BetaTehPro

MAVLink FTP Implementation on Black Box Device with LittleFS File System

Vuk Stijepović

Filip Radojević

Aleksa Lazić

May 2025

Contents

1 Introduction

This document presents the implementation of the MAVLink FTP protocol, specifically designed for managing data stored within the black box system. The protocol facilitating data transfer between the ground station and the black box is described in detail. The gateways, GW gnd and GW sky, subscribe to FTP messages, which are enqueued by the black box's WORK task. The LOG task within the black box processes these messages and, based on the opcode contained in the payload, invokes appropriate functions located in the driver module of the code.

Test data have been transmitted using a Python script available in the samples section. Prior to testing, it is necessary to manually configure the computer's IP address to 192.168.1.151, which corresponds to the ground station. UDP packets are expected to be received on port 1024.

2 Message Format

MAVLink FTP messages follow a standardized packet structure defined by the MAVLink protocol. Each message consists of a header containing metadata such as the system ID, component ID, message ID (110 in FTP case), and sequence number, followed by the payload section which carries the actual FTP data. The payload size is fixed, ensuring consistent message length for reliable transmission. Finally, a checksum is appended to verify data integrity during communication between the ground station and the vehicle's black box system. This format enables efficient and robust file transfer operations over the MAVLink protocol.

In the MAVLink FTP protocol, a message consists of several fixed fields:

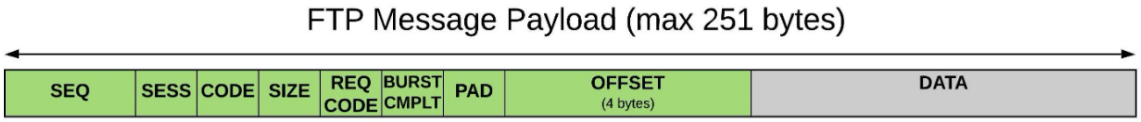
- **target network** — the network ID (0 indicates broadcast),
- **target system** — the system ID ,
- **target component** — the component ID within the system,
- and finally the **payload**, which is a 251-byte array.

The payload contains the actual FTP protocol data, such as commands (opcodes), offsets, data lengths, and the data itself (e.g., file blocks, directory names, status codes, etc.).

Therefore, what you have provided is an accurate description of the FTP payload within a MAVLink FTP message — it represents the data that is transmitted and decoded at the higher FTP protocol level.

The maximum file path length is limited to 118 bytes, while the maximum allowed size of the directory containing that file is 100 bytes.

The following table presents and briefly explains each field contained in the payload.



Byte Index	C version	Content	Value	Explanation
0 to 1	`uint16_t seq_number`	Sequence number for message	0 – 65535	All new messages between the GCS and drone iterate this number. Re-sent commands/ACK/NAK should use the previous response's sequence number.
2	`uint8_t session`	Session id	0 – 255	Session id for read/write operations (the server may use this to reference the file handle and progress).
3	`uint8_t opcode`	OpCode (id)	0 – 255	Ids for particular commands and ACK/NAK messages.
4	`uint8_t size`	Size	0 – 255	For Reads/Writes this is the size of the `data` transported. For NAK it is the number of bytes used for error information.
5	`uint8_t req_opcode`	Request OpCode	0 – 255	OpCode (of original message) returned in an ACK or NAK response.
6	`uint8_t burst _complete`	Burst complete	0, 1	1 = set of burst packets complete, 0 = more burst packets coming. Only used if `req_opcode` is BurstReadFile.
7	`uint8_t padding`	Padding		32-bit alignment padding.
8 to 11	`uint32_t offset`	Content offset		Offsets into data to be sent for ListDirectory and ReadFile commands.
12 to 251	`uint8_t data[]`	Data		Varies by OpCode. Contains command/response data, like path or error code.

The most crucial field is the opcode, as it directly determines which function should

be executed on the black box. The following table lists all available opcodes and their meanings.

Depending on the opcode, the interpretation of the remaining fields in the message may vary. Therefore, each opcode will be discussed in more detail in its respective section.

3 Op Codes

The following table shows a list of all OP codes.

Opcode	Name
0	`None`
1	TerminateSession
2	ResetSessions
3	ListDirectory
4	OpenFileRO
5	ReadFile
6	CreateFile
7	WriteFile
8	RemoveFile
9	CreateDirectory
10	RemoveDirectory
11	OpenFileWO
12	TruncateFile
13	Rename
14	CalcFileCRC32
15	BurstReadFile
128	ACK
129	NAK

Before diving into each individual function, it is necessary to consider some basic concepts. The drone's response to every message sent by the ground station should always include an opcode indicating either `ACK` or `NAK`. Meanwhile, the original function being executed is copied into the request opcode block.

In the case of a `NAK` response, the specific error that occurred is written into the data section. In the `size` section, the size of the data section (in bytes) must be

specified, both when sending in one direction and in the other, unless that field is reserved for something else (which will be noted in later sections of the text).

The following table presents every pre-defined errors.

Error	Name	Description
0	None	No error
1	Fail	Unknown failure
2	FailErrno	Command failed, error number sent back in `PayloadHeader.data[1]`. This is a file-system error number understood by the server operating system.
3	InvalidDataSize	Payload `size` is invalid
4	InvalidSession	Session is not currently open
5	NoSessionsAvailable	All available sessions are already in use.
6	EOF	Offset past end of file for `ListDirectory` and `ReadFile` commands.
7	UnknownCommand	Unknown command / opcode
8	FileExists	File/directory already exists
9	FileProtected	File/directory is write protected
10	FileNotFound	File/directory not found

One more thing worth mentioning is that during error enumeration, directories were referred to as files. It's a bit awkward, but we followed an already established standard.

3.1 None

The None function serves as a ping mechanism. The ground station sends a message filled with zeros to the drone, and the drone responds with a message in which the OpCode is set to ACK, while all other fields remain zero.

OP CODE	**Sender**
NONE	GCS
ACK	BB

3.2 Terminate Session

One of the key concepts in the MAVLink FTP protocol is **sessions**. Every time a file is opened or created, it is assigned a unique session ID. The system has a limited

maximum number of active sessions, so periodically it is necessary to free up some sessions.

Closing a session not only releases the session ID but also closes the file associated with that session. Because the session ID is frequently used in the payload, there is a dedicated block in the message structure specifically for this ID.

When the function to close a session is called, the ground station sends the session ID in this dedicated payload block. The rest of the message is empty, except for the opcode, which logically has the value 1 (indicating the close session command).

