string	yes	Security key to set in the radio to encrypt all the data going over the link.
tx_power	int	no	Transmit power to be used by the vehicle when sending data to the GCS. If not set the vehicle can use it's default power setting.
device_ip	string	yes	GCS IP.

Attribute	Type	Required	Description
mavlink_protocol	string	no	Use TCP or UDP to stream MAVLink. This is actually decided by the side that binds to the telemetry port. If the vehicle binds to the telemetry port and the GCS connects to it then the vehicle should specify this in the response to this message. If the CGS specifies this then the vehicle will need to use this protocol.
mavlink_port	int	no	Port to connect to to receive MAVLink. Same consideration as for the mavlink_protocol field.

Response


```yaml
response: connect
hostname: vehicle-0080

drivers:
  - type: microhard

    # device configuration
    device_ip: 192.168.168.4

    # MAVLink service: this is only required for systems that expose a MAVLink port so
    mavlink_protocol: udp
    mavlink_port: 15667

  - type: psmpu5

    # device configuration
    device_ip: 172.20.100.104

    # MAVLink service: this is only required for systems that expose a MAVLink port so
    mavlink_protocol: udp
    mavlink_port: 15667
```

General Settings

Attribute	**Type**	**Required**	**Description**
response	string	yes	Set to “connect” when sending this message.
hostname	string	yes	Vehicle hostname. Mainly used for caching purposes on the GCS side and to display the vehicle name in the GCS.
drivers	string	yes	A list that will contain all the available radios in the system. Each element in the list will contain all the settings for the specific radio.

Radio Specific Settings

Attribute	**Type**	**Required**	**Description**
device_ip	string	yes	Vehicle IP.
mavlink_protocol	string	no	Use TCP or UDP to stream MAVLink. This is actually decided by the side that binds to the telemetry port. If the GCS binds to the telemetry port and the vehicle connects to it then the GCS should specify this in the request for this message. If the vehicle specifies this then the GCS will need to use this protocol.

Attribute	Type	Required	Description
mavlink_port	int	no	Port to connect to to receive MAVLink. Same consideration as for the mavlink_protocol field.

QR Code Pairing The idea behind this mechanism is to serialize the radio configurations and embed them into a QR code. There are two possible ways to exchange information between the GCS and vehicle using the QR code:

- Add a sticker with a QR code to the vehicle that contains the default pairing settings. The GCS can scan the QR code with its camera and apply the same pairing settings on the GCS in order to establish the radio link.
- Every time pairing is triggered from the GCS UI a QR code is generated on the screen. The vehicle can scan the QR code with its camera and apply the same settings. This QR code is more flexible because it can be generated from user inputs in the UI, and can be changed as needed (with the first approach the QR code is hardcoded).

USB Pairing (Ethernet over USB) To make communication simpler and more flexible Ethernet over USB is used. This has the advantage of having a predefined link that goes through the USB cable, but we will still need some discovery mechanism where one of the two sides advertises its IP to the other side. For this mechanism the steps are:

1. Employ an Ethernet connection over usb between the GCS and the vehicle to make communication simpler and more flexible
2. Plug in a usb cable from the vehicle to the GCS
3. Vehicle sends “discovery” message as broadcast
4. GCS receives the message, stores the vehicle IP and sends the “connect” message with the desired radio settings
5. Vehicle responds to “connect” request to acknowledge and include the MAVLink port that it is exposing to telemetry
6. Now the radio link is fully established and the two sides can start communicating over it and the user could unplug the USB cable from this point
7. GCS connects to the telemetry port over the radio link

Message/Enum Summary

Table 29: Message Description

Message	Description
HEARTBEAT	Broadcast that a MAVLink component is present and responding, along with its type (MAV_TYPE) and other properties.

Table 30: ENUM Description

Enum	Description
MAV_TYPE	Type of the component. Flight controllers must report the type of the vehicle on which they are mounted (e.g. MAV_TYPE_OCTOROTOR). All other components must report a value appropriate for their type (e.g. a camera must use MAV_TYPE_CAMERA). For generic RAS-A components, [MAV_TYPE_GENERIC_COMPONENT][MAV_TYPE_GENERIC_COMPONENT] should be used as the MAV_TYPE.
MAV_AUTOPILOT	Autopilot type / class.
MAV_MODE_FLAG	System mode bitmap.
MAV_STATE	System status flag.

HEARTBEAT Broadcast Frequency

Components must regularly broadcast their HEARTBEAT and monitor for heartbeats from other components/systems.

The rate at which the HEARTBEAT message must be broadcast, and how many messages may be "missed" before a system is considered to have timed out/disconnected from the network, depends on the channel (it is not defined by MAVLink). On RF telemetry links, components typically publish their heartbeat at 1 Hz and consider another system to have disconnected if four or five messages are not received.

A component may choose not to send or broadcast information on a channel (other than the HEARTBEAT) if it does not detect another system, and it will continue to send messages to a system while it is receiving heartbeats. Therefore it is important that systems:

- broadcast a heartbeat even when not commanding the remote system.
- do not broadcast a heartbeat when they are in a faulted state (i.e. do not publish a heartbeat from a separate thread that is unaware of the state of the rest of the component).

Connecting to a GCS or MAVLink API

The `HEARTBEAT` may also be used by GCS (or Developer API) to determine if it **can** connect to a vehicle in order to collect telemetry and send missions/commands.

For example, *QGroundControl* will only connect to a vehicle system (i.e. not another GCS, gimbal, or onboard controller) before displaying the vehicle connected message. QGC also uses the specific type of vehicle and other heartbeat information to control layout of the GUI.

The specific code for connecting to QGroundControl can be found in MultiVehicleManager.cc (see void void MultiVehicleManager::_vehicleHeartbeatInfo).

Component Identity

The *type* (`MAV_TYPE`) of a component is obtained from its `HEARTBEAT.type` field.

- A flight controller component will use a `MAV_TYPE` corresponding to a particular vehicle - e.g. `MAV_TYPE_FIXED_WING`, `MAV_TYPE_QUADROTOR` etc. (The use of any of these "vehicle types" indicates the component is a flight controller).
- For generic RAS-A components, use [`MAV_TYPE_GENERIC_COMPONENT`][MAV_TYPE_GENERIC_COMPONENT].
- Prior to the RAS-A IOP v1.2 other components used their actual type, e.g. `MAV_TYPE_GIMBAL`, `MAV_TYPE_BATTERY`, etc. | deprecated

Every component must have a system-unique component ID, which is used for routing and for identifying multiple instances of a particular component type.

Historically the component ID was also used to determine the component type. For any MAVLink system implementing the generic ras-a payload interface, it is encouraged to think of the unique `MAV_COMPONENT` field in heartbeats as simply a unique component number used for routing and topological purposes. New code must not make any assumption about the type from the ID used (type is determined from `HEARTBEAT.type`).

MAVLink recommends that *by default* components use a type-appropriate component id from `MAV_COMPONENT`, and provide an interface to change the component ID if needed. For example, a camera component might use any of the `MAV_COMP_ID_CAMERAn` IDs, and should not use `MAV_COMP_ID_GPS2`.

`MAV_COMP_ID_MONOLITHIC` is used for systems which implement many different component behaviors which are not well captured by a single component ID.

Using type-specific component IDs:

- Makes ID clashes less likely "out of the box" (unless two components of the same type are present on the same system).
- Reduces the impact on legacy code that determines component type from the ID. No breaking changes are allowed to component identity methods

until the RAS-A IOP v2.0 is released.

Generic Payload Attribute Protocol

Any system component which presents a heartbeat containing `MAV_TYPE_GENERIC_COMPONENT` implements the Generic Payload Attribute Protocol.

Any `MAV_TYPE_GENERIC_COMPONENT` implements the Parameter Protocol and expresses what type of payload it is and its attributes via mavlink parameter exchange.

Attribute Parameter Schema

The parameter naming schema for generic component attributes parameters are as follows:

All RAS-A specifc standard parameters have 4 blocks in the parameter's string separated by periods (ASCII `0x2E`): ___

`RAS.<x>.<y>.<n>` - `RAS`: standard start - `<x>`: type - `<y>`: attribute name - `n`: group number (may be omitted, may be no larger than 255) ___

The convention for the second block specifies what block `<y>` contains: ___ `RAS.<A>.<y>.<n>` - An Attribute which can be associated with a capability by group `<n>`, or is standalone

`RAS.<C>.<y>.<n>` - A Capability where the string in `<y>` specifies a RAS-A capaibility

`RAS.<x>.<y>.<n>` - `<n>` contains a base 10 integer group number. If omitted from the parameter string, the parameter belongs to the default group. ___

All parameters are grouped. Any repeating attributes or capabilities must be part of separate groups, repeating attributes within a group are not allowed. If the number in the last block is omitted, it belongs to the default group. ***`RAS.<x>.<y>` - Belongs to the default group `RAS.<x>.<y>.0` - Belongs to group 0 `RAS.C.EMIT.0` - Denotes an emitter capabilty in group 0 `RAS.C.EMIT.1` - Denotes an second emitter capability in group 1***

Default Parameters Contents of the payload / capability attribute parameter payload are to be defined in the IOP and convey properties of the attribute. As RAS-A versions increment, this list will grow.

RAS-A Version	Parameter Name	Description
1.2	RAS.C.EMIT	Attribute belongs to a component that emits something, intended to be ued with components that emit light, or sound.
1.2	RAS.A.POWER	Attribute that describes a power level in percent from 0 to 100
1.2	RAS.A.PERIOD	Period of some sort of emitter frequency. This value is in seconds.
1.2	RAS.A.DUTY	Emitter period PWM duty cycle.
1.2	RAS.A.ONLY	Within a component if ONLY exists within a group and is set to 1, only that group may be “ACTIVE” at any time. E.g. you have 3 emitter groups (0,1, and 2) with ONLY and group 0 is set to ONLY (1), group 0 may only be on if 1 and 2 are off. An exclusivity trait.
1.2	RAS.A.TEMP	Read-only attribute for reading temperature. Value is in celsius.
1.2	RAS.C.DEPLOY	Component supports mavlink command for payload deployment control.
1.2	RAS.C.ARM	Component supports the arming authorization protocol.
1.2	RAS.C.PARENT	Value of this paramter contains the parents compoenent number that this component is attached to.
1.2	RAS.A.ACTIVE	Attribute used to set a component group to active.

RAS-A Version	Parameter Name	Description
1.2	RAS.C.GIMB	Component implements gimbal microservice with version number contained in parameter value.
1.2	RAS.C.FRAME	The frame of reference for the component. Value contains `MAV_FRAME`.
1.2	RAS.C.T1	First translation parameter in the RAS.C.FRAME
1.2	RAS.C.T2	Second translation parameter in the RAS.C.FRAME
1.2	RAS.C.T3	Third translation parameter in the RAS.C.FRAME
1.2	RAS.C.QUATX	First quaternion rotation parameter in the RAS.C.FRAME
1.2	RAS.C.QUATY	Second quaternion rotation parameter in the RAS.C.FRAME
1.2	RAS.C.QUATZ	Third quaternion rotation parameter in the RAS.C.FRAME
1.2	RAS.C.QUATW	Fourth quaternion rotation parameter in the RAS.C.FRAME
1.2	RAS.C.MAVCAM	Component implements the camera microservice

Example Generic Payload: Gimbal with Camera

This example describes a gimbal with two components, which all-together: - Implements Gimbal Control V2 - Has a camera on it - Separate payload - Has a laser pointer on it - Gimbal Controlled Capability - Has an illuminator on it - Gimbal Controlled Capability - Has a widget on it - Gimbal Controlled Capability - Has 5 distinct elements: Camera, illuminator, widget, laser, gimbal.

This gimbal has a gimbal controller which is conneted over UART to a mission computer, and a camera which is IP addressable. In this case the gimbal would emit two distinct payload heartbeats, one for the gimbal, and one for the camera.

Through the generic payload parameter interface, two heartbeats expose the following capabilities via parameter exchange: Capabilities: Illumination (Green), Laser (LWIR), Widget (Trumpet Sounds), Gimbal (V2), Known Translation / Rotation.

Heartbeat 1 leads to discovery of the following parameters: PARAM: `RAS.C.GIMB` - Version lives in param_value (V2) PARAM: `RAS.C.EMIT.0` - TYPE lives in the param_value (green led) PARAM: `RAS.C.EMIT.1` - TYPE lives in the param_value (LWIR laser CW ON/OFF) PARAM: `RAS.C.EMIT.2` - TYPE lives in the param_value (Trumpets) PARAM: `RAS.C.FRAME` - Reference frame for translation lives in param_value (LOCAL_NED) PARAM: `RAS.C.T1` - Translation value 1 in the param_value PARAM: `RAS.C.T2` - Translation value 2 in the param_value PARAM: `RAS.C.T3` - Translation value 3 in the param_value PARAM: `RAS.C.QUATX` - Quaternion x in the param_value PARAM: `RAS.C.QUATY` - Quaternion y in the param_value PARAM: `RAS.C.QUATZ` - Quaternion z in the param_value PARAM: `RAS.C.QUATW` - Quaternion w in the param_value

Heartbeat 2 leads to discovery of the following parameters: PARAM: `RAS.C.MAVCAM` - Version lives in param_value PARAM: `RAS.C.PARENT` - Gimbal Component ID 1 in the param_value

Monolithic Case Here we have all of the following capabilities presented by a fully monolithic component instead of via two separate components.

Example Generic Payload: Parachute

Example 3: Mavlink-controlled Chute Chute - Mavlink Controlled - Controlled Directly From A Flight Computer Component ID: `MAV_COMP_CHUTE` Features: Capability: `ON/OFF` Has 1 distinct elements: `Fuselage` 1 Generic payload heartbeats: `Chute with parent fuselage Chute`: Capability: `ON/OFF` Flight controller would know that it can send a MAV_CMD to this component ID based on its capability attributes discovered via parameter exchange.

Heartbeat 1 leads to discovery of the following parameters: PARAM: `RAS.C.DEPLOY` - Type lives in `param_value` (`BINARY`) PARAM: `RAS.C.RETRACT` - Supports the retraction command / message set (TBD) PARAM: `RAS.C.INTERLOCK` - Supports the interlock command / message set (TBD) PARAM: `RAS.C.ARM` - Supports the arming command / message set PARAM: `RAS.C.PARENT` - Parent component number lives in `param_value`

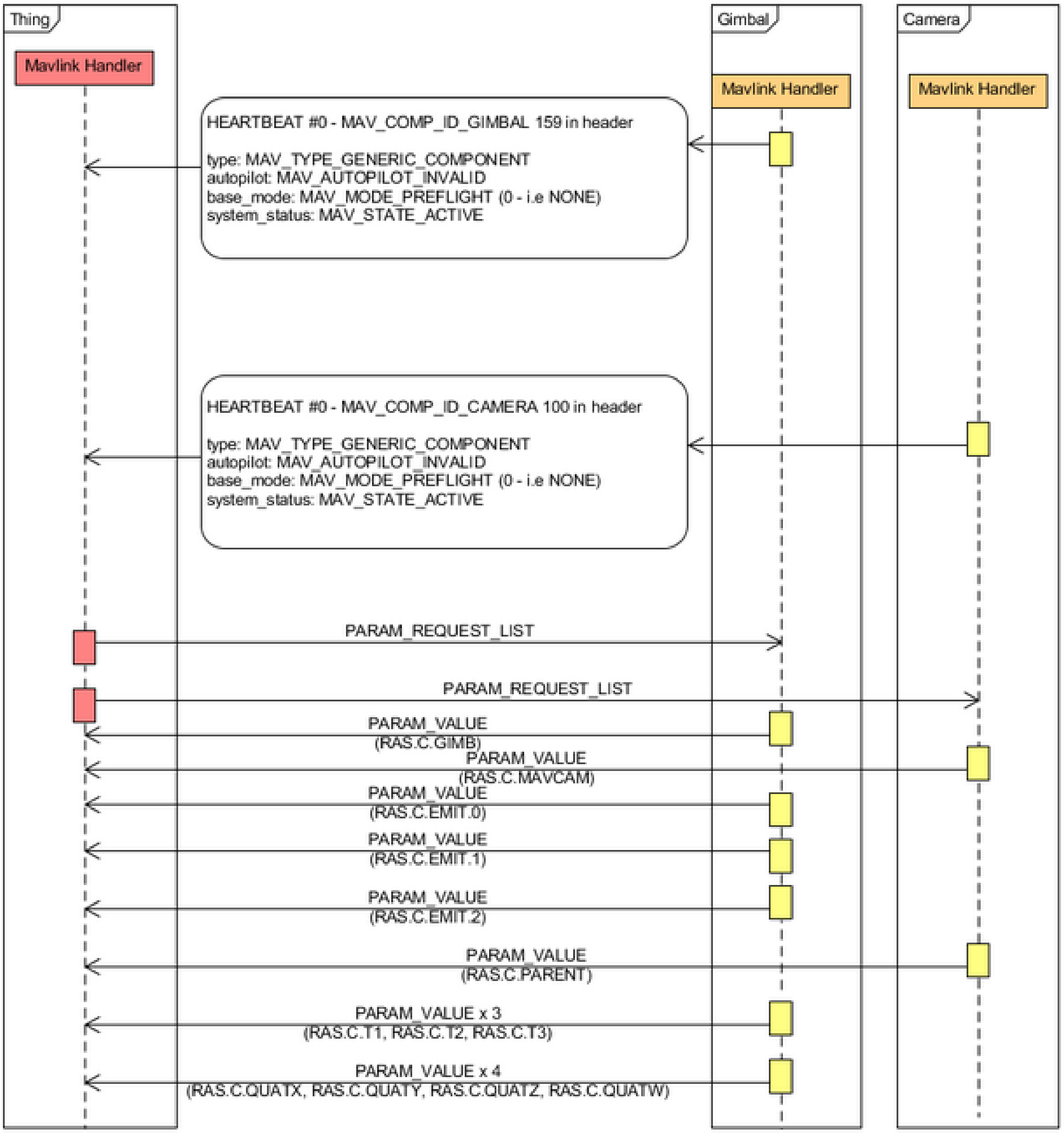

Figure 8: Parameter Exchange For Gimbal and Camera

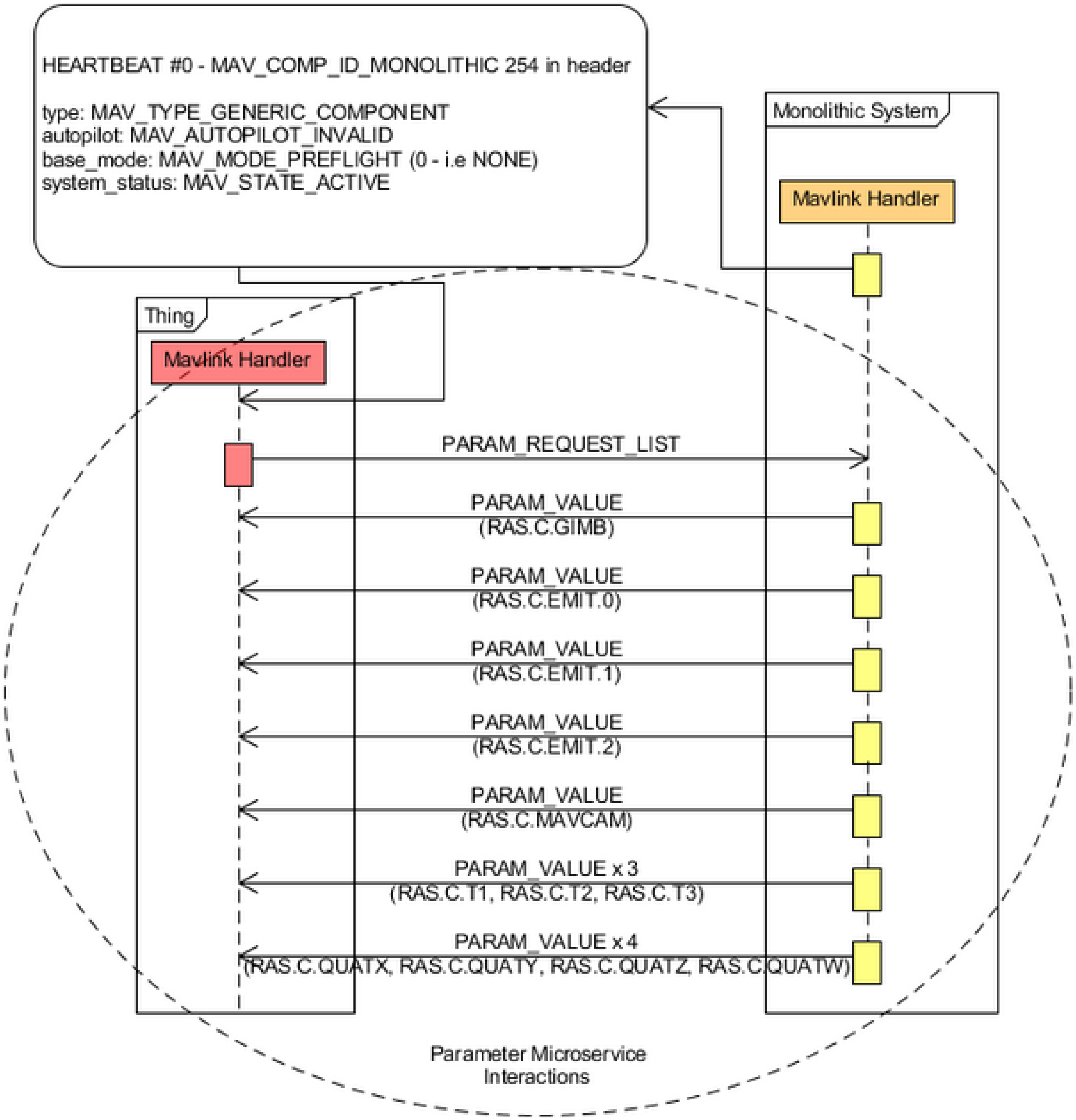

Figure 9: Parameter Exchange For Monolithic Gimbal and Camera

Telemetry

This section includes the required and recommended telemetry to be sent from the vehicle and to be processed by a GCS implementation that is compliant with this IOP. Note that although not clearly part of a microservice, the vehicle telemetry is considered a must in order to provide appropriate flight controller, sensors and peripheral status (like radios or batteries) to the operator.

General requirements and recommendations

The following table provides the required and recommended telemetry to be sent from the drone and/or its peripherals. Note that by default these telemetry messages come from the MAVLink component ID 1, which represents the flight controller component. "One-shot" messages should be requested using the `MAV_CMD_REQUEST_MESSAGE` command.

Message	Description	Required minimum rate (Hz)	Max rate (Hz)
Required			
ALTITUDE	The current system altitude. Should be sent when GLOBAL_POSITION_INT is not available to be sent.	5	10
ATTITUDE	The attitude of the vehicle in the aeronautical frame.	15	30
BATTERY_STATUS	Part of the Battery Protocol. Battery information. Updates GCS with flight controller battery status. Smart batteries also use this message	1	1
SMART_BATTERY_STATUS	Part of the Battery Protocol. Smart Battery information (static/infrequent update). Use for updates from: smart battery to flight stack, flight stack to GCS.	"One-shot"	"One-shot"
EXTENDED_SYS_STATE	Provides state for additional features. Required to provide the landing state to the GCS.	1	1
GLOBAL_POSITION_INT	The filtered global position (e.g. fused GPS and accelerometers). The position is in GPS-frame (right-handed, Z-up).	5	10
GPS_RAW_INT	The global position, as returned by the Global Positioning System (GPS).	5	5

Message	Description	Required minimum rate (Hz)	Max rate (Hz)
HEARTBEAT	The heartbeat message shows that a system or component is present and responding.	1	1
HOME_POSITION	Contains the home position. The home position is the default position that the system will return to and land on.	"One-shot"	"One-shot"
RADIO_STATUS	Status generated by radio and injected into MAVLink stream.	1	1
SYS_STATUS	The general system state.	1	1
Recommended			
ACTUATOR_OUTPUT_STATUS	The raw values of the actuator outputs.	5	20
ESTIMATOR_STATUS	Estimator status message including flags, innovation test ratios and estimated accuracies.	1	1
LOCAL_POSITION_NED	The filtered local position (e.g. fused computer vision and accelerometers). Coordinate frame is right-handed, Z-axis down.	5	30

Manual Control Protocol

The Manual Control Protocol enables controlling a system using a "standard joystick" (or joystick-like input device that supports the same axes nomenclature).

The protocol is implemented with just the MANUAL_CONTROL message. It defines the target system to be controlled, the movement in four primary axes (`x`, `y`, `z`, `r`) and two extension axes (`s`, `t`), and two 16-bit fields to represent the states of up to 32 buttons (`buttons`, `buttons2`). Unused axes can be disabled, and the extension axes must be explicitly enabled using bits 0 and 1 of the enabled_extensions field.

The protocol is by intent relatively simple and abstract, and provides a simple way of controlling the main motion of a vehicle, along with several arbitrary features that can be triggered using buttons.

This allows GCS software to provide a simple level of control for many types of vehicles, and allows new vehicle types with unusual functions to operate with minimal (if any) changes to the MAVLink protocol or existing GCS software.

Mapping Axes

Manual control is performed in the vehicle-frame. All axis values are normalized to the range -1000 to 1000.

Note: the GCS implementation might opt to normalize between 0 and 1000 for some of the axis. E.g. the Z-axis when used for throttle or altitude control), since in some flight control implementations, a negative value on this axis means a negative thrust values (used for example with rovers). **QGC Government Edition sends by default the throttle values normalized between 0 and 1000, unless the operator tips the “Allow negative Thrust” option on the Joystick configuration panel.**

Rotation-Focused Control The typical axis assignments for a thrust- and rotation-controlled vehicle (e.g. planes, multi-copters) are listed below.

Field	Motion Axis	+VE direction	-VE direction
x	pitch	forward/nose-down	backward/nose-up
y	roll	right-down	left-down
z	thrust	positive	negative
r	yaw	counter-clockwise	clockwise

Directional Control Vehicles with direct control over vehicle translation directions (multicopters) typically use the following mappings.

Field	Motion Axis	+VE direction	-VE direction
x	forward	forward	backward
y	lateral	right	left
z	vertical	up	down
r	yaw	counter-clockwise	clockwise
s	pitch	forward/nose-down	backward/nose-up
t	roll	right-down	left-down

Mapping Buttons

Button functions are vehicle/flight-stack dependent. The following are reference implementations that can be used as examples:

- ArduPilot treats button values as user-configurable using firmware parameters (e.g. ArduCopter's `BTN_FUNCn` or ArduSub's `BTNn_FUNCTION`), through the Parameter or Extended Parameter protocols.
- PX4 defines fixed meanings to some of the `buttons` values, and these are mapped to user-selected functions by the ground control station (applicable also to QGC-Gov).

The `buttons` field is required, and corresponds to the first 16 buttons.

`buttons2` is an extension, and corresponds to the optional second set of 16 buttons.

Alternatives

Vehicles may alternatively be controlled by sending information as a set of up to 18 channel values using RC_CHANNELS_OVERRIDE. Channels can be mapped to firmware parameters using PARAM_MAP_RC, and the autopilot can use the current parameter values at each point in time to determine control actions.

It's worth noting that the generality of RC channels control is a double-edged sword. It is incredibly versatile, and can be used to provide support for several arbitrary control axes, but the user-defined in-vehicle nature of the mapped parameters means additional setup is frequently required for compatibility with GCSs, and there are no guarantees that multiple vehicles running the same firmware will have the same channel-parameter mapping. This is a similar issue to the `MANUAL_CONTROL` buttons, so to minimize firmware complexity and maximize interoperability between a vehicle type and GCSs it's recommended to use targeted MAVLink commands where possible.

Note: Under this IOP, the supported way to manually control the vehicle through a joystick is using the MANUAL_CONTROL message.

Implementations

The protocol has been implemented in various GCSs and vehicle firmwares. These implementations can be used in your own code within the terms of their software licenses.

Ground Control Stations The protocol has been implemented in QGroundControl/QGC-Gov and Mission Planner. These can be used as reference implementations:

QGroundControl / QGC-Gov implementation

- src/Joystick/Joystick.cc (in `_handleAxis` method)

Note: QGC sends the MANUAL_CONTROL messages at a fixed rate of 30hz and *always* streams the message when a joystick is connected or the virtual joystick is enabled. This has implications on the implementation on the vehicle side, since it will have to verify if there were updates on the sticks to consider an update on the manual control of the vehicle.

***MissionPlanner* implementation**

- MainV2.cs (in `joysticksend` method)

Vehicle Firmwares The protocol has been implemented in PX4, and in the Copter, Plane, Rover, and Sub vehicles firmware in ArduPilot.

PX4 Implementation

- mavlink_receiver.cpp (in `handle_message_manual_control` method)

ArduPilot Implementations

- ArduCopter/GCS_Mavlink.cpp (in `handleMessage` method)
- ArduPlane/GCS_Mavlink.cpp (in `handleMessage` method)
- ArduRover/GCS_Mavlink.cpp (in `handle_manual_control` method)
- ArduSub/joystick.cpp (in `transform_manual_control_to_rc_override` method)

Future Extensions

Future extensions are likely to be handled with additional targeted MAVLink commands rather than mapping functionality in the flight controller (i.e. handling more complex inputs in the GCS to reduce vehicle firmware complexity).

Mission Protocol

The mission sub-protocol allows a GCS or developer API to exchange *mission* (flight plan), *geofence* and *safe point* information with a drone/component.

The protocol covers:

- Operations to upload, download and clear missions, set/get the current mission item number, and get notification when the current mission item has changed.
- Message type(s) and enumerations for exchanging mission items.
- Mission Items (“MAVLink commands”) that are common to most systems.

The protocol supports re-request of messages that have not arrived, which allows missions to be reliably transferred over a lossy link.

Mission Types

MAVLink 2 supports three types of "missions": flight plans, geofences and rally/safe points. The protocol uses the same sequence of operations for all types (albeit with different types of Mission Items). The mission types must be stored and handled separately/independently.

Mission protocol messages include the type of associated mission in the mission_type field (a MAVLink 2 message extension). The field takes one of the `MAV_MISSION_TYPE` enum values: `MAV_MISSION_TYPE_MISSION`, `MAV_MISSION_TYPE_FENCE`, `MAV_MISSION_TYPE_RALLY`.

MAVLink 1 supports only "regular" flight-plan missions (this is implied/not explicitly set).

Mission Items (MAVLink Commands)

Mission items for all the mission types are defined in the `MAV_CMD` enum.

`MAV_CMD` is used to define commands that can be used in missions ("mission items") and commands that can be sent outside of a mission context (using the Command Protocol). Some `MAV_CMD` can be used with both mission and command protocols. Not all commands/mission items are supported on all systems (or for all flight modes).

The items for the different types of mission are identified using a simple name prefix convention:

- *Flight plans*:
  - NAV commands (`MAV_CMD_NAV_*`) for navigation/movement (e.g. `MAV_CMD_NAV_WAYPOINT`, `MAV_CMD_NAV_LAND`)
  - DO commands (`MAV_CMD_DO_*`) for immediate actions like changing speed or activating a servo (e.g. `MAV_CMD_DO_CHANGE_SPEED`).
  - CONDITION commands (`MAV_CMD_CONDITION_*`) for changing the execution of the mission based on a condition - e.g. pausing the mission for a time before executing the next command (`MAV_CMD_CONDITION_DELAY`).
- *Geofence mission items*:
  - Prefixed with `MAV_CMD_NAV_FENCE_` (e.g. `MAV_CMD_NAV_FENCE_RETURN_POINT`).
- *Rally point mission items*:
  - There is just one rally point `MAV_CMD`: `MAV_CMD_NAV_RALLY_POINT`.

Mission items (`MAV_CMD`) are transmitted/encoded in `MISSION_ITEM_INT` messages. This message includes fields to identify the particular mission item (command id) and up to 7 command-specific optional parameters.

Field Name	Type	Values	Description
command	uint16_t	MAV_CMD	Command id, as defined in MAV_CMD.
param1	float		Param #1.
param2	float		Param #2.
param3	float		Param #3.
param4	float		Param #4.
param5 (x)	int32_t		X coordinate (local frame) or latitude (global frame) for navigation commands (otherwise Param #5).
param6 (y)	int32_t		Y coordinate (local frame) or longitude (global frame) for navigation commands (otherwise Param #6).
param7 (z)	float		Z coordinate (local frame) or altitude (global - relative or absolute, depending on frame) (otherwise Param #7).

Table 11:Command-specific Optional Parameters

The first four parameters (shown above) can be used for any purpose - this depends on the particular command. The last three parameters (x, y, z) are used for positional information in `MAV_CMD_NAV_*` commands, but can be used for any purpose in other commands.

The remaining message fields are used for addressing, defining the mission type, specifying the reference frame used for x, y, z in `MAV_CMD_NAV_*` messages, etc.:

Field Name	Type	Values	Description
target_system	uint8_t		*Deprecated. It is expected that the message only gets sent to the correct system by the underlying routing protocol*
target_component	uint8_t		Component ID
seq	uint16_t		Sequence number for item within mission (indexed from 0).
frame	uint8_t	MAV_FRAME	The coordinate system of the waypoint.
mission_type	uint8_t	MAV_MISSION_TYPE	Mission type.

Field Name	Type	Values	Description
`current`	`uint8_t`	false:0, true:1	When downloading, whether the item is the current mission item.
`autocontinue`	`uint8_t`		Autocontinue to the next waypoint when the command completes.

Table 12: Message Fields Description

Message/Enum Summary

The following messages and enums are used by the service.

Table 37: Messages

Message	Description
`MISSION_REQUEST_LIST`	Initiate mission download from a system by requesting the list of mission items.
`MISSION_COUNT`	Send the number of items in a mission. This is used to initiate mission upload or as a response to `MISSION_REQUEST_LIST` when downloading a mission.
`MISSION_REQUEST_INT`	Request mission item data for a specific sequence number be sent by the recipient using a `MISSION_ITEM_INT` message. Used for mission upload and download.
`MISSION_ITEM_INT`	Message encoding a mission item/command (defined in a `MAV_CMD`). Used for mission upload and download.
`MISSION_ACK`	Acknowledgment message when a system completes a mission operation (e.g. sent by autopilot after it has uploaded all mission items). The message includes a `MAV_MISSION_RESULT` indicating either success or the type of failure.
`MISSION_CURRENT`	Message containing the current mission item sequence number. This is emitted when the current mission item is set/changed.
`MISSION_SET_CURRENT`	Set the current mission item by sequence number (continue to this item on the shortest path).
`STATUSTEXT`	Sent to notify systems when a request to set the current mission item fails.
`MISSION_CLEAR_ALL`	Message sent to clear/delete all mission items stored on a system.

Message	Description
MISSION_ITEM_REACHED	Message emitted by system whenever it reaches a new waypoint. Used to monitor progress.

Table 38: Enum

Enum	Description
MAV_MISSION_TYPE	Mission type for message (mission, geofence, rally points).
MAV_MISSION_RESULT	Used to indicate the success or failure reason for an operation (e.g. to upload or download a mission). This is carried in a MISSION_ACK.
MAV_FRAME	Coordinate frame for position/velocity/acceleration data in the message.
MAV_CMD	Mission Items (and MAVLink commands) sent in MISSION_ITEM_INT.

