1. Introduction

1.1. ArduPilot Version

All the research and tests done in this article were using ArduPilot version **Copter-4.1**.

1.2. Goal

Our goal is to disable the ArduPilots auto-land feature, without effecting any of its other failsafes such as RTL in case of RC disconnection.

Landing procedure can take place in multiple scenarios, including fence failsafe, RTL failure and probably more. But in this article we are going to talk only about those auto-land triggers:

- GCS failure
- EKF failure
- RC failure
- Low battery

1.3. Table of Contents

2. GCS Failure

GCS failsafe is triggerd when hearbeats and RC overrides from the GCS are missing for more then `FS_GCS_TIMEOUT` (see RC Failsafe Code Research).

For configuring the GCS failsafe behavior, we can use 3 parameters:

- `FS_GCS_TIMEOUT` - The timeout untill the GCS failsafe will kick in.
- `FS_GCS_ENABLE` - The action that the failsafe will trigger. The options that we are intrested in are:
    - **Disabled** (=0)
    - **`RTL`** (=1)
    - (*deprecated*) `RTL` or Continue with Mission in Auto Mode (=2)
    - **`SmartRTL` or `RTL`** (=3)
    - `SmartRTL` or `LAND` (=4)
    - `LAND` (=5)
    - Auto `DO_LAND_START` or `RTL` (=6)
- `FS_OPTIONS` - Aditional options for battery, RC, & GCS failsafes. The option that we are intrested in is: `bitmask#4 (16) - Continue if in pilot controlled modes on GCS failsafe`. This means that if the drone is in `ALT_HOLD` mode - the GCS failsafe will not be triggered at all, but if in `GUIDED`, it will.

3. EKF Failure

EKF failsafe kicks in when the EKF variance gets **over** the configured threshold. The default action the ArduPilot does in case of EKF failure is mode change to `LAND`.

3.1. Option 1: Changing EKF Failsafe Action

- The `FS_EKF_ACTION` controls what action will be triggerd in case of EKF failsafe. The option that we are intrested in is:
    - Change to mode `LAND` (=1)
    - **Change to mode `ALT_HOLD`** (=2)
    - Change to mode `LAND` even in `STABALIZE` mode (=3)

3.2. Option 2: Changing EKF Failsafe Threshold

- Setting the `FS_EKF_THRESH` parameter to `0` seems to disable the EKF failsafe.
    - This parameter contolls the EKF variance threshold that affects the EKF failsafe trigger.

All the usages of this parameter in the code are:

- AP_Arming_Copter::mandatory_gps_checks - There is no reference to the `fs_ekf_thresh` parameter before the first code block mentioned there. This may mean that we can not arm when there is no valid position (which is different then having high EKF variance), even if this parameter is set to 0.
- Copter::ekf_check - This means that EKF failsafe will **not** be triggered when this fucntion is called.
- Copter::ekf_over_threshold - This means that every place in the code that will check the EKF status using this function will get a response that indicates that everything is ok.

Because it is not officialy documented that this will disable the EKF failsafe, it is recommended to check if there are more places that check the ekf or gps status without the two last functions. For example - see the discussion at the RC failure section.

A simple test using the ArduPilot SITL shows that if the drone is in guided mode, after setting this parameter to 0, and disabling the GPS in the simulation, nothing happens. The drone stays in `GUIDED` mode and slowly drifts.

4. RC Failure

- RC failure by its own will trigger a failsafe that will do the action that is configured in `FS_THR_ENABLE`.
- In case of RC failure and EKF failure **at the same time**, a special failsafe will kick in, and will change mode to `LAND`.

We are intrested in mode change to `RTL` in case of RC failsafe with healthy EKF, and whould like to disable any mode change to `LAND` that is caused by an RC failsafe.

When a GCS has a joistick connected to it, it sends its commands using an RC override MAVLink message, which is one of: `RC_CHANNELS_OVERRIDE (#70)` or `MANUAL_CONTROL (#69)`. Each override of this kind, has a timeout that is being determined by the `RC_OVERRIDE_TIME` parameter.

The term "regular" or "real" RC (not override), is referring to an RC that is connected physically to the pixhawk.

If choosing the [second EKF failure option](), it can become difficult to preserve the auto `RTL` behavior, because the ArduPilot will try and return home but in case of no actual GPS signal, the behavior may be un-predictable. In a short test with the ArduPilot SITL I found that in case of loosing GPS and then RC (with `FS_EKF_THRESH` set to 0 and `FS_THR_ENABLE` set to `RTL`), it changes mode to `LAND` although the EKF threshold is 0 (probably because there is no valid position from the GPS). If there is a GPS signal it does changes mode to `RTL`.