The drone checks if the session with the given session ID is active. If it is, the session is freed, and the drone responds with a message where the opcode is set to `ACK`, while all other values (except the request opcode) are zero.

In case of an error, the drone sends a `NAK` message, with the `size` field set to 1. The two considered errors are:

- The provided session ID is greater than the maximum allowed sessions (`NO_SESSION_AVAILABLE`)
- The session ID to be freed is not allocated (`INVALID_SESSION`)

It is important to note that in the standard protocol specification, only read sessions can be terminated; however, in our implementation, it is also possible to close write sessions.

OP CODE	**Sender**	**Data**	**Size**	**Session**	**Description**
TERMINATE SESSION	GCS	NULL	0	ID	Termination request
ACK	BB	NULL	0	ID	SUCCESSFUL
NAK	BB	NO SESSION AVAILABLE	1	ID	Requested ID is greater than the maximum allowed
NAK	BB	INVALID SESSION	1	ID	Session is not allocated

3.3 Reset Session

The Reset Sessions command is very similar to the Terminate Session command. The main difference is that the ground station does not send anything in the session block; instead, the drone automatically frees all active sessions.

The drone's responses to both functions are essentially the same, with the exception that, in the case of Reset Sessions, an invalid session error cannot occur — since the user does not directly interact with session IDs.

As with the previously mentioned Terminate Session command, our implementation also extends support to write sessions, not just read sessions.

OP CODE	Sender	Data	Size	Description
RESET SESSION	GCS	NULL	0	Resset request
ACK	BB	NULL	0	SUCCESSFUL
NAK	BB	INVALID SESSION	1	Session is not allocated

3.4 List Directory

This function is intended to list the contents of a directory on the file system. It will parse the target path from the incoming FTP payload and attempt to open the directory using the LittleFS API. For each valid entry in the directory (files and subdirectories), the function will prepare and send directory entries back to the client over multiple FTP response messages, depending on the number of entries and payload size limitations.

The sequence of operations is as follows:

The ground station sends the `ListDirectory` command, specifying the path to the directory and the index of the entry it wants to retrieve.

The request must contain the following in the payload:

- `data[0]` = the path to the directory
- `size` = length of the path string
- `offset` = the index of the first entry to retrieve (0 for the first, 1 for the second, etc.)

It is not explicitly necessary to open the directory being read, since the `ListDirectory` function will automatically open and close every time.

The drone's response is an `ACK`, which contains one or more entries (the first entry corresponds to the offset specified in the request).

The response payload must contain:

- `data[0]` = information for one or more consecutive entries, starting from the requested offset index. Each entry is separated by a null terminator (`\0`) and follows the format (where `type` is one of the letters: `F` = file, `D` = directory)

  This is how the representation of a single file entry should look:

  ```
  <type><file_or_folder_name>\t<file_size_in_bytes>\0
  ```

- `size` = The size of the `data`
- `offset` = how many entries have already been read from the file

It is necessary for the station to keep track of the current offset and repeat the same operation only for the shifted offsets. It is sufficient to take the offset from the response message and increment it by one. For each request, the drone will respond

with an ACK response as long as there is data to read. When the entire file has been read, the drone returns a NAK message with EOF data. EOF is sent in a separate message even if there is space left in the last ACK response.

The following table shows an example of the communication.

OP CODE	Sender	Data	Offset	Description
LIST DIRECTORY	GCS	DIR PATH	0	List request
ACK	BB	DATA	X	SUCCESSFULY sent X files.
LIST DIRECTORY	GCS	DIR PATH	X+1	List request
ACK	BB	DATA	X+Y	SUCCESSFULY sent more Y files
LIST DIRECTORY	GCS	DIR PATH	X+Y+1	List request
NAK	BB	EOF	0	END OF FILE

The data looks something like this:

`Ffile1.log\t223\0Ffile2.log\t755568\0Ffile3.log\t11111\0`

The following table shows all possible errors that may occur in sky (sender is always black box, size is always 1 and OP_code is always NAK).

Data	Description
INVALID DATA SIZE	Path length out of range
FILE NOT FOUND	Directory doesnt exist
FAIL	SD Card is busy
EOF	This is not an error, but the opcode is still NAK

3.5 Open File Read Only

Opening a file is possible in two modes: read-only and write-only. Each option has its own separate OPcode. Once the file is opened, a session must be assigned to it, which the ground station retrieves from the response message. The file remains open after the operation, and must eventually be closed by Reset or Terminate functions.

A major difference between our implementation and the recommended one is that the size of the opened file is not written in the size field, but rather in the data block. The size field is only 8 bits wide, meaning the maximum file size it can represent is 255 bytes, whereas our system is designed to handle files of megabyte size.

OP CODE	Sender	Data	Session id	Description
OPEN FILE RO	GCS	FILE PATH	0	Open file in read only mode request
ACK	BB	FILE SIZE	X	SUCCESSFULY

In the table above, x represents the session assigned to that file. In the case of an ACK, the file size is stored in 4 bytes — which is somewhat of an overkill — but since the data block isn't used for anything else anyway, it's acceptable.

If you need to open a file from the root directory, it's enough to specify just the filename, but it's also possible to access it using ./filename (the dot (.) represents the current directory, just like in Linux).

Data	Description
INVALID DATA SIZE	Path length out of range
FILE NOT FOUND	File or directory in the path doesnt exist
FAIL	File can't be opened
NO SESSION AVAILABLE	Each session is in use — the user must terminate one to proceed.
FAIL ERRNO	File u want to read is empty

3.6 Read File

This function reads size bytes from offset in session. Seeks to (offset) in the file opened in (session) and reads (size) bytes into the result buffer. Sends an ACK packet with the result buffer on success, otherwise a NAK packet with an error code. For short reads or reads beyond the end of a file, the (size) field in the ACK packet will indicate the actual number of bytes read. Reads can be issued to any offset in the file for any number of bytes, so reconstructing portions of the file to deal with lost packets should be easy.

The `ReadFile` function itself does not open or close files. The file must be opened beforehand using the `OpenFile RO` command, and the session should be properly closed afterward using `TerminateSession`.

- `size` = Specifies the number of bytes to read from the file. This can be any value (e.g., 50, 100, 200), but must not exceed the maximum capacity of the `data` block.
- `offset` = Indicates the position (in bytes) within the file where reading starts. This enables random access to file contents.

- `session` = Represents the session ID that was previously assigned using the `OpenFile RO` command. This session must be valid and active for the operation to succeed.