Deprecated Types: MISSION_ITEM

The legacy version of the protocol also supported MISSION_REQUEST for requesting that a mission be sent as a sequence of MISSION_ITEM messages.

Both MISSION_REQUEST and MISSION_ITEM messages are now deprecated, and should no longer be sent. If MISSION_REQUEST is received the system should instead respond with MISSION_ITEM_INT items (as though it received MISSION_REQUEST_INT).

Frames & Positional Information

By convention, mission items use param5, param6, param7 for positional information when needed (and otherwise as “free use” parameters). The table below shows that the positional parameters can be local (x, y, z), global (latitude, longitude, altitude), and also the data type used to store the parameters in the MISSION_ITEM_INT message.

Table 39: Positional Parameters

param	type	Local	Global
param5	int32_t	x	Latitude
param6	int32_t	y	Longitude
param7	float	z	Altitude (global - relative or absolute)

The coordinate frame of positional parameters is defined in the MISSION_ITEM_INT.frame field using a `MAV_FRAME` value.

The global frames are prefixed with `MAV_FRAME_GLOBAL_*`. Mission items should use frame variants that have the suffix `_INT`: e.g. `MAV_FRAME_GLOBAL_RELATIVE_ALT_INT`, `MAV_FRAME_GLOBAL_INT`, `MAV_FRAME_GLOBAL_TERRAIN_ALT_INT`. When using these frames, latitude and longitude values must be encoded by multiplying the degrees by 1E7 (e.g. the latitude 69.69000000 would be sent as 69.69000000x1E7 = 696900000). Using int32 of degrees `* 10^7` has higher resolution than could be achieved with a single floating point.

A number of local frames are also specified. Local frame position values that are sent in integer field parameters must be encoded as *position in meters x 1E4* (e.g. 5m would be encoded and sent as 50000). If sent in messages float parameter fields the value should be sent as-is.

Don't use the non-INT global frames in mission items (e.g. `MAV_FRAME_GLOBAL_RELATIVE_ALT`). These are intended to be used with messages that have float fields for positional information, e.g.: `MISSION_ITEM` (deprecated), `COMMAND_LONG`. If these frames are used, position values should be sent unencoded (i.e. no need to multiply by 1E7).

As above, in theory if a global non-INT frame variant is set for a `MISSION_ITEM_INT` the position value should be sent as-is (not encoded). This will result in the value being rounded when it is sent in the integer value, which will make the value unusable. In practice, many systems will assume you have encoded the value, but you should test this for your particular flight stack. Better just to use the correct frames!

Don't use `MAV_FRAME_MISSION` for mission items that contain positional data; this does not correspond to any particular real frame, and so will be ambiguous. `MAV_FRAME_MISSION` should be used for mission items that use `params5` and `param6` for other purposes.

Param 5, 6 For Non-Positional Data

Param5, param6, param7 may also be used for non-positional information. In this case the `MISSION_ITEM_INT.frame` should be set to `MAV_FRAME_MISSION` (this is equivalent to say "the frame data is irrelevant").

As param5 and param6 are sent in *integer* fields, generally you should design mission items/MAV_CMDs such that these only include integer data (and are sent as-is/unscaled). If these must be used for real numbers and scaling is required, then this must be noted in the mission item itself.

Operations

This section defines all the protocol operations.

Upload a Mission to the Vehicle Uploading a mission can be performed from a ground control station or from a different device / vehicle. The diagram below shows the communication sequence to upload a mission to a drone (assuming all operations succeed).

Mission update must be robust! A new mission should be fully uploaded and accepted before the old mission is replaced/removed.

In more detail, the sequence of operations is:

1. GCS sends `MISSION_COUNT` including the number of mission items to be uploaded (count).
   - A timeout must be started for the GCS to wait on the response from Drone (`MISSION_REQUEST_INT`).
2. Drone receives a message and responds with `MISSION_REQUEST_INT` requesting the first mission item (seq==0).
   - A timeout must be started for the Drone to wait on the MISSION_ITEM_INT response from GCS.
3. GCS receives `MISSION_REQUEST_INT` and responds with the requested mission item in a `MISSION_ITEM_INT` message.
4. Drone and GCS repeat the `MISSION_REQUEST_INT/MISSION_ITEM_INT` cycle, iterating seq until all items are uploaded (`seq==count-1`).
5. After receiving the last mission item the drone responds with `MISSION_ACK` with the type of `MAV_MISSION_ACCEPTED` indicating mission upload completion/success.
   - The drone should set the new mission to be the current mission, discarding the original data.
   - The drone considers the upload complete.
6. GCS receives `MISSION_ACK` containing `MAV_MISSION_ACCEPTED` to indicate the operation is complete.

Note:

- A timeout is set for every message that requires a response (e.g. `MISSION_REQUEST_INT`). If the timeout expires without a response being received then the request must be resent.
- Mission items must be received in order. If an item is received out-of-sequence the expected item should be re-requested by the vehicle (the out-of-sequence item is dropped).

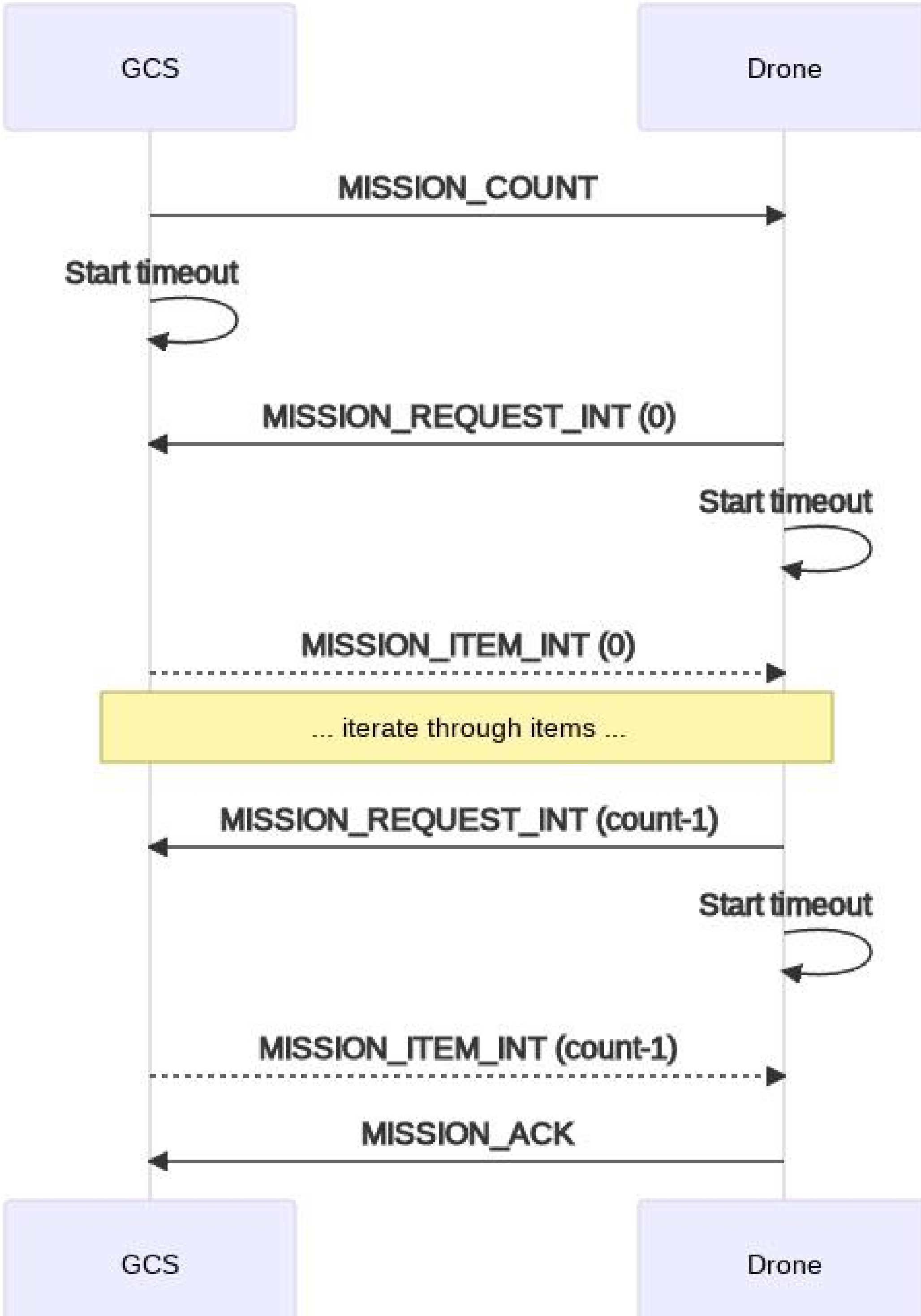

Figure 10: Mission Upload Communication Sequence Diagram

- An error can be signaled in response to any request using a MISSION_ACK message containing an error code. This must cancel the operation and restore the mission to its previous state. For example, the drone might respond to the MISSION_COUNT request with a MAV_MISSION_NO_SPACE if there isn't enough space to upload the mission.
- The sequence above shows the mission items packaged in MISSION_ITEM_INT messages. Protocol implementations must also support MISSION_ITEM and MISSION_REQUEST in the same way.
- Uploading an empty mission (MISSION_COUNT is 0) has the same effect as clearing the mission.

Download a Mission from the Vehicle The diagram below shows the communication sequence to download a mission from a drone (assuming all operations succeed).

The sequence is similar to that for uploading a mission. The main difference is that the client (e.g. GCS) sends MISSION_REQUEST_LIST, which triggers the autopilot to respond with the current count of items. This starts a cycle where the GCS requests mission items, and the drone supplies them.

Note:

- A timeout is set for every message that requires a response (e.g. MISSION_REQUEST_INT). If the timeout expires without a response being received then the request must be resent.
- Mission items must be received in order. If an item is received out-of-sequence the expected item should be re-requested by the GCS (the out-of-sequence item is dropped).
- An error can be signaled in response to any request using a MISSION_ACK message containing an error code. This must cancel the operation.
- The sequence above shows the mission items packaged in MISSION_ITEM_INT messages. Protocol implementations must also support MISSION_ITEM and MISSION_REQUEST in the same way.

Set Current Mission Item The diagram below shows the communication sequence to set the current mission item.

In more detail, the sequence of operations is:

1. GCS/App sends MISSION_SET_CURRENT, specifying the new sequence number (seq).
2. Drone receives a message and attempts to update the current mission sequence number.
   - On success, the Drone must *broadcast* a MISSION_CURRENT message containing the current sequence number (seq).

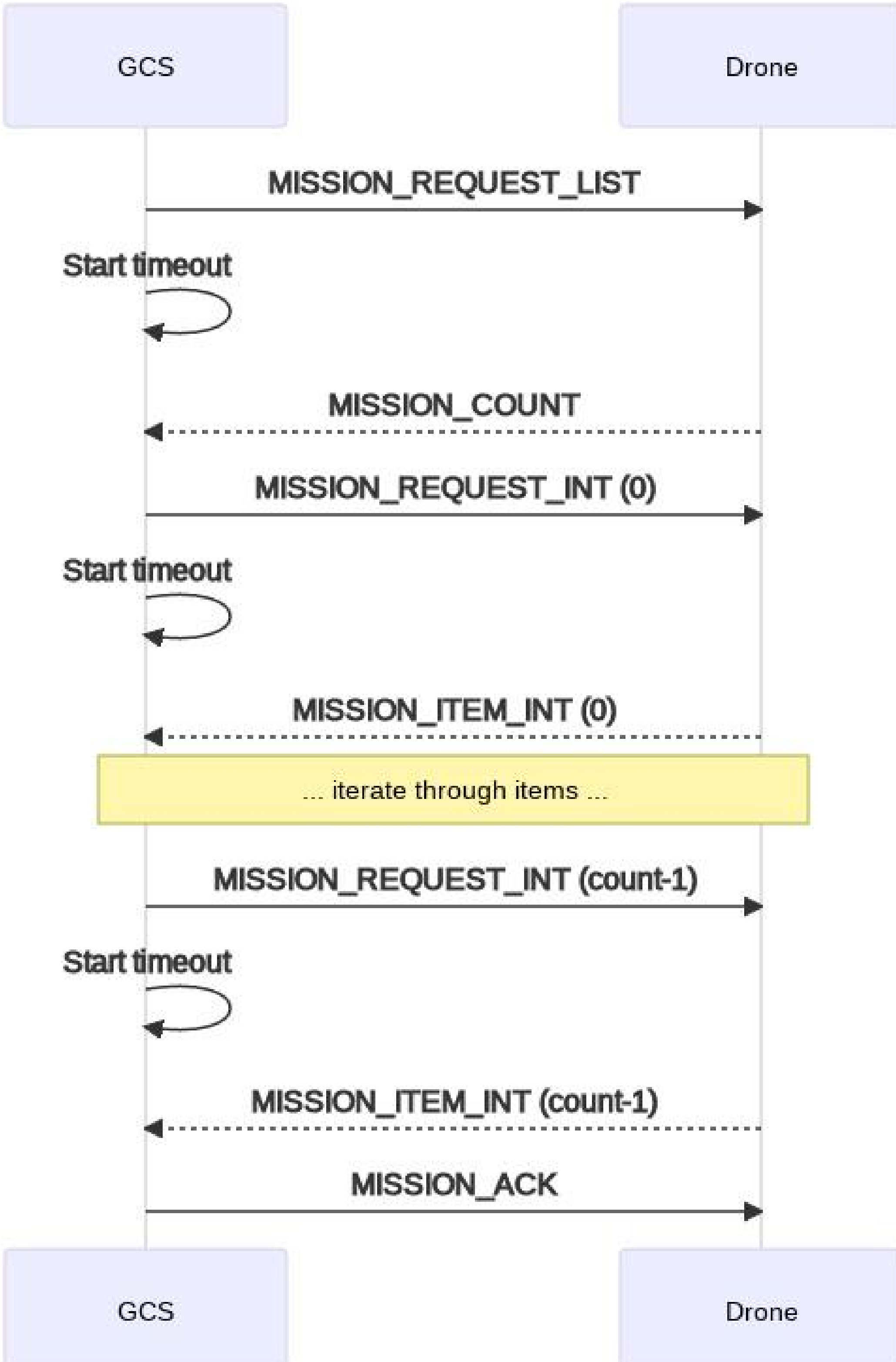

Figure 11: Mission Download Communication Sequence Diagram

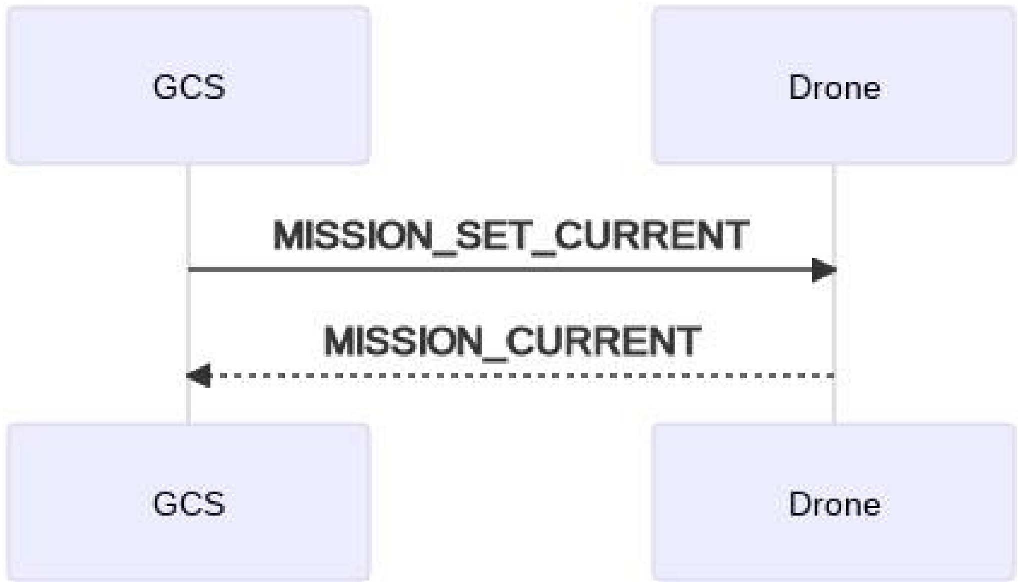

Figure 12: Mission Setting Communication Sequence Diagram

- On failure, the Drone must *broadcast* a `STATUSTEXT` with a `MAV_SEVERITY` and a string stating the problem. This may be displayed in the UI of receiving systems.

Notes:

- There is no specific timeout on the `MISSION_SET_CURRENT` message.
- The acknowledgment of the message is via broadcast of mission/system status, which is not associated with the original message. This differs from error handling in other operations. This approach is used because the success/failure is relevant to all mission-handling clients.

Monitor Mission Progress GCS/developer API can monitor progress by handling the appropriate messages sent by the drone:

- The vehicle must broadcast a `MISSION_ITEM_REACHED`) message whenever a new mission item is reached. The message contains the seq number of the current mission item.
- The vehicle must also broadcast a `MISSION_CURRENT` message if the current mission item is changed.

Clear Missions The diagram below shows the communication sequence to clear the mission from a drone (assuming all operations succeed).

In more detail, the sequence of operations is:

1. GCS/API sends `MISSION_CLEAR_ALL`

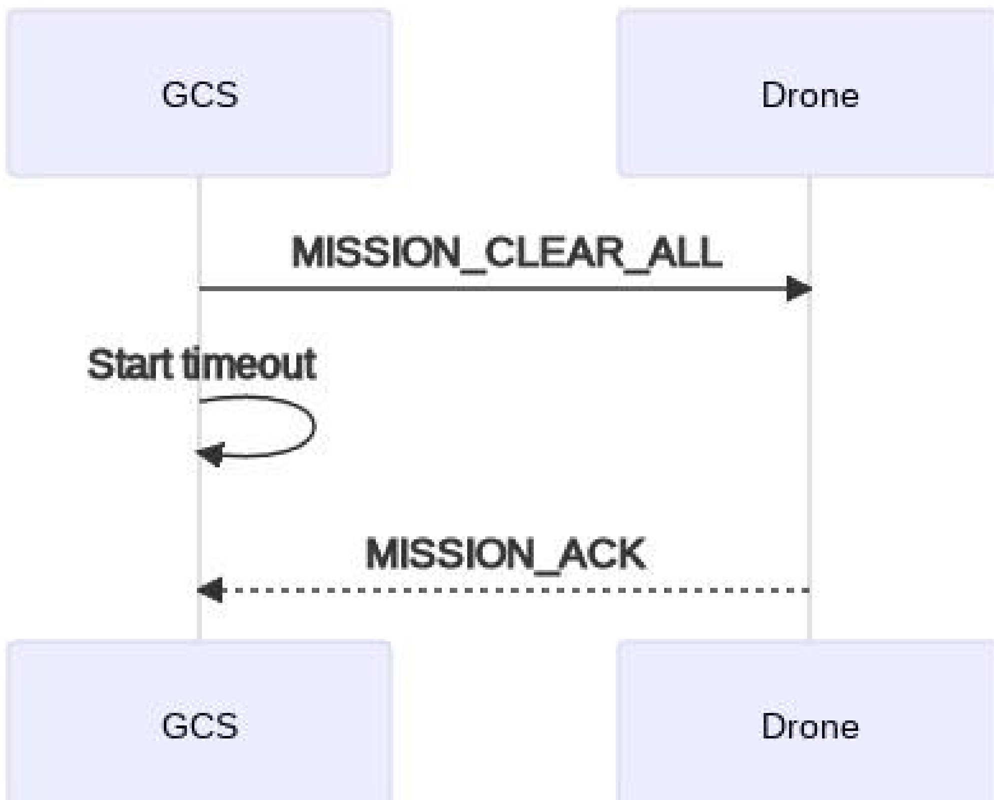

Figure 13: Mission Clearance Communication Sequence Diagram

  - A timeout is started for the GCS to wait on MISSION_ACK from Drone.

2. Drone receives the message, and clears the mission from storage.
3. Drone responds with `MISSION_ACK` with the result type of `MAV_MISSION_ACCEPTED` `MAV_MISSION_RESULT`.
4. GCS receives MISSION_ACK and clears its own stored information about the mission. The operation is now complete.

Note:

- A timeout is set for every message that requires a response (e.g. `MISSION_CLEAR_ALL`). If the timeout expires without a response being received then the request must be resent.
- An error can be signaled in response to any request (in this case, just `MISSION_CLEAR_ALL`) using a `MISSION_ACK` message containing an error code. This must cancel the operation. The GCS record of the mission (if any) should be retained.

Canceling Operations The above mission operations may be canceled by responding to any request (e.g. `MISSION_REQUEST_INT`) with a `MISSION_ACK` message containing the `MAV_MISSION_OPERATION_CANCELLED` error.

Both systems should then return themselves to the idle state (if the system does not receive the cancellation message it will resend the request; the recipient will then be in the idle state and may respond with an appropriate error for that state).

Operation Exceptions

Timeouts and Retries A timeout should be set for all messages that require a response. If the expected response is not received before the timeout then the message must be resent. If no response is received after a number of retries then the client must cancel the operation and return to an idle state.

The recommended timeout values before resending, and the number of retries are:

- Timeout (default): 1500 ms
- Timeout (mission items): 250 ms.
- Retries (max): 5

Errors/Completion All operations complete with a `MISSION_ACK` message containing the result of the operation (`MAV_MISSION_RESULT`) in the type field.

On successful completion, the message must contain type of MAV_MISSION_ACCEPTED; this is sent by the system that is receiving the command/data (e.g. the drone for mission upload or the GCS for mission download).

An operation may also complete with an error - `MISSION_ACK.type` set to MAV_MISSION_ERROR or some other error code in MAV_MISSION_RESULT. This can occur in response to any message/anywhere in the sequence.

Errors are considered unrecoverable. If an error is sent, both ends of the system should reset themselves to the idle state and the current state of the mission on the vehicle should be unaltered.

Note:

- timeout are not considered errors.
- Out-of-sequence messages in mission upload/download are recoverable, and are not treated as errors.

Mission File Formats

The *de facto* standard file format for exchanging missions/plans is discussed in: File Formats > Mission Plain-Text File Format.

Mission Command Detail

This section is for clarifications and additional information about common mission items. In particular it is intended for cases that are difficult to document in the specification XML, or when images will much better describe expected behavior.

Loiter Commands (`MAV_CMD_NAV_LOITER_*`) Loiter commands are provided to allow a vehicle to hold at a location for a specified time or number of turns, until it reaches the specified altitude, or indefinitely. Multicopter vehicles stop at the specified point (within a *vehicle-specific* acceptance radius that is not set by the mission item). Forward-moving vehicles (e.g. fixed-wing) *circle* the point with the specified radius/direction.

The commands are:

- MAV_CMD_NAV_LOITER_TIME - Loiter at specified location for a given amount of time after reaching the location.
- MAV_CMD_NAV_LOITER_TURNS - Loiter at specified location for a given number of turns.
- MAV_CMD_NAV_LOITER_TO_ALT - Loiter at specified location until desired altitude is reached.
- MAV_CMD_NAV_LOITER_UNLIM - Loiter at specified location for an unlimited amount of time, yawing to face a given direction.

The location and fixed-wing loiter radius parameters are common to all commands:

Table 40: Fixed-wing Loiter and Location Parameters I

Param	Description	Units
3: Radius	Radius around the waypoint. If positive loiter clockwise, else counter-clockwise	m
5: Latitude	Latitude	
6: Longitude	Longitude	
7: Altitude	Altitude	m

The loiter time and turns are set in param 1 for the respective messages. The direction of the loiter for `MAV_CMD_NAV_LOITER_UNLIM` can be set using `param4` (Yaw).

The remaining parameters (xtrack and heading) apply only to forward flying aircraft (not multicopters!)

Xtrack and heading define the location at which a forward flying (fixed wing) vehicle will *exit the loiter circle, and its path to the next waypoint* (these apply only to apply to only `MAV_CMD_NAV_LOITER_TIME` and `MAV_CMD_NAV_LOITER_TURNS`).

Table 41: Fixed-wing Loiter and Location Parameters II

Param	Description	Units
2: Heading Required	Leave loiter circle only once heading towards the next waypoint (0 = False)	min:0. max:1, increment:1
4: Xtrack Location	Sets xtrack path or exit location: 0 for the vehicle to converge towards the center xtrack when it leaves the loiter (the line between the centers of the current and next waypoint), 1 to converge to the direct line between the location that the vehicle exits the loiter radius and the next waypoint. Otherwise the angle (in degrees) between the tangent of the loiter circle and the center xtrack at which the vehicle must leave the loiter (and converge to the center xtrack). NaN to use the current system default xtrack behavior.	

The recommended values (and resulting paths) are those shown below.

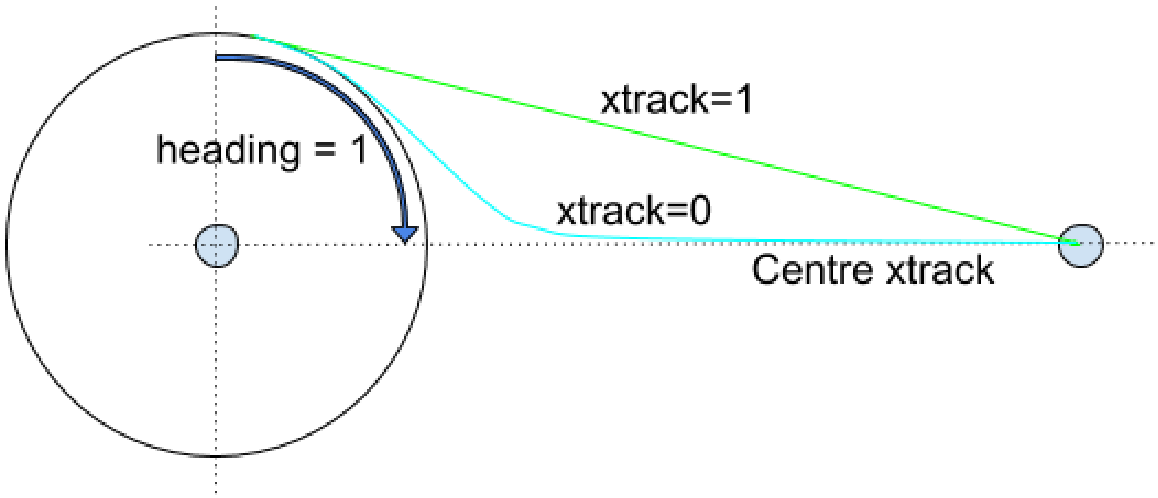

Figure 14: Fixed-wing Loiter Configuration Recommendation I

The vehicle leaves the loiter after it reaches the desired number of turns or time *and* based on **both** the heading required and xtrack params.

A heading required of 1 prevents the vehicle from exiting the loiter unless it is heading towards the next waypoint (if 0 it can leave at any point provided the other conditions are met). With this setting the vehicle can leave at any point in the arc shown, provided it meets the other conditions (e.g. xtrack). If necessary (i.e. it is not in the arc when the other conditions are met), the vehicle will loop back around the loiter before it evaluates the xtrack condition.

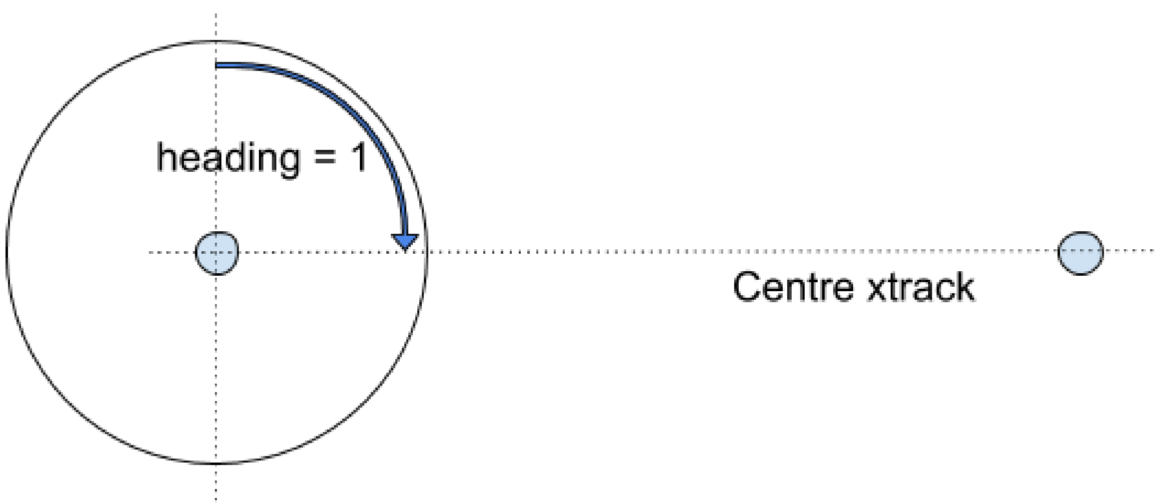

Figure 15: Fixed-wing Loiter Configuration Recommendation II

The Xtrack parameter independently defines the path and exit location:

- xtrack=0: Exit the loiter circle and converge to the centre xtrack between this and the next waypoint.

  - If the heading required parameter is not set it will exit the loiter immediately.
  - Otherwise it will leave as soon as it is heading towards the next waypoint (which may also be immediately!)

- xtrack=1: Exit the loiter circle and fly/converge to the straight line between the exit point and the centre of the next waypoint (i.e. don't converge to the centre xtrack).
  - If the heading required parameter is set it will exit the loiter as soon as it is heading towards the next waypoint (which may be immediately!).
  - If the heading required parameter is not set it will exit the loiter immediately (note that this exit path does not make much sense unless the heading parameter is set).
- xtrack=NaN: Exit the loiter using "system specific default behavior".
  - The vehicle must still respect the heading required param.
  - Usually this is synonymous with xtrack=0
- xtrack=any other value: Exit the loiter when the vehicle heading (tangent) makes the specified angle in degrees to the center xtrack. Converge to the center xtrack. The vehicle must still respect the heading required param (some xtrack values may not be possible with this condition true). This allows callers to specify how quickly the vehicle converges to the center xtrack. For example, the image below shows the vehicle exiting the loiter at 30 degrees.

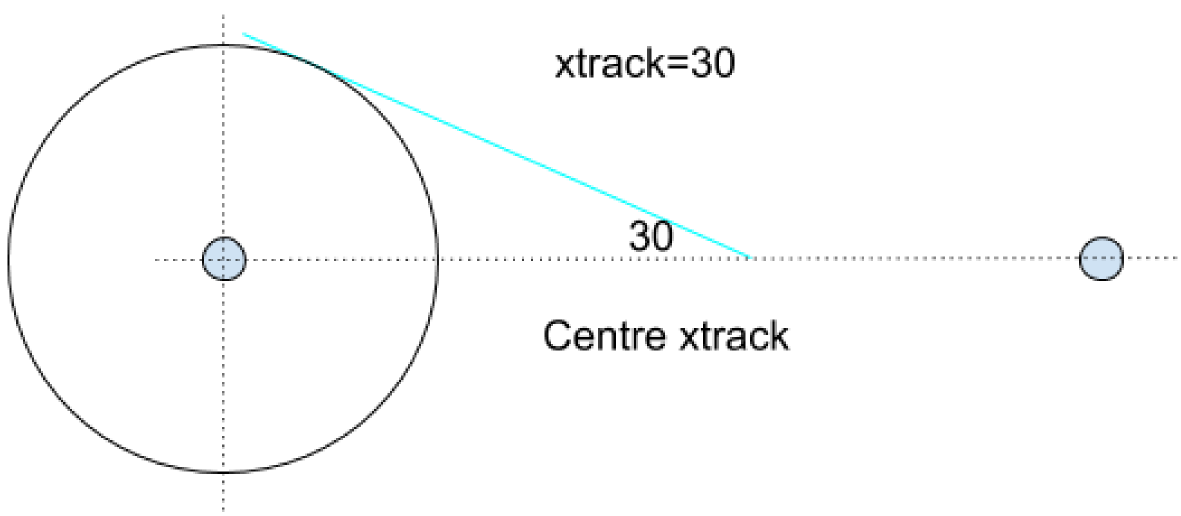

Figure 16: Fixed-wing Loiter Configuration Recommendation III

Plan File Format

This file format is not part of the MAVLink transfer, but is standardized as a ground or vehicle side storage format. Future versions of the protocol might adopt a file-based mission transfer which allows direct uploads in this form. Plan files are stored in JSON file format and contain mission items and (optional) geofence and rally-points. Below you can see the top level format of a Plan file.

This is "near-minimal" - a plan must contain at least one mission item. The plan fence and rally points are also used in modes when no mission is running.

Plan File Format example:

```json
{
    "fileType": "Plan",
    "geoFence": {
        "circles": [
        ],
        "polygons": [
        ],
        "version": 2
    },
    "groundStation": "QGroundControl",
    "mission": {
    },
    "rallyPoints": {
        "points": [
        ],
        "version": 2
    },
    "version": 1
}
```

The main fields are:

Table 42: Plan File Main Fields Description

Key	Description
`version`	The version for this file. Current version is 1.
`fileType`	Must be "Plan".
`groundStation`	The name of the ground station which created this file (here *QGroundControl*)
`mission`	The mission associated with this flight plan.
`geoFence`	(Optional) Geofence information for this plan.
`rallyPoints`	(Optional) Rally/Safe point information for this plan.

Mission Object The structure of the mission object is shown below. The items field contains a comma-separated list of mission items (it must contain at least one mission item, as shown below). The list may be a mix of both SimpleItem and ComplexItem objects.

Mission Object Structure example:

```
"mission": {
    "cruiseSpeed": 15,
    "firmwareType": 12,
    "hoverSpeed": 5,
    "items": [
        {
            "AMSLAltAboveTerrain": null,
            "Altitude": 50,
            "AltitudeMode": 0,
            "autoContinue": true,
            "command": 22,
            "doJumpId": 1,
            "frame": 3,
            "params": [
                15,
                0,
                0,
                null,
                47.3985099,
                8.5451002,
                50
            ],
            "type": "SimpleItem"
        }
    ],
    "plannedHomePosition": [
        47.3977419,
        8.545594,
        487.989
    ],
    "vehicleType": 2,
    "version": 2
},
```

The following values are required:

Table 43: Required Parameter Values for Mission Object

Key	**Description**
version	The version for the mission object. Current version is 2.
firmwareType	The firmware type for which this mission was created. This is one of the MAV_AUTOPILOT enum values.
vehicleType	The vehicle type for which this mission was created. This is one of the MAV_TYPE enum values.
cruiseSpeed	The default forward speed for Fixed wing or VTOL vehicles (i.e. when moving between waypoints).
hoverSpeed	The default forward speed for multi-rotor vehicles.
items	The list of mission item objects associated with the mission . The list may contain either/both SimpleItem and ComplexItem objects.
plannedHomePosition	The planned home position is shown on the map and used for mission planning when no vehicle is connected. The array values shown above are (from top): latitude, longitude and AMSL altitude.