- *A fast dive in to the scenario described here, took me to the function `Copter::position_ok` that eventually uses the function [NavEKF2_core::healthy]() to determine if the position is OK or not. This function **does not** refer to the threshold configured (i.e. any place in the code that will use the function `Copter::position_ok` will receive an answer that does not take into account the EKF threshold).*

4.1. Option 1: Sending Fake RC Overrides

- The ArduPilot treats RC overrides the same as regular RC commands. This means that when sending fake RC override messages from the **mission computer**, the ArduPilot will behave as if the overrides are comming from its real GCS, and will not activate any RC failsafe, or will disable an previously activated RC failsafe.
  - RC loss can be identified by missing GCS heartbeats or by an unintended mode change to `LAND`.
    - **We need to make sure not to cancel wanted lands**.
    - If setting the `RC_OVERRIDE_TIME` parameter to be bigger then what we consider as RC failure, RC failure can be detected before the failsafe kicks in (by identifying missing GCS heartbeats), and by that RC failsafe will be never triggerd.
  - After identifying an RC loss, we can start sending overrides and change mode to `ALT_HOLD` or `RTL` (determined by the EKF variance).
- If needed, we can set the `RC_OVERRIDE_TIME` parameter to be a big number, or `-1` for disabling the timeout. In case of using the `-1` option, the ways to disable an override is to send a newer override,

change this parameter back to a valid timeout or to disable overrides using the `RC_OPTIONS` parameter (see Good to know).
- RC overrides will be accepted only if the sender *system ID* will be the same as the `SYSID_MYGCS` parameter (see RC Override Code Research).
  - Because of that, for sending overrides, we need to set the `SYSID_MYGCS` parameter to be the mission computer *system ID*, and eventually set it back to be the real GCS *system ID*.

4.2. Option 2: Changing RC Failsafe Timeout

There are multiple ways to set the RC failsafe timeout, although only one seems to work.

*Using the `AFS_RC_FAIL_TIME` parameter is not relevant becase all it does is disables the RC **termination** failsafe, which is a failsafe that will be activeted only if we enable the advanced failsafe mechanism. The RC termination failsafe is a failsafe that terminates (crashes) the drone in case of RC loss for more then what is configured in `AFS_RC_FAIL_TIME`.*

Setting `RC_OVERRIDE_TIME` to `-1` will not work as well (see RC failsafe trigger code research).

The option that seems to work is:

- Change a bit the code by setting the `FS_RADIO_TIMEOUT_MS` define and the `FS_RADIO_RC_OVERRIDE_TIMEOUT_MS` define in the `APM_Config.h` file to be a very big number. Those values are used **only** by the function Copter::read_radio and only for enabling the RC failsafe.

4.3. Option 3: Changing RC Failsafe Action To Disabled

- It is documented in the ArduPilot Radio Failsafe page that the `FS_THR_ENABLE` parameter can controll the RC failsafe action, after a check in the SITL, when setting the value to `0 - Disabled`, no RC failsafe will be triggered.
  - This will disable auto-RTL in case of radio failure with healthy EKF. We can overcome this by using the GCS failsafe mechanism.
  - This will disable auto-landing in case of radio failure and EKF failure at the same time (If using option 1).
    - There is one rare edge case where the vehicle may change mode to `LAND` (assuming we are using the GCS failsafe mechanism to overcome the RTL problem described in the previous point).

4.4. Good to know

- If setting the second bit in the `RC_OPTIONS` parameter, the ArduPilot will ignore MAVLink overrides.
  - This can help us gain controll if the auto pilot does nonesense actions.
    - This of course assumes that there is a real RC connected, it is important to remeber that **the joystick in the GCS is commanding the drone via overrides too**.
- In general if we fell into an EKF failsafe, because the way we do DR (sending RC overrides) - a GCS or RC failsafe will never be present.

4.5. Preserving The Auto `RTL` Failsafe In Case Of RC Failure And Healty EKF

This is relevant only for option 3. If option 1 or option 2 is used, the ArduPilot will think that the RC overrides are comming from its real GCS and because of that, will never activate the GCS failsafe.