OP CODE	Sender	Offset	Size	Session id	Description
READ FILE	GCS	OFFSET	SIZE TO READ	ID	Read opened file request
ACK	BB	OFSET + SIZE OF DATA READ	SIZE OF DATA READ	ID	SUCCESSFUL (maybe?)
NAK	BB	LEN (FILE)	1	ID	END OF FILE (this is in data field)

In this function, an error may occur even if the response is an ACK. There is a possibility that certain files were read only up to the point where file corruption occurred. Therefore, on the ground station side, it is always necessary to check whether the size field in the request and the response match.

Data	Description
INVALID SESSION	Session with this ID is not alocated
FILE NOT FOUND	File doesnt exist with this session ID
FAIL ERRNO	Problem with seek function
FAIL	File can't be opened
INVALID DATA SIZE	File u want to read is empty
FILE PROTECTED	File is open in Write Only mode

3.7 Create File

The purpose of this OPcode is to create new file, which will later be used for writing data. The station sends the file path (in the form of folder1/folder2/file) in the data block, and expects an ID of the assigned session for that file in the response. When creating a file, the entire path to that file must already exist (e.g., folder1/folder2 must have been previously created in an earlier step). In the conceptual MAVLink implementation, it is recommended that if a file already exists at the same path, the file should be truncated. In our implementation, however, a NAK with the message FILE_EXISTS is expected. The file remains open after this function completes, so it is immediately ready for writing data.

OP CODE	Sender	Data	Session	Description
CREATE FILE	GCS	FILE PATH	0	Try to create file
ACK	BB	0	ID	SUCCESSFULY

Data	Description
INVALID DATA SIZE	Path size larger than allowed
FILE NOT FOUND	The file path does not exist
FILE EXISTS	The file already exists
NO SESSION AVAILABLE	No free session (need to release one)
FAIL	Unable to open file

3.8 Write File

This function handles the FTP write file operation. It expects:

- `session_id` = ID of the active session used to identify the file being written.
- `offset` = starting position in the file for the write operation.
- `size` = number of bytes to write.
- `data` = actual data bytes to be written into the file.

Before using the write function, it is necessary to either open a file in write-only mode or create a new file. After uploading a larger file, the session must be terminated. Just like in reading, writing to a file is done in multiple iterations, and before each new write, the offset must be set to match the current write position.

OP CODE	Sender	Offset	Size	Session	Data	Description
WRITE FILE	GCS	OFFSET	SIZE TO WRITE	ID	DATA TO WRITE	Write file request
ACK	BB	0	0	ID	0	SUCCESSFULY

A discussion of the complete file upload protocol — as a non-atomic version of this function, merged with other supporting operations — will be presented in a separate section.

Data	Description
INVALID SESSION	Session with this ID is not allocated
FILE NOT FOUND	File doesn't exist with this session ID
END OF FILE	Offset range is out of bounds
FAIL	File can't be opened
FILE PROTECTED	File is open in Write Only mode

3.9 Remove File

This function deletes a file at the specified path. It is required that the path to the file exists (for example, if the file path is ./dir1/file1, the directory ./dir1 must exist), otherwise the function will return an error. Since there are separate functions for deleting files and directories, it is important to ensure what exactly is located at the given path. The following tables shows the request sent by the ground station and the corresponding responses from the black box, both ACK and NAK.

OP CODE	Sender	Data	Description
REMOVE FILE	GCS	FILE PATH	Remove file request
ACK	BB	0	SUCCESSFULY

Data	Description
FAIL ERRNO	Directory doesn't exist
FILE NOT FOUND	Directory exists, but the file within it does not exist
FAIL	File can't be removed
FILE EXISTS	There is a file at the specified path, not a directory

The name of the FILE EXISTS error is quite confusing, but since no better alternatives are available and there aren't enough distinct errors to justify adding more ERRNO codes, we chose to use this option — especially to keep it consistent with the remove directory function.

3.10 Create Directory

This function handles the FTP "create directory" operation. It is responsible for processing a request to create a new directory in the filesystem.

It expects the following parameter:

- `data` = Filepath of the directory to be created (e.g., `home/folder`). Do not append a trailing slash (`/`).

The payload format is:

```
home/folder
```

The root directory is implicit (represented as `.`) and should not be included.

For example, to create the folder `folder` inside `home`, use the above format. To create a directory directly in the root, simply use `name of directory that you want`.

The function performs the following actions: it first extracts the full directory filepath from the FTP payload using a helper function. Then, it checks whether a directory or a file with the same name already exists in the filesystem. If it does, the function responds with a NAK message indicating that the directory already exists and the operation cannot proceed.

OP CODE	Sender	Data	Description
CREATE DIRECTORY	GCS	FILE PATH	Request to create a new directory
ACK	BB	0	Successful directory creation acknowledgment

If the directory does not exist, the function attempts to create it. Upon success, it sends an ACK message. If the creation fails, a NAK message is sent with one of the following error codes:

Data	Description
FILE EXISTS	The file or directory already exists
FAIL ERRNO	Insufficient memory to create the directory
INVALID DATA SIZE	Invalid data size or incorrect parameter
FAIL	Unknown error; general failure indication using the `FAIL` macro

3.11 Remove Direcotry

Remove directory closely resembles remove file. The difference between the original and our implementation is that it is important whether the given path points to a file or a directory — the function will return NAK if it encounters a file. The ground station is not required to release the sessions held by the files within the directory, because the black box will recursively go through the directory, release all sessions, and delete all files and subdirectories at the given path.

OP CODE	Sender	Data	Description
REMOVE DIRECTORY	GCS	DIR PATH	Remove directory request
ACK	BB	0	SUCCESSFUL

Data	Description
FILE NOT FOUND	Directory doesn't exist
FAIL	Directory can't be removed
END OF FILE	Problems with root directory
FAIL ERRNO	SD card is busy
FILE EXISTS	There is a directory at the specified path, not a file

An additional functionality is triggered if the value of data is set to a `dot` (.). In that case, everything in the root directory is deleted except for the boot count folder.

3.12 Open File Write Only

This function allocates a session for the file and grants write-only permission to it.

It expects parameters:

- `session_id` = The ID of the active session used to identify the file being written.
- `data` = The filepath of the file to be written.

The function performs the following actions: it first extracts the full directory filepath from the FTP payload using a helper function. Then, it checks whether a directory or a file with the same name already exists in the filesystem. If it does, the function responds with a NAK message indicating that the directory already exists and the operation cannot proceed.