The format of the simple and complex items is given below.

SimpleItem - Simple Mission Item A simple item represents a single MAVLink MISSION_ITEM command.

Simple Item Definition example:

```
{
     "AMSLAltAboveTerrain": null,
     "Altitude": 50,
     "AltitudeMode": 0,
     "autoContinue": true,
     "command": 22,
     "doJumpId": 1,
     "frame": 3,
     "params": [
         15,
         0,
         0,
         null,
         47.3985099,
         8.5451002,
         50
```

```
                ],
                "type": "SimpleItem"
            }
```

The field mapping is shown below.

Table 44: Simple Item Field Mapping

Key	Description
`type`	`SimpleItem` for a simple item
`AMSLAltAboveTerrain`	Altitude value shown to the user.
`Altitude`	
`AltitudeMode`	
`autoContinue`	`MISSION_ITEM.autoContinue`
`command`	The command (`MAV_CMD`) for this mission item - see `MISSION_ITEM.command`.
`doJumpId`	The target id for the current mission item in DO_JUMP commands. These are auto-numbered from 1.
`frame`	`MAV_FRAME` (see `MISSION_ITEM.frame`)
`params`	`MISSION_ITEM`.param1,2,3,4,x,y,z (values depend on the particular `MAV_CMD`).

Complex Mission Items A complex item is a higher level encapsulation of multiple `MISSION_ITEM` objects treated as a single entity.

There are currently three types of complex mission items:

- Survey
- Corridor Scan
- Structure Scan

Survey Scan Complex Mission Item

The object definition for a Survey complex mission item is given below.

```
{
                "TransectStyleComplexItem": {
                    ...
                },
                "angle": 0,
                "complexItemType": "survey",
                "entryLocation": 0,
                "flyAlternateTransects": false,
                "polygon": [
```

```
                    [
                        -37.75170619863631,
                        144.98414811224316
                    ],
                    ...
                    [
                        -37.75170619863631,
                        144.99457681259048
                    ]
                ],
                "type": "ComplexItem",
                "version": 4
            },
```

Complex items have these values associated with them:

Table 45: Survey Complex Mission Key Values

Key	Description
`version`	The version number for this survey definition. Current version is 3.
`type`	ComplexItem (this is a complex item).
`complexItemType`	survey
`TransectStyleComplexItem`	The common base definition for `Survey` and `CorridorScan` complex items.
`angle`	The angle for the transect paths (degrees).
`entryLocation`	?
`flyAlternateTransects`	If true, the vehicle will skip every other transect and then come back at the end and fly these alternates. This can be used for fixed wing aircraft when the turnaround would be too acute for the vehicle to make the turn.
`polygon`	The polygon array which represents the polygonal survey area. Each point is a latitude, longitude pair for a polygon vertex.

Corridor Scan Complex Mission Item
The object definition for a `CorridorScan` complex mission item is given below.

```
            {
                "CorridorWidth": 50,
                "EntryPoint": 0,
                "TransectStyleComplexItem": {
                    ...
```

```
            },
        },
        "complexItemType": "CorridorScan",
        "polyline": [
            [
                -37.75234887156983,
                144.9893624624168
            ],
            ...
            [
                -37.75491914850321,
                144.9893624624168
            ]
        ],
        "type": "ComplexItem",
        "version": 2
    },
```


Table 46: CorridorScan Complex Mission Key Values

Key	Description
`version`	The version for this CorridorScan definition. Current version is 3.
`type`	ComplexItem (this is a complex item).
`complexItemType`	CorridorScan
`CorridorWidth`	?
`EntryPoint`	?
`TransectStyleComplexItem`	The common base definition for `Survey` and `CorridorScan` complex items.
`polyline`	?

Structure Scan Complex Mission Item
The object definition for a `StructureScan` complex mission item is given below.


```
    {
        "Altitude": 50,
        "CameraCalc": {
            "AdjustedFootprintFrontal": 25,
            "AdjustedFootprintSide": 25,
            "CameraName": "Manual (no camera specs)",
            "DistanceToSurface": 10,
            "DistanceToSurfaceRelative": true,
```

```
                    "version": 1
                },
                "Layers": 1,
                "StructureHeight": 25,
                "altitudeRelative": true,
                "complexItemType": "StructureScan",
                "polygon": [
                    [
                        -37.753184359536355,
                        144.98879374063998
                    ],
                    ...
                    [
                        -37.75408368012594,
                        144.98879374063998
                    ]
                ],
                "type": "ComplexItem",
                "version": 2
            }
```

Table 47: StructureScan Complex Mission Key Values

Key	Description
version	The version for this StructureScan definition. Current version is 2.
type	ComplexItem (this is a complex item).
complexItemType	StructureScan
Altitude	?
CameraCalc	?
Layers	?
StructureHeight	?
altitudeRelative	true: altitude is relative to home, false: altitude is AMSL.
polygon	?

TransectStyleComplexItem

TransectStyleComplexItem contains the common base definition for survey and CorridorScan complex items.

The object definition for a TransectStyleComplexItem complex mission item is given below.

```
"TransectStyleComplexItem": {
    "CameraCalc": {
        ...
    },
    "CameraTriggerInTurnAround": true,
    "FollowTerrain": false,
    "HoverAndCapture": false,
    "Items": [
        ...
    ],
    "Refly90Degrees": false,
    "TurnAroundDistance": 10,
    "VisualTransectPoints": [
        [
            -37.75161626657736,
            144.98414811224316
        ],
        ...
        [
            -37.75565155437309,
            144.99438539496475
        ]
    ],
    "version": 1
},
```


Table 48: TransectStyle Complex Mission Key Values

Key	**Description**
`version`	The version for this `TransectStyleComplexItem` definition. Current version is 1.
`CameraCalc`	?
`CameraTriggerIn-TurnAround`	? (boolean)
`FollowTerrain`	? (boolean)
`HoverAndCapture`	? (boolean)
`Items`	?
`Refly90Degrees`	? (boolean)
`TurnAroundDistance`	The distance to fly past the polygon edge prior to turning for the next transect.
`VisualTransectPoints`	?

CameraCalc Complex Mission Item

The `CameraCalc` complex mission item contains camera information used for a survey, corridor or structure scan.

The object definition for a `CameraCalc` complex mission item is given below:

```
"CameraCalc": {
    "AdjustedFootprintFrontal": 272.4,
    "AdjustedFootprintSide": 409.2,
    "CameraName": "Sony ILCE-QX1",
    "DistanceToSurface": 940.6896551724138,
    "DistanceToSurfaceRelative": true,
    "FixedOrientation": false,
    "FocalLength": 16,
    "FrontalOverlap": 70,
    "ImageDensity": 25,
    "ImageHeight": 3632,
    "ImageWidth": 5456,
    "Landscape": true,
    "MinTriggerInterval": 0,
    "SensorHeight": 15.4,
    "SensorWidth": 23.2,
    "SideOverlap": 70,
    "ValueSetIsDistance": false,
    "version": 1
},
```

Key	Description
`version`	The version for this `CameraCalc` definition. Current version is 1.
`AdjustedFootprintFrontal`	?
`AdjustedFootprintSide`	?
`DistanceToSurface`	? Units?
`DistanceToSurfaceRelative`	?
`CameraName`	Name of camera being used (must correspond to one of the cameras known to *QGroundControl* or: Manual (no camera specs) for manual setup, Custom Camera for a custom setup. The keys listed after this point are not specified for a “Manual” camera definition.
`FixedOrientation`	? (boolean)
`FocalLength`	Focal length of camera lens in millimeters.
`FrontalOverlap`	Percentage of frontal image overlap.
`ImageDensity`	?

Key	**Description**
`ImageHeight`	Image height in px
`ImageWidth`	Image width in px
`Landscape`	true: Camera installed in landscape orientation on vehicle, false: Camera installed in portrait orientation on vehicle.
`MinTriggerInterval`	?
`SensorHeight`	Sensor height in millimeters.
`SensorWidth`	Sensor width in millimeters.
`SideOverlap`	Percentage of side image overlap.
`ValueSetIsDistance`	? (boolean)

Table 25: CameraCalc Complex Mission Item Key Values

GeoFence Geofence information is optional. The plan can contain an arbitrary number of geofences defined in terms of polygons and circles.

A minimal GeoFence definition is given below:

```
  "geoFence": {
       "circles": [
       ],
       "polygons": [
       ],
       "version": 2
   },
```

The fields are:

Key	**Description**
`version`	The version number for the geofence plan format. The documented version is 2.
`circles`	List containing circle geofence definitions (comma separated).
`polygons`	List containing polygon geofence definitions (comma separated).

Table 26: GeoFence Key Values

Circle Geofence Each circular geofence is defined in a separate item, as shown below (multiple comma-separated items can be defined). The items define the centre and radius of the circle, and whether or not the specific geofence is activated.

Circle GeoFence Definition:

```
{
    "circle": {
        "center": [
            47.39756763610029,
            8.544649762407738
        ],
        "radius": 319.85
    },
    "inclusion": true,
    "version": 1
}
```

The fields are:

Key	Description
version	The version number for the geofence "circle" plan format. The documented version is 1.
circle	The definition of the circle. Includes centre (latitude, longitude) and radius as shown above.
inclusion	Whether or not the geofence is enabled (true) or disabled.

Table 27: Circle GeoFence Key Values

Polygon Geofence Each polygon geofence is defined in a separate item, as shown below (multiple comma-separated items can be defined). The geofence includes a set of points defined with a clockwise winding (i.e. they must enclose an area).

The object definition for a polygon geofence is given below:

```
{
    "inclusion": true,
    "polygon": [
        [
            47.39807773798406,
            8.543834631785785
        ],
        [
            47.39983519888905,
            8.550024648373267
        ],
        [
```

```
                        47.39641100087146,
                        8.54499282423751
                    ],
                    [
                        47.395590322265186,
                        8.539435808992085
                    ]
                ],
                "version": 1
            }
        ],
        "version": 2
    }
```

The fields are:

Table 52: Polygon GeoFence Key Values

Key	Description
`version`	The version number for the geofence "polygon" plan format. The documented version is 2.
`polygon`	A list of points for the polygon. Each point contains a latitude and longitude. The points are ordered in a clockwise winding.
`inclusion`	Whether or not the geofence is enabled (true) or disabled.
`inclusion`	Whether or not the geofence is enabled (true) or disabled.

Rally Points Rally point information is optional. The plan can contain an arbitrary number of rally points, each of which has a latitude, longitude, and altitude (above home position).

The object definition for two rallypoints is given below:

```
    "rallyPoints": {
        "points": [
            [
                47.39760401,
                8.5509154,
                50
            ],
            [
                47.39902017,
```

```
                8.54263274,
                50
            ]
        ],
        "version": 2
    }
```

The fields are:

Table 53: RallyPoints Key Values

Key	Description
`version`	The version number for the rally point plan format. The documented version is 2.
`points`	A list of rally points.

Parameter Protocol

The parameter microservice is used to exchange configuration settings between MAVLink components.

Each parameter is represented as a key/value pair. The key is usually the human-readable name of the parameter (maximum of 16 characters) and a value - which can be one of a number of types.

The key/value pair has a number of important properties:

- The human-readable name is small but useful (it can encode parameter names from which users can infer the purpose of the parameter).
- Unknown autopilots that implement the protocol can be supported "out of the box".
- A GCS does not *have* to know in advance what parameters exist on a remote system (although in practice a GCS can provide a *better* user experience with additional parameter metadata like maximum and minimum values, default values, etc.).
- Adding a parameter only requires changes to the system with parameters. A GCS that loads the parameters, and the MAVLink communication libraries, should not require any changes.

Message/Enum Summary

Message	Description
PARAM_REQUEST_LIST	Request all parameters. The recipient broadcasts all parameter values using PARAM_VALUE.
PARAM_REQUEST_READ	Request a single parameter. The recipient broadcasts the specified parameter value using PARAM_VALUE.
PARAM_SET	Send command to set a specified parameter to a value. After the value has been set (whether successful or not), the recipient should broadcast the current value using PARAM_VALUE.
PARAM_VALUE	The current value of a parameter, broadcast in response to a request to get one or more parameters (PARAM_REQUEST_READ, PARAM_REQUEST_LIST) or whenever a parameter is set (PARAM_SET) or changes.

Table 30: Message

Enum	Description
MAV_PARAM_TYPE	PARAM_SET and PARAM_VALUE store/encode parameter values within a float field. This type conveys the real type of the encoded parameter value, e.g. MAV_PARAM_TYPE_UINT16, MAV_PARAM_TYPE_INT32, etc.

Table 31: Enum

Parameter Encoding

Parameter names/ids are set in the param_id field of messages where they are used. The `param_id` string can store up to 16 characters. The string is terminated with a NULL (\0) character if there are less than 16 human-readable chars, and without a null termination byte if the length is exactly 16 chars.

Values are byte-wise encoded *within* the param_value field, an IEE754 single-precision, 4 byte, floating point value. The `param_type` (MAV_PARAM_TYPE) is used to indicate the actual type of the data so that it can be decoded by the recipient. Supported types are: 8, 16, 32 and 64-bit signed and unsigned integers, and 32 and 64-bit floating point numbers.

A byte-wise conversion is needed, rather than a simple cast, to enable larger integers to be exchanged (e.g. 1E7 scaled integers can be useful for encoding some types of data, but lose precision if cast directly to floats).

Mavgen C API The C API provides a convenient union that allows you to bytewise convert between any of the supported types: `mavlink_param_union_t` (mavlink_types.h). For example, below we show how you can set the union integer field but pass the float value to the sending function:

```
mavlink_param_union_t param;
int32_t integer = 20000;
param.param_int32 = integer;
param.type = MAV_PARAM_TYPE_INT32;
// Then send the param by providing the float bytes to the send function
mavlink_msg_param_set_send(xxx, xxx, param.param_float, param.type, xxx);
```

Mavgen Python API (Pymavlink) Pymavlink does not include special support to byte-wise encode the non-float data types (unsurprisingly, because Python effectively "hides" low level data types from users). When working with a system that supports non-float parameters you will need to do the encoding/decoding yourself when sending and receiving messages.

There is a good example of how to do this in the Pymavlink mavparm.py module (see MAVParmDict.mavset()).

Parameter Caching

A GCS or other component may choose to maintain a cache of parameter values for connected components/systems, in order to reduce the time required to display values and reduce MAVLink traffic.

The cache can be populated initially by first reading the full parameter list at least once, and then updated by monitoring for `PARAM_VALUE` messages (which are emitted whenever a parameter is written or otherwise changed).

Cache synchronisation is not guaranteed; a component may miss update messages due to parameter changes by other components.

Multi-System and Multi-Component Support

MAVLink supports multiple systems in parallel on the same link, and multiple MAVLink enabled components within a system.

Requests to get and set parameters can be sent to individual systems or components. To get a complete parameter list from a system, send the request parameter message with target_component set to `MAV_COMP_ID_ALL`.

All components must respond to parameter request messages addressed to their ID or the ID `MAV_COMP_ID_ALL`.

QGroundControl by default queries all components of the currently connected system (it sends ID `MAV_COMP_ID_ALL`).

Limitations

Parameters Table is Invariant The protocol *requires* that the parameter set does not change during normal operation/after parameters have been read.

If a component can add parameters during (or after) initial synchronization the protocol cannot guarantee reliable/robust synchronization, because there is no way to notify that the parameter set has changed and a new sync is required.

If working with a non-compliant component, the risk of problems when working with parameters can be *reduced* (but not removed) if:

- The param_id is used to read parameters where possible (the mapping of param_index to a particular parameter might change on systems where parameters can be added/removed).
- `PARAM_VALUE.param_count` is monitored and the parameter set re-synchronised on change.

Parameter Synchronisation Can Fail A GCS (or other component) that wants to cache parameters with a component and keep them synchronised, should first get all parameters, and then track any new parameter changes by monitoring for `PARAM_VALUE` messages (updating their internal list appropriately).

This works for the originator of a parameter change, which can resend the request if an expected `PARAM_VALUE` is not received. This approach may fail for components that did not originate the change, as they will not know about updates they do not receive (i.e. if messages are dropped).

A component may mitigate this risk by, for example, sending the `PARAM_VALUE` multiple times after a parameter is changed.

Parameter Operations

This section defines the state machine/message sequences for all parameter operations.

Read All Parameters The read-all operation is started by sending the `PARAM_REQUEST_LIST` message. The target component must start to broadcast the parameters individually in `PARAM_VALUE` messages after receiving this message. The drone should allow a pause after sending each parameter to ensure that the operation doesn't consume all of the available link bandwidth (30 - 50 percent of the bandwidth is reasonable).

The sequence of operations is:

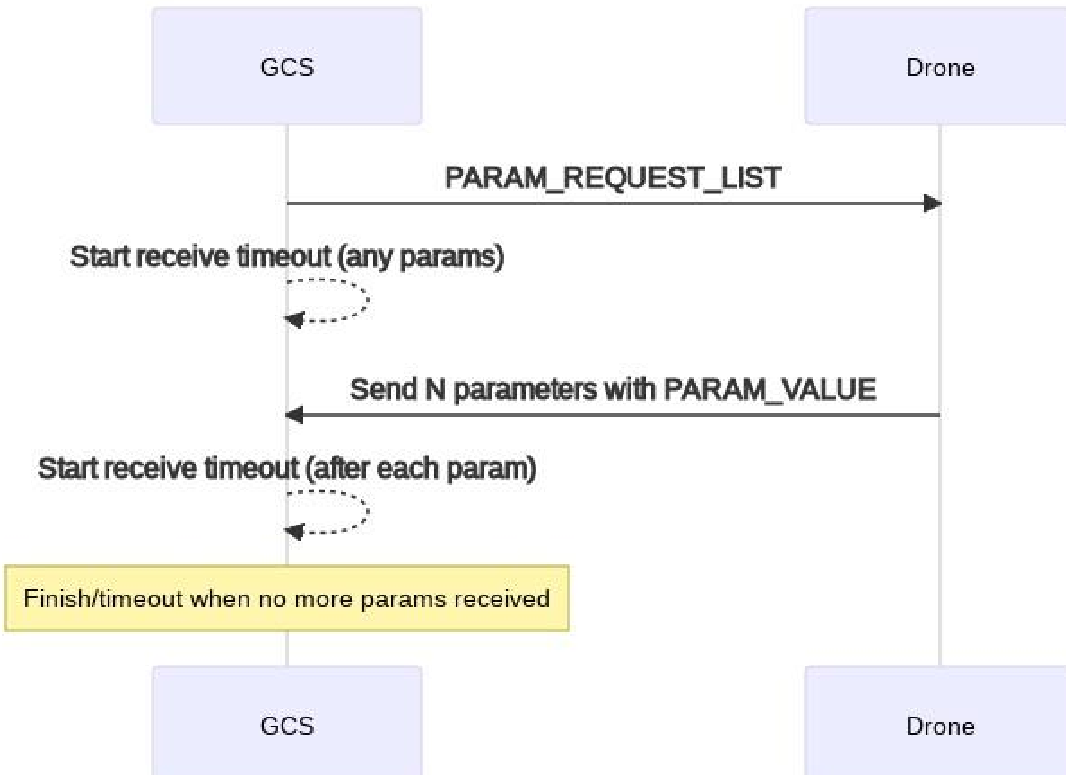

Figure 17: Read-All Operations Diagram

1. GCS (client) sends PARAM_REQUEST_LIST specifying a target system/component.
   - Broadcast addresses may be used. All targeted components should respond with parameters (or ignore the request if they have none).
   - The GCS is expected to accumulate parameters from all responding systems.
   - The timeout/retry behavior is GSC dependent.
2. The targeted component(s) should respond, sending all parameters individually in PARAM_VALUE messages.
   - Allow breaks between each message in order to avoid saturating the link.
   - Components with no parameters should ignore the request.
3. GCS starts timeout after each PARAM_VALUE message in order to detect when parameters are no longer being sent (that the operation has completed).

Notes:

- The GCS/API may accumulate the received parameters for each component and can determine if any are missing/not received (PARAM_VALUE contains the total number of params and index of current param).
- Handling of missing params is GCS-dependent. *QGroundControl*, for example, individually requests each missing parameter by index.
- If a component does not have any parameters then it will ignore a PARAM_REQUEST_LIST request. The sender should simply timeout (after resending) if no PARAM_VALUE is received.

Read Single Parameter A single parameter can be read by sending the PARAM_REQUEST_READ message, as shown below:

The sequence of operations is:

1. GCS (client) sends PARAM_REQUEST_READ specifying either the parameter id (name) or parameter index.
2. GCS starts timeout waiting for acknowledgment (in the form of a PARAM_VALUE message).
3. Drone responds with PARAM_VALUE containing the parameter value. This is a broadcast message (sent to all systems).

The drone may restart the sequence if the PARAM_VALUE acknowledgment is not received within the timeout.

There is no formal way for the drone to signal when an invalid parameter is requested (i.e. for a parameter name or id that does not exist). In this case the drone should emit STATUSTEXT. The GCS may monitor for the specific notification, but will otherwise fail the request after any timeout/resend cycle completes.

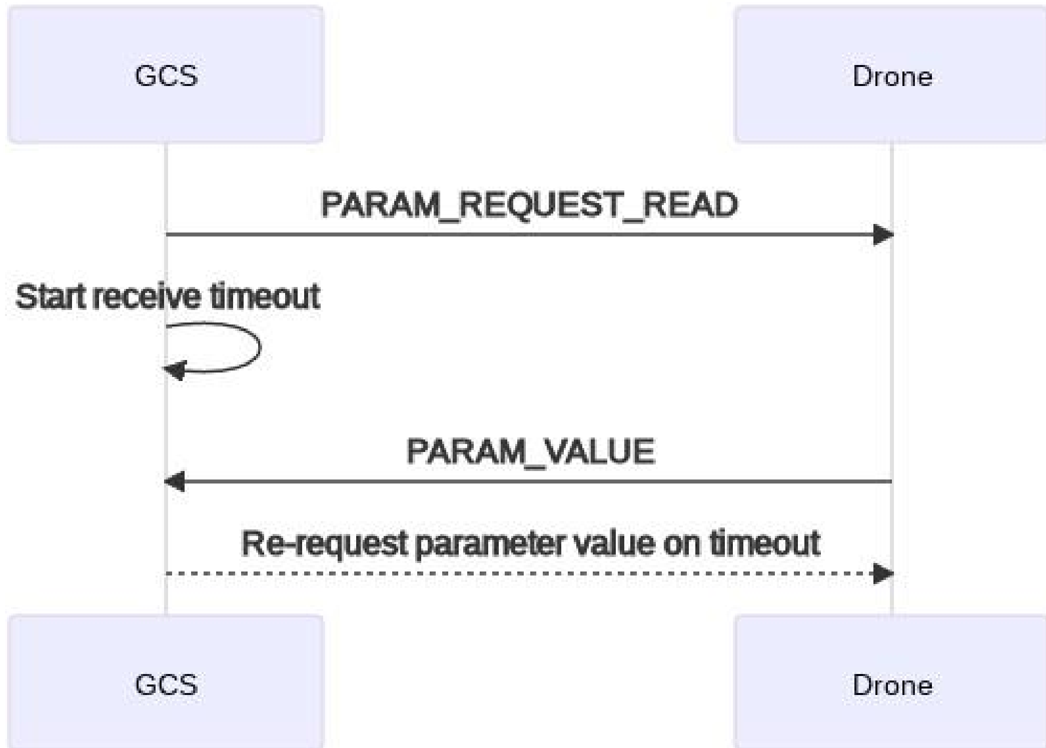

Figure 18: Read Single Diagram

Write Parameters Parameters can be written individually by sending the parameter name and value pair to the GCS, as shown:

The sequence of operations is:

1. GCS (client) sends `PARAM_SET` specifying the param name to update and its new value (also target system/component and the param type).
2. GCS starts timeout waiting for acknowledgment (in the form of a `PARAM_VALUE` message).
3. Drone writes parameters and responds by *broadcasting* a PARAM_VALUE containing the updated parameter value to all components/systems.
4. The Drone must acknowledge the PARAM_SET by broadcasting a PARAM_VALUE even if the write operation fails. In this case the PARAM_VALUE will be the current/unchanged parameter value.
5. GCS should update the parameter cache (if used) with the new value.
6. The GCS may restart the sequence if the expected `PARAM_VALUE` is not received within the timeout, or if the write operation fails (the value returned in `PARAM_VALUE` does not match the value set).

The command `MAV_CMD_DO_SET_PARAMETER` is not part of the parameter protocol. If implemented it can be used to set the value of a parameter using the enumeration of the parameter within the remote system is known (rather than the id). This has no particular advantage over the parameter protocol methods.

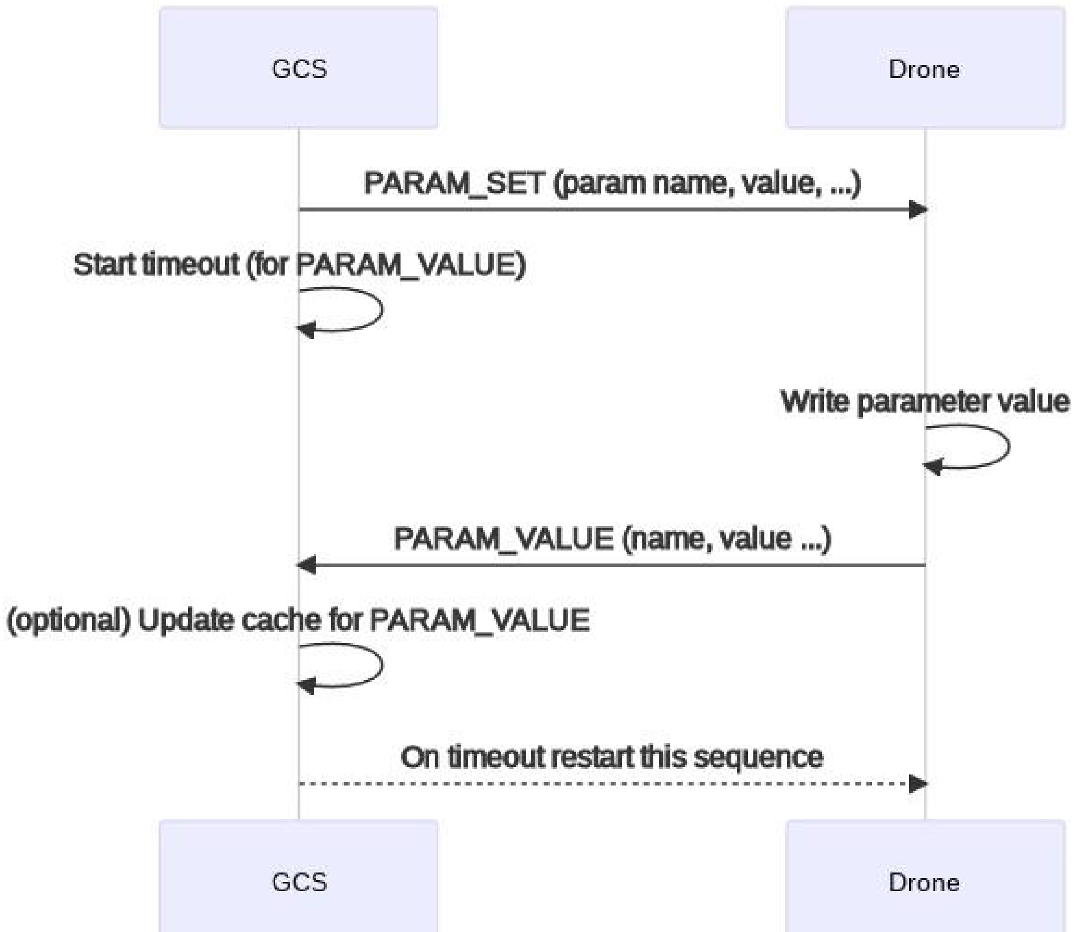

Figure 19: Write Parameter Diagram

List of parameters

The following list of parameters provide the required and optional parameters under this IOP, according to the vehicles available functionality - i.e if the vehicle provides the specified functionality, then the parameters that configure that functionality should be exposed. These parameters should be accessible and configurable through the Parameter Protocol. Note that this list can be expanded on further iterations of the IOP, according to the requirements.

Name	Type	Description	Min > Max (Incr.)	Default	Units
`GCS_LOSS_T`	`INT32`	GCS connection loss time threshold. After this amount of seconds without datalink, the GCS_LOSS_ACT mode triggers	5 > 300 (1)	10	s
`GCS_LOSS_ACT`	`INT32`	GCS connection loss failsafe action. The GCS connection loss failsafe will only be entered after a timeout, set by `GCS_LOSS_T` in seconds. Once the timeout occurs the selected action will be executed. Values: 0: Disabled; 1: Hold; 2: Return; 3: Land; 4: Terminate; 5: Disarm.	0 > 6 (1)	0	
`LED_ILLUM_MODE`	`UINT32`	Defines the illumination mode. Uses the following BIT Flags: Navigation lights on: 0x1; Navigation lights strobe: 0x2; IR on: 0x4; IR strobe: 0x8; White on: 0x10; White strobe: 0x20.		0x00	

Name	Type	Description	Min > Max (Incr.)	Default	Units
`COM_OBS_AVOID`	`INT32`	Defines the obstacle avoidance level. The level of obstacle avoidance is platform specific, i.e. the definition of what represents this level depends on how the vehicle platform implements it (proximity to obstacles, speed, etc.). Values: 0: Off; 1: Low; 2: Medium; 3: High.	0 > 4 (1)	0	
`RTX_RETURN_ALT`	`FLOAT`	Minimum traversal altitude above RTX destination (defined by RTX_TYPE). The vehicle will ascend to this altitude when Return mode is engaged, unless currently flying higher.	0 > `MAX_HAGL` (0.5)	60.0	m
`RTX_TYPE`	`INT32`	Return mode destination and flight path (direct if not specified). Values: 0: Return to closest safe point (home, rally point, takeoff, GCS, planned mission landing); 1: Return to takeoff; 2: Return to closest rally point or home; 3: Return to GCS or home if GCS position not known; 4: Return to mission start via reverse path; 5. Return to planned mission landing along mission path.	[0, 5]	0	

Name	Type	Description	Min > Max (Incr.)	Default	Units
`LOW_BAT_ACT`	`INT32`	Action the system take at battery conditions. Emergency level is minimum battery needed to land safely. Critical level is minimum battery needed to return to `RTX_TYPE` and land safely. Values: 0: Warning (Alert User); 1: Land immediately at emergency; 2: Land at critical; 3: Enter Return mode at critical, land immediately emergency level	0 > 3 (1)	1	
`GND_SPD_LIM`	`FLOAT`	Maximum ground speed. If higher speeds are commanded in a mission they are capped at this speed. Speed less than 0 indicates unlimited.	0 > Inf (0.5)	12.0	m/s
`MAX_HAGL`	`FLOAT`	Maximum Height AGL. Vehicles who cannot compute HAGL, will use height above home. Values less than 0 indicates unlimited	0.0 > Inf (0.5)	121.0	m
`MAX_DIST_RTX`	`FLOAT`	Maximum slant range to RTX point specified by `RTX_TYPE`. Value < 0 indicates disabled.	0 > Inf (0.5)	-1.0	m
`MAX_DIST_GCS`	`FLOAT`	Maximum slant range to GCS. Value < 0 indicates disabled	0 > Inf (0.5)	-1.0	m

Extended Parameter Protocol

The *Extended Parameter Protocol* is an extended version of the Parameter Protocol that adds support for larger custom parameter types e.g. strings. It can be used to exchange configuration settings between MAVLink components, and in particular configuration settings that may be more than just numeric values.

The protocol shares most of the same benefits and limitations of the original protocol, and similar (but not identical) operation sequences. The main difference is that when writing a parameter the system emits one or more `PARAM_EXT_ACK` messages (rather than `PARAM_EXT_VALUE`, as you would expect from the original protocol). This allows the *Extended Parameter Protocol* to differentiate between the case where a write fails (or is in progress) and the case where the value update simply went missing.

The extensions were invented for the Camera Protocol, which uses them to request/set parameter values specified in a Camera Definition File. At time of writing the protocol is supported by *QGroundControl* for this purpose, but is not otherwise supported by flight stacks.

Message/Enum Summary

Table 57: Message

Message	**Description**
`PARAM_EXT_REQUEST_LIST`	Request all parameters of this component. On receiving this request, the requested component will emit all parameter values using `PARAM_EXT_VALUE`.
`PARAM_EXT_VALUE`	Emit the value of a parameter, following a `PARAM_EXT_REQUEST_LIST` or `PARAM_EXT_REQUEST_READ`. The message includes param_count and param_index which the recipient can use to track received parameters and re-request missing parameters after a timeout.
`PARAM_EXT_REQUEST_READ`	Request the value of a specific parameter using either its param_id or param_index. Expects a response in a `PARAM_EXT_VALUE`.
`PARAM_EXT_SET`	Set a parameter value. Expects immediate response `PARAM_EXT_ACK` with result indicating success, failure, or that the request is still in progress (`PARAM_ACK_IN_PROGRESS`). If in progress, additional update `PARAM_EXT_ACK` messages are expected.
`PARAM_EXT_ACK`	Response from a `PARAM_EXT_SET` message, which indicates whether the value was accepted (set), failed, setting is still in progress, or that the specified parameter is invalid/unsupported.

Table 58: Enum

Enum	Description
MAV_PARAM_EXT_TYPE	Specifies the datatype of a MAVLink extended parameter (parameter values are encoded within the a `char[128]` array in the messages). This type conveys the *real type* of the encoded parameter value, e.g. `MAV_PARAM_EXT_TYPE_REAL32`.
PARAM_ACK	Request acknowledgment status value, sent in an PARAM_EXT_ACK as a response to a PARAM_EXT_SET message. A request can be accepted, fail, in-progress, or unsupported (indicating the specified parameter does not exist or has an invalid value or value type).

Parameter Encoding

Parameter names/ids are set in the param_id field of messages where they are used. The param_id string can store up to 16 characters. The string is terminated with a NULL (\0) character if there are less than 16 human-readable chars, and without a null termination byte if the length is exactly 16 chars.

Names (as above) are the same as for the Parameter Protocol.

Values are byte-wise encoded *within* the param_value field, which is a `char[128]`. The param_type (MAV_PARAM_EXT_TYPE) is used to indicate the actual type of the data so that it can be decoded by the recipient. Supported types are: 8, 16, 32 and 64-bit signed and unsigned integers, 32 and 64-bit floating point numbers, and a "custom type" which may be used for e.g. strings.

The encoding is best described by example as shown below.

C Encoding/Decoding

To send the parameter, the data is written into a union structure then memcpy used to copy the data into the message `char[128]` field.