The RC commands the pilot is sending are anyway being sent via MAVLink (because they are sent using the GCS), so RC failure (missing regular RC and overrides) is equivilant to GCS failure (missing heartbeats and RC overrides, see the code research).

Because of that, if configuring the folowing:

- GCS failsafe action: mode change to `RTL`
- GCS failsafe disabled on pilot-controlled modes (see GCS Failure)
- EKF failure action: mode change to `ALT_HOLD`

Then:

- In case of EKF failure, the EKF failsafe will change mode to `ALT_HOLD`.
- In case of EKF failure and then RC failure, the EKF failsafe will change mode to `ALT_HOLD` and then the GCS failsafe will not kick in.
- In case of RC failure, a GCS failure will be present as well, so `RTL` mode will be enabled.
- In case of RC failure and then EKF failure, the GCS failsafe will trigger `RTL` and then the EKF failsafe will trigger mode change to `ALT_HOLD`.

And this is the wanted behavior.

5. Battery Failure

For configuring the battery failsafe we can use:

- `BATT_LOW_TIMER` - Set timeout before a low battery voltage failsafe will be triggerd.
- `BATT_FS_VOLTSRC` - Set the source from which low voltage events will come from.
- `BATT_FS_LOW_ACT` - The action to take if the battery gets low. (0 is disable)
  - `BATT_LOW_VOLT` - Set minimum voltage to trigger the low battery failsafe.
  - `BATT_LOW_MAH` - Set minimum battery capacity to trigger the low battery failsafe.
- `BATT_FS_CRT_ACT` - The action to take if the battery gets critical. (0 is disable)
  - `BATT_CRT_VOLT` - Set minimum voltage to trigger the critical battery failsafe.
  - `BATT_CRT_MAH` - Set minimum battery capacity to trigger the critical battery failsafe.

Those parameters exist in the form of BATTx_... for x in {2, 3, 4, 5, 6, 7, 8, 9} too.

6. Pros and Cons

6.1. Changing The EKF Failsafe Action

- Pros
  - Stable - Mode change is a very common (and simple, conceptually) action that is being tested all over the world by probably almost every ArduPilot user. Moreover, `ALT_HOLD` mode is a common mode, and for the same reasons, it is stable and reliable.
- Cons

  - In case of active RC failsafe - the EKF failsafe action will be mode change to `LAND` (see the function ekf_check).

6.2. Changing The EKF Failsafe Threshold

- Pros
  - Any piece of code that will ask the Copter class for its EKF status will receive a message that everything is OK, so if there are hidden places in the code, or if changes will be made in the future, the EKF failure will stay transperent.
- Cons
  - This is not a common thing to do, and may lead to un-predictable behavior in varios edge cases (for example, see the discussion at the RC failure chapter). For proving that this approach is safe, a deep code research needs to be done every time we upgrade the ArduPilot version.
    - In the case of this version - the behavior may not be stable enough.
  - The responsability on changing mode to a GPS-free mode is on the mission computer. If the mission computer for some reason resets or fails, the drone will stay on a mode that needs accurate GPS, wihtout accurate GPS, and that can lead to un-predictable behavior.
    - This adds more **critical** and **realtime** responsability on the mission computer.
  - As described in the discussion at the RC failure chapter, this option may complicate the RTL - RC failure behavior implementation. For implementing this behavior, we need to change modes to `RTL` or `ALT_HOLD` manualy, and as long as the drone is in `RTL` mode we need to check that the it did not accidently change its mode to `LAND`, and if so, change it back to `ALT_HOLD`.
    - As with the previous drawback, this adds more **critical** and **realtime** responsability on the mission computer.

6.3. Sending Fake RC Overrides

- Pros
  - Simple - sending the same RC override MAVLink messages will probably behave the same, the change is minor from the ArduPilot perspective. In fact, after chaning the main GCS *system ID*, from the ArduPilot point of view - this will be exactly the same.
- Cons
  - The ArduPilot will accept only overrides from its main GCS *system ID* (see RC Override Code Research), so changing the ArduPilot main GCS *system ID* is required. If this will be done, and not rolled back, the GCS may loose control.
    - This is not destructive because the GCS itself can change this paramter too and by that gain controll, but the dealing with this parameter may let a race condition be present - were the mission computer tries to set this parameter back to itself to gain control.
    - Moreover, the DR will use overrides so the usage of this mechanism will anyway be used. The main difference is that here it is going to be used more.
  - Every override has expiring time (default is 1 second). If the mission computer will fail or restart, the ArduPilot will enter the RC failsafe.
    - For solving this the override timeout can be configured to be long (for example, longer then the mission computer restart time) but this can lead to **unwanted behavior** (see Changing RC failsafe timeout).