After that, it allocates a session for the file and binds it to the specified filepath. The key point is that this function **OPENS** the file and grants write-only permission. **THIS IS IMPORTANT TO EMPHASIZE,** as the file must be explicitly closed or the session terminated afterwards. This function only prepares the file for writing; it does not perform any write operations itself.

OP CODE	Sender	Session	Data	Description
OPEN FILE WRITE ONLY	GCS	0	FILE PATH	Open file in write only mode request
ACK	BB	ID	0	SUCCESSFULY

If error appears somewhere in a program it will return NAK with following payload answer:

Data	Description
INVALID DATA SIZE	The filepath name is too long.
FILE NOT FOUND	The file that we wanna open is doesn't exist
NO SESSION AVAILABLE	All sessions in use
FAIL	Unknown error; general failure indication using the `FAIL` macro

3.13 Truncate File

This function handles the FTP truncate file operation. It expects:

- `session` = ID of the active session identifying the file to be truncated.
- `offset` = new size (in bytes) to which the file should be truncated.

Truncating a file involves removing all content after the specified `offset`. The session must be previously opened in write or read-write mode before calling this function. If the specified offset is greater than the current file size, the file is extended with zeros so that its size matches the `offset`. More on this will be discussed later, but before truncating a file, it is necessary to either open the file in write-only mode or create a new one. After truncation, the session remains open and can continue to be used for further operations (e.g., writing or closing).

OP CODE	Sender	Session	Offset	Description
TRUNCATE FILE	GCS	ID	OFFSET	truncate file request
ACK	BB	ID	0	SUCCESSFULY

Data	Description
INVALID SESSION	Session is not alocated
FILE PROTECTED	The file is in read only mode
FILE NOT FOUND	File does not exist
FAIL	Unknown error

3.14 Rename File/Directory

This function handles the FTP operation for renaming a file or directory.

It expects the following parameter:

- `data` = The payload containing the old and new file/directory paths.

The payload format is:

```
OldPath\0NewPath\0
```

Where both paths are `null-terminated` strings.

The function attempts to rename the specified file or directory using the provided `NewPath`. If successful, it responds with an `ACK` message.

OP CODE	Sender	Data	Description
RENAME	GCS	FILE PATH	Request to rename a file or directory
ACK	BB	0	SUCCESSFULY

If an error occurs, the system will respond with a `NAK` message containing one of the following payload values:

Data	Description
`FILE EXISTS`	The old and new file/directory paths are identical; renaming is not needed
`INVALID DATA SIZE`	The file or directory name is too long or improperly formatted
`FILE NOT FOUND`	The specified old file or directory does not exist in the filesystem
`FAIL`	An unknown error occurred; general failure indicated using the `FAIL` macro

3.15 Calculate File CRC32

This function handles the FTP operation for calculating the CRC32 checksum of a file.

It expects the following parameter:

- `data` = The payload containing the filepath of the file for which the CRC32 checksum will be calculated.

The payload format is:

```
Filepath (e.g.  home/file1)
```

The function expects the payload to contain the filepath of the file whose CRC32 checksum is to be computed. The CRC calculation follows the algorithm described here: https://mavlink.io/en/guide/crc.html. If the operation is successful, an `ACK` message is returned.

OP CODE	Sender	Data	Description
CALCULATE FILE CRC32	GCS	FILE PATH	Request to calculate CRC32 checksum of a file
ACK	BB	CRC32 (first 4 bytes in payload)	Successful CRC32 calculation; CRC32 value returned in payload

If an error occurs, the system responds with a `NAK` message containing one of the following payload values:

Data	Description
INVALID DATA SIZE	Filepath is empty or too long
FILE NOT FOUND	The specified file does not exist
FAIL	General failure during CRC calculation

3.16 Burst Read

This function handles the FTP burst read operation. It expects:

- `session` = ID of the active session identifying the file to be read.
- `size` = how mutch data you wnat to read in one message.
- `offset` = position in the file from which the read operation should start.

Burst read enables continuous reading of file data in large blocks with reduced protocol overhead. Unlike standard read, which requires specifying the number of bytes to read in each request, burst read operates in a loop where the sender continues to deliver data until the file is fully read or an error occurs. After the operation is completed, the session remains open and may be used for further operations such as closing or truncating the file. After sending a burst read response, the ground station does not send any further requests until the burst read response includes a message where the burst read block value is equal to one. This indicates the end of the read operation.

OP CODE	Sender	Session	Offset	Size	Burst Complete	Description
BURST READ FILE	GCS	ID	OFFSET	DATA PER BURST	0	Burst file read request
ACK	BB	ID	CURRENT OFFSET	DATA PER BURST	0	SUCCESSFULY
...						
ACK	BB	ID	CURRENT OFFSET	0	1	SUCCESSFULY

In the offset field of ACK messages, it is possible to see how much of the file has been read, so that in case of an error, reading does not have to start from zero.

Data	Description
INVALID SESSION	Session is not alocated
INVALID DATA SIZE	Size is to big
FILE NOT FOUND	File does not exist
FAIL ERRNO	Offset is to big
FAIL	Unknown error