The union structure might look like this:

```
MAVPACKED(
typedef struct {
    union {
        float       param_float;
        double      param_double;
        int64_t     param_int64;
        uint64_t    param_uint64;
```

```
        int32_t     param_int32;
        uint32_t    param_uint32;
        int16_t     param_int16;
        uint16_t    param_uint16;
        int8_t      param_int8;
        uint8_t     param_uint8;
        uint8_t     bytes[MAVLINK_MSG_PARAM_EXT_SET_FIELD_PARAM_VALUE_LEN];
    };
    uint8_t type;
}) param_ext_union_t;
```

To send the parameter, the data is written into the union value of the correct type and then `memcpy` used to copy it to the message data.

```
// Create C object for message data and zero fill
mavlink_param_ext_set_t p;
memset(&p, 0, sizeof(mavlink_param_ext_set_t));

// Store type of data to be sent in message
p.param_type = /* Value for type from MAV_PARAM_EXT_TYPE */;

// Create union value to assign data to
param_ext_union_t union_value;

// Assign data to union value (usually in a case statement based on type).
union_value.param_uint16 = static_cast<uint16_t>(aUint16Value);

// memcpy the union bytes value into the message data array.
memcpy(&p.param_value[0], &union_value.bytes[0], MAVLINK_MSG_PARAM_EXT_SET_FIELD_PARAM_V
```

Receiving and decoding a parameter is even simpler:

```
// 'value' is the char[128] from the message
// 'param_type' is the param_type value from the message

// Create union value to assign data to
param_ext_union_t union_value;

// memcpy the received value into the union_value bytes field.
memcpy(union_value.bytes, value, MAVLINK_MSG_PARAM_EXT_SET_FIELD_PARAM_VALUE_LEN);

// Assign the union_value of correct type to a variable for use
switch (param_type) {
    ...
    case MAV_PARAM_EXT_TYPE_INT16:
```

```
        auto var = union_value.param_int16;
        break;
    ...
}
```

QGroundControl provides real code examples here:

- Union structure: QGCCameraIO.h::param_ext_union_t
- Send a parameter (encode in `char[128]`): QGCCameraIO.cc::QGCCameraParamIO::_sendParameter()
- Receive a parameter and get typed value: QGCCameraIO.cc::QGCCameraParamIO::_valueFromMessage()

Parameter Caching

A GCS or other component may choose to maintain a cache of parameter values for connected components/systems, in order to reduce the time required to display values and reduce MAVLink traffic.

The cache can be populated initially by first reading the full parameter list at least once, and then updated by monitoring for `PARAM_EXT_ACK` messages with `PARAM_ACK_ACCEPTED` (which are emitted whenever a parameter is successfully written/changed).

A system may also monitor for `PARAM_EXT_VALUE` originating from other components/systems requesting parameter values.

Cache synchronisation is not guaranteed; a component may miss parameter update messages due to changes by other components.

Limitations

Parameters Table is Invariant The protocol *requires* that the parameter set does not change during normal operation/after parameters have been read.

If a component can add parameters during (or after) initial synchronization the protocol cannot guarantee reliable/robust synchronization, because there is no way to notify that the parameter set has changed and a new sync is required.

When requesting parameters from such a components, the risk of problems can be *reduced* (but not removed) if:

- The param_id is used to read parameters where possible (the mapping of param_index to a particular parameter may change on systems where parameters can be added/removed).
- `PARAM_EXT_VALUE.param_count` may be monitored. If this changes the parameter set should be re-synchronised.

Parameter Synchronisation Can Fail A GCS (or other system) that wants to cache parameters from a component and keep them synchronised should first get all parameters, and then track changes by monitoring for `PARAM_EXT_ACK` messages (updating their internal list appropriately).

This works for the originator of a parameter change, which can resend the request if an expected `PARAM_EXT_ACK` is not received. This approach may fail for components that did not originate the change, as they will not know about updates they do not receive (i.e. if messages are dropped).

A component may mitigate this risk by, for example, sending the `PARAM_EXT_ACK` multiple times after a parameter is changed.

Parameter Operations

This section defines the state machine/message sequences for all parameter operations.

Read All Parameters The read-all operation is started by sending the `PARAM_EXT_REQUEST_LIST` message. The target component must start to broadcast the parameters individually in `PARAM_EXT_VALUE` messages after receiving this message. The drone should allow a pause after sending each parameter to ensure that the operation doesn't consume all of the available link bandwidth (30 - 50 percent of the bandwidth is reasonable).

The sequence of operations is:

1. GCS (client) sends `PARAM_EXT_REQUEST_LIST` specifying a target system/component.
   - Broadcast addresses may be used. All targeted components should respond with parameters (or ignore the request if they have none).
   - The GCS is expected to accumulate parameters from all responding systems.
   - The timeout/retry behavior is GSC dependent.
2. The targeted component(s) should respond, sending all parameters individually in `PARAM_EXT_VALUE` messages.
   - Allow breaks between each message in order to avoid saturating the link.
   - Components with no parameters should ignore the request.
3. GCS starts timeout after each `PARAM_EXT_VALUE` message in order to detect when parameters are no longer being sent (that the operation has completed).

Notes:

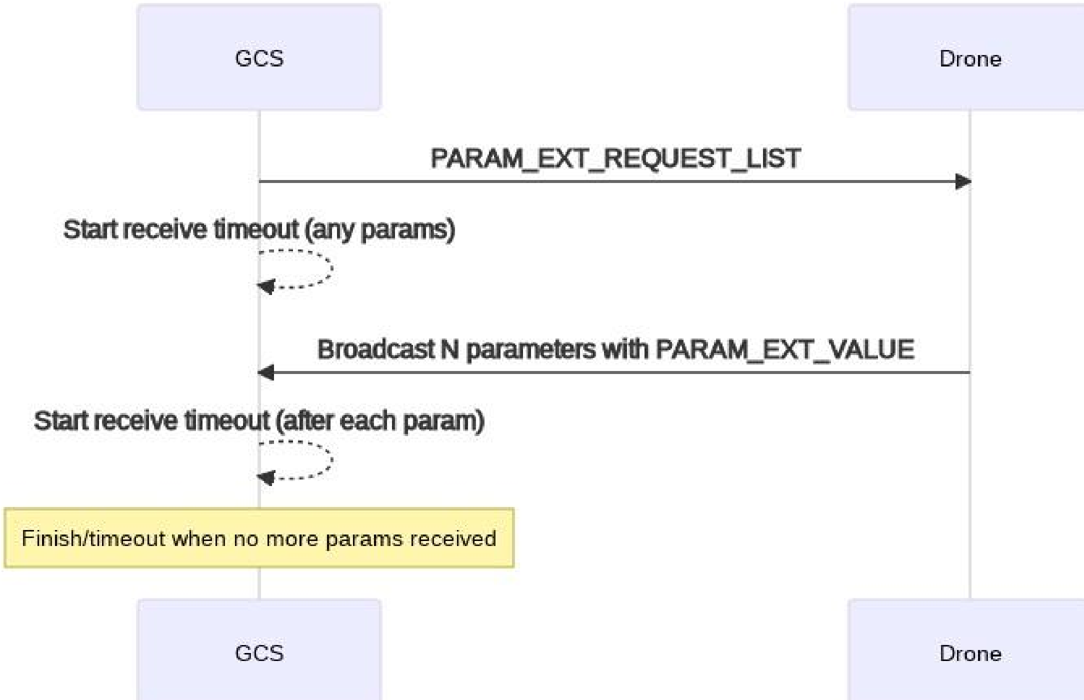

Figure 20: Read All Parameters

- The GCS/API may accumulate the received parameters for each component and can determine if any are missing/not received (PARAM_EXT_VALUE contains the total number of params and index of current param).
- Handling of missing params is GCS-dependent. *QGroundControl*, for example, individually requests each missing parameter by index.
- If a component does not have any parameters then it will ignore a PARAM_EXT_REQUEST_LIST request. The sender should simply timeout (after resending) if no PARAM_EXT_VALUE is received.

Read Single Parameter A single parameter can be read by sending the PARAM_EXT_REQUEST_READ message, as shown below:

The sequence of operations is:

1. GCS (client) sends PARAM_EXT_REQUEST_READ specifying either the parameter id (name) or parameter index.
2. GCS starts timeout waiting for acknowledgment (in the form of a PARAM_EXT_VALUE message).
3. Drone responds with PARAM_EXT_VALUE containing the parameter value. This is a broadcast message (sent to all systems).

The drone may restart the sequence if the PARAM_EXT_VALUE acknowledgment is not received within the timeout.

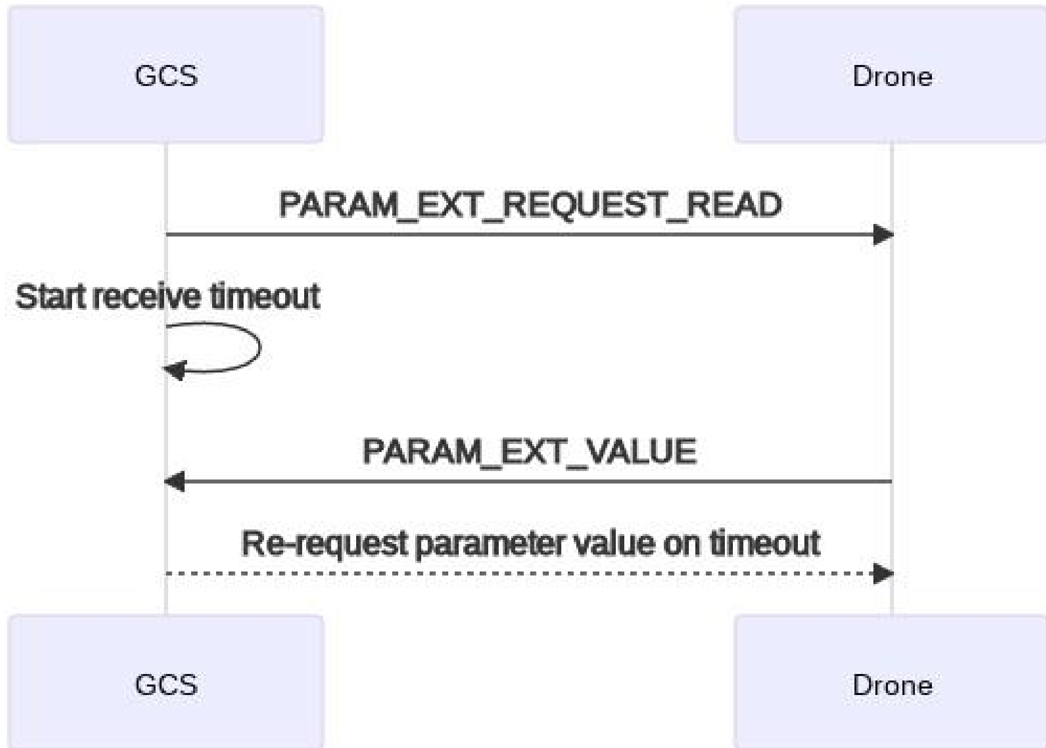

Figure 21: Read Single Diagram

Write Parameters Parameters are written individually using PARAM_EXT_SET. The recipient will respond with PARAM_EXT_ACK indicating success, failure, or that the write is still in progress (PARAM_ACK_IN_PROGRESS). On receipt of PARAM_ACK_IN_PROGRESS the component setting the parameter will extend its timeout (PARAM_EXT_ACK will be re-sent when the write completes)

Parameters can be written individually by sending the parameter name and value pair to the GCS, as shown:

For long-running write operations drone may initially respond with PARAM_ACK_IN_PROGRESS:

The sequence of operations is:

1. GCS (client) sends PARAM_EXT_SET specifying the param name to update and its new value (also target system/component and the param type).
2. GCS starts timeout waiting for acknowledgment (in the form of a PARAM_EXT_ACK message).
3. Drone (starts to) write parameters and responds by *broadcasting* a PARAM_EXT_ACK.
   - If the write succeeded the PARAM_EXT_ACK will contain a result of PARAM_ACK_ACCEPTED and the updated parameter value.

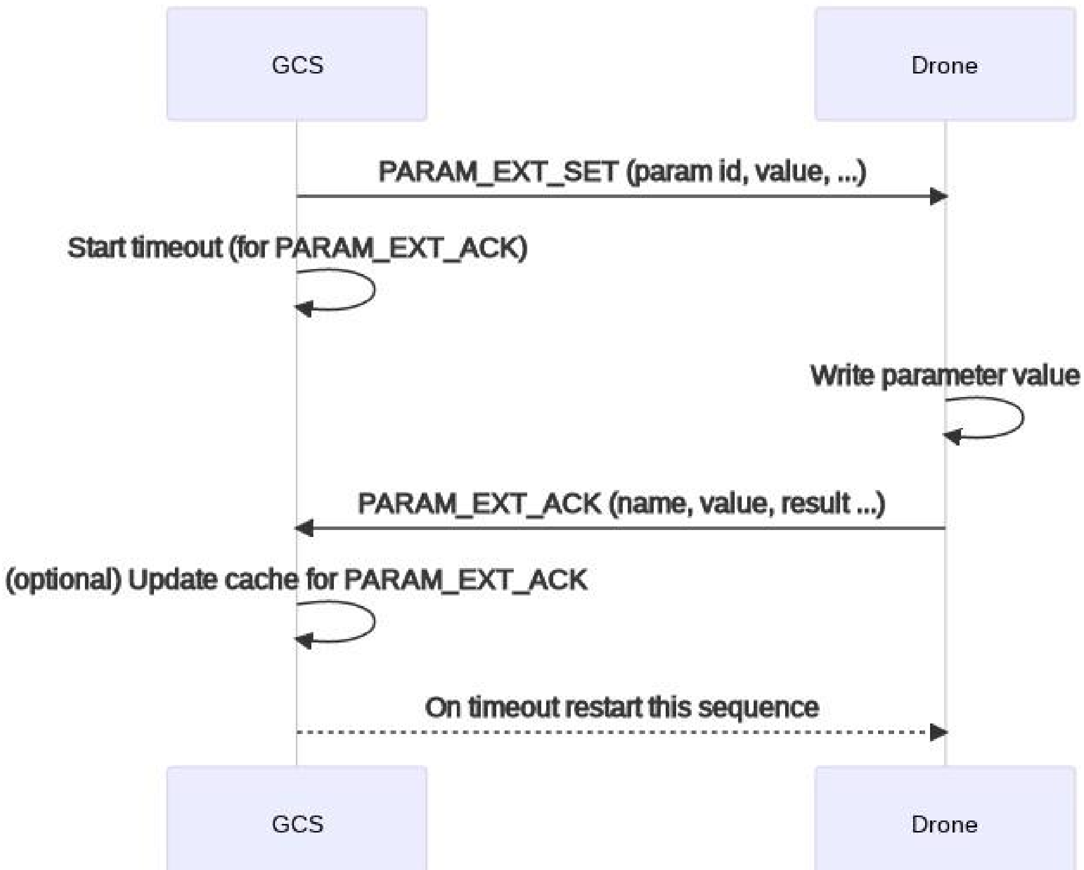

Figure 22: Write Parameters Diagram

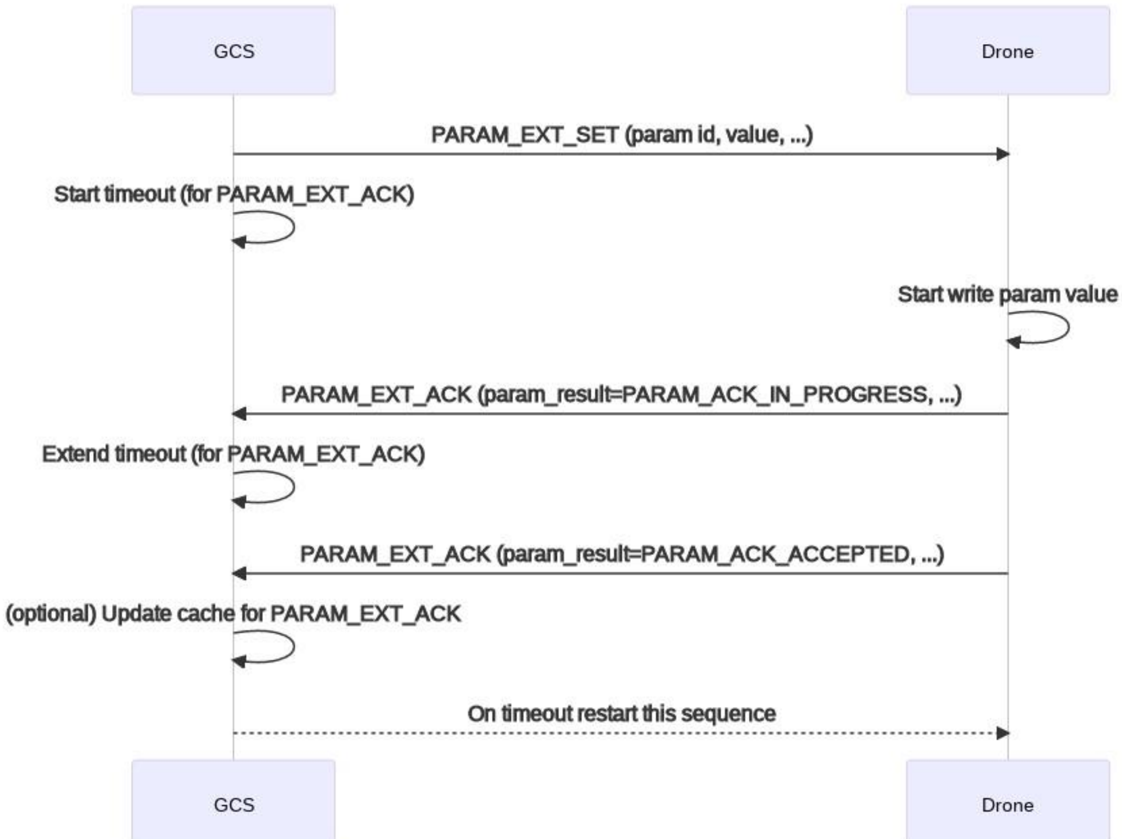

Figure 23: Long-Running Write Parameter Diagram

- If the parameter was unknown or of an unsupported type PARAM_EXT_ACK will contain a result of PARAM_ACK_VALUE_UNSUPPORTED and the current parameter value will be XXXXX.
- If the write failed for another reason then PARAM_EXT_ACK will contain a result of PARAM_ACK_FAILED and the current parameter value.
- If the write operation is long-running the PARAM_EXT_ACK will contain a result of PARAM_ACK_IN_PROGRESS and the XXXX parameter value. In this case the recipient should increase their timeout and way for another PARAM_EXT_ACK. PARAM_EXT_ACK should be present when the operation completes.

4. GCS should update the parameter cache (if used) with the new value.
5. The GCS may restart the sequence if an expected PARAM_EXT_ACK is not received within the timeout, or if the write operation fails.

Command Protocol

The MAVLink command protocol allows guaranteed delivery of MAVLink commands.

Commands are values of MAV_CMD that define the values of up to 7 parameters. These parameters and the command id are encoded in COMMAND_INT or COMMAND_LONG for sending.

The protocol provides reliable delivery by expecting a matching acknowledgement (COMMAND_ACK) from commands to indicate command arrival, and result. If no acknowledgement is received the command must be automatically re-sent.

COMMAND_INT is generally recommended when sending positional information as it allows greater precision, and is explicit about the coordinate frame. Commands that require float-only properties in parameters 5, 6 must be sent in COMMAND_LONG (e.g. commands where NaN has an explicit meaning).

Note: Under this IOP, support for both COMMAND_INT and COMMAND_LONG generation and parsing are required for both vehicle and GCS implementers.

Message/Enum Summary

Table 59: Message

Message	Description
COMMAND_INT	Message for encoding a command (MAV_CMD). The message encodes commands into up to 7 parameters: parameters 1-4, 7 are floats, and parameters 5,6 are scaled integers. The scaled integers are used for positional information (scaling depends on the actual command value). The coordinate frame of positional parameters is explicitly specified in a frame field. Commands that require float-only properties in parameters 5, 6 cannot be sent in this message (e.g. commands where NaN has an explicit meaning).
COMMAND_LONG	Message for encoding a command (MAV_CMD). The message encodes commands into up to 7 float parameters. The coordinate frame used for positional coordinates is implementation dependent. Any command may be packaged in this message, but there may be some loss of precision for positional coordinates (latitude, longitude).
COMMAND_ACK	Command acknowledgement. Includes results (success, failure, still in progress) and may include progress information and additional detail about failure reasons.
COMMAND_CANCEL	Cancel a long running command.

Table 60: Enum

Enum	Description
MAV_CMD	Commands to be executed/sent in the command messages.
MAV_FRAME	Coordinate frame. Used in COMMAND_INT to specify the coordinate frame of any positional parameters.
MAV_RESULT	Result of command, included in COMMAND_ACK.result.

Sequences

If the command drops the sender should increase the confirmation field:

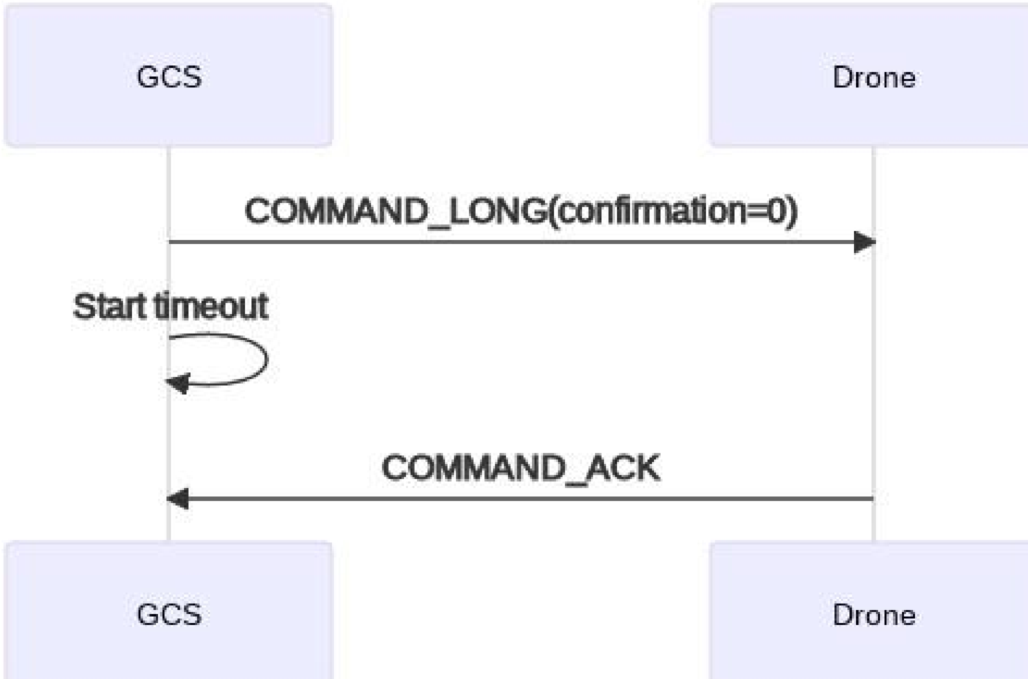

Figure 24: Sequence Diagram

Long Running Commands

Some commands are *long running*, and cannot be completed immediately. The drone reports its progress by sending COMMMAND_ACK messages with `COMMAND_ACK.result=MAV_RESULT_IN_PROGRESS` and the progress as a percentage in `COMMMAND_ACK.progress` ([0-100] percent complete, 255 if progress not supplied). When the operation completes, the drone must terminate with a `COMMMAND_ACK` containing the final result of the operation: `MAV_RESULT_ACCEPTED`, `MAV_RESULT_FAILED`, `MAV_RESULT_CANCELLED`).

Long running operations may be cancelled by sending the `COMMAND_CANCEL` message. The drone should cancel the operation and complete the sequence by sending `COMMAND_ACK` with `COMMAND_ACK.result=MAV_RESULT_CANCELLED`.

- If cancellation is not supported the drone can just continue to send progress updates until completion.
- If the sequence has already completed (or is idle) the cancel command should be ignored.

If another command is received while handling a command (long running or otherwise) the new command should be rejected with `MAV_RESULT_TEMPORARILY_REJECTED`. What this means is that to restart an operation (i.e. with new parameters) it must first be cancelled.

The rate at which progress messages are emitted is system-dependent. Gen-

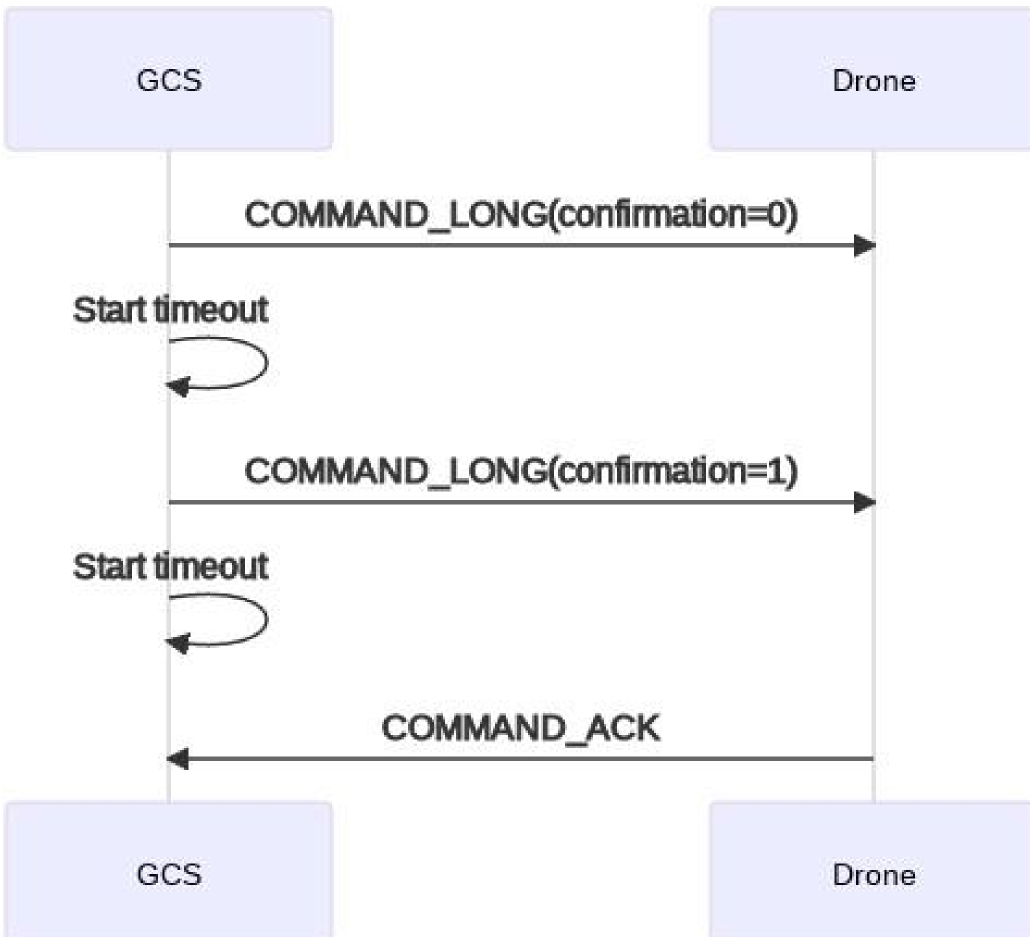

Figure 25: Sequence with Dropped Command Diagram

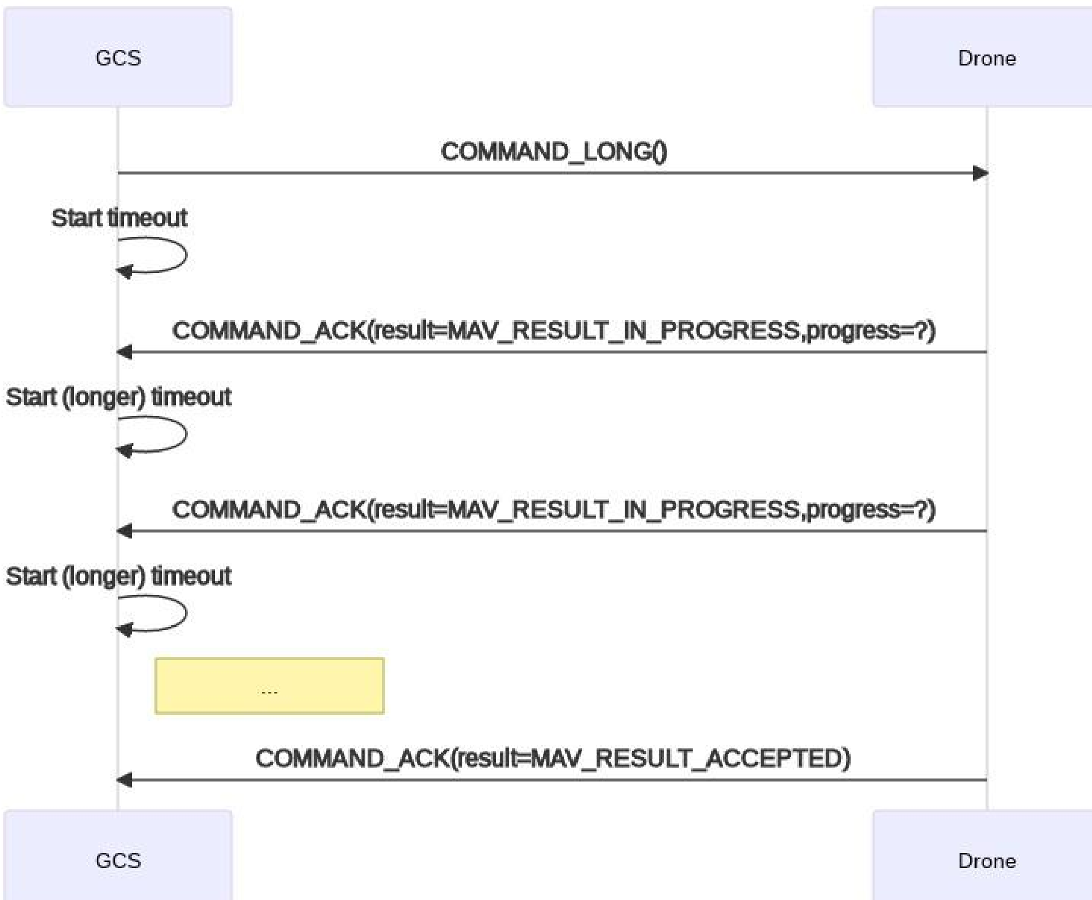

Figure 26: Long Running Commands Diagram

erally though, the GCS should have a much increased timeout after receiving an ACK with `MAV_RESULT_IN_PROGRESS`.

If a timeout is triggered when waiting for a progress or completion update, the GCS should terminate the sequence (return to the idle state) and notify the user if appropriate.

Commands to support

Commands to be executed by the vehicle, either independently and when executing a mission through the Mission Protocol. Note that similar to the parameters, if the vehicle supports the functionality (like loiter, "follow-me" mode, or others), then it should be capable of processing a command that interacts with the respective function. Required commands are marked with the **REQUIRED** tag, and are independent of the available capabilities in the vehicle. They can be executed on user request, or as part of a mission script. If the action is used in a mission, the parameter mapping to the waypoint/mission message is as follows: Param 1, Param 2, Param 3, Param 4, X: Param 5, Y:Param 6, Z:Param 7. This command list is similar to what ARINC 424 is for commercial aircraft: A data format how to interpret waypoint/mission data. `NaN` and `INT32_MAX` may be used in float/integer params (respectively) to indicate optional/default values (e.g. to use the component's current yaw or latitude rather than a specific value). See `MAV_CMD` for information about the structure of the `MAV_CMD` entries.

MAV_CMD_NAV_WAYPOINT (16)

REQUIRED. Navigate to waypoints.

Table 61: Waypoint Navigation Configuration

Param	**Description**	**Values**	**Units**
1: Hold	Hold time. (ignored by fixed wing, time to stay at waypoint for rotary wing)	*min:*0	s
2: Accept Radius	Acceptance radius (if the sphere with this radius is hit, the waypoint counts as reached)	*min:*0	m
3: Pass Radius	0 to pass through the WP, if > 0 radius to pass by WP. Positive value for clockwise orbit, negative value for counter-clockwise orbit. Allows trajectory control.		m

Param	Description	Values	Units
4: Yaw	Desired yaw angle at waypoint (rotary wing). NaN to use the current system yaw heading mode (e.g. yaw towards next waypoint, yaw to home, etc.).		deg
5: Latitude	Latitude		
6: Longitude	Longitude		
7: Altitude	Altitude		m

MAV_CMD_NAV_LOITER_UNLIM (17)

Loiter around this waypoint an unlimited amount of time

Table 62: Around Waypoint Navigation Configuration

Param	Description	Values	Units
1	Empty		
2	Empty		
3: Radius	Loiter radius around waypoints for forward-only moving vehicles (not multicopters). If positive loiter clockwise, else counter-clockwise	m	
4: Yaw	Desired yaw angle. NaN to use the current system yaw heading mode (e.g. yaw towards next waypoint, yaw to home, etc.).	deg	
5: Latitude	Latitude		
6: Longitude	Longitude		
7: Altitude	Altitude	m	

MAV_CMD_NAV_LOITER_TURNS (18)

Loiter around this waypoint for X turns.

Table 63: Specific Around Waypoint Navigation Configuration

Param	**Description**	**Values**	**Units**
1: Turns	Number of turns.	*min:*0	
2: Heading Required	Leave loiter circle only once heading towards the next waypoint (0 = False)	*min:*0, *max:*1 *increment:*1	
3: Radius	Loiter radius around waypoints for forward-only moving vehicles (not multicopters). If positive loiter clockwise, else counter-clockwise		m
4: Xtrack Location	Loiter circle exit location and/or path to next waypoint (”xtrack”) for forward-only moving vehicles (not multicopters). 0 for the vehicle to converge towards the center xtrack when it leaves the loiter (the line between the centers of the current and next waypoint), 1 to converge to the direct line between the location that the vehicle exits the loiter radius and the next waypoint. Otherwise the angle (in degrees) between the tangent of the loiter circle and the center xtrack at which the vehicle must leave the loiter (and converge to the center xtrack). NaN to use the current system default xtrack behavior.		
5: Latitude	Latitude		
6: Longitude	Longitude		
7: Altitude	Altitude	m	

MAV_CMD_NAV_LOITER_TIME (19) Loiter at the specified latitude, longitude and altitude for a certain amount of time. Multicopter vehicles stop at the point (within a vehicle-specific acceptance radius). Forward-only moving vehicles (e.g. fixed-wing) circle the point with the specified radius/direction. If the Heading Required parameter (2) is non-zero, forward moving aircraft will only leave the loiter circle once heading towards the next waypoint.