- Either way, if the mission comupter will restart or fail the outcome is not satisfactory. Because of that, this option adds more **critical** and **realtime** responsability on the mission computer.

6.4. Changing RC Failsafe Timeout

- Pros
  - Will disable any RC failsafe that is coused because of RC disconnection.
- Cons
  - This can lead to unwanted behavior in case of a mission computer restart (when RC failure is present). After a movement RC command - the command will stay active untill the mission computer wakes up.
  - Includes recompiling the ArduPilot with different paramteres, in case of every ArduPilot version update, **a new code research needs to be done**.
  - This solution is tightly tied to the internal implementation of the RC failsafe mechanism, that in any time may change.

6.5. Changing RC Failsafe Action To Disabled

- Pros
  - Documented in the official docs that this is the way to disable RC failsafes.
  - Preseves the `RTL` failsafe in case of RC failure with healthy EKF (using the GCS failsafe mechanism).
- Cons
  - It is not documented what will be the behavior in case of RC and EKF failure at the **exact** same time, see (GCS failsafe code research). From the research done for this article we can see that there is an edge case in which the drone will eventually enter mode `LAND`.
    - We should handle this and change mode manually to `ALT_HOLD` if needed. This requiers the mission computer to be a *realtime* component, but in a **very** rare scenario and for a **very** short time.

7. Mixing it All, Recommendations

7.1. GCS

The GCS failsafe is configurable and handy, so there are no real dilemas regarding the GCS failsafe system. The real dilemas involving that module, are related to the RC failure topic.

7.2. EKF

Changing the EKF threshold seems much more vulnrable and less stable or predictable, while the option of changing the EKF failsafe action is a builtin feature that is relativly simple (conceptually) and straight forward.

Because of that, my recommendation is to **use the `FS_EKF_ACTION` parameter to disable auto-landing at EKF failure**.

7.3. RC

The option of sending fake RC overrides is risky because in any of the following 2 options, the defenition of the mission computer will change to be more *realtime* (which is something that want to avoid):

- **Setting the override timeout to be long** - It is possible that the drone will receive RC commands without monitoring of any kind (In case that the mission computer failes or restarts)
- **Else** - The auto-land will not be disabled untill an active action from the mission computer.

Changing RC failsafe timeout is better in the context of auto-landing disabling because it does not allow RC failsafe in any price (even in case of a total failure of the mission comupter), but still for the same reason (the first point) - it makes the mission computer role become *critical* and *realtime*. But is also worse in the context of the way we upgrade the ArduPilot version because the research that needs to be done every time and because it is not guaranteed to be a relevant solution in future ArduPilot releases.

What we get with the RC failsafe action change to disabled option is the same outcome of full RC failsafe disable but with auto `RTL` binded in the ArduPilot (i.e. RTL can be performed even if the mission computer is dead), and **much** less probability of loosing control for a long time. This solution in contrast to the previous one, is much more likely to be relevant in future ArduPilot releases.

Because of that, my recommendation is to **use `FS_THR_ENABLE` to disable RC failsafes, `FS_GCS_ENABLE` to set GCS failsafe to change mode to `RTL` and `FS_OPTIONS` to enable any GCS failsafe only in non-pilot controled modes**.

In all cases if DR will start, the mission computer will send RC overrides. So all that is said here in context of the mission computer becoming a realtime component refers only for if DR is not active. The behavior in case of a mission computer failure at the middle of a DR, seems to be to continue with the same RC commands (see RC failsafe trigger code research).

7.4. Battery

If wanted, it is possible to disable any battery failsafe (see Battery Failure).

8. Code Research

8.1. GCS Failsafe Flow Code Research

8.1.1. Code Analysis

- First of all, we assume that the failsafe_gcs_on_event function is called in case of a gcs failsafe.
  - This can be assumed because there is a loop that runs at 3.3 hz and calles an GCS check. This check will evantually check when was the last gcs update (see GCS Failsafe Trigger code research) and if the time is less then the GCS failsafe timeout configured using the `FS_GCS_TIMEOUT` (see GCS timout parameter) the function will call the failsafe_gcs_on_event
- First of all, we can see that it disables all the active overrides. This means that any RC override command that should have been active because of the `RC_OVERRIDE_TIME`, will not be active after a GCS failsafe.
- From there, lets assume we configured that GCS failure will trigger `RTL` (and `FS_OPTIONS` configured to disable gcs failsafe in pilot-controlled modes) and that the EKF failure will trigger mode change to `ALT_HOLD` (see changing EKF failsafe action and GCS failure).