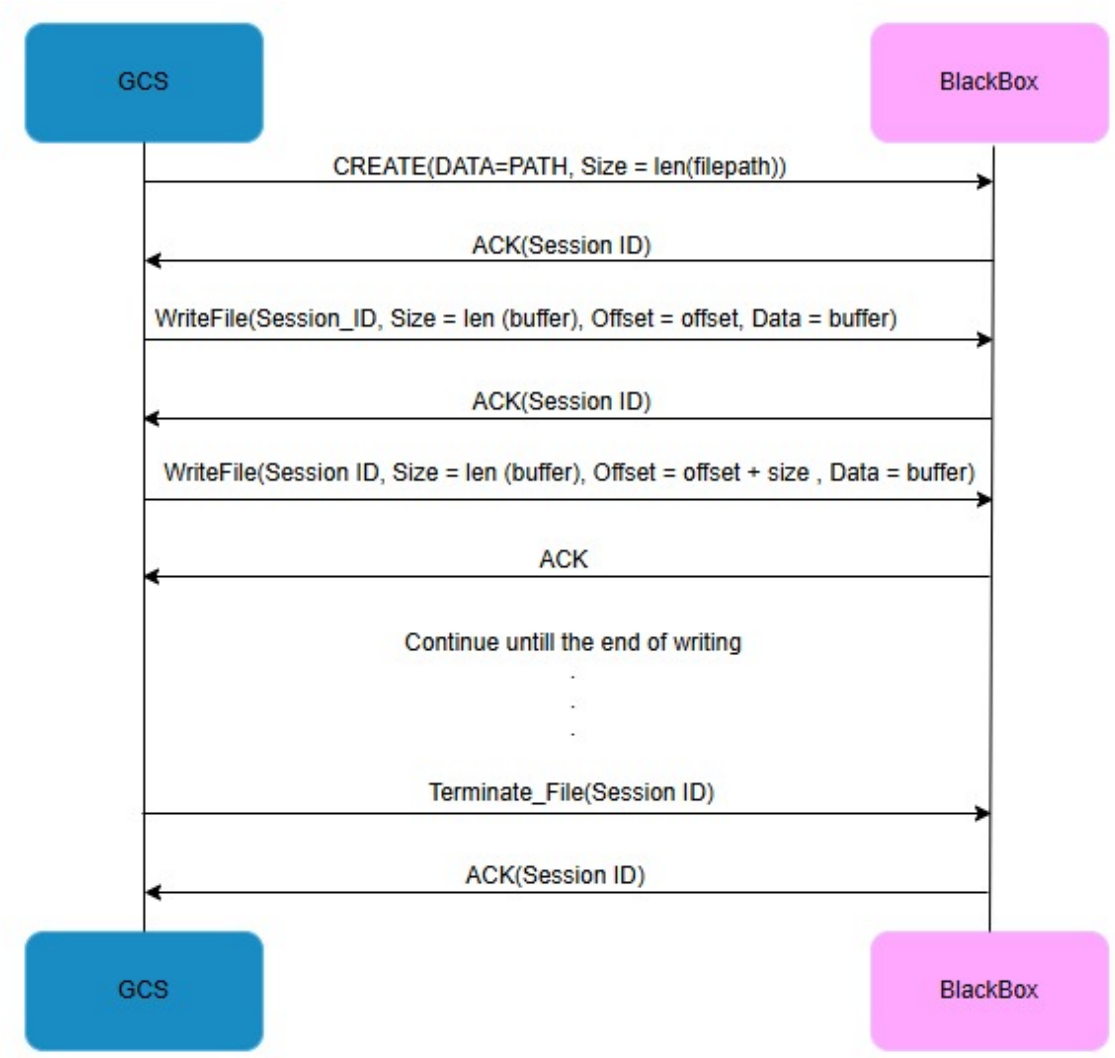
In every response sent by the drone to the station, the offset block contains the number of files delivered so far; this part differs from the original documentation. Also, the last message, in which the burst complete field equals one, is empty. This significantly complicates the implementation (mainly due to safety checks), while not adding much overhead. The file remains open after the reading is completed.

4 Non Atomic functions

The previous functions have mostly served as tools used by more complex, higher-level functions. Complex functions require a handshake between the ground station and the black box, which forms the basis of the file transfer protocol itself. All non-atomic instructions will be discussed later in the text.

4.1 Upload file

Uploading a file generally implies creating a new file and writing specific data into it. A typical upload cycle proceeds as follows:



- The ground control center (GCS) creates the desired file at the specified location.
- The drone responds with an `ACK` or a `NAK`. In case of a `NAK`, the GCS must inspect the response and identify the problem from the `data` block. This

response also includes a session ID, which will be used by the GCC for future access to the newly opened file.

- The GCS writes the file chunk by chunk to the black box.
- The drone responds to each chunk individually. The GCS waits for a response between each chunk.
- After writing is completed, the GCS closes the session.
- The drone replies with an `ACK`, which marks the successful completion of the upload process.

All possible errors are already defined in the section describing OP codes.

4.2 Download file (Not Recommended)

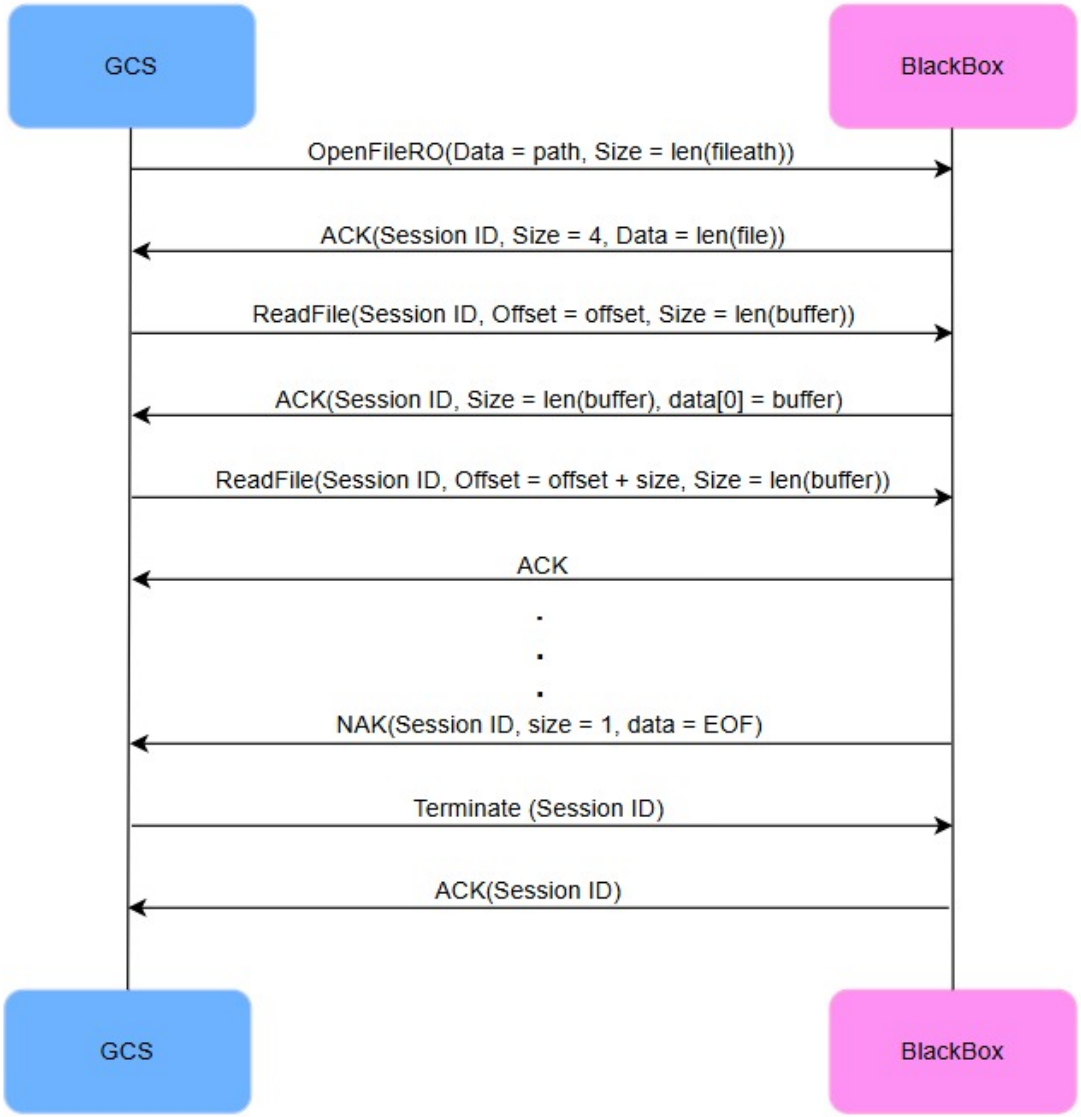
The non-atomic version of the `Read File` operation closely resembles the file upload process:

- The ground control center (GCC) first opens the desired file in read-only mode at the specified location.
- The drone responds with an `ACK` or a `NAK`. If an `ACK` is received, it includes the file size and the assigned session ID.
- The GCC reads the file chunk by chunk from the drone.
- The drone responds to each chunk individually. The GCC waits for a response between each chunk.
- The GCC waits for a `NAK` message in which `data = END_OF_FILE`, indicating the end of the transmission.

The name read file is already used in the non-atomic version, which is why the function is called download file here, even though the name isn't entirely ideal.

his requires a large handshake, so this method of reading is not recommended for use, but it is important that it exists for cases where establishing a secure connection is not possible. In normal scenarios, it is preferable to use the burst variant.

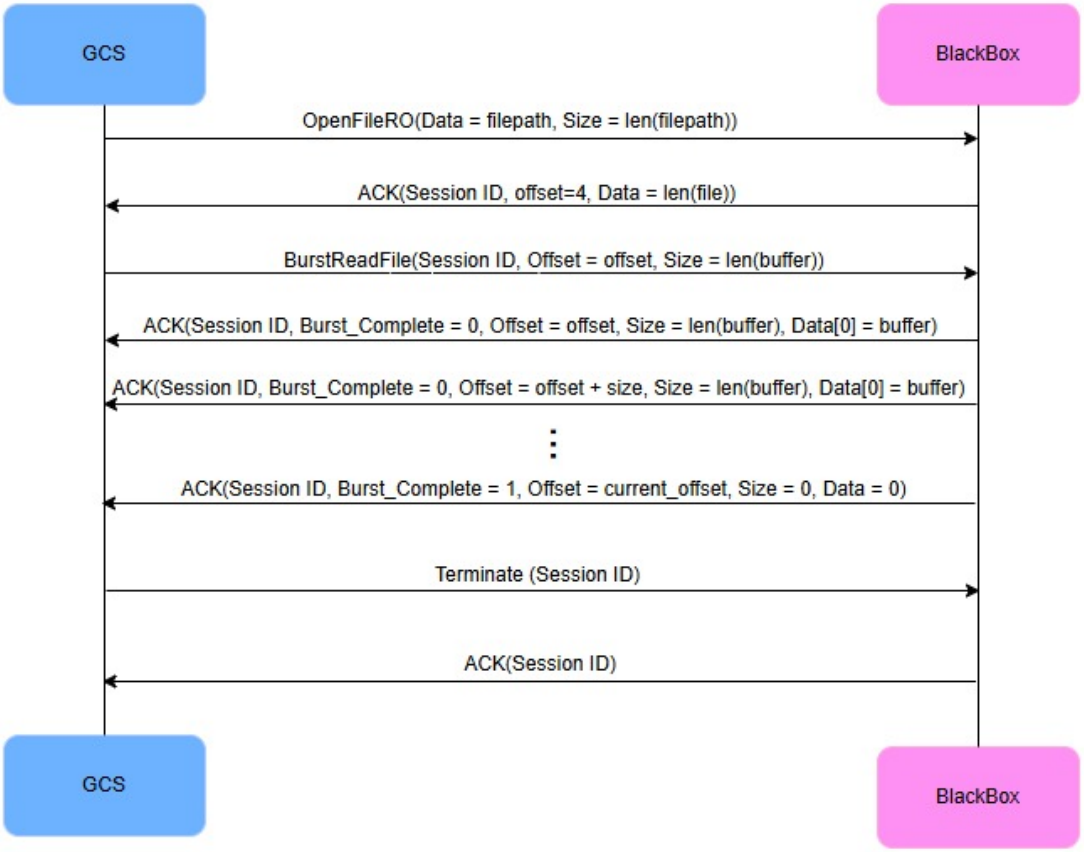
By len(buffer) in messages related to initialization of reading, it is meant the desired size of the reception packet.