Table 64: Latitude Loiter Navigation Configuration

Param	**Description**	**Values**	**Units**
1: Time	Loiter time (only starts once Lat, Lon and Alt is reached).	*min:*0	s
2: Heading Required Leave loiter circle only once heading towards the next waypoint (0 = False)	*min:*0 *max:*1 *increment:*1		
3: Radius	Loiter radius around waypoints for forward-only moving vehicles (not multicopters). If positive loiter clockwise, else counter-clockwise.		m
4: Xtrack Location	Loiter circle exit location and/or path to next waypoint (”xtrack”) for forward-only moving vehicles (not multicopters). 0 for the vehicle to converge towards the center xtrack when it leaves the loiter (the line between the centers of the current and next waypoint), 1 to converge to the direct line between the location that the vehicle exits the loiter radius and the next waypoint. Otherwise the angle (in degrees) between the tangent of the loiter circle and the center xtrack at which the vehicle must leave the loiter (and converge to the center xtrack). NaN to use the current system default xtrack behavior.		
5: Latitude	Latitude		
6: Longitude	Longitude		
7: Altitude	Altitude		m

MAV_CMD_NAV_RETURN_TO_LAUNCH (20)

REQUIRED. Return to launch location.

Note: this command should change the the vehicle to AUTO:RTL mode.

Table 65: Return to Launch Navigation Configuration

Param	Description
1	Reserved for future use
2	Reserved for future use
3	Reserved for future use
4	Reserved for future use
5	Reserved for future use
6	Reserved for future use
7	Reserved for future use

MAV_CMD_NAV_LAND (21)

REQUIRED. Land at location.

Note: this command should change the the vehicle to AUTO:LAND mode.

Table 66: Land at Location Navigation Configuration

Param	Description	Values	Units
1: Abort Alt	Minimum target altitude if landing is aborted (0 = undefined/use system default).		m
2: Land Mode	Precision land mode.	`PRECISION_LAND_MODE`	
3: Empty.			
4: Yaw Angle	Desired yaw angle. NaN to use the current system yaw heading mode (e.g. yaw towards next waypoint, yaw to home, etc.).		deg
5: Latitude	Latitude.		
6: Longitude	Longitude.		
7: Altitude	Landing altitude (ground level in current frame).		m

MAV_CMD_NAV_TAKEOFF (22) REQUIRED. Takeoff from ground / hand. Vehicles that support multiple takeoff modes (e.g. VTOL quadplane) should take off using the currently configured mode.

Note: this command should change the the vehicle to AUTO:TAKEOFF mode.

Table 67: Take off Navigation Configuration

Param	Description	Units
1: Pitch	Minimum pitch (if airspeed sensor present), desired pitch without sensor	deg
2:	Empty	
3:	Empty	
4: Yaw	Yaw angle (if magnetometer present), ignored without magnetometer. NaN to use the current system yaw heading mode (e.g. yaw towards next waypoint, yaw to home, etc.).	deg
5: Latitude	Latitude	
6: Longitude	Longitude	
7: Altitude	Altitude	m

MAV_CMD_DO_ORBIT (34)

Start orbiting on the circumference of a circle defined by the parameters. Setting values to NaN/INT32_MAX (as appropriate) results in using defaults.

Table 68: Orbit Navigation Configuration

Param	Description	Values	Units
1: Radius	Radius of the circle. Positive: orbit clockwise. Negative: orbit counter-clockwise. NaN: Use vehicle default radius, or current radius if already orbiting.		m
2: Velocity	Tangential Velocity. NaN: Use vehicle default velocity, or current velocity if already orbiting.		m/s
3: Yaw Behavior	Yaw behavior of the vehicle.	`ORBIT_YAW_BEHAVIOUR`	

Param	Description	Values	Units
4: Orbits	Orbit around the centre point for this many radians (i.e. for a three-quarter orbit set 270*Pi/180). 0: Orbit forever. NaN: Use vehicle default, or current value if already orbiting.	*min:*0	rad
5: Latitude/X	Center point latitude (if no `MAV_FRAME` specified) / X coordinate according to `MAV_FRAME`. `INT32_MAX` (or NaN if sent in `COMMAND_LONG`): Use current vehicle position, or current center if already orbiting.		
6: Longitude/Y	Center point longitude (if no `MAV_FRAME` specified) / Y coordinate according to `MAV_FRAME`. `INT32_MAX` (or NaN if sent in `COMMAND_LONG`): Use current vehicle position, or current center if already orbiting.		
7: Altitude/Z	Center point altitude (MSL) (if no `MAV_FRAME` specified) / Z coordinate according to `MAV_FRAME`. NaN: Use current vehicle altitude.		

MAV_CMD_DO_SET_MODE (176)

REQUIRED. Set system mode.

Note: This command changes the vehicle to any of the supported modes defined in this IOP.

Table 69: Set Mode Configuration

Param	Description	Values	Units
1: Mode	Mode	`MAV_MODE`	
2: Custom Mode	Custom mode - this is system specific, please refer to the individual autopilot specifications for details.		
3: Custom Submode	Custom sub mode - this is system specific, please refer to the individual autopilot specifications for details.		

Param	Description	Values	Units
4:			
5:			
6:			
7:			

MAV_CMD_DO_CHANGE_SPEED (178)

REQUIRED. Change speed and/or throttle set points.

Table 70: Change Speed Configuration

Param	Description	Values	Units
1: Speed Type	Speed type (0=Airspeed, 1=Ground Speed, 2=Climb Speed, 3=Descent Speed)	min:0 max:3 increment:1	
2: Speed	Speed (-1 indicates no change).	min: -1	m/s
3: Throttle	Throttle (-1 indicates no change).	min: -1	%
4:			
5:			
6:			
7:			

MAV_CMD_DO_SET_HOME (179)

REQUIRED. Sets the home position to either to the current position or a specified position. The home position is the default position that the system will return to and land on. The position is set automatically by the system during the takeoff (and may also be set using this command). Note: the current home position may be emitted in a `HOME_POSITION` message on request (using `MAV_CMD_REQUEST_MESSAGE` with param1=242).

Table 71: Set Home Position Configuration

Param	Description	Values	Units
1: Use Current	Use current (1=use current location, 0=use specified location)	min:0 max:1 increment:1	
2:			
3:			

Param	Description	Values	Units
4: Yaw	Yaw angle. NaN to use default heading		deg
5: Latitude	Latitude		
6: Longitude	Longitude		
7: Altitude	Altitude		m

MAV_CMD_DO_FLIGHTTERMINATION (185)

REQUIRED. Terminate flight immediately. Flight termination immediately and irreversibly terminates the current flight, returning the vehicle to ground. The vehicle will ignore RC or other input until it has been power-cycled. Termination may trigger safety measures, including: disabling motors and deployment of parachute on multicopters, and setting flight surfaces to initiate a landing pattern on fixed-wing). On multicopters without a parachute it may trigger a crash landing. Support for this command can be tested using the protocol bit: `MAV_PROTOCOL_CAPABILITY_FLIGHT_TERMINATION`. Support for this command can also be tested by sending the command with param1=0 (< 0.5); the ACK should be either `MAV_RESULT_FAILED` or `MAV_RESULT_UNSUPPORTED`.

Table 72: Flight Termination Configuration

Param	Description	Values	Units
1: Terminate	Flight termination activated if > 0.5. Otherwise not activated and ACK with `MAV_RESULT_FAILED`.	min:0 max:1 increment:1	
2:			
3:			
4:			
5:			
6:			
7:			

MAV_CMD_DO_CHANGE_ALTITUDE (186) **REQUIRED.** Change altitude set point.

Table 73: Change Altitude Navigation Configuration

Param	Description	Values	Units
1: Altitude	Altitude.		m
2: Frame	Frame of new altitude.	`MAV_FRAME`	
3:			
4:			
5:			
6:			
7:			

MAV_CMD_DO_REPOSITION (192)

REQUIRED. Reposition the vehicle to a specific WGS84 global position.

Table 74: Reposition Navigation Configuration

Param	Description	Values	Units
1: Speed	Ground speed, less than 0 (-1) for default	*min:* -1	m/s
2: Bitmask	Bitmask of option flags.	MAV_DO_REPOSITION_FLAGS	
3:	Reserved		
4: Yaw	Yaw heading. NaN to use the current system yaw heading mode (e.g. yaw towards next waypoint, yaw to home, etc.). For planes indicates loiter direction (0: clockwise, 1: counter clockwise)		deg
5: Latitude	Latitude		
6: Longitude	Longitude		
7: Altitude	Altitude		m

MAV_CMD_DO_PAUSE_CONTINUE (193)

REQUIRED. If in a GPS controlled position mode, hold the current position or continue.

Note: If the vehicle is in AUTO:MISSION mode, when processing this command, the vehicle should change to AUTO:LOITER mode. If in AUTO:LOITER mode, then when processing the command, the vehicle should change to AUTO:MISSION mode.

Table 75: Pause and Continue Navigation Configuration

Param	**Description**	**Values**
1: Continue	0: Pause current mission or reposition command, hold current position. 1: Continue the mission. A VTOL capable vehicle should enter hover mode (multicopter and VTOL planes). A plane should loiter with the default loiter radius.	*min:*0 *max:*1 *increment:*1
2:	Reserved	
3:	Reserved	
4:	Reserved	
5:	Reserved	
6:	Reserved	
7:	Reserved	

MAV_CMD_DO_SET_ROI_LOCATION (195)

Sets the region of interest (ROI) to a location. This can then be used by the vehicle's control system to control the vehicle attitude and the attitude of various sensors such as cameras. This command can be sent to a gimbal manager but not to a gimbal device. A gimbal is not to react to this message.

Table 76: ROI Location Configuration

Param	**Description**	**Values**	**Units**
1: Gimbal device ID	Component ID of gimbal device to address (or 1-6 for non-MAVLink gimbal), 0 for all gimbal device components. Send command multiple times for more than one gimbal (but not all gimbals).		
2:			
3:			
4:			

Param	Description	Values	Units
5: Latitude	Latitude of ROI location		degE7
6: Longitude	Longitude of ROI location		degE7
7: Altitude	Altitude of ROI location		m

MAV_CMD_DO_SET_ROI_WPNEXT_OFFSET (196)

Sets the region of interest (ROI) to be toward next waypoint, with optional pitch/roll/yaw offset. This can then be used by the vehicle's control system to control the vehicle attitude and the attitude of various sensors such as cameras. This command can be sent to a gimbal manager but not to a gimbal device. A gimbal device is not to react to this message.

Table 77: ROI Next Waypoint Offset Configuration

Param	Description	Values	Units
1: Gimbal device ID	Component ID of gimbal device to address (or 1-6 for non-MAVLink gimbal), 0 for all gimbal device components. Send command multiple times for more than one gimbal (but not all gimbals).		
2:			
3:			
4:			
5: Pitch Offset	Pitch offset from next waypoint, positive pitching up		rad
6: Roll Offset	Roll offset from next waypoint, positive rolling to the right		rad
7: Yaw Offset	Yaw offset from next waypoint, positive yawing to the right		rad

MAV_CMD_DO_SET_ROI_NONE (197)

Cancels any previous ROI command returning the vehicle/sensors to default flight characteristics. This can then be used by the vehicle's control system to control the vehicle attitude and the attitude of various sensors such as cameras. This command can be sent to a gimbal manager but not to a gimbal device. A gimbal device is not to react to this message. After this command the gimbal manager should go back to manual input if available, and otherwise assume a neutral position.

Table 78: Cancel ROI Configuration

Param	Description	Values	Units
1: Gimbal device ID	Component ID of gimbal device to address (or 1-6 for non-MAVLink gimbal), 0 for all gimbal device components. Send command multiple times for more than one gimbal (but not all gimbals).		
2:			
3:			
4:			
5:			
6:			
7:			

MAV_CMD_DO_MOUNT_CONTROL (205)

This command is deprecated, but still considered under this IOP while Gimbal Protocol v1 is marked as supported. To be removed on a next IOP release.

Mission command to control a camera or antenna mount. The message can still be used to communicate with legacy gimbals implementing it.

Table 79: Mount Control Configuration

Param	Description	Values	Units
1: Pitch	Pitch depending on mount mode (degrees or degrees/second depending on pitch input).		
2: Roll	Roll depending on mount mode (degrees or degrees/second depending on roll input).		
3: Yaw	Yaw depending on mount mode (degrees or degrees/second depending on yaw input).		
4: Altitude	Altitude depending on mount mode.		m
5: Latitude	Latitude, set if appropriate mount mode.		degE7

Param	Description	Values	Units
6: Longitude	Longitude, set if appropriate mount mode.		degE7
7: Mode	Mount mode.	`MAV_MOUNT_MODE`	

MAV_CMD_DO_SET_CAM_TRIGG_DIST (206)

Mission command to set camera trigger distance for this flight. The camera is triggered each time this distance is exceeded. This command can also be used to set the shutter integration time for the camera.

Table 80: Camera Trigger Distance Configuration

Param	Description	Values	Units
1: Distance	Camera trigger distance. 0 to stop triggering.	min:0	m
2: Shutter	Camera shutter integration time. -1 or 0 to ignore	min: -1 increment:1	ms
3: Trigger	Trigger camera once immediately. (0 = no trigger, 1 = trigger)	min:0 max:1 increment:1	
4:			
5:			
6:			
7:			

MAV_CMD_DO_SET_CAM_TRIGG_INTERVAL (214)

Mission command to set camera trigger interval for this flight. If triggering is enabled, the camera is triggered each time this interval expires. This command can also be used to set the shutter integration time for the camera.

Table 81: Camera Trigger Interval Configuration

Param	Description	Values	Units
1: Trigger Cycle	Camera trigger cycle time. -1 or 0 to ignore.	min: -1 increment:1	ms

Param	Description	Values	Units
2: Shutter Integration	Camera shutter integration time. Should be less than trigger cycle time. -1 or 0 to ignore.	min: -1 increment:1	ms
3:			
4:			
5:			
6:			
7:			

MAV_CMD_PREFLIGHT_CALIBRATION (241)

Trigger calibration. This command will be only accepted if in pre-flight mode. Except for Temperature Calibration, only one sensor should be set in a single message and all others should be zero.

Note: This command is sent by QGC-Gov when executing a sensor calibration procedure.

Table 82: Pre Flight Calibration

Param	Description	Values
1: Gyro Temperature	1: gyro calibration, 3: gyro temperature calibration	*min:*0 *max:*3 *increment:*1
2: Magnetometer	1: magnetometer calibration	*min:*0 *max:*1 *increment:*1
3: Ground Pressure	1: ground pressure calibration	*min:*0 *max:*1 *increment:*1
4: Remote Control	1: radio RC calibration, 2: RC trim calibration	*min:*0 *max:*1 *increment:*1
5: Accelerometer	1: accelerometer calibration, 2: board level calibration, 3: accelerometer temperature calibration, 4: simple accelerometer calibration	*min:*0 *max:*4 *increment:*1
6: Compass or Airspeed	1: Compass/motor interference calibration, 2: airspeed calibration	*min:*0 *max:*2 *increment:*1
7: ESC or Baro	1: ESC calibration, 3: barometer temperature calibration	*min:*0 *max:*3 *increment:*1

MAV_CMD_MISSION_START (300)

REQUIRED. Start running a mission.

Note: this command should change the the vehicle to AUTO:MISSION mode.

Table 83: Mission Start Configuration

Param	Description	Values	Units
1: First Item	first_item: the first mission item to run.	min:0 increment:1	
2: Last Item	last_item: the last mission item to run (after this item is run, the mission ends)	min: 0 increment:1	
3:			
4:			
5:			
6:			
7:			

MAV_CMD_COMPONENT_ARM_DISARM (400)

REQUIRED. Arms / Disarms a component. Used to arm / disarm a vehicle. While in air, this command should only execute if param2=21196.

This command can be process as a long running command.

Table 84: Arm and Disarm

Param	Description	Values	Units
1: Arm	0: Disarm, 1: Arm	min:0 max:1 increment:1	
2: Force	0: Arm-disarm unless prevented by safety checks (i.e. when landed), 21196: force arming/disarming (e.g. allow arming to override preflight checks and disarming in flight)	min:0 max:21196 increment:21196	
3:			
4:			
5:			
6:			
7:			

MAV_CMD_START_RX_PAIR (500)

Starts receiver pairing.

Note: This command is required when the vehicle supports datalink radio pairing and there is no direct connection between pairing trigger (for example, a pairing or safety button press), and the service running the Pairing Slave. An example of this scenario is when the pairing button is connected to an embedded autopilot flight controller board, and the Pairing Slave is running on a connected companion onboard computer.

Table 85: Pairing Start Configuration

Param	Description	Values	Units
1: Spektrum	0: Spektrum.		
2: RC Type	RC type.	`RC_TYPE`	
3:			
4:			
5:			
6:			
7:			

MAV_CMD_SET_MESSAGE_INTERVAL (511)

REQUIRED. Set the interval between messages for a particular MAVLink message ID. This interface replaces `REQUEST_DATA_STREAM`.

Note: Under this IOP, any required and (supported) optional message stream should be have its interval configurable through this command.

Table 86: Message Interval Configuration

Param	Description	Values	Units
1: Message ID	The MAVLink message ID	min:0 max:16777215 increment:1	
2: Interval	The interval between two messages. Set to -1 to disable and 0 to request default rate.	min: -1 increment:1	us
3:			

Param	Description	Values	Units
4:			
5:			
6:			
7: Response Target	Target address of message stream (if message has target address fields). 0: Flight-stack default (recommended), 1: address of requestor, 2: broadcast.	min:0 max:2 increment:1	

MAV_CMD_REQUEST_MESSAGE (512)

REQUIRED. Request the target system(s) emit a single instance of a specified message (i.e. a "one-shot" version of `MAV_CMD_SET_MESSAGE_INTERVAL`). The list of the required messages, that should be sent when requested through this command, can be found below:

- `HOME_POSITION`
- `FLIGHT_INFORMATION`
- `MESSAGE_INTERVAL`
- `PROTOCOL_VERSION` (Handshake)
- `AUTOPILOT_VERSION` (Handshake)
- `CAMERA_INFORMATION` (Camera Protocol, when supported)
- `CAMERA_SETTINGS` (Camera Protocol, when supported)
- `CAMERA_CAPTURE_STATUS` (Camera Protocol, when supported)
- `CAMERA_IMAGE_CAPTURED` (Camera Protocol, when supported)
- `STORAGE_INFORMATION` (Camera Protocol, when supported)
- `VIDEO_STREAM_STATUS` (Camera Protocol, when supported)
- `VIDEO_STREAM_INFORMATION` (Camera Protocol, when supported)
- `GIMBAL_MANAGER_INFORMATION` (Gimbal Protocol v2, when supported)
- `GIMBAL_DEVICE_INFORMATION` (Gimbal Protocol v2, when supported)

Table 87: Request Message Configuration

Param	Description	Values
1: Message ID	The MAVLink message ID of the requested message.	*min:*0 *max:*16777215 *increment:*1
2: Req Param 1	Use for index ID, if required. Otherwise, the use of this parameter (if any) must be defined in the requested message. By default assumed not used (0).	

Param	Description	Values
3: Req Param 2	The use of this parameter (if any), must be defined in the requested message. By default assumed not used (0).	
4: Req Param 3	The use of this parameter (if any), must be defined in the requested message. By default assumed not used (0).	
5: Req Param 4	The use of this parameter (if any), must be defined in the requested message. By default assumed not used (0).	
6: Req Param 5	The use of this parameter (if any), must be defined in the requested message. By default assumed not used (0).	
7: Response Target	Target address for requested message (if message has target address fields). 0: Flight-stack default, 1: address of requester, 2: broadcast.	*min:*0 *max:*2 *increment:*1

MAV_CMD_SET_CAMERA_MODE (530)

Set camera running mode. Use NaN for reserved values. GCS will send a `MAV_CMD_REQUEST_VIDEO_STREAM_STATUS` command after a mode change if the camera supports video streaming.

Table 88: Camera Mode Configuration

Param	Description	Values	Units
1:			
2: Camera Mode	Camera mode	`CAMERA_MODE`	
3:			
4:			
5:			
6:			
7:			

MAV_CMD_SET_CAMERA_ZOOM (531)

Set camera zoom. Camera must respond with a `CAMERA_SETTINGS` message (on success).

Table 89: Camera Zoom Configuration

Param	Description	Values	Units
1: Zoom Type	Zoom type	`CAMERA_ZOOM_TYPE`	
2: Zoom Value	Zoom value. The range of valid values depend on the zoom type.		
3:			
4:			
5:			
6:			
7:			

MAV_CMD_SET_CAMERA_FOCUS (532)

Set camera focus. Camera must respond with a `CAMERA_SETTINGS` message (on success).

Table 90: Camera Focus Configuration

Param	Description	Values	Units
1: Focus Type	Focus type	`SET_FOCUS_TYPE`	
2: Focus Value	Focus value.		
3:			
4:			
5:			
6:			
7:			

MAV_CMD_SET_STORAGE_USAGE (533)

Set that a particular storage is the preferred location for saving photos, videos, and/or other media (e.g. to set that an SD card is used for storing videos). There can only be one preferred save location for each particular media type: setting a media usage flag will clear/reset that same flag if set on any other storage. If no flag is set the system should use its default storage. A target system can choose to always use default storage, in which case it should ACK the command with `MAV_RESULT_UNSUPPORTED`. A target system can choose to not allow a particular storage to be set as preferred storage, in which case it should ACK the command with `MAV_RESULT_DENIED`.

Table 91: Storage Usage Configuration

Param	Description	Values	Units
1: Storage ID	Storage ID (1 for first, 2 for second, etc.)	min:0 increment:1	
2: Usage	Usage flags.	`STORAGE_USAGE_FLAG`	
3:			
4:			
5:			
6:			
7:			

MAV_CMD_DO_GIMBAL_MANAGER_PITCHYAW (1000) REQUIRED when using Gimbal Protocol V2. High level setpoint to be sent to a gimbal manager to set a gimbal attitude. It is possible to set combinations of the values below. E.g. an angle as well as a desired angular rate can be used to get to this angle at a certain angular rate, or an angular rate only will result in continuous turning. `NaN` is to be used to signal unset.

Note: a gimbal is never to react to this command but only the gimbal manager.

Table 92: Gimbal Manager Configuration I

Param	Description	Values	Units
1: Pitch angle	Pitch angle (positive to pitch up, relative to vehicle for FOLLOW mode, relative to world horizon for LOCK mode).	*min:* -180 *max:*180	deg
2: Yaw angle	Yaw angle (positive to yaw to the right, relative to vehicle for FOLLOW mode, absolute to North for LOCK mode).	*min:* -180 *max:*180	deg
3: Pitch rate	Pitch rate (positive to pitch up).		deg/s
4: Yaw rate	Yaw rate (positive to yaw to the right).		deg/s
5: Gimbal manager flags	Gimbal manager flags to use.	GIMBAL_MANAGER_FLAGS	

Param	Description	Values	Units
7: Gimbal device ID	Component ID of gimbal device to address (or 1-6 for non-MAVLink gimbal), 0 for all gimbal device components. Send command multiple times for more than one gimbal (but not all gimbals).		

MAV_CMD_DO_GIMBAL_MANAGER_CONFIGURE (1001) REQUIRED when using Gimbal Protocol V2. Gimbal configuration to set which sysid/compid is in primary and secondary control.

Table 93: Gimbal Manager Configuration II

Param	Description
1: sysid primary control	Sysid for primary control (0: no one in control, -1: leave unchanged, -2: set itself in control (for missions where the own sysid is still unknown), -3: remove control if currently in control).
2: compid primary control	Compid for primary control (0: no one in control, -1: leave unchanged, -2: set itself in control (for missions where the own sysid is still unknown), -3: remove control if currently in control).
3: sysid secondary control	Sysid for secondary control (0: no one in control, -1: leave unchanged, -2: set itself in control (for missions where the own sysid is still unknown), -3: remove control if currently in control).
4: compid secondary control	Compid for secondary control (0: no one in control, -1: leave unchanged, -2: set itself in control (for missions where the own sysid is still unknown), -3: remove control if currently in control).
7: Gimbal device ID	Component ID of gimbal device to address (or 1-6 for non-MAVLink gimbal), 0 for all gimbal device components. Send command multiple times for more than one gimbal (but not all gimbals).

MAV_CMD_IMAGE_START_CAPTURE (2000)

Start image capture sequence. Sends `CAMERA_IMAGE_CAPTURED` after each capture. Use NaN for reserved values.

Table 94: Start Capture Configuration

Param	Description	Values	Units
1:	Reserved (Set to 0)		
2: Interval	Desired elapsed time between two consecutive pictures (in seconds). Minimum values depend on hardware (typically greater than 2 seconds).	*min:*0	s
3: Total Images	Total number of images to capture. 0 to capture forever/until MAV_CMD_IMAGE_STOP_CAPTURE.	*min:*0 *increment:*1	
4: Sequence Number	Capture sequence number starting from 1. This is only valid for single-capture (param3 == 1), otherwise set to 0. Increment the capture ID for each capture command to prevent double captures when a command is re-transmitted.	*min:*1 *increment:*1	
5:			
6:			
7:			

MAV_CMD_IMAGE_STOP_CAPTURE (2001)
Stop image capture sequence Use NaN for reserved values.

Table 95: Stop Capture Configuration

Param	Description	Values	Units
1:			
2:			
3:			
4:			
5:			
6:			
7:			

MAV_CMD_DO_TRIGGER_CONTROL (2003)

Enable or disable on-board camera triggering system.

Table 96: Trigger Control Configuration

Param	Description	Values	Units
1: Enable	Trigger enable/disable (0 for disable, 1 for start), -1 to ignore	min: -1 max:1 increment:1	
2: Reset	1 to reset the trigger sequence, -1 or 0 to ignore	min: -1 max:1 increment:1	
3: Pause	1 to pause triggering, but without switching the camera off or retracting it. -1 to ignore	min: -1 max:1 increment:2	
4:			
5:			
6:			
7:			

MAV_CMD_CAMERA_TRACK_POINT (2004)
If the camera supports point visual tracking (`CAMERA_CAP_FLAGS_HAS_TRACKING_POINT` is set), this command allows to initiate the tracking.

Table 97: Track Point Configuration

Param	Description	Values
1: Point x	Point to track x value (normalized 0..1, 0 is left, 1 is right).	*min:*0 *max:*1
2: Point y	Point to track y value (normalized 0..1, 0 is top, 1 is bottom).	*min:*0 *max:*1
3: Radius	Point radius (normalized 0..1, 0 is image left, 1 is image right).	*min:*0 *max:*1

MAV_CMD_CAMERA_TRACK_RECTANGLE (2005)
If the camera supports rectangle visual tracking (`CAMERA_CAP_FLAGS_HAS_TRACKING_RECTANGLE` is set), this command allows to initiate the tracking.

Table 98: Track Rectangle Configuration

Param	Description	Values
1: Top left corner x	Top left corner of rectangle x value (normalized 0..1, 0 is left, 1 is right).	*min:*0 *max:*1

Param	Description	Values
2: Top left corner y	Top left corner of rectangle y value (normalized 0..1, 0 is top, 1 is bottom).	*min:*0 *max:*1
3: Bottom right corner x	Bottom right corner of rectangle x value (normalized 0..1, 0 is left, 1 is right).	*min:*0 *max:*1
4: Bottom right corner y	Bottom right corner of rectangle y value (normalized 0..1, 0 is top, 1 is bottom).	*min:*0 *max:*1

MAV_CMD_CAMERA_STOP_TRACKING (2010)

Stops ongoing tracking.

Table 99: Stop Tracking Configuration

Param	Description

MAV_CMD_VIDEO_START_CAPTURE (2500)

Starts video capture (recording).

Table 100: Video Start Capture Configuration

Param	Description	Values	Units
1: Stream ID	Video Stream ID (0 for all streams)	*min:*0 *increment:*1	
2: Status Frequency	Frequency CAMERA_CAPTURE_STATUS messages should be sent while recording (0 for no messages, otherwise frequency)	*min:*0	Hz
3:			
4:			
5:			
6:			
7:			

MAV_CMD_VIDEO_STOP_CAPTURE (2501) Stop the current video capture (recording).

Table 101: Video Stop Capture Configuration

Param	Description	Values
1: Stream ID	Video Stream ID (0 for all streams)	*min:*0 *increment:*1
2:		
3:		
4:		
5:		
6:		
7:		

MAV_CMD_DO_VTOL_TRANSITION (3000) REQUIRED for VTOL aircraft.
Request VTOL transition.

Table 102: VTOL Transition Configuration

Param	Description	Values
1: State	The target VTOL state. For normal transitions, only `MAV_VTOL_STATE_MC` and `MAV_VTOL_STATE_FW` can be used.	`MAV_VTOL_STATE`
2: Immediate	Force immediate transition to the specified `MAV_VTOL_STATE`. 1: Force immediate, 0: normal transition. Can be used, for example, to trigger an emergency “Quadchute”. Caution: Can be a dangerous/damaged vehicle, depending on autopilot implementation of this command.	

MAV_CMD_NAV_FENCE_POLYGON_VERTEX_INCLUSION (5001)

Fence vertex for an inclusion polygon (the polygon must not be self-intersecting). The vehicle must stay within this area. Minimum of 3 vertices required.

Table 103: Geofence Polygon Vertex Inclusion Configuration

Param	Description	Values	Units
1: Vertex Count	Polygon vertex count	min:3 increment:1	
2: Inclusion Group	Vehicle must be inside ALL inclusion zones in a single group, vehicle must be inside at least one group, must be the same for all points in each polygon	min:0 increment:1	
3:			
4:			
5: Latitude	Latitude		
6: Longitude	Longitude		
7:			

MAV_CMD_NAV_FENCE_POLYGON_VERTEX_EXCLUSION (5002)

Fence vertex for an exclusion polygon (the polygon must not be self-intersecting). The vehicle must stay outside this area. Minimum of 3 vertices required.

Table 104: Geofence Polygon Vertex Inclusion Configuration

Param	Description	Values	Units
1: Vertex Count	Polygon vertex count	min:3 increment:1	
2:			
3:			
4:			
5: Latitude	Latitude		
6: Longitude	Longitude		
7:			

MAV_CMD_NAV_FENCE_CIRCLE_INCLUSION (5003)

Circular fence area. The vehicle must stay inside this area.

Table 105: Geofence Circle Inclusion Configuration

Param	Description	Values	Units
1: Radius	Polygon vertex count		m
2: Inclusion Group	Vehicle must be inside ALL inclusion zones in a single group, vehicle must be inside at least one group.	min:0 increment:1	
3:			
4:			
5: Latitude	Latitude		
6: Longitude	Longitude		
7:			

MAV_CMD_NAV_FENCE_CIRCLE_EXCLUSION (5004)

Circular fence area. The vehicle must stay outside this area.

Table 106: Geofence Circle Exclusion Configuration

Param	Description	Values	Units
1: Radius	Polygon vertex count		m
2:			
3:			
4:			
5: Latitude	Latitude		
6: Longitude	Longitude		
7:			

MAV_CMD_NAV_RALLY_POINT (5100)

Rally point. You can have multiple rally points defined.

Table 107: Rally Point Configuration

Param	Description	Values	Units
1:			
2:			
3:			

Param	Description	Values	Units
4:			
5: Latitude	Latitude		
6: Longitude	Longitude		
7: Altitude			m

MAV_RESULT

[Enum] Result from a MAVLink command (MAV_CMD)

Table 108: MAVResult Enum

Value	Field Name	Description
0	`MAV_RESULT_ACCEPTED`	Command is valid (is supported and has valid parameters), and was executed.
1	`MAV_RESULT_TEMPORARILY_REJECTED`	Command is valid, but cannot be executed at this time. This is used to indicate a problem that should be fixed just by waiting (e.g. a state machine is busy, can't arm because it does not have a GPS lock, etc.). Retrying later should work.
2	`MAV_RESULT_DENIED`	Command is invalid (is supported but has invalid parameters). Retrying the same command and parameters will not work.
3	`MAV_RESULT_UNSUPPORTED`	Command is not supported (unknown).
4	`MAV_RESULT_FAILED`	Command is valid, but execution has failed. This is used to indicate any non-temporary or unexpected problem, i.e. any problem that must be fixed before the command can succeed/be retried. For example, attempting to write a file when out of memory, attempting to arm when sensors are not calibrated, etc.

Value	Field Name	Description
5	MAV_RESULT_IN_PROGRESS	Command is valid and is being executed. This will be followed by further progress updates, i.e. the component may send further COMMAND_ACK messages with result MAV_RESULT_IN_PROGRESS (at a rate decided by the implementation), and must terminate by sending a COMMAND_ACK message with final result of the operation. The COMMAND_ACK.progress field can be used to indicate the progress of the operation.
6	MAV_RESULT_CANCELLED	Command has been cancelled (as a result of receiving a COMMAND_CANCEL message).

Camera Protocol

The camera protocol is used to configure camera payloads and request their status. It supports photo capture, video capture, and streaming. It also includes messages to query and configure the onboard camera storage.

The Dronecode Camera Manager provides an example implementation of this protocol.

We are transitioning from specific request commands to a single generic requestor. GCS and MAVLink SDKs/apps should support both approaches as we migrate to exclusive use of the new method (documented here). For more information see Migration Notes for GCS & Camera Servers.

Camera Connection

Camera components are expected to follow the Heartbeat/Connection Protocol and send a constant flow of heartbeats (nominally at 1Hz). Each camera must use a different predefined camera component ID: `MAV_COMP_ID_CAMERA` to `MAV_COMP_ID_CAMERA6`.

The first time a heartbeat is detected from a new camera, a GCS (or other receiving system) should start the Camera Identification process.

If a receiving system stops receiving heartbeats from the camera it is assumed to be disconnected, and should be removed from the list of available cameras. If heartbeats are again detected, the camera identification process below must be restarted from the beginning.

Basic Camera Operations

The CAMERA_INFORMATION.flags provides information about camera capabilities. It contains a bitmap of CAMERA_CAP_FLAGS values that tell the GCS if the camera supports still image capture, video capture, or video streaming, and if it needs to be in a certain mode for capture, etc.

Camera Identification The camera identification operation identifies all the available cameras and determines their capabilities.

Camera identification must be carried out before all other operations!

The first time a heartbeat is received from a new camera component, the GCS will send it a MAV_CMD_REQUEST_MESSAGE message asking for CAMERA_INFORMATION (message id 259). The camera will then respond with a COMMAND_ACK message containing a result. On success (result is MAV_RESULT_ACCEPTED) the camera component must then send a CAMERA_INFORMATION message.

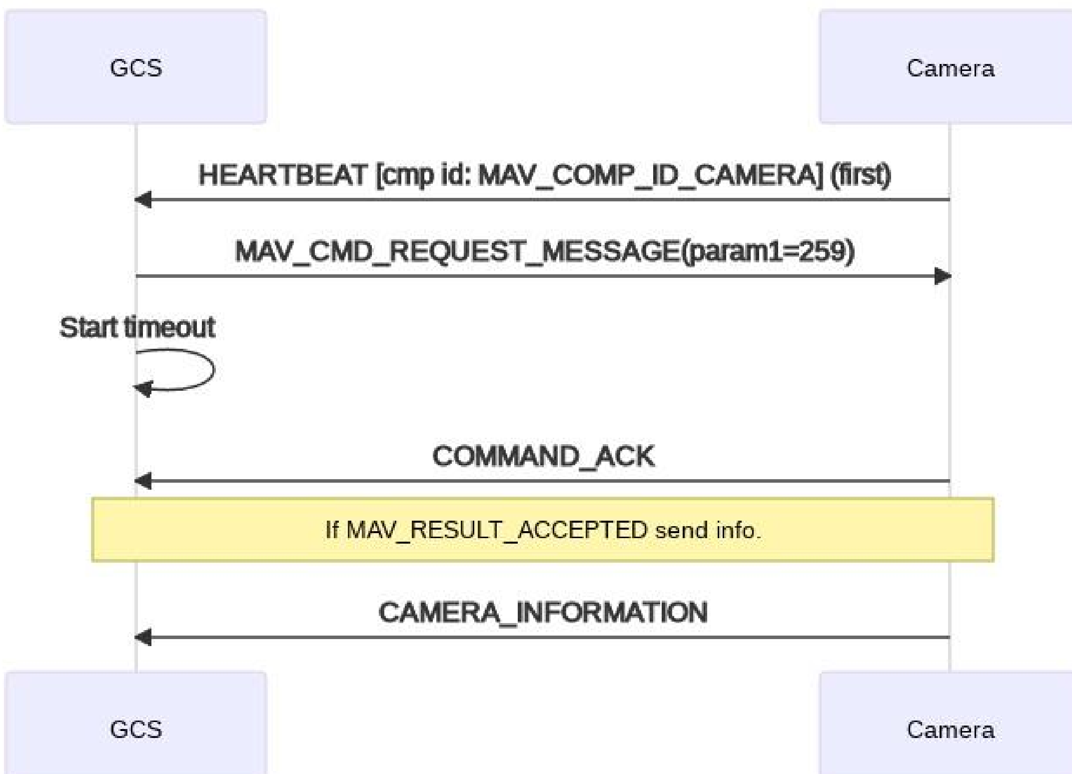

Figure 27: Camera Identification Diagram

The operation follows the normal Command Protocol rules for command/acknowledgment (if no COMMAND_ACK response is received for MAV_CMD_REQUEST_MESSAGE the command will be re-sent a number of times before failing). If CAMERA_INFORMATION is not received after receiving an ACK with MAV_RESULT_ACCEPTED, the protocol assumes the message was lost, and the cycle of sending MAV_CMD_REQUEST_MESSAGE is repeated. If CAMERA_INFORMATION is still not received after three cycle repeats, the GCS may assume that the camera is not supported.

The CAMERA_INFORMATION response contains the bare minimum information about the camera and what it can or cannot do. This is sufficient for basic image and/or video capture.

If a camera provides finer control over its settings CAMERA_INFORMATION.cam_definition_uri will include a URI to a Camera Definition File. If this URI exists, the GCS will request it, parse it and prepare the UI for the user to control the camera settings.

A GCS that implements this protocol is expected to support HTTP (http://) or MAVLink FTP (mavlinkftp://) URIs for download of the camera definition file. If the camera provides an HTTP or MAVLink FTP interface, the definition file can be hosted on the camera itself. Otherwise, it can be hosted anywhere (on any reachable server).

The CAMERA_INFORMATION.cam_definition_version field should provide a version for the definition file, allowing the GCS to cache it. Once downloaded, it would only be requested again if the version number changes.

If a vehicle has more than one camera, each camera will have a different component ID and send its own heartbeat. The GCS should create multiple instances of a camera controller based on the component ID of each camera. All commands are sent to a specific camera by addressing the command to a specific component ID.

Camera Modes Some cameras must be in a certain mode for still and/or video capture.

The GCS can determine if it needs to make sure the camera is in the proper mode prior to sending a start capture (image or video) command by checking whether the CAMERA_CAP_FLAGS_HAS_MODES bit is set true in CAMERA_INFORMATION.flags.

In addition, some cameras can capture images in any mode but with different resolutions. For example, a 20 megapixel camera would take a full resolution image when set to CAMERA_MODE_IMAGE but only at the current video resolution if it is set to CAMERA_MODE_VIDEO.

To get the current mode, the GCS would send a MAV_CMD_REQUEST_MESSAGE command asking for CAMERA_SETTINGS. The camera component will then respond with a COMMAND_ACK message containing a result. On success (COMMAND_ACK.result is MAV_RESULT_ACCEPTED) the camera must then send a CAMERA_SETTINGS message. The current mode is the CAMERA_SETTINGS.mode_id field.

The sequence is shown below:

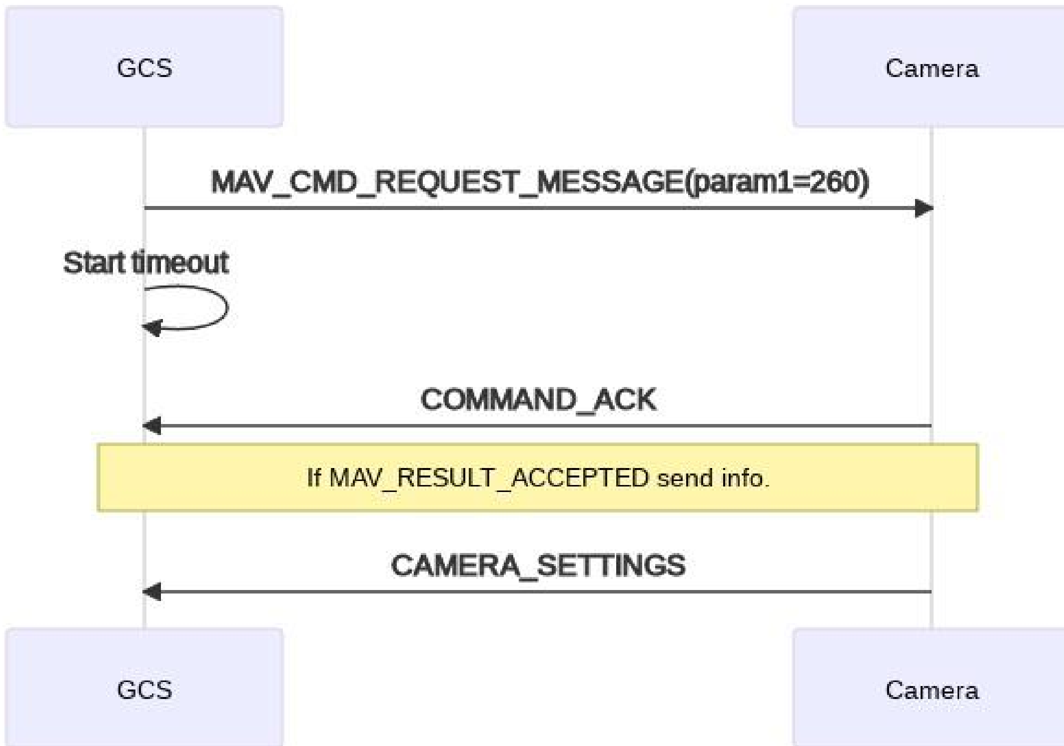

Figure 28: Camera Setting Diagram

Command acknowledgment and message resending is handled in the same way as for camera identification (if a successful ACK is received the camera will expect the CAMERA_SETTINGS message, and repeat the cycle - up to 3 times - until it is received).

To set the camera to a specific mode, the GCS would send the `MAV_CMD_SET_CAMERA_MODE` command with the appropriate mode.

The sequence is shown below:

The operation follows the normal Command Protocol rules for command/acknowledgment.

Storage Status Before capturing images and/or videos, a GCS should query the storage status to determine if the camera has enough free space for these operations (and provide the user with feedback as to the current storage status). The GCS will send the `MAV_CMD_REQUEST_MESSAGE` command and it expects a `COMMAND_ACK` message back as well as a `STORAGE_INFORMATION` response. For formatting (or erasing depending on your implementation), the GCS will send a `MAV_CMD_STORAGE_FORMAT` command.

Camera Capture Status In addition to querying about storage status, the GCS will also request the current *Camera Capture Status* in order to provide

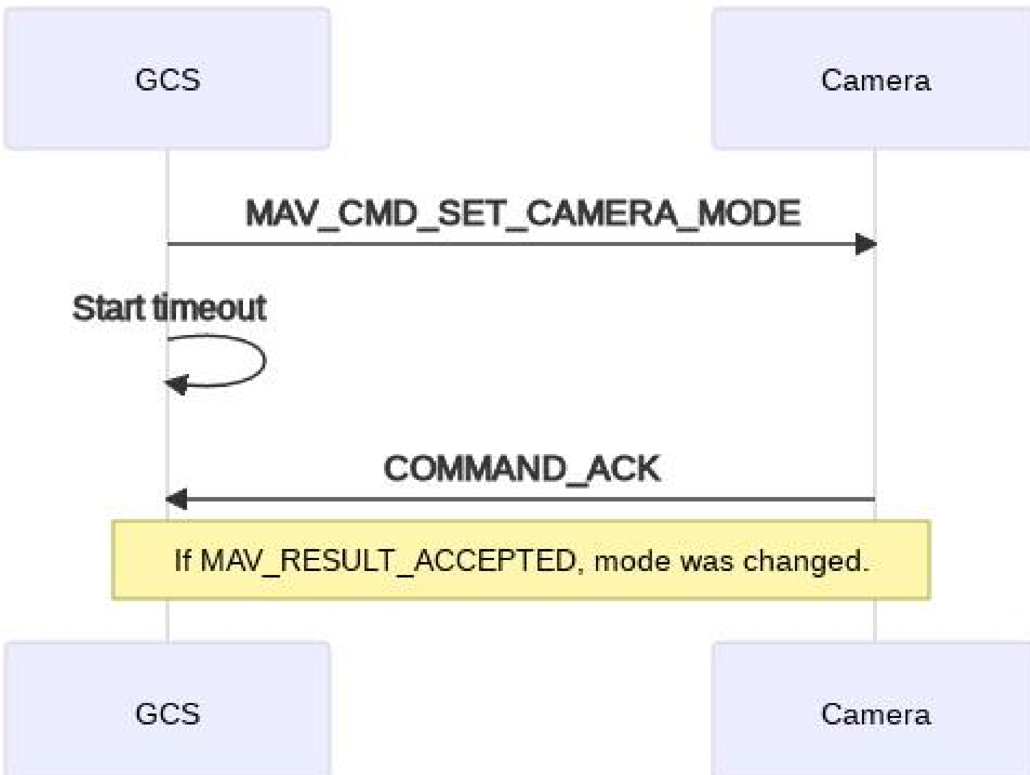

Figure 29: Specific Mode Camera Setting Diagram

the user with proper UI indicators. The GCS will send a `MAV_CMD_REQUEST_MESSAGE` command asking for `CAMERA_CAPTURE_STATUS` and it expects a `COMMAND_ACK` message back as well as a `CAMERA_CAPTURE_STATUS` response.

Still Image Capture A camera supports *still image capture* if the `CAMERA_CAP_FLAGS_CAPTURE_IMAGE` bit is set in `CAMERA_INFORMATION.flags`.

A GCS/MAVLink app uses the `MAV_CMD_IMAGE_START_CAPTURE` command to request that the camera capture a specified number of images (or forever), and the duration between them. The camera immediately returns the normal command acknowledgment (`MAV_RESULT_ACCEPTED`).

Each time an image is captured, the camera *broadcasts* a `CAMERA_IMAGE_CAPTURED` message. This message not only tells the GCS the image was captured, it is also intended for geo-tagging.

The camera must iterate `CAMERA_IMAGE_CAPTURED.image_index` and the counter used in `CAMERA_CAPTURE_STATUS.image_count` for every new image capture (these values iterate until explicitly cleared using `MAV_CMD_STORAGE_FORMAT`). The index and total image count can be used to re-request missing images (e.g. images captured when the vehicle was out of telemetry range).

The `MAV_CMD_IMAGE_STOP_CAPTURE` command can optionally be sent to stop an image capture sequence (this is needed if image capture has been set to continue forever).

The still image capture message sequence *for missions* (as described above) is shown below:

The message sequence for *interactive user-initiated image capture* through a GUI is slightly different. In this case the GCS should:

- Confirm that the camera is *ready* to take images before allowing the user to request image capture.
  - It does this by sending `MAV_CMD_REQUEST_MESSAGE` asking for `CAMERA_CAPTURE_STATUS`.
  - The camera should return a `MAV_RESULT` and then `CAMERA_CAPTURE_STATUS`.
  - The GCS should check that the status is “Idle” before enabling camera capture in the GUI.
- Send `MAV_CMD_IMAGE_START_CAPTURE` specifying a single image (only).

The sequence is as shown below:

Request Lost CAMERA_IMAGE_CAPTURED Messages The camera broadcasts a `CAMERA_IMAGE_CAPTURED` every time a new image is captured, iterating

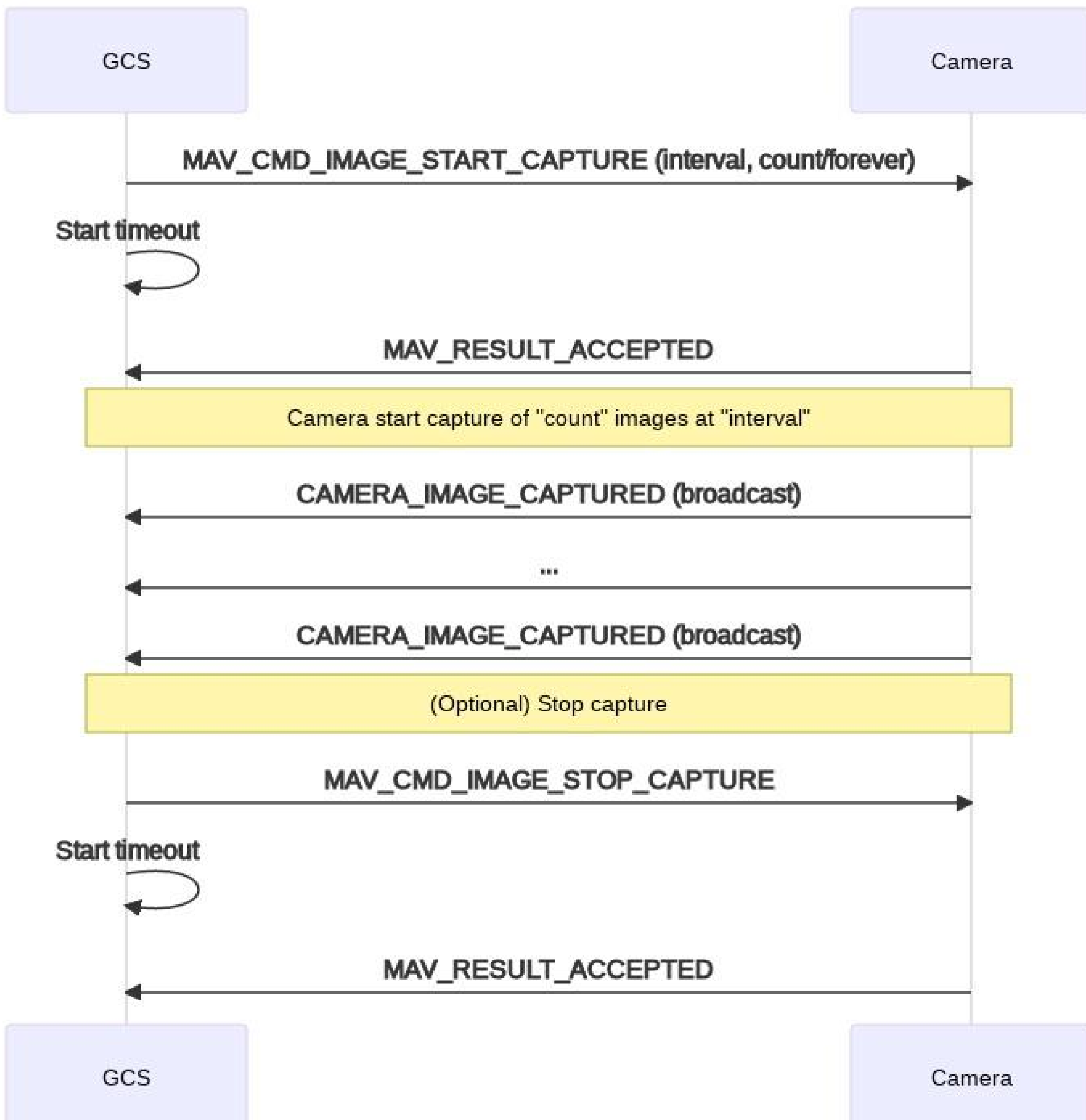

Figure 30: Image Capturing Diagram

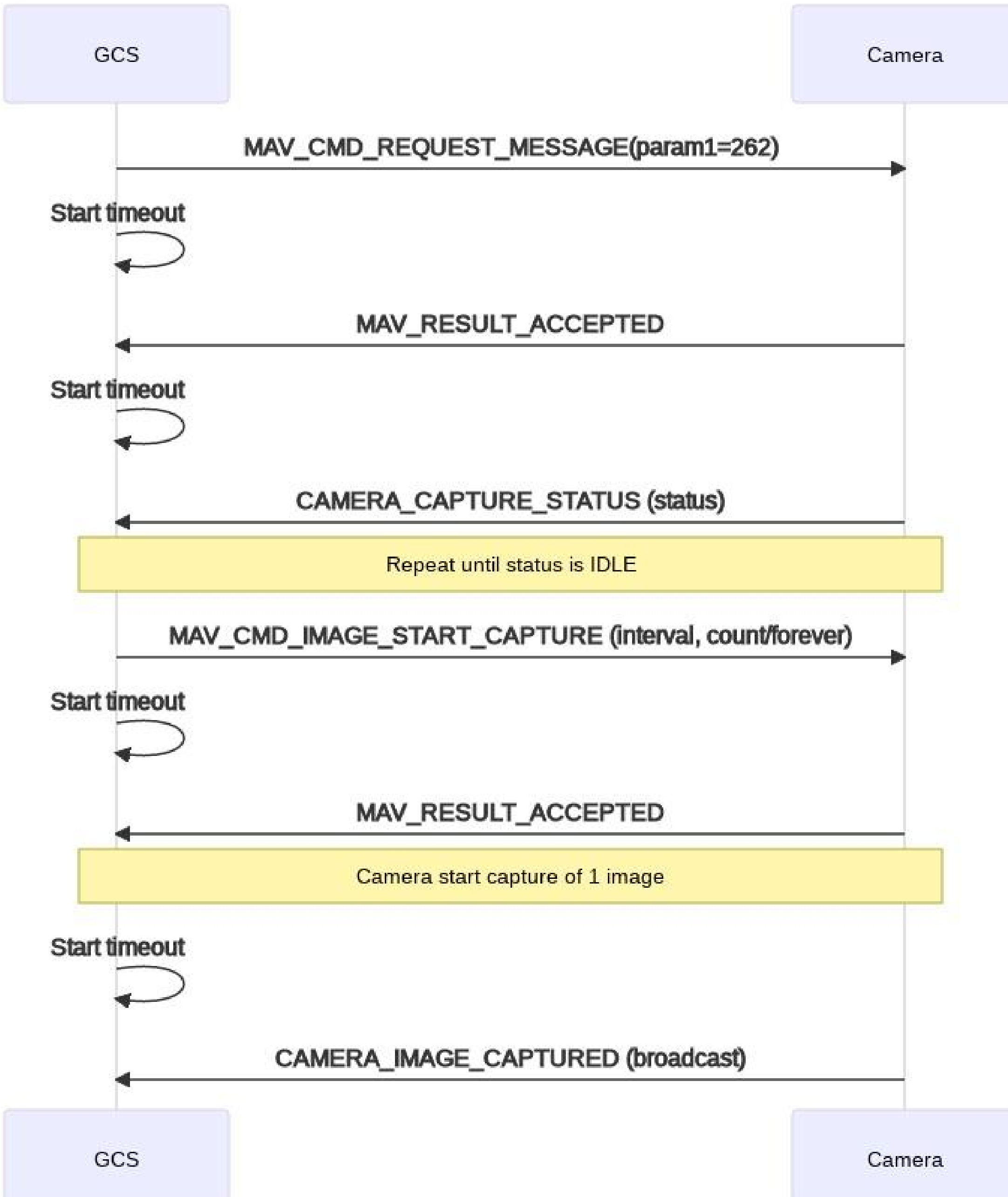

Figure 31: Single Image Capturing Diagram

both the current image index (`CAMERA_IMAGE_CAPTURED.image_index`) and the total image count (CAMERA_CAPTURE_STATUS.image_count).

These messages can be lost during transmission; for example if the vehicle is out of datalink range of the ground stations.

Lost image capture messages can be detected by comparing GCS and camera image counts. Individual entries can be requested using `MAV_CMD_REQUEST_MESSAGE`, where `param1="MAVLINK_MSG_ID_CAMERA_IMAGE_CAPTURED"` and `param2="the index of the missing image"`.

The camera image log iterates "forever" (but may be explicitly reset using `MAV_CMD_STORAGE_FORMAT.param3=1`).

Video Capture A camera supports video capture if the `CAMERA_CAP_FLAGS_CAPTURE_VIDEO` bit is set in `CAMERA_INFORMATION.flags`.

To start recording videos, the GCS uses the `MAV_CMD_VIDEO_START_CAPTURE` command. If requested, the `CAMERA_CAPTURE_STATUS` message is sent to the GCS at a set interval.

To stop recording, the GCS uses the `MAV_CMD_VIDEO_STOP_CAPTURE` command.

Video Streaming The GCS should already have identified all connected cameras by their heartbeat and followed the Camera Identification steps to get `CAMERA_INFORMATION` for every camera.

A camera is capable of streaming video if it sets the `CAMERA_CAP_FLAGS_HAS_VIDEO_STREAM` bit set in `CAMERA_INFORMATION.flags`.

The sequence for requesting *all* video streams from a particular camera is shown below:

The steps are:

1. GCS follows the Camera Identification steps to get `CAMERA_INFORMATION` for every camera.
2. GCS checks if `CAMERA_INFORMATION.flags` contains the `CAMERA_CAP_FLAGS_HAS_VIDEO_STREAM` flag.
3. If so, the GCS sends the `MAV_CMD_REQUEST_MESSAGE` message to the camera requesting the video streaming configuration (param1=269) for all streams (`param2=0`). A GCS can also request information for a particular stream by setting its id in param2.
4. Camera returns a `VIDEO_STREAM_INFORMATION` message for the specified stream or all streams it supports.

If your camera only provides video streaming and nothing else (no camera features), the `CAMERA_CAP_FLAGS_HAS_VIDEO_STREAM` flag is the only flag you need

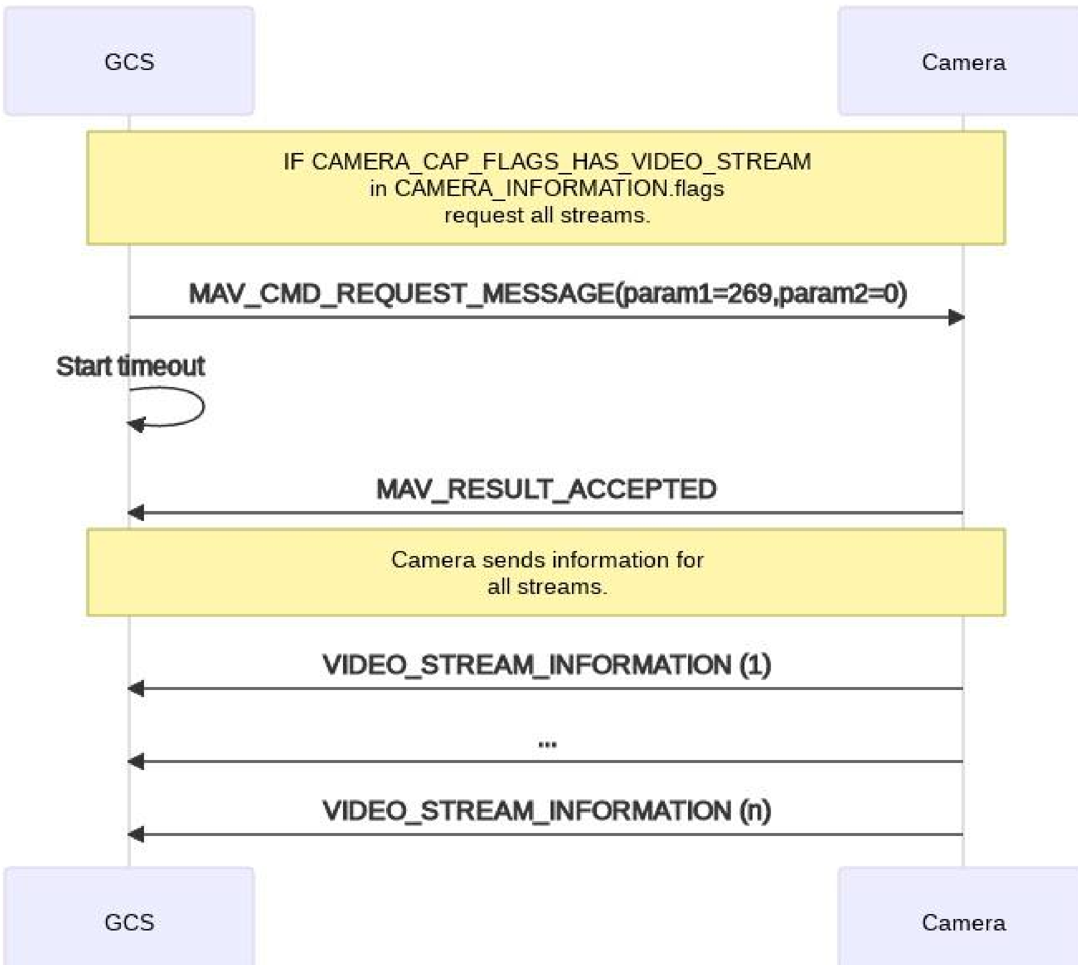

Figure 32: Video Streaming Diagram

to set. The GCS will then provide video streaming support and skip camera control.

Note: Each camera component is supposed to report a single video stream URI. A different video stream or source should be associated with a different camera component ID.

Battery Status Camera components that are powered from their own battery should publish BATTERY_STATUS messages.

Other components like a GCS will typically only use the camera BATTERY_STATUS.battery_remaining field (or possibly time_remaining); generally other fields can be set as "not supported".

Message/Enum Summary

Table 109: Messages

Message	Description	Status
MAV_CMD_REQUEST_MESSAGE	Send command to request any message	to be used
CAMERA_INFORMATION	Basic information about camera including supported features and URI link to extended information (cam_definition_uri field).	
CAMERA_SETTINGS	Timestamp and camera mode information.	
MAV_CMD_SET_CAMERA_MODE	Send command to set CAMERA_MODE.	
VIDEO_STREAM_INFORMATION	Information defining a video stream configuration. If a camera has more than one video stream, it would send one of these for each video stream, with their specific configuration. Each stream must have its own, unique stream_id.	
VIDEO_STREAM_STATUS	Information updating a video stream configuration.	
STORAGE_INFORMATION	Storage information (e.g. number and type of storage devices, total/used/available capacity, read/write speeds).	
MAV_CMD_STORAGE_FORMAT	Send command to format the specified storage device.	

Message	Description	Status
CAMERA_CAPTURE_STATUS	Camera capture status, including current capture type (if any), capture interval, available capacity.	
MAV_CMD_IMAGE_START_CAPTURE	Send command to start image capture, specifying the duration between captures and total number of images to capture.	
MAV_CMD_IMAGE_STOP_CAPTURE	Send command to stop image capture.	
CAMERA_IMAGE_CAPTURED	Information about image captured (returned to GPS every time an image is captured).	
MAV_CMD_VIDEO_START_CAPTURE	Send command to start video capture, specifying the frequency that CAMERA_CAPTURE_STATUS messages should be sent while recording.	
MAV_CMD_VIDEO_STOP_CAPTURE	Send command to stop video capture.	
CAMERA_IMAGE_CAPTURED	Information about image captured (returned to GPS every time an image is captured).	
MAV_CMD_VIDEO_START_STREAMING	Send command to start video streaming for the given Stream ID (stream_id.) This is mostly for streaming protocols that *push* a stream. If your camera uses a connection based streaming configuration (RTSP, TCP, etc.), you may ignore it if you don't need it but note that you still must ACK the command, like all MAV_CMD_XXX commands. When using a connection based streaming configuration, the GCS will connect the stream from its side. When a camera offers more than one stream and the user switches from one stream to another, the GCS will send a MAV_CMD_VIDEO_STOP_STREAMING command targeting the current Stream ID followed by a MAV_CMD_VIDEO_START_STREAMING targeting the newly selected Stream ID.	

Message	Description	Status
MAV_CMD_VIDEO_STOP_STREAMING	Send command to stop video streaming for the given Stream ID (stream_id.) This is mostly for streaming protocols that *push* a stream. If your camera uses a connection based streaming configuration (RTSP, TCP, etc.), you may ignore it if you don't need it but note that you still must ACK the command, like all MAV_CMD_XXX commands. When using a connection based streaming configuration, the GCS will disconnect the stream from its side. When a camera offers more than one stream and the user switches from one stream to another, the GCS will send a MAV_CMD_VIDEO_STOP_STREAMING command targeting the current Stream ID followed by a MAV_CMD_VIDEO_START_STREAMING targeting the newly selected Stream ID.	
MAV_CMD_REQUEST_CAMERA_SETTINGS	Send command to request CAMERA_SETTINGS.	deprecated
MAV_CMD_REQUEST_CAMERA_INFORMATION	Send command to request CAMERA_INFORMATION.	deprecated
MAV_CMD_REQUEST_VIDEO_STREAM_INFORMATION	Send command to request VIDEO_STREAM_INFORMATION. This is sent once for each camera when a camera is detected and it has set the CAMERA_CAP_FLAGS_HAS_VIDEO_STREAM flag within the CAMERA_INFORMATION message flags field.	deprecated
MAV_CMD_REQUEST_VIDEO_STREAM_STATUS	Send command to request VIDEO_STREAM_STATUS. This is sent whenever there is a mode change (when MAV_CMD_SET_CAMERA_MODE is sent.) It allows the camera to update the stream configuration when a camera mode change occurs.	deprecated
MAV_CMD_REQUEST_STORAGE_INFORMATION	Send command to request STORAGE_INFORMATION.	deprecated

Message	Description	Status
MAV_CMD_REQUEST_CAMERA_CAPTURE_STATUS	Send command to request CAMERA_CAPTURE_STATUS.	deprecated

Table 110: Enum

Enum	Description
CAMERA_CAP_FLAGS	Camera capability flags (Bitmap). For example: ability to capture images in video mode, support for survey mode etc. Received in CAMERA_INFORMATION.
CAMERA_MODE	Camera mode (image, video, survey etc.). Received in CAMERA_SETTINGS.
VIDEO_STREAM_TYPE	Type of stream - e.g. RTSP, MPEG. Received in VIDEO_STREAM_INFORMATION.
VIDEO_STREAM_STATUS_FLAGS	Bitmap of stream status flags - e.g. zoom, thermal imaging, etc. Received in VIDEO_STREAM_INFORMATION.

Gimbal Protocol v2

Introduction

The gimbal protocol allows MAVLink control over the attitude/orientation of cameras (or other sensors) mounted on the drone. The orientation can be: controlled by the pilot in real time (e.g. using a joystick from a ground station), set as part of a mission, or moved based on camera tracking.

The protocol also defines what status information is published for developers, configurators, as well as users of the drone. It additionally provides ways to assign control to different sources.

The protocol supports a number of hardware setups, and enables gimbals with varying capabilities.

Concepts

Gimbal Manager and Gimbal Device To accommodate gimbals with varying capabilities, and various hardware setups, “a gimbal” is conceptually split into two parts:

- **Gimbal Device:** the actual gimbal device, hardware and software.
- **Gimbal Manager:** software to deconflict gimbal messages and commands from different sources, and to abstract the capabilities of the **Gimbal Device** from gimbal users.

The *Gimbal Manager* and *Gimbal Device* expose respective *message sets* that can be used for: gimbal manager/device discovery, querying capabilities, publishing status, and various types of orientation/attitude control.

The key concept to understand is that a *Gimbal Manager* has a 1:1 relationship with a particular *Gimbal Device*, and is the only party on the MAVLink network that is allowed to directly command that device - it does so using the *Gimbal Device message set*.

MAVLink applications (ground stations, developer APIs like the MAVSDK, etc.), and any other software that wants to control a particular gimbal, must do so via its *Gimbal Manager*, using the *Gimbal Manager message set*.

Note that the gimbal manager is (by default) implemented on the autopilot.

Common Set-ups This section outlines the three most common hardware set-ups.

Simple Gimbal Directly Connected to Autopilot In this (default) set-up the autopilot takes the role of the gimbal manager.

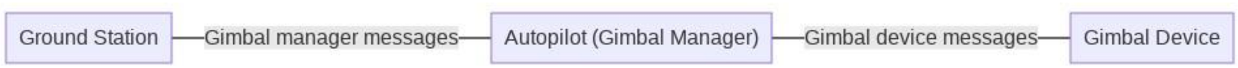

Figure 33: Simple Gimbal Diagram

Standalone Integrated Camera/Gimbal In this set-up the integrated camera/gimbal itself can be the *Gimbal Manager*.

Therefore, the gimbal device interface is internal (no implementation is required).

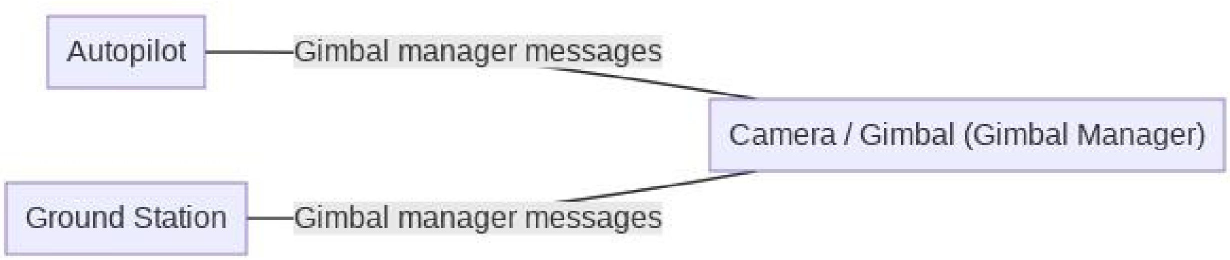

Figure 34: Standalone Integrated Camera/Gimbal Diagram

Onboard Computer with Camera and Gimbal Connected to Autopilot In this set-up the *Gimbal Manager* can be on the onboard computer.

Commands from the GCS (etc.) are sent to the *Gimbal Manager* on the companion computer. Messages from the *Gimbal Manager* to the *Gimbal Device* need to be sent to/routed through the autopilot.