4.3 Download files in burst

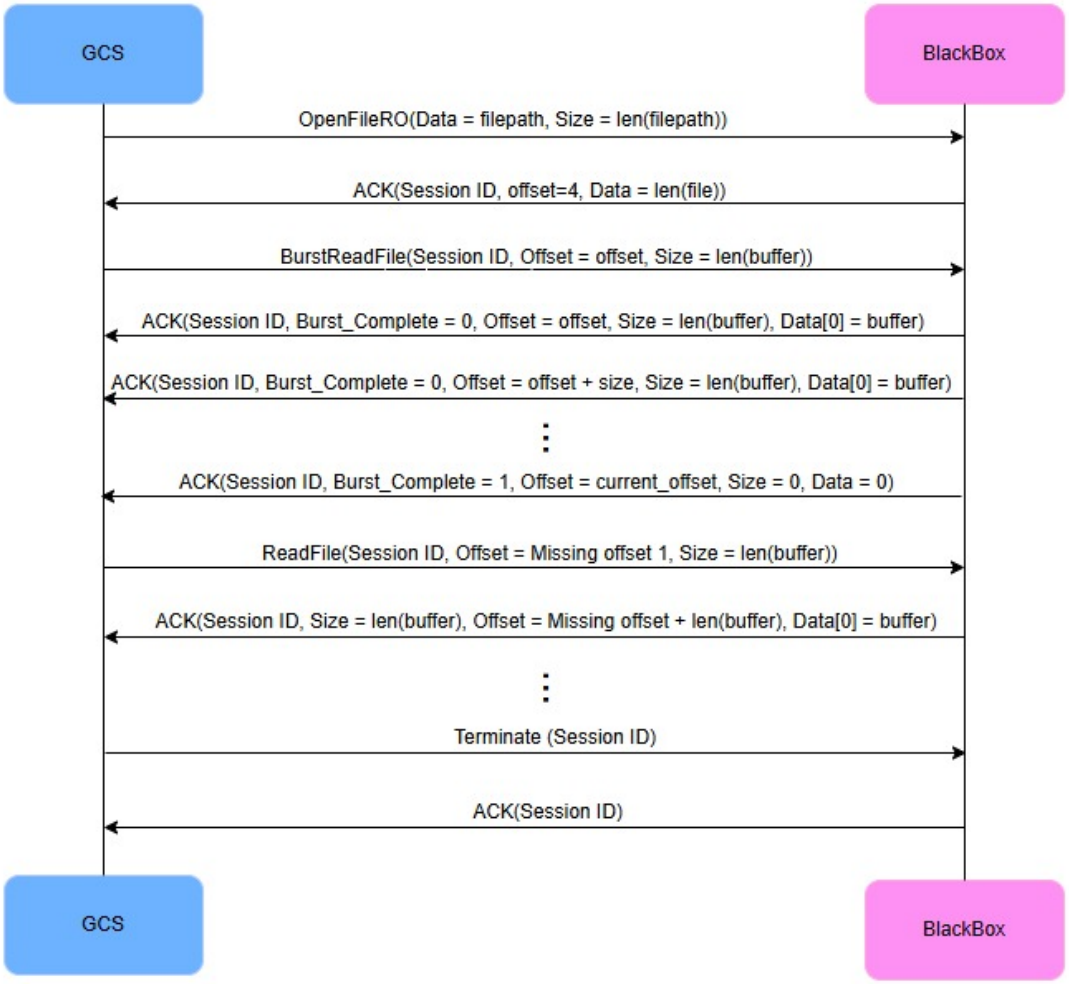
The only reason this function is separated is because it requires adding an open file (read-only) operation before execution and a terminate file operation afterward. It has already been discussed in the section about opcodes, so it will not be mentioned further here.

The first image shows the system flow diagram in the case where all files are successfully transmitted, which unfortunately is often not feasible—especially when the transfer becomes wireless. It is likely that some files will not reach their destination, and since the base station does not return ACKs (due to the speed of the transmission), it is necessary to re-request the missing files once the transfer is complete. Therefore, on the base station side, it is necessary to keep track of the offsets of the messages that did not arrive, in order to request them again later.



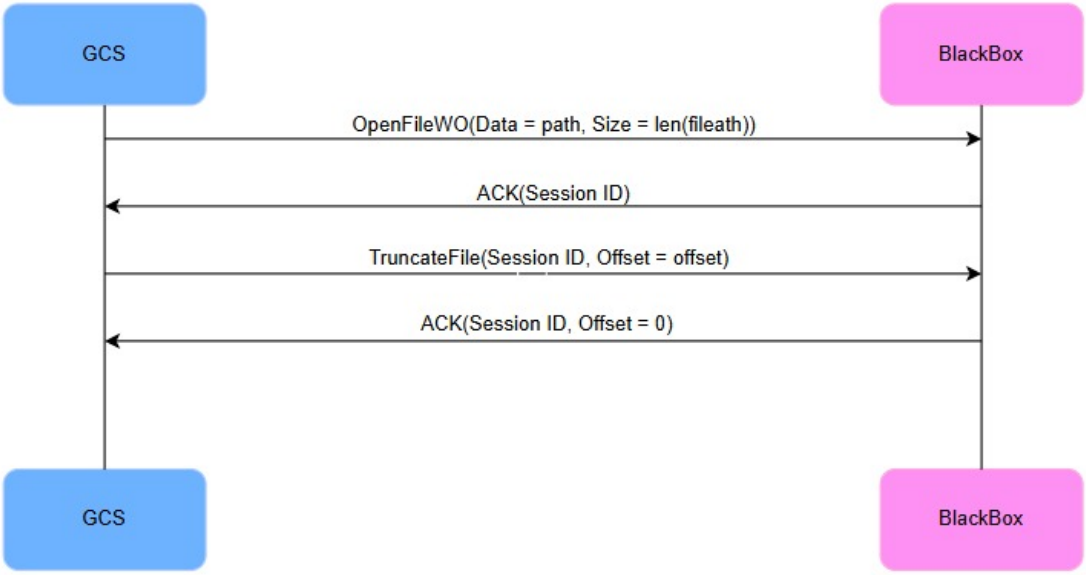
The second image illustrates an example of a successful (thats hope) data transfer.

It is also important to note that it is recommended to track the offset returned even after regular read functions, in order to ensure that file corruption has not occurred. By len(buffer) in messages related to initialization of reading, it is meant the desired size of the reception packet.


GCS
BlackBox
OpenFileRO(Data = filepath, Size = len(filepath))
ACK(Session ID, offset=4, Data = len(file))
BurstReadFile(Session ID, Offset = offset, Size = len(buffer))
ACK(Session ID, Burst_Complete = 0, Offset = offset, Size = len(buffer), Data[0] = buffer)
ACK(Session ID, Burst_Complete = 0, Offset = offset + size, Size = len(buffer), Data[0] = buffer)
ACK(Session ID, Burst_Complete = 1, Offset = current_offset, Size = 0, Data = 0)
ReadFile(Session ID, Offset = Missing offset 1, Size = len(buffer))
ACK(Session ID, Size = len(buffer), Offset = Missing offset + len(buffer), Data[0] = buffer)
Terminate (Session ID)
ACK(Session ID)
GCS
BlackBox