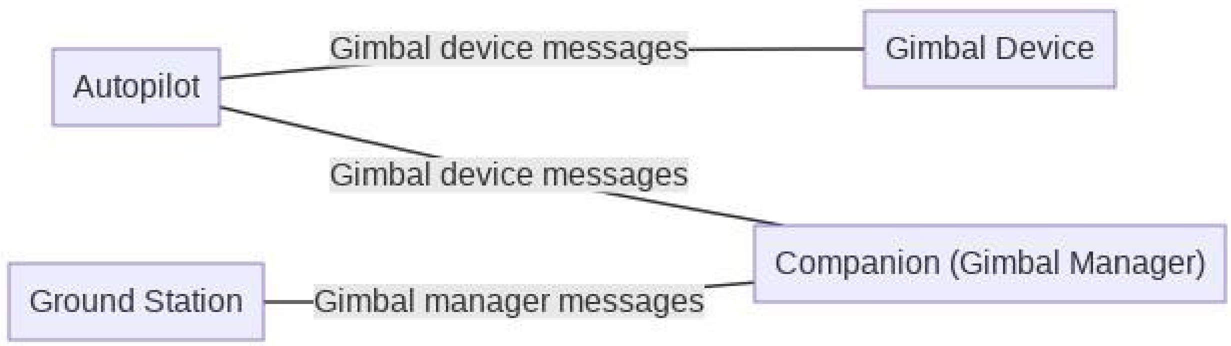

Figure 35: Onboard Computer with Camera and Gimbal Diagram

Multiple Gimbals Multiple gimbals per drone are supported.

Component IDs Multiple component IDs are reserved for gimbal devices: MAV_COMP_ID_GIMBAL, MAV_COMP_ID_GIMBAL2, MAV_COMP_ID_GIMBAL3, MAV_COMP_ID_GIMBAL4, MAV_COMP_ID_GIMBAL5, MAV_COMP_ID_GIMBAL6.

The listed component IDs should be used where possible (other ids may be used as long as the `MAV_TYPE` is correctly set to `MAV_TYPE_GIMBAL`).

Mapping from Gimbal Managers to Gimbal Devices Every *Gimbal Manager* must publish its associated *Gimbal Device* (there is a 1:1 relationship) in its `GIMBAL_MANAGER_INFORMATION` message.

A particular MAVLink component can implement multiple gimbal managers (e.g. an autopilot can implement two gimbal managers in order to control two gimbal devices).

Addressing of Gimbal Devices *Gimbal Manager* commands and messages have a param field to indicate the component ID of the *Gimbal Device* that they intend to control.

A system that wants to control a *particular* gimbal device will send messages to the component that has the manager(s), specifying the particular device to be controlled.

If all gimbal devices should be controlled (on the component that has the gimbal managers), this param/field can be set to 0 (signalling "all").

Implementation and Messages

Messages between Ground Station and Gimbal Manager

Discovery of Gimbal Manager A ground station should initially discover all gimbal managers by sending a broadcast `MAV_CMD_REQUEST_MESSAGE` for `GIMBAL_MANAGER_INFORMATION`. Every gimbal manager should respond with GIMBAL_MANAGER_INFORMATION.

The GIMBAL_MANAGER_INFORMATION contains important information such as gimbal capabilities (`GIMBAL_MANAGER_CAP_FLAGS`), maximum angles and angle rates, as well as the gimbal_component which is the component ID of the *Gimbal Device* controlled by this *Gimbal Manager*.

Gimbal Manager Status A *Gimbal Manager* should send out `GIMBAL_MANAGER_STATUS` at a low regular rate (e.g. 5 Hz) to inform the ground station about its status.

Starting / Configuring Gimbal Control It is possible for multiple components to want to control a gimbal at the same time, e.g.: a ground station, a companion computer, or the autopilot running a mission.

In order to start controlling a gimbal, a component first needs to send the `MAV_CMD_DO_GIMBAL_MANAGER_CONFIGURE` command. This allows setting which MAVLink component (set by system ID and component ID) is in primary control and which one is in secondary control. The gimbal manager is to ignore any gimbal controls which come from MAVLink components that are not explicitly set to "in control". This should prevent conflicts between various inputs as long as all components are fair/co-operative when using the configure command.

To be co-operative entails the following rules:

- Don't send the configure manager configure command continuously but only once to initiate and once to stop control again.
- Check the `GIMBAL_MANAGER_STATUS` about who is in control first and - if possible - warn user about planned action. For example, if the autopilot is in control of the gimbal as part of a mission, the ground station should ask the user first (i.e. via a pop-up) if they really want to take over manual control.
- Don't forget to release control when an action/task is finished and set the sysid/compid to 0.

It is possible to assign control to another component too, not just to itself. For example, a smart shot running on a companion computer can set itself to be in primary control but assign a ground station for secondary control to e.g. nudge during the smart shot.

Note The implementation of how primary and secondary control are combined or mixed is not defined by the protocol but up to the implementation. This allows flexibility for different use cases.

Manual Gimbal Control using MAVLink A ground station can manually control a gimbal by sending `GIMBAL_MANAGER_SET_MANUAL_CONTROL`. This allows controlling the gimbal with either angles, or angular rates, using a normalized unit (-1..1). The gimbal device is responsible for translating the input based on angle, speed, and "smoothness" settings.

This input can additionally be scaled by the gimbal manager depending on its state. For example, if the gimbal manager is on a camera and knows the current zoom level / focal length of the camera, it can scale the angular rate down to support smooth panning and tilting.

Controlling Gimbal Angle and/or Angular Rate using MAVLink A ground station, companion computer, or other MAVLink component can set the gimbal angle and/or angular rates using the messages `GIMBAL_MANAGER_SET_ATTITUDE` or `GIMBAL_MANAGER_SET_TILTPAN`.

Messages between Gimbal Manager and Gimbal Device

Discovery of Gimbal Device The MAVlink node where the *Gimbal Manager* is implemented needs to discover *Gimbal Devices* by sending a broadcast `MAV_CMD_REQUEST_MESSAGE` for `GIMBAL_DEVICE_INFORMATION`. Every gimbal device should respond with `GIMBAL_DEVICE_INFORMATION`.

The MAVLink node should then create as many *Gimbal Manager* instances as *Gimbal Devices* found.

Control of a Gimbal Device To control the angle and/or angular rate of the *Gimbal Device*, use the message `GIMBAL_DEVICE_SET_ATTITUDE`. If the gimbal manager has multiple gimbal control inputs available it should deconflict them as explained below.

Autopilot State for Gimbal Device The autopilot should also send the message `AUTOPILOT_STATE_FOR_GIMBAL_DEVICE` to the gimbal device. This data is required by the *Gimbal Device* attitude estimator (horizon compensation), as well as to anticipate the vehicle's movements (e.g. the feed forward angular velocity in z-axis, so the current yaw intention).

Gimbal Device Broadcast/Status Messages The gimbal device should send out its attitude and status in GIMBAL_DEVICE_ATTITUDE_STATUS at a regular rate, e.g. 10 Hz.

This message is meant as broadcast, so it's set to the GCS, *Gimbal Manager*, and all parties on the network (not just *Gimbal Manager*, like all other messages).

Custom Gimbal Device Settings Custom gimbal settings can be accomplished using the component information microservice which is based on a component information file (this is similar to the camera definition file).

Message/Command/Enum Summary

Gimbal Manager Messages This is the set of messages/enums for communicating with the gimbal manager (by ground station, autopilot, etc.).

Table 111: Messages

Message	Description
GIMBAL_MANAGER_INFORMATION	Information about a high level gimbal manager. This message should be requested by a ground station using MAV_CMD_REQUEST_MESSAGE.
GIMBAL_MANAGER_STATUS	Current status about a high level gimbal manager. This message should be broadcast at a low regular rate (e.g. 5Hz).
GIMBAL_MANAGER_SET_ATTITUDE	High level message to control a gimbal's attitude. This message is to be sent to the gimbal manager (e.g. from a ground station).

Table 112: Commands

Command	Description
MAV_CMD_REQUEST_MESSAGE	Request the target system(s) emit a single instance of a specified message. This is used to request GIMBAL_MANAGER_INFORMATION.
MAV_CMD_DO_GIMBAL_MANAGER_CONFIGURE	Gimbal configuration to set which sysid/compid is in primary and secondary control.

Command	**Description**
GIMBAL_MANAGER_SET_MANUAL_CONTROL	High level message to control a gimbal manually, so without units. The actual angles or angular rates will be produced by the gimbal manager based on settings. This message is to be sent to the gimbal manager (e.g. from a ground station). Angles and rates can be set to NaN according to use case.
MAV_CMD_DO_GIMBAL_MANAGER_TILTPAN	High level setpoint to be sent to a gimbal manager to set a gimbal attitude. Note: a gimbal is never to react to this command but only the gimbal manager.
MAV_CMD_DO_SET_ROI_LOCATION	Sets the region of interest (ROI) to a location. This can then be used by the vehicle's control system to control the vehicle attitude and the attitude of various sensors such as cameras. This command can be sent to a gimbal manager but not to a gimbal device. A gimbal is not to react to this message.
MAV_CMD_DO_SET_ROI_WPNEXT_OFFSET	Sets the region of interest (ROI) to be toward the next waypoint, with optional pitch/roll/yaw offset. This can then be used by the vehicle's control system to control the vehicle attitude and the attitude of various sensors such as cameras. This command can be sent to a gimbal manager but not to a gimbal device. A gimbal device is not to react to this message.
MAV_CMD_DO_SET_ROI_SYSID	Mount tracks system with specified system ID. Determination of target vehicle position may be done with GLOBAL_POSITION_INT or any other means. This command can be sent to a gimbal manager but not to a gimbal device. A gimbal device is not to react to this message.
MAV_CMD_DO_SET_ROI_NONE	Cancels any previous ROI command returning the vehicle/sensors to default flight characteristics. This can then be used by the vehicle's control system to control the vehicle attitude and the attitude of various sensors such as cameras. This command can be sent to a gimbal manager but not to a gimbal device. A gimbal device is not to react to this message. After this command the gimbal manager should go back to manual input if available, and otherwise assume a neutral position.

Command	Description
MAV_CMD_DO_GIMBAL_MANAGER_TRACK_POINT	If the gimbal manager supports visual tracking (GIMBAL_MANAGER_CAP_FLAGS_HAS_TRACKING_POINT is set), this command initiates the tracking. Such a tracking gimbal manager would usually be an integrated camera/gimbal, or alternatively a companion computer connected to a camera.
MAV_CMD_DO_GIMBAL_MANAGER_TRACK_RECTANGLE	If the gimbal supports visual tracking (GIMBAL_MANAGER_CAP_FLAGS_HAS_TRACKING_RECTANGLE is set), this command initiates the tracking. Such a tracking gimbal manager would usually be an integrated camera/gimbal, or alternatively a companion computer connected to a camera.

Table 113: Enum

Enum	Description
GIMBAL_MANAGER_FLAGS	Flags for high level gimbal manager operation. The first 16 bytes are identical to the GIMBAL_DEVICE_FLAGS. Used in MAV_CMD_DO_GIMBAL_MANAGER_TILTPAN, GIMBAL_MANAGER_STATUS, GIMBAL_MANAGER_SET_ATTITUDE.
GIMBAL_MANAGER_CAP_FLAGS	Gimbal manager high level capability flags (bitmap). The first 16 bits are identical to the GIMBAL_DEVICE_CAP_FLAGS which are identical with GIMBAL_DEVICE_FLAGS. However, the gimbal manager does not need to copy the flags from the gimbal but can also enhance the capabilities and thus add flags. Used in GIMBAL_MANAGER_INFORMATION

Gimbal Device Messages This is the set of messages/enums for communication between the gimbal manager and the gimbal device.

Table 114: Messages

Message	Description
GIMBAL_DEVICE_INFORMATION	Information about a low level gimbal. This message should be requested by the gimbal manager or a ground station using MAV_CMD_REQUEST_MESSAGE.

Message	Description
GIMBAL_DEVICE_SET_ATTITUDE	Low level message to control a gimbal device's attitude. This message is to be sent from the gimbal manager to the gimbal device component. Angles and rates can be set to NaN according to use case.
GIMBAL_DEVICE_ATTITUDE_STATUS	Message reporting the status of a gimbal device. This message should be broadcasted by a gimbal device component.

Table 115: Commands

Command	Description
MAV_CMD_REQUEST_MESSAGE	Request the target system(s) emit a single instance of a specified message. This is used to request GIMBAL_DEVICE_INFORMATION.

Table 116: Enum

Enum	Description
GIMBAL_DEVICE_CAP_FLAGS	Gimbal device (low level) capability flags (bitmap). Used in GIMBAL_DEVICE_INFORMATION.
GIMBAL_DEVICE_FLAGS	Flags for gimbal device (lower level) operation. Used in GIMBAL_DEVICE_ATTITUDE_STATUS and GIMBAL_DEVICE_SET_ATTITUDE.
GIMBAL_DEVICE_ERROR_FLAGS	Gimbal device (low level) error flags (bitmap, 0 means no error). Used in GIMBAL_DEVICE_ATTITUDE_STATUS.

Sequences

Depicted below are message sequences for some common scenarios.

Discovery This shows a possible sequence on startup. Note that the gimbal manager could already discover the gimbal device before the ground station asks for the information.

Normal Manual Control During the normal manual control, all messages are streamed at a regular rate. Note that GIMBAL_DEVICE_ATTITUDE_STATUS is broadcast to anyone, so to the gimbal manager and also the ground station.

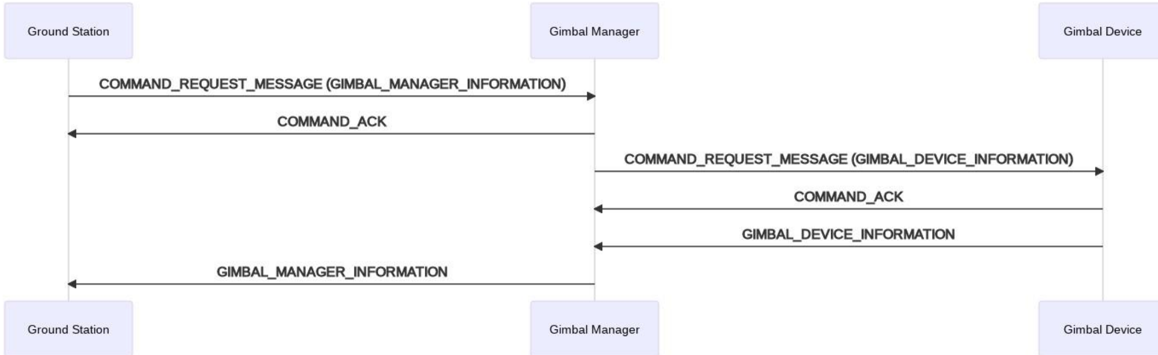

Figure 36: Discovery Diagram

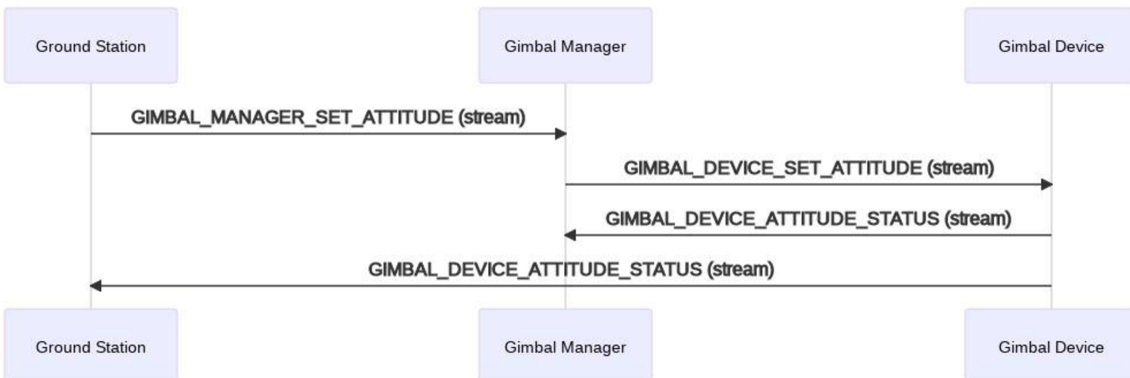

Figure 37: Normal Manual Control Diagram

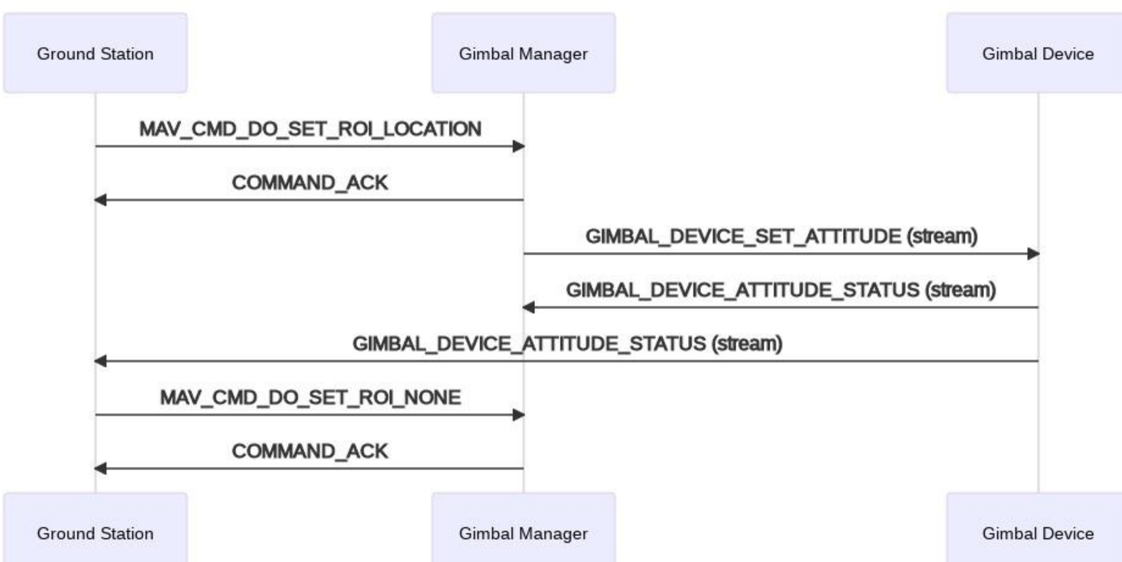

Figure 38: ROI Ground Station Initiated Diagram

ROI Initiated from Ground Station ROI can be started using a command and should also be stopped again with a command. The ROI command is translated to a gimbal attitude in the gimbal manager.

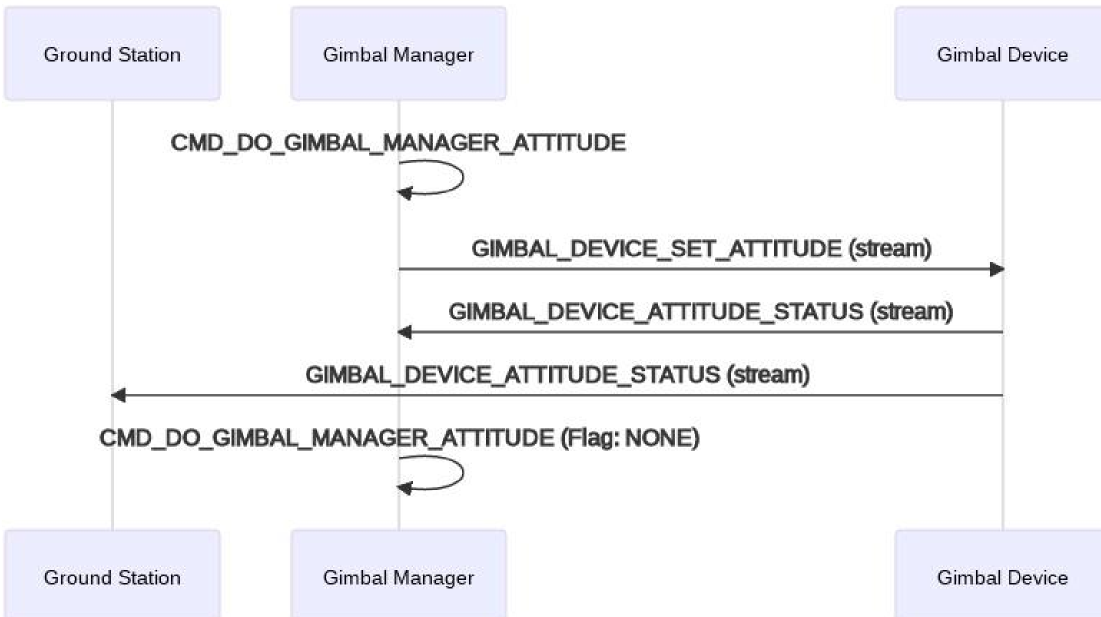

Figure 39: Attitude Set during Mission Diagram

Attitude Set During Mission In this case the gimbal manager is implemented by the autopilot which "sends" the attitude command (for instance for a survey).

How to Implement the Gimbal Device Interface

Below is a short summary of all messages that a gimbal device should implement.

A Gimbal Device can be tested by connecting it to an autopilot with a Gimbal Manager. To avoid having to do a full setup including autopilot, a direct test using MAVSDK is available.

Messages to Send The messages listed should be sent to all connections (sent to everyone).

HEARTBEAT Heartbeats should always be sent (usually at 1 Hz).

- `sysid`: *Ignored, can be any number*
- `compid`: `MAV_COMP_ID_GIMBAL`
- `type`: `MAV_TYPE_GIMBAL`

- `autopilot`: `MAV_AUTOPILOT_INVALID`
- `base_mode`: 0
- `custom_mode`: 0
- `system_status`: `MAV_STATE_UNINIT`

GIMBAL_DEVICE_ATTITUDE_STATUS

The gimbal device should send out its attitude status at a regular rate, e.g. 10 Hz. The fields target_system and target_component can be set to 0 (broadcast) by default.

GIMBAL_DEVICE_INFORMATION

The static information about the gimbal device needs to be sent out when requested using `MAV_CMD_REQUEST_MESSAGE`.

Messages to Listen To/Handle

GIMBAL_DEVICE_SET_ATTITUDE

This is the actual attitude setpoint that the gimbal device should follow. Note that the frame of the quaternion setpoint depends on the `GIMBAL_DEVICE_FLAGS`.

AUTOPILOT_STATE_FOR_GIMBAL_DEVICE

The gimbal device should be able to get all the information from the autopilot that it requires in this one message. If something is missing that should be streamed at a high rate, it should be added to this message.

If this message is not sent by default by the autopilot, or the rate is not ok, the command `MAV_CMD_SET_MESSAGE_INTERVAL` can be used to request it at a certain rate.

COMMAND_LONG

The gimbal device needs to check for commands. See below which commands should get answered.

Commands to Answer

`MAV_CMD_REQUEST_MESSAGE`

The gimbal device should send out messages when they get requested, e.g. `GIMBAL_DEVICE_INFORMATION`.

MAV_CMD_SET_MESSAGE_INTERVAL

The gimbal device should stream messages at the rate requested.

Terrain Protocol

The Terrain Protocol provides a mechanism for a vehicle to get terrain information (tiles) from a ground station, and for a ground station to check the autopilot terrain cache for a tile at a particular location. Support for this protocol is indicated by `AUTOPILOT_VERSION.capabilities` by the `MAV_PROTOCOL_CAPABILITY_TERRAIN` flag.

A vehicle that supports this capability must also support terrain following in missions using the data. Note however that a vehicle may also support terrain handling in missions using a distance sensor, even if this protocol is not supported and capability flag is not set.

Message/Enum Summary

Table 117: Messages

Message	Description
`TERRAIN_REQUEST`	Request from drone (to GCS) for terrain data. The message specifies a mask indicating what tiles are required, and the GCS responds by sending `TERRAIN_DATA` for each tile. The drone will also stream `TERRAIN_REPORT` messages to provide progress updates while it is waiting for data.
`TERRAIN_DATA`	Terrain data from GCS for a particular tile (sent in response to a `TERRAIN_REQUEST`. The lat/lon and grid_spacing must be the same as the lat/lon from a `TERRAIN_REQUEST`.
`TERRAIN_REPORT`	The drone will stream `TERRAIN_REPORT` to indicate progress of terrain download, and in response to a `TERRAIN_CHECK`.
`TERRAIN_CHECK`	Request that the vehicle report terrain height at the given location (expected response is a `TERRAIN_REPORT`). Used by GCS to check if a vehicle has all terrain data needed for a mission.

Autopilot Terrain Map Request

The sequence for a drone to update its terrain altitude information is entirely driven by the drone, and is shown below.

In summary, the sequence is:

1. Drone sends `TERRAIN_REQUEST` to the GCS to request a set of tiles (specified in a mask).

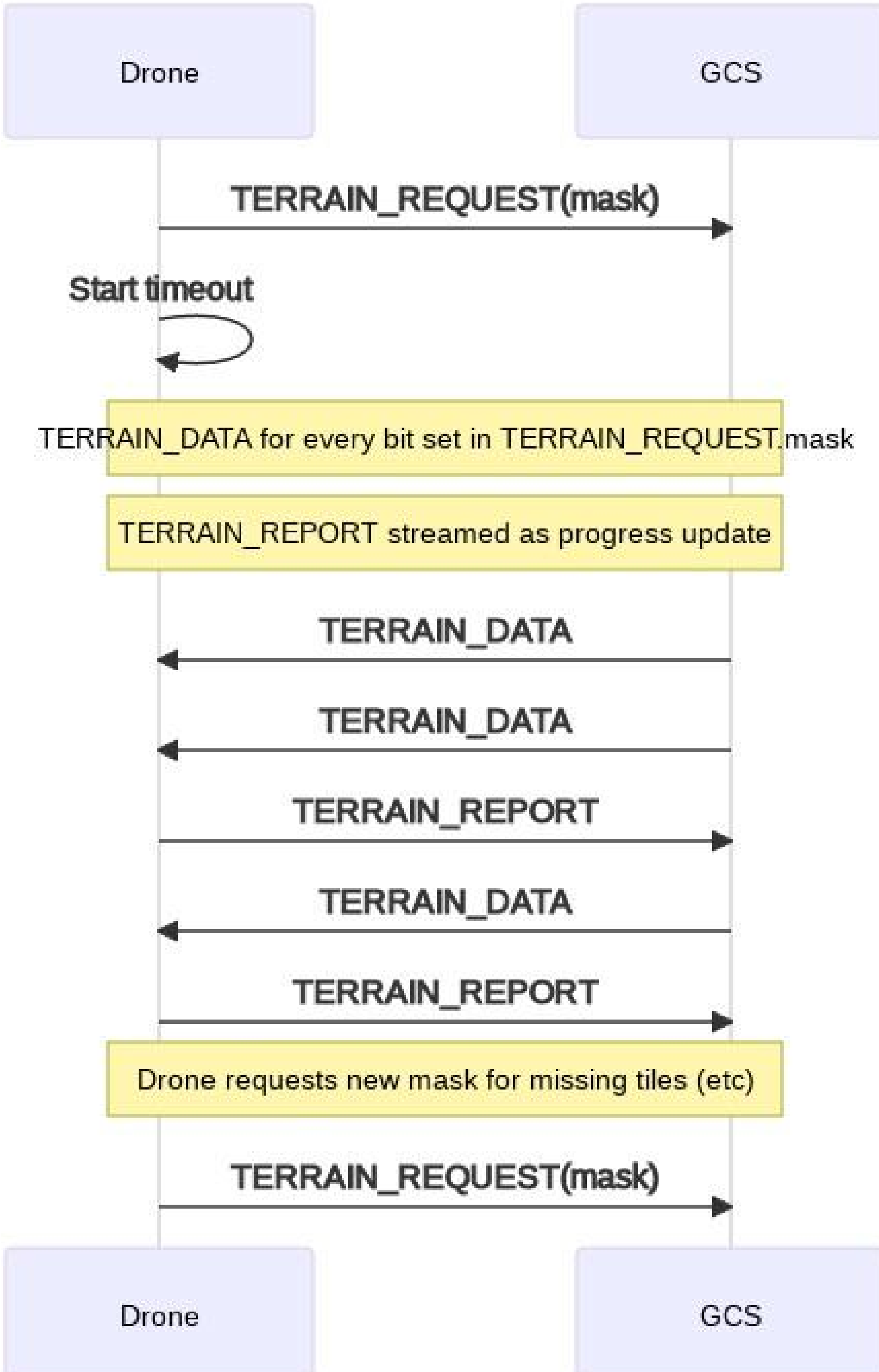

Figure 40: Autopilot Terrain Map Diagram

2. The GCS responds by sending a TERRAIN_DATA message for each tile set in the mask

3. The drone also streams TERRAIN_REPORT) messages back to the GCS indicating the current state of the download

    - TERRAIN_REPORT.pending and TERRAIN_REPORT.loaded indicate how many tiles are expected and have arrived, respectively.
    - TERRAIN_REPORT.lat, .lon, .terrain_height, while duplicated in other messages, are useful for debugging (a GCS can check its own internal terrain data against the information).

4. The drone must maintain its own record of what tiles have arrived/not arrived, and can re-request any that are missing using a further TERRAIN_REQUEST (with mask indicating just the missing tiles).

The diagram below shows the way the data is encoded within the TERRAIN_REQUEST and TERRAIN_DATA.

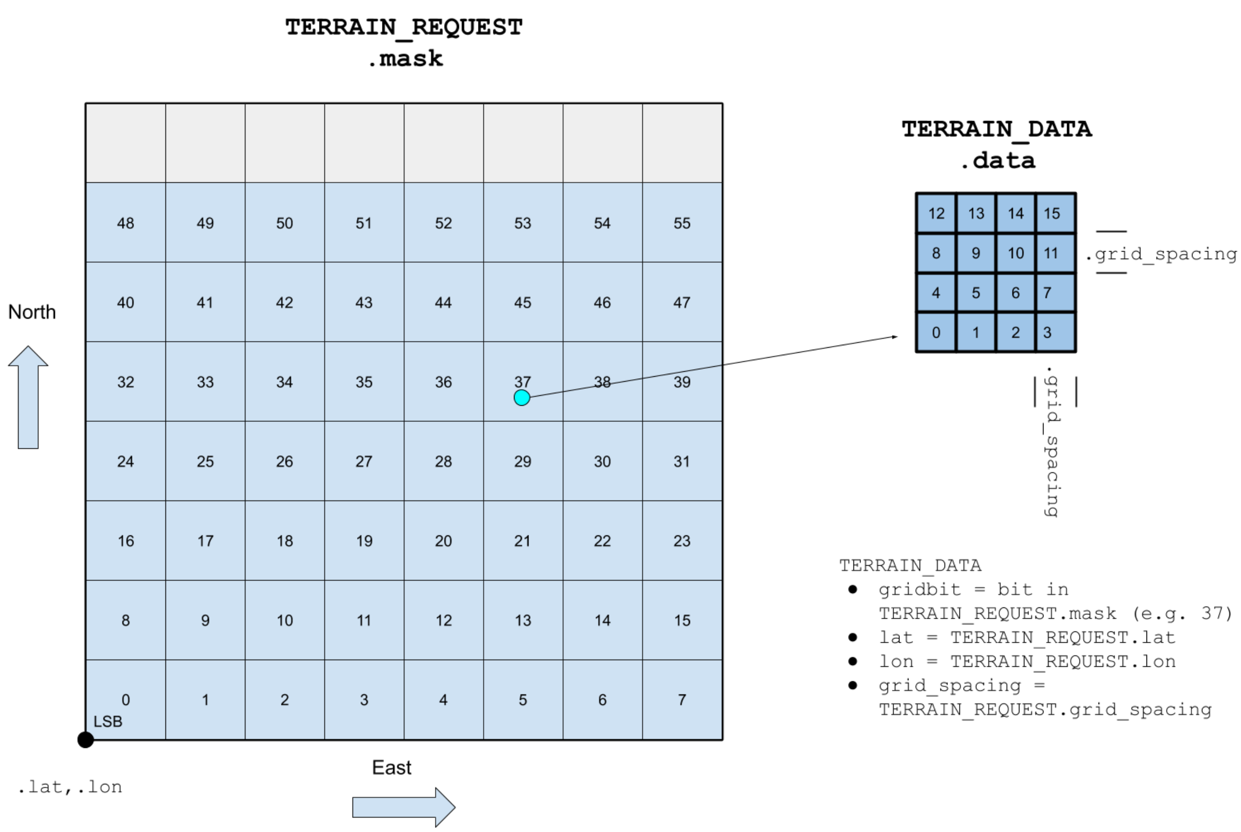

Figure 41: Terrain Request Data Table

TERRAIN_REQUEST.mask is a 64-bit value that represents a row major 8x7 array of (4x4) tiles. The lat, lon fields indicate the position of the South-West corner of the first grid position (tile). The tiles are allocated sequentially in rows (West to East) starting from the lowest significant bit of mask, and then in columns (South to North).

Each tile represents a 4x4 grid of altitude information. The spacing between the rows/columns in the tile is indicated by grid_spacing (the same value must be used in both request and data messages).

GCS Terrain Tile Check

The sequence for a GCS to check the autopilot terrain cache at a particular location is shown below.

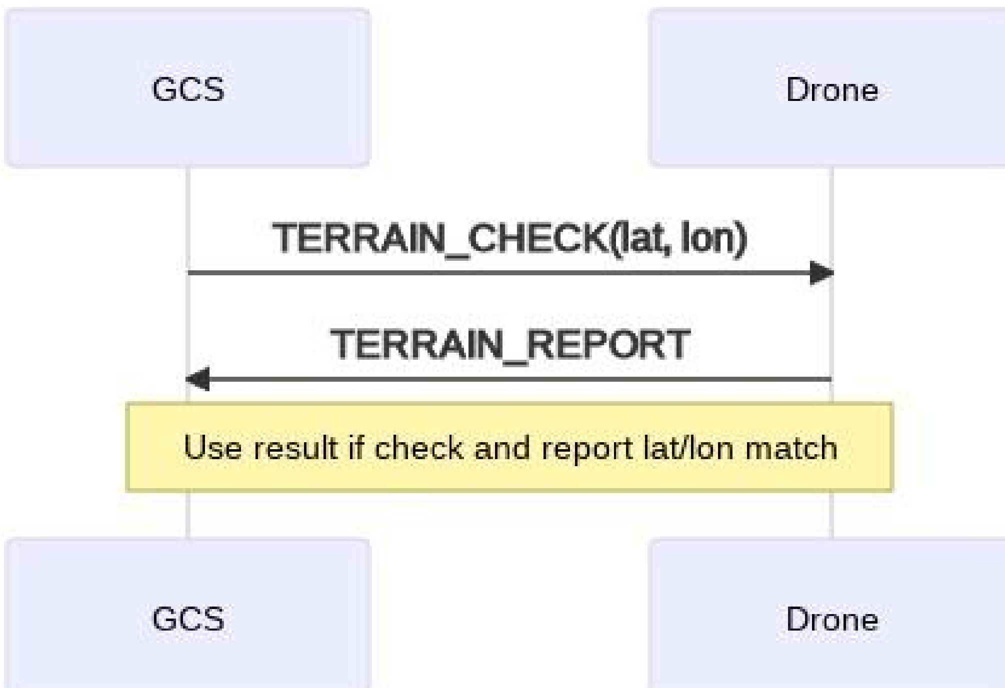

Figure 42: GCS Terrain Tile Check Diagram

In summary, the sequence is:

1. GCS sends `TERRAIN_CHECK` to the vehicle to request terrain information at a specific location.
2. The drone responds with a `TERRAIN_REPORT` message containing the tile information it has for that location. If it does not have tile information for the specified location, then the request is ignored.
3. GCS can verify that the terrain report matches a terrain check by comparing the latitude/longitude fields for both messages.

The protocol does not define how the ground station handles the case if no `TERRAIN_REPORT` is received (although it might resend the request after a timeout).

Exploration Protocol

The Exploration Protocol is applicable to vehicles that have the capability to perform autonomous exploration tasks in indoor and outdoor environments, either in mapped or unmapped areas. While the decision making process that takes the vehicles to specific areas to explore can be conditioned or configured by user input, as of now, this microservice controls where the exploration starts and stops, and provides status over the exploration behavior and allows to setup different control points, like ingress and egress portals or return point post-exploration. A vehicle executing an exploration should be set to `Exploration` flight mode.

When the system (vehicle plus GCS) have computer vision and image processing capabilities that allow POI detection and tracking, and when using the POI_REPORT message, the POI_REPORT uid field can be used to identify the ingress or egress portals. Note though that while the POI_REPORT uid field is a 64-bit integer, the portal IDs are limited to a 32-bit field. It is though advised to just use the range 0x00000000 to 0xFFFFFFFF to identify any portal on the POI_REPORT message, and so that ID can also be used as the same identifier for the portal in the commands and messages used in this microservice.

Exploration task definition and configuration

An exploration task is identified by a task ID. A new task always gets assigned a new task ID. When stopping and resuming a task, multiple tasks can be queued, and MAV_CMD_DO_EXPLORATION allows to define if one wants to start a new task, resume the current task, or resume a queued task

The possible configurations for a task are the time limit, in seconds, for executing a task, the ingress and egress (when applicable) portals and the exploration boundaries. The boundaries can be set using MAV_CMD_SET_EXPLORATION_BOUNDARIES and are defined by a cardinal-direction-aligned rectangular solid (cuboid) defined by 3 points: p1, p2 and p3. In order to set the exploration task world boundaries. p1 and p2 identify the vertices of a rectangle, which define a 2D localization of the exploration area, parallel to the ground plane, while p3, matching the same coordinates of p1, provides the height limitation of the 3D volume to explore. In short, the values to be set are x1, y1, x2, y2, z and the height (h) of the cuboid / 3D volume, where the point coordinates can be represented by p1(x1, y1, z+h), p2(x2, y2, z+h) and p3(x1, y1, z).

Message/Enum Summary

Message	Description
EXPLORATION_STATUS	Provides status over an exploration task. The message should, by default, be streamed at 1Hz.
EXPLORATION_INFO	Provides configuration information about an exploration task.
EXPLORATION_RETURN_POSITION	Provides the return-from-exploration position when an exploration is completed or canceled.

Enumeration	Description
MAV_EXPLORATION_STATUS	Flag bitmap to provide status of the exploration task metrics.

Commands Summary

Command	Description
MAV_CMD_GO_THROUGH_PORTAL	Go through a portal. In an indoor exploration context, a portal represents an structural identifiable entry or pass point to start, stop or continue an exploration.
MAV_CMD_DO_EXPLORATION	Start or continue the exploration task.
MAV_CMD_STOP_EXPLORATION	Stop the current exploration task. The behavior after stopping is defined by a parameter.
MAV_CMD_SET_EXPLORATION_RETURN_POS	Sets the return position after stopping or finishing an exploration task. Used when the vehicle autonomy engine does not set this position or used to overwrite it.
MAV_CMD_DO_EXPLORATION_RETURN	Return to defined position after exploration.
MAV_CMD_SET_INGRESS_PORTAL	Sets a specific portal to be an ingress portal.
MAV_CMD_SET_EGRESS_PORTAL	Sets a specific portal to be an egress portal.
MAV_CMD_SET_EXPLORATION_BOUNDARIES	Set exploration task world boundaries.

Message definitions

EXPLORATION_STATUS Provides status over an exploration task. The message should, by default, be streamed at 1Hz.

Field name	Type	Units	Values	Description
time_usec	uint64_t	us	-	Timestamp (UNIX Epoch time or time since system boot). The receiving end can infer timestamp format (since 1.1.1970 or since system boot) by checking for the magnitude of the number.
time_to_timeout	uint64_t	us	-	Remaining time for the vehicle to execute the exploration task, after which another predefined behavior is triggered. UINT64_MAX when unknown or not applicable.
exploration_id	uint8_t	-	-	ID of the exploration task. 255 if not applicable or unknown.
progress	uint16_t	-	-	Progress measurement of the exploration task. Specific meaning may vary by implementation, but in general, increasing values mean more has been explored. UINT16_MAX when unknown or not applicable.

Field name	Type	Units	Values	Description
denominator	uint16_t	-	-	Measurement of the known size of the exploration task. Specific meaning may vary, but when progress == denominator, this should imply that exploration is complete. This value may increase as more need to explore is discovered, or may be fixed (100 recommended) if the end state is known (e.g., exploration in a known mapped environment). 0 when no meaningful size can be communicated.
flags	uint8_t	-	MAV_EXPLORATION_STATUS_FLAGS	Bitmap of the exploration task status flags.
level	int8_t	-	-	In an indoor exploration task, it indicates the floor/level of the structure that is currently being explored. The level where the vehicle started the exploration is considered the level 0. INT8_MAX when unknown, not capable to provide or not applicable.

EXPLORATION_INFO Provides configuration information about an exploration task. The message can be requested using the MAV_CMD_REQUEST_MESSAGE command, where param 3 should be used to set which exploration task to get. To determine all coordinates of the cuboid, consider: p1_x equals p3_x, p1_y equals p3_y, p1_z equals p2_z, p1_lat equals p3_lat, p1_lon equals p3_lon and p1_alt equals p2_alt.

Field name	Type	Units	Values	Description
exploration_id	uint8_t	-	-	ID of the exploration task. 255 to get the information of the current running exploration task.
time_limit	uint32_t	s	-	Time limit to execute the exploration. Reaching this time triggers the behavior defined in the behaviour_after_stopping field. Set to 0 when the exploration time is not limited.
behaviour_after_stopping	uint8_t	-	-	The behavior after stopping the exploration task. 0: Hold, 1: Land, 2: Return (to exploration return position).
boundaries_p1_lat	int32_t	degE7	-	Exploration cuboid boundaries point 1 Latitude (WGS84). INT32_MAX if unknown. p1_lat == p3_lat.
boundaries_p1_lon	int32_t	degE7	-	Exploration cuboid boundaries point 1 Longitude (WGS84). INT32_MAX if unknown. p1_lon == p3_lon.
boundaries_p1_alt	float	m	-	Exploration cuboid boundaries point 1 Altitude (MSL). NaN if unknown. p1_alt == p2_alt.
boundaries_p1_x	float	m	-	Exploration cuboid boundaries point 1 local NED X coordinate. NaN if unknown. p1_x == p3_x.
boundaries_p1_y	float	m	-	Exploration cuboid boundaries point 1 local NED Y coordinate. NaN if unknown. p1_y == p3_y.
boundaries_p1_z	float	m	-	Exploration cuboid boundaries point 1 local NED Z coordinate. NaN if unknown. p1_z == p2_z.

Field name	Type	Units	Values	Description
boundaries_p2_lat	int32_t	degE7	-	Exploration cuboid boundaries point 2 Latitude (WGS84). INT32_MAX if unknown.
boundaries_p2_lon	int32_t	degE7	-	Exploration cuboid boundaries point 2 Longitude (WGS84). INT32_MAX if unknown.
boundaries_p2_x	float	m	-	Exploration cuboid boundaries point 2 local NED X coordinate. NaN if unknown.
boundaries_p2_y	float	m	-	Exploration cuboid boundaries point 2 local NED Y coordinate. NaN if unknown.
boundaries_p3_alt	float	m	-	Exploration cuboid boundaries point 3 Altitude (MSL). NaN if unknown.
boundaries_p3_z	float	m	-	Exploration cuboid boundaries point 3 local NED Z coordinate. NaN if unknown.
ingress_portal_id	uint32_t	-	-	Currently defined ingress portal ID. This portal ID can either be set by the system autonomy engine or by an offboard controller or operator using MAV_CMD_SET_INGRESS_PORTAL. When unknown, not applicable or not deterministic, set to UINT32_MAX.
ingress_portal_lat	int32_t	degE7	-	Currently defined ingress portal Latitude (WGS84). INT32_MAX if unknown, not applicable or when ingress_portal_id is set and different from UINT32_MAX.

Field name	Type	Units	Values	Description
ingress_portal_lon	int32_t	degE7	-	Currently defined ingress portal Longitude (WGS84). INT32_MAX if unknown, not applicable or when ingress_portal_id is set and different from UINT32_MAX.
ingress_portal_alt	float	m	-	Currently defined ingress portal Altitude (MSL). NaN if unknown, not applicable or when ingress_portal_id is set and different from UINT32_MAX.
egress_portal_id	uint32_t	-	-	Currently defined egress portal ID. This portal ID can either be set by the system autonomy engine or by an offboard controller or operator using MAV_CMD_SET_EGRESS_PORTAL. When unknown, not applicable or not deterministic, set to UINT32_MAX.
egress_portal_lat	int32_t	degE7	-	Currently defined egress portal Latitude (WGS84). INT32_MAX if unknown, not applicable or when egress_portal_id is set and different from UINT32_MAX.
egress_portal_lon	int32_t	degE7	-	Currently defined egress portal Longitude (WGS84). INT32_MAX if unknown, not applicable or when egress_portal_id is set and different from UINT32_MAX.

Field name	Type	Units	Values	Description
egress_portal_alt	float	m	-	Currently defined egress portal Altitude (MSL). NaN if unknown, not applicable or when egress_portal_id is set and different from UINT32_MAX.

EXPLORATION_RETURN_POSITION Provides the return-from-exploration position when an exploration is completed (e.g. volume set by the exploration boundaries does not have new open areas for the vehicle to explore) or canceled (e.g. the operator stops the exploration task and requests the vehicle to leave the defined exploration area). Can either be set by the vehicle's onboard autonomy engine or set by the user MAV_CMD_SET_EXPLORATION_RETURN_POS. A MAV_CMD_DO_EXPLORATION_RETURN can be used to send the vehicle to the position defined by this message. This message can be requested by sending the MAV_CMD_REQUEST_MESSAGE command.

Field name	Type	Units	Values	Description
time_usec	uint64_t	us	-	Timestamp (UNIX Epoch time or time since system boot). The receiving end can infer timestamp format (since 1.1.1970 or since system boot) by checking for the magnitude of the number.
latitude	int32_t	degE7		Latitude (WGS84). INT32_MAX when unknown.
longitude	int32_t	degE7		Longitude (WGS84). INT32_MAX when unknown.
altitude	int32_t	mm	-	Altitude (MSL). Positive for up. Note that virtually all GPS modules provide both WGS84 and MSL. INT32_MAX when unknown.
relative_alt	int32_t	mm	-	Altitude above ground. INT32_MAX when unknown.
geoid_alt	int32_t	mm	-	Altitude relative to WGS84 geoid. INT32_MAX when unknown.

Field name	Type	Units	Values	Description
x	float	m		Local X position of this position in the local coordinate NED frame. NaN when unknown.
y	float	m		Local Y position of this position in the local coordinate NED frame. NaN when unknown.
z	float	m		Local Z position of this position in the local coordinate NED frame. NaN when unknown.
yaw	float	-		World to surface heading transformation of the return-from-exploration position. Used to indicate the heading with respect to the ground. NaN when unknown.

Command definitions

MAV_CMD_GO_THROUGH_PORTAL Go through a portal. In an indoor exploration context, a portal represents an structural identifiable entry or pass point to start, stop or continue an exploration. COMMAND_INT should be used so to set the MAV_FRAME and consequently, frame and coordinates of the portal position (MAV_FRAME_GLOBAL means global coordinates are being used, while MAV_FRAME_LOCAL_NED means local NED coordinates are being used).

Param (:Label)	Description	Values	Units
1: Portal ID	ID of the portal. If the ID is unknown or not deterministic, set it to UINT32_MAX and use params 5, 6 and 7 to define the local or global position of the portal.	min: 0 max: 4294967295 increment: 1	
2	Reserved		
3	Reserved		
4	Reserved		

Param (:Label)	Description	Values	Units
5: Position X or Latitude	X or Latitude (WGS84) coordinate of the portal position. NaN or INT32_MAX if unknown (or if using param 1 to set the portal).		degE7 or mE4
6: Position Y or Longitude	Y or Longitude (WGS84) coordinate of the portal position. NaN or INT32_MAX if unknown (or if using param 1 to set the portal).		degE7 or mE4
7: Position Z or Altitude (MSL)	Z or Altitude (MSL) coordinate of the portal position. NaN when unknown or when using param 1 to set the portal.		m

MAV_CMD_DO_EXPLORATION Start or continue the exploration task. If starting a new exploration, but requiring to get through a portal first, params 4 or 5 to 7 should be set and different from the specified ignore values. Continuing an exploration can be done without necessarily setting the ingress portal, unless the exploration requires the definition of more than the initial ingress portal. If the passed exploration ID does not exist or is not listed as a valid exploration ID in the vehicle autonomy engine, then this command should be rejected. COMMAND_INT should be used so to set the MAV_FRAME and consequently, frame and coordinates of the portal position (MAV_FRAME_GLOBAL means global coordinates are being used, while MAV_FRAME_LOCAL_NED means local NED coordinates are being used).

Param (:Label)	Description	Values	Units
1: Exploration task ID	Sets the ID of the exploration task to start. If the ID already exists, it resumes that (queued/listed) exploration task. Set to UINT8_MAX if one wants to resume the last exploration task.	min: 0 max: 255 increment: 1	
2: Time limit	Time limit to execute the exploration. Reaching this time triggers the behavior defined in param 4. Set to 0 when there is no time limit.	min: 0	s

Param (:Label)	Description	Values	Units
3: Behavior after stopping	The behavior after stopping the exploration task. 0: Hold, 1: Land, 2: Return to position.	min: 0 max: 2 increment: 1	
4: Ingress Portal ID	ID of the ingress portal. If the ID is unknown, not deterministic, or when already in the area to explore after going through a portal, set it to UINT32_MAX and use params 5, 6 and 7 to define the local or global position of the portal. Note that an egress portal is not defined by this command but can be consequently set either by the vehicle autonomy engine or by an operator/external controller using MAV_CMD_SET_EGRESS_PORTAL.	min: 0 max: 4294967295 increment: 1	
5: Position X or Latitude	X or Latitude (WGS84) coordinate of the portal position. NaN or INT32_MAX if unknown (or if using param 4 to set the portal).		degE7 or mE4
6: Position Y or Longitude	Y or Longitude (WGS84) coordinate of the portal position. NaN or INT32_MAX if unknown (or if using param 4 to set the portal).		degE7 or mE4
7: Position Z or Altitude (MSL)	Z or Altitude (MSL) coordinate of the portal position. NaN when unknown or when using param 4 to set the portal ID.		m

MAV_CMD_STOP_EXPLORATION Stop an exploration task. The behavior after stopping is defined by a parameter. Return to position should take the vehicle to the defined exit portal first, and then to the return post-exploration point (accessible through MAV_CMD_SET_EXPLORATION_RETURN_POS . If the passed exploration ID does not exist or is not listed as a valid exploration ID in the vehicle autonomy engine, then this command should be rejected.

Param (:Label)	Description	Values	Units
1: Exploration task ID	Sets the ID of the (current or queued) exploration task to stop. Set to UINT32_MAX if one wants to stop the current exploration task.	min: 0 max: UINT32_MAX increment: 1	
2: Behavior after stopping	The behavior after stopping the exploration task. 0: Hold, 1: Land, 2: Return to position.	min: 0 max: 2 increment: 1	
3	Reserved		
4	Reserved		
5	Reserved		
6	Reserved		
7	Reserved		

MAV_CMD_SET_EXPLORATION_RETURN_POS Sets the return position after stopping or finishing an exploration task. Used when the vehicle autonomy engine does not set this position or to overwrite it. COMMAND_INT should be used so to set the MAV_FRAME and consequently, frame and coordinates of the position (MAV_FRAME_GLOBAL means global coordinates are being used, while MAV_FRAME_LOCAL_NED means local NED coordinates are being used).

Param (:Label)	Description	Values	Units
1	Reserved		
2	Reserved		
3	Reserved		
4: Yaw or Heading	Yaw or heading of the return position.		rad
5: Position X or Latitude	X or Latitude (WGS84) coordinate of the return position.		degE7 or mE4
6: Position Y or Longitude	Y or Longitude (WGS84) coordinate of the return position.		degE7 or mE4
7: Position Z or Altitude (MSL)	Z or Altitude (MSL) coordinate of the return position.		m

MAV_CMD_DO_EXPLORATION_RETURN Return to a system-defined position after exploration.

Param (:Label)	Description	Values	Units
1	Reserved		
2	Reserved		
3	Reserved		
4	Reserved		
5	Reserved		
6	Reserved		
7	Reserved		

MAV_CMD_SET_INGRESS_PORTAL Sets a specific portal to be an entry portal. Defines also the approach vector for a vehicle to approach and go through the portal. COMMAND_INT should be used so to set the MAV_FRAME and consequently, frame and coordinates of the approach vector (MAV_FRAME_GLOBAL means global coordinates are being used, while MAV_FRAME_LOCAL_NED means local NED coordinates are being used).

Param (:Label)	Description	Values	Units
1: Portal ID	ID of the portal to be set as an ingress point.	min: 0 max: 4294967295 increment: 1	
2: Approach vector initial point X or Latitude	X (mE4) or Latitude (degE7) (WGS84) coordinate of the initial point of the approach vector to the portal. NaN or INT32_MAX if unknown.		degE7 or mE4
3: Approach vector initial point Y or Longitude	Y (mE4) or Longitude (degE7) (WGS84) coordinate of the initial point of the approach vector to the portal. NaN or INT32_MAX if unknown.		degE7 or mE4
4: Approach vector initial point Z or Altitude (MSL)	Z or Altitude (MSL) coordinate of the initial point of the approach vector to the portal. NaN if unknown.		m
5: Approach vector final point X or Latitude	X (mE4) or Latitude (degE7) (WGS84) coordinate of the final point of the approach vector to the portal. NaN or INT32_MAX if unknown.		degE7 or mE4

Param (:Label)	Description	Values	Units
6: Approach vector final point Y or Longitude	Y (mE4) or Longitude (degE7) (WGS84) coordinate of the final point of the approach vector to the portal. NaN or INT32_MAX if unknown.		degE7 or mE4
7: Approach vector final point Z or Altitude (MSL)	Z or Altitude (MSL) coordinate of the final point of the approach vector to the portal. NaN if unknown.		m

MAV_CMD_SET_EGRESS_PORTAL Sets a specific portal to be an exit portal. Defines also the approach vector for a vehicle to approach and go through the portal. COMMAND_INT should be used so to set the MAV_FRAME and consequently, frame and coordinates of the approach vector (MAV_FRAME_GLOBAL means global coordinates are being used, while MAV_FRAME_LOCAL_NED means local NED coordinates are being used).

Param (:Label)	Description	Values	Units
1: Portal ID	ID of the portal to be set as an egress point.	min: 0 max: 4294967295 increment: 1	
2: Approach vector initial point X or Latitude	X (mE4) or Latitude (degE7) (WGS84) coordinate of the initial point of the approach vector to the portal. NaN if unknown. NaN or INT32_MAX if unknown.		degE7 or mE4
3: Approach vector initial point Y or Longitude	Y (mE4) or Longitude (degE7) (WGS84) coordinate of the initial point of the approach vector to the portal. NaN if unknown. NaN or INT32_MAX if unknown.		degE7 or mE4
4: Approach vector initial point Z or Altitude (MSL)	Z or Altitude (MSL) coordinate of the initial point of the approach vector to the portal. NaN if unknown. NaN if unknown.		m

Param (:Label)	Description	Values	Units
5: Approach vector final point X or Latitude	X (mE4) or Latitude (degE7) coordinate of the final point of the approach vector to the portal. NaN or INT32_MAX if unknown.		degE7 or mE4
6: Approach vector final point Y or Longitude	Y (mE4) or Longitude (degE7) (WGS84) coordinate of the final point of the approach vector to the portal. NaN or INT32_MAX if unknown.		degE7 or mE4
7: Approach vector final point Z or Altitude (MSL)	Z or Altitude (MSL) coordinate of the final point of the approach vector to the portal. NaN if unknown.		m

MAV_CMD_SET_EXPLORATION_BOUNDARIES Set exploration task world boundaries. When starting a new behavior, either the system has default boundaries or it's boundless. This message will be accepted when the task ID already exists, or otherwise should fail. In order to set the exploration task world boundaries, p1 and p2 identify the vertices of a rectangle, which define a 2D localization of the exploration area, while p3, matching the same coordinates of p1, provides the height limitation of the 3D volume to explore. In short, the coordinates are x1, y1, x2, y2, z and the height of the cuboid / 3D volume, where the point coordinates can be represented by p1(x1, y1, z+h), p2(x2, y2, z+h) and p3(x1, y1, z). COMMAND_INT should be used so to set the MAV_FRAME and consequently, frame and coordinates of the boundaries cuboid (MAV_FRAME_GLOBAL means global coordinates are being used, while MAV_FRAME_LOCAL_NED means local NED coordinates are being used).

Param (:Label)	Description	Values	Units
1: Exploration task ID	ID of the exploration task to set these boundaries to. Set to UINT8_MAX if not applicable or/and to set these boundaries to the current running or active exploration task.	min: 0 max: 255 increment: 1	

Param (:Label)	Description	Values	Units
2: Cuboid height	Exploration 3D space boundaries cuboid height. The Z local coordinate or altitude (MSL) of point 1 and 2 are computed by the sum of this height with the local Z or altitude (MSL) of point 3, i.e Z1 = Z2 = Z3 + cuboid height. NaN if not applicable.		m
3: X1 or Latitude 1	Local X (mE4) or Latitude (degE7) (WGS84) of point 1 of the exploration 3D space boundaries cuboid. NaN or INT32_MAX if not applicable.		degE7 or mE4
4: Y1 or Longitude 1	Local Y (mE4) or Longitude (degE7) (WGS84) of point 1 of the exploration 3D space boundaries cuboid. NaN or INT32_MAX if not applicable.		degE7 or mE4
5: X2 or Latitude 2	Local X (mE4) or Latitude (degE7) (WGS84) of point 2 of the exploration 3D space boundaries cuboid. NaN or INT32_MAX if not applicable.		degE7 or mE4
6: Y2 or Longitude 2	Local Y (mE4) or Longitude (degE7) (WGS84) of point 2 of the exploration 3D space boundaries cuboid. NaN or INT32_MAX if not applicable.		degE7 or mE4
7: Z3 or Altitude 3	Local Z or Altitude (MSL) point 3 of the exploration 3D space boundaries cuboid. This also represents the height of the bottom plane of the cuboid. NaN if not applicable.		m

Vehicle dynamics, states and configuration

This section covers other details that are not specific to the definition of the MAVLink protocol but that require standardization under this IOP.

System modes

The system modes, under the MAVLink spec, are a combination of a base mode and custom mode. Under this IOP, and following the approach taken in the PX4 Autopilot, a mode is identified using a `<MAIN_MODE>` or `<MAIN_MODE>:<SUBMODE>` formats. Although `MAV_CMD_DO_SET_MODE` allows to set the mode the vehicle should go into, other commands will force the vehicle into going into a specific mode (e.g. `MAV_CMD_NAV_TAKEOFF` should take the vehicle into an `AUTO:TAKEOFF` mode). The minimal modes to be supported are:

MAIN_ MODE:SUBMODE	**PX4** `main_mode` **value**	**PX4** `custom_mode` **value**	**Description**
INIT	10		Initial state of the vehicle before it takes-off. Vehicle should get back to this state after landing.
MANUAL	1		Manual control of the vehicle attitude.
ALTCTL	2		Assisted manual control of the vehicle altitude.
POSCTL:POSCTL	3	0	Assisted manual control of the vehicle position. Holds position with no input from the operator, but expects the operator to provide input.
POSCTL:ORBIT	3	1	Assisted manual control of the vehicle position while orbiting. Holds an orbit with no input from the operator, but expects the operator to provide input.
AUTO:LOITER	4	3	Autonomously holding position, without intervention of the operator.
AUTO:TAKEOFF	4	2	Executing an autonomous takeoff.
AUTO:LAND	4	6	Executing an autonomous landing.
AUTO:PREC_LAND	4	9	Executing an autonomous precision landing.

MAIN_ MODE:SUBMODE	PX4 `main_mode` value	PX4 `custom_mode` value	Description
AUTO:MISSION	4	4	Executing an autonomous mission.
AUTO:RTL	4	5	Autonomously returning to the launch point.
AUTO:FOLLOW_ TARGET	4	8	Autonomously tracking and following a target.
AUTO:GO_ THROUGH	4	10	Autonomously going through a spacial entity in the environment (portal, door, etc.).
AUTO:EXPLORATION	4	11	Executing an autonomous exploration task.

The following diagram provides the possible mode transitions.

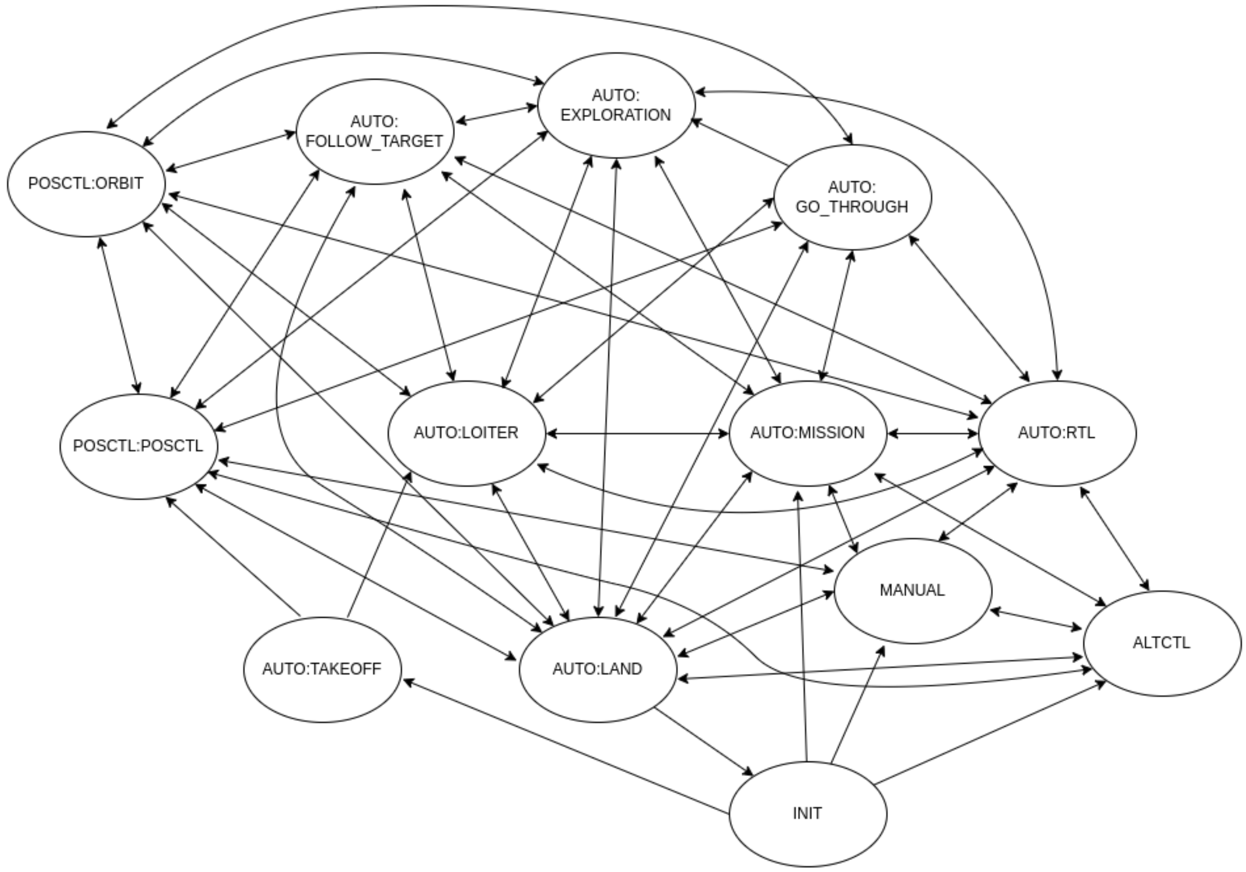

Figure 43: System Modes State Machine

MAV_CMD_DO_SET_MODE can switch the system to any mode (considering the conditions are met), but some other commands enforce switching to specific modes:

- MAV_CMD_NAV_LAND switches the system mode to **AUTO:LAND** or **AUTO:PREC_LAND** (depending on param2)
- MAV_CMD_NAV_TAKEOFF switches the system mode to **AUTO:TAKEOFF**
- MAV_CMD_DO_FOLLOW and MAV_CMD_DO_FOLLOW_REPOSITION switch the system mode to **AUTO:FOLLOW_TARGET**
- MAV_CMD_DO_ORBIT switches the system mode to **POSCTL:ORBIT**
- MAV_CMD_MISSION_START switches the system mode to **AUTO:MISSION**
- MAV_CMD_DO_FLIGHTTERMINATION switches the system mode to **INIT**

Arming procedure

This IOP considers that an arming procedure does note necessarily means a quick state transition but rather a process that can take several seconds. The reason for that is that the required mechanism that need to be enabled for a vehicle to be considered "armed" and ready-to-fly might vary from vehicle to vehicle. For that same reason, it is considered that the MAV_CMD_COMPONENT_ARM_DISARM can be processed as a long running command, depending on the first ACK that is sent from the vehicle side as a response to this command being sent from the GCS. This means that the progress of the arming procedure can also be captured through these same ACKs.

Autonomy Engine

Vehicles that supports advanced features via an "autonomy engine" may perform more as part of the arming sequence. A vehicle should advertise this requirement by broadcasting an additional `HEARTBEAT` for this component with component ID `MAV_COMP_ID_AUTONOMY_ENGINE`. The starting and control of this component is done through the `MAV_CMD_COMPONENT_CONTROL` command, which allows control of system components with MAVLink interfaces much like the UNIX `systemctl`.

A GCS will be required to send a MAV_CMD_COMPONENT_CONTROL command directed to the vehicle's MAV_COMP_ID_AUTONOMY_ENGINE (setting *param1* of the command to MAV_COMP_ID_AUTONOMY_ENGINE) requesting the component to be started - this is achieved by setting *param2* of the command to take the value of COMPONENT_CONTROL_START. The GCS should only send this command if it receives `HEARBEAT` messages from the `MAV_COMP_ID_AUTONOMY_ENGINE` component, meaning that the component is present.

The autonomy engine component should use the `system_status` field of the `HEARTBEAT` message to communicate whether it has started or not. A GCS should

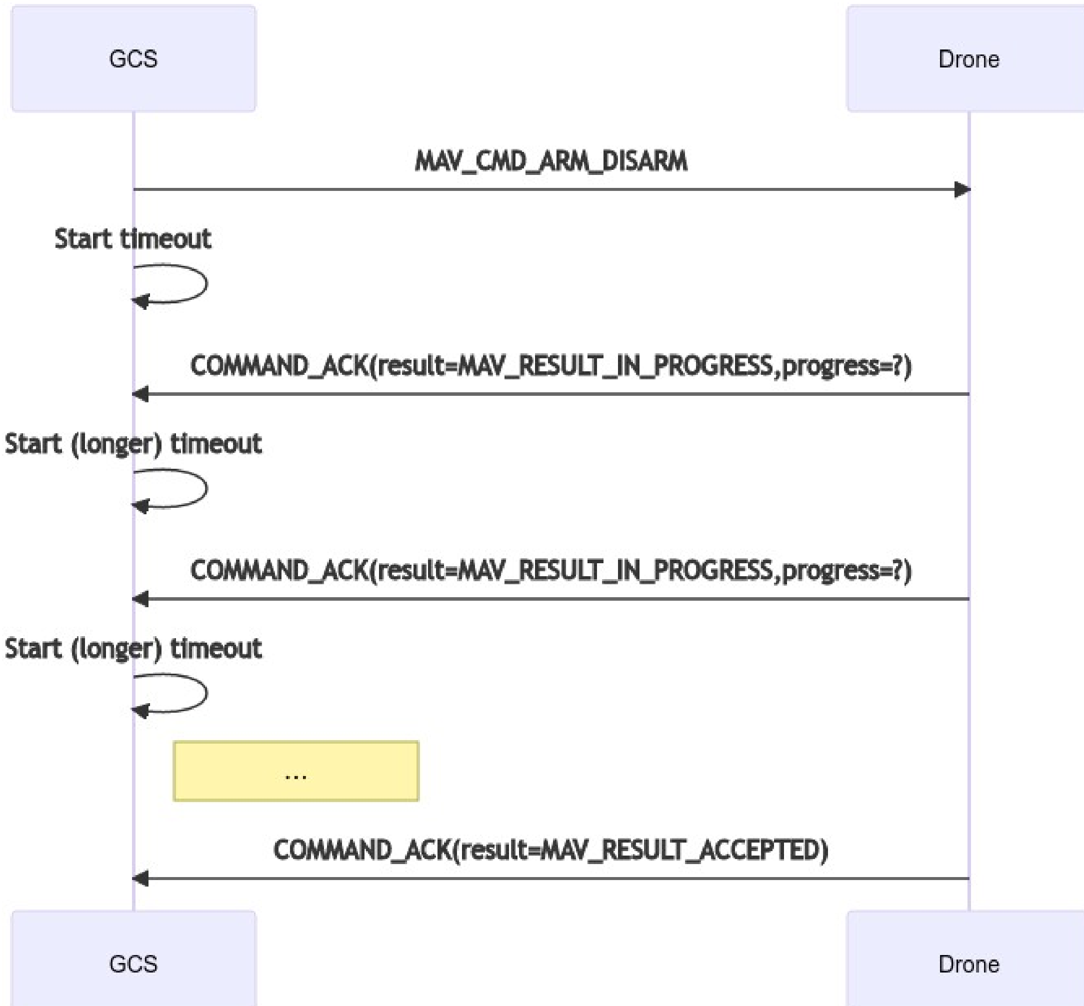

Figure 44: Arming Procedure

interpret the system_status as follows: - `MAV_STATE_UNINIT`, indicates it is waiting for receipt of the COMPONENT_CONTROL_START. - `MAV_STATE_ACTIVE`, indicates the autonomy engine is activated and ready.

On receipt of a `MAV_CMD_COMPONENT_ARM_DISARM`, if not already received externally the vehicle should generate and send the `MAV_CMD_COMPONENT_CONTROL` to start the autonomy engine. This moves the complexity of the workflow to the vehicle side, but guarantees that the GCS doesn't have to send yet an extra command. STATUSTEXT messages can be send from the vehicle to the GCS to provide status on the start of the autonomy engine component. Further iteration of this IOP will consider a mechanism to send this status in a more convinient and adequate way, potentially through the MAVLink Events Interface Protocol.