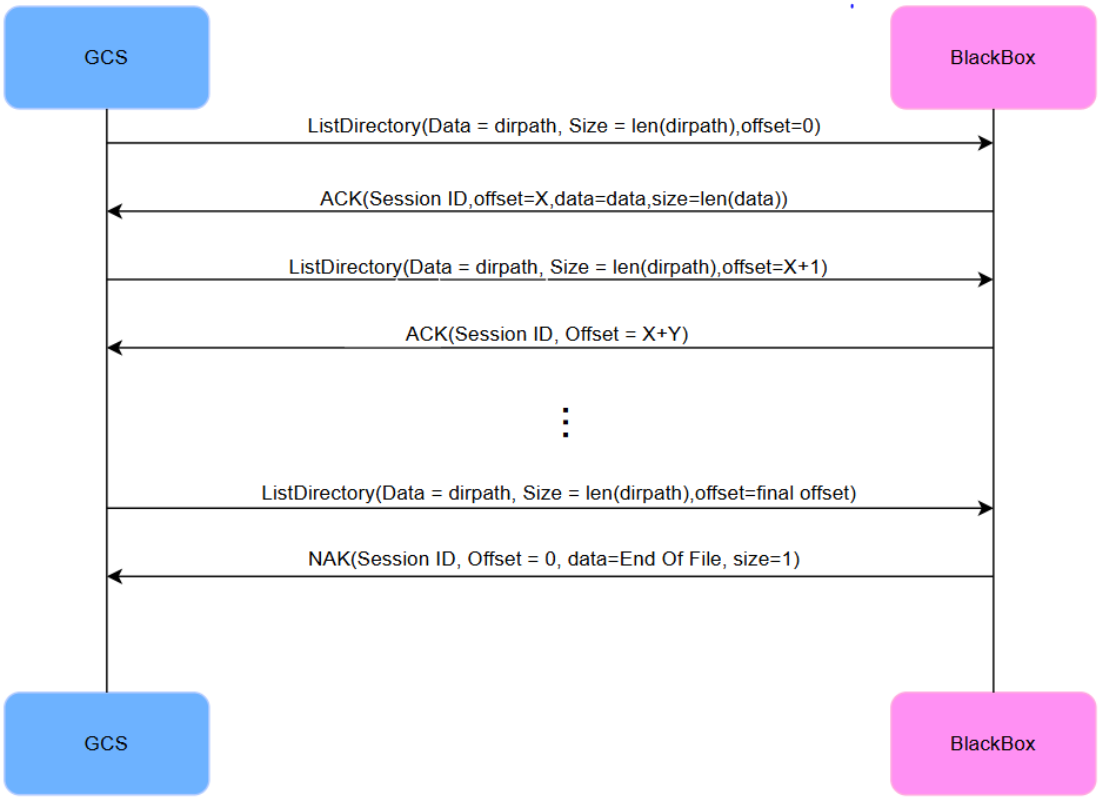
4.4 Truncate file

The only reason this function is separated is because it requires opening the file in write-only mode before execution, and then truncating the file as the operation demands. This has already been discussed in the section about opcodes, so it will not be mentioned again here.



4.5 List Directory

The list directory command requires a handshake, which is why its diagram is provided, although the entire communication process has already been explained in detail in the previous section



5 Timeout Logic

The timeout logic on the GCS side works as follows: if the GCS does not receive the expected ACK within approximately **50 ms**, it will resend the request. This process can repeat up to **six times**, as defined by the MAVLink FTP standard.

On the BlackBox side, the implementation ensures that if a request with the same **sequence number** is received again, the system will resend the previous response. This is because the GCS may have sent the original request but failed to receive the corresponding ACK/NAK, prompting it to retry.

In this mechanism, any data corruption can only occur on the BlackBox side, since the **GCS** is **correctly resending the request**, and **BlackBox** is responsible for providing a **consistent and correct response** based on the sequence number.

Furthermore, if BlackBox receives a sequence number that is **same as the previously processed one**, it will interpret this as a delayed message from the GCS and respond with the **previously sent data**. This behavior ensures consistency in communication, although it highlights that the system experienced timing irregularities and did not operate as intended.

If the BlackBox receives a sequential number that is **lower than the sequential number of the last received data**, that message is discarded. Therefore, it is important that the ground stations do not send requests until an ACK, NAK, or burst complete message is received.

As part of the MAVLink FTP protocol, a timeout mechanism is implemented for handling **BURST READ** operations. When the **BURST COMPLETE flag** is set in the response, the system initiates a retry logic that monitors for repeated requests within a 50 ms time window.

Upon sending a response with **BURST COMPLETE = 1**, the system waits up to **50 milliseconds** to detect whether the same request is received again (based on the sequence number). This mechanism assumes that the ground control station may retransmit the same request if it did not receive the expected ACK.

If a repeated request is detected within the timeout window, the system responds with **the same data as previously**, maintaining consistency. This process may repeat up to **six times**. If no repeated request is detected within each interval, the response is resent to ensure delivery.

This mechanism ensures reliable communication and proper handling of retransmissions in time-sensitive FTP operations.

6 LittleFS Fuse

With this library, we can read our SD card via the `FUSE` filesystem. The following steps need to be performed:

1. Check the version of `fusermount`:

   ```
   fusermount -V
   ```

2. Install the required libraries:

   ```
   sudo apt-get install libfuse-dev
   ```

3. Compile the project:

   ```
   make
   ```

4. Find a free loop device:

   ```
   sudo losetup --find
   ```

   Use the output of this command in the next step:

5. Assign permissions to the loop device:

   ```
   sudo chmod a+rw /dev/loopX
   ```

   Where `loopX` is the device you obtained in the previous command.

6. Create a directory to mount the SD card:

   ```
   mkdir mnt
   ```

7. Check which device corresponds to your SD card:

   ```
   lsblk
   ```

   Identify the appropriate device, e.g. `/dev/sdb`, `/dev/sdc`, `/dev/sdd`...

8. Mount the SD card using the LittleFS Fuse program:

   ```
   sudo ./lfs --blocksize=2048 --block_count=15589376 /dev/sdX mnt
   ```

   Replace `/dev/sdX` with the correct device from the previous step, and `mnt` is the mount directory.

9. List the files and directories in the mounted folder:

   ```
   sudo ls -la ~/littlefs-fuse/mnt
   ```

After these steps, you will be able to access the contents of the SD card via the `mnt` directory.

7 Reference

- https://mavlink.io/en/services/ftp.html – MAVLink FTP documentation
- https://github.com/PX4/PX4-Autopilot/blob/main/src/modules/mavlink/mavlink_ftp.h – PX4 Autopilot Project on GitHub
- https://github.com/ArduPilot/pymavlink/blob/master/mavftp.py – ArduPilot Project on GitHub
- https://github.com/littlefs-project/littlefs – LittleFS on GitHub
- https://github.com/littlefs-project/littlefs-fuse – lfsFuse (tool for reading lfs files)
- https://github.com/filipradojevic/FTP – Project on GitHub