- If we are not in a pilot-conrolled mode (such as `GUIDED`), the next thing to happen is a call to do_failsafe_action with `Failsafe_Action_RTL`.
    - The do_failsafe_action function will call to set_mode_RTL_or_land_with_pause that will try to set the mode to `RTL`, and if the mode change failes for any reason, the mode will change to `LAND`.
    - **Important:** If an GPS failure accures just before the check that the flight mode is pilot-controlled, the EKF variance may not be high enough to trigger the EKF failsafe and because of that - the flight mode may not be changed quick enough to `ALT_HOLD` so the code will attempt to change mode to `RTL`, fail because there is no valid position, and then change mode to `LAND`.
        - This is an hypothesis, for being sure, a test or a more deep research is needes.
- If we are in a pilot-controlled mode, no failsafe action will be triggerd.

8.1.2. Conclusions

Assuming that if we have GPS, we are in an auto-pilot (`GUIDED`, `AUTO`, etc..) mode and these configurations:

- `FS_GCS_ENABLE`: **`RTL`** (=1)
- `FS_OPTIONS`: **Continue if in pilot controlled modes on GCS failsafe** (=16)
- `FS_EKF_ACTION`: **`ALT_HOLD`** (=2)

Then:

- If we have valid EKF and valid GPS - the autopilot will change mode to `RTL`.
- (*hypothysis*) If we have valid EKF and invalid GPS - the autopilot will change mode to `LAND` (see the "Important" point above).
- If we do not have valid EKF (the EKF failsafe should have changed the mode to `ALT_HOLD` when the EKF variance was to high) - nothing will happen.

It is important to pay attention that if a GCS failsafe kicks in, any active RC override will be disabled, even if `RC_OVERRIDE_TIME` is -1 (=forever (well, I guess almost)).

8.2. GCS Failsafe Trigger Code Research

8.2.1. Code Analysis

- The ArduPilot checks for GCS failure with the function failsafe_gcs_check.
- In it, it is triggering a failsafe if the GCS last seen time is bigger then the `FS_GCS_TIMEOUT` parameter.
- The last seen time is calculated using the function sysid_myggcs_last_seen_time_ms, which just returns the `_sysid_mygcs_last_seen_time_ms` variable.
- The `_sysid_mygcs_last_seen_time_ms` variable is being set only in the sysid_myggcs_seen function that is being called in the following functions (In all of those functions, the last seen is being updated only if the sender *system ID* is the same as the configured `SYSID_MYGCS`):
    - The function HandleMessage that is being called when receiving `MANUAL_CONTROL (#69)`.
    - The function handle_rc_channels_override] that is being called when receiving `RC_CHANNELS_OVERRIDE (#70)`.
    - The function GCS_MAVLINK::handle_heartbeat that is being called for every heartbeat message.

8.2.2. Conclusions

A GCS failsafe will be triggerd after `FS_GCS_TIMEOUT` that no heartbeat, or RC override (of any kind) were received from the configured `SYSID_MYGCS`.

8.3. RC Override Code Research

8.3.1. Code Analysis

There are 2 ways to send RC commands via mavlink:

1. `MANUAL_CONTROL (#69)`
2. `RC_CHANNELS_OVERRIDE (#70)`

- Every MAVLink message is being handled by GCS_MAVLINK_Copter::handleMessage.
    - In it, if the message is `MANUAL_CONTROL (#69)`, then **only accept it if the *system ID* of the sender is the same as the configured `SYSID_MYGCS`**.
    - If the message is `RC_CHANNELS_OVERRIDE (#70)`, then send it to handle_common_message.
        - In it, if the message is `RC_CHANNELS_OVERRIDE (#70)`, then send it to handle_rc_channels_override.
            - In it, **return imidiatly if the *system ID* of the sender is not the same as the configured `SYSID_MYGCS`**.

8.3.2. Conclusions

It does not matter in what way an RC override command was sent, either way it will be accepted only if the sender *system ID* is the same as the configured GCS (`SYSID_MYGCS` parameter).

8.4. RC Failsafe Trigger Code Research

8.4.1. Code Analysis

- The function that we are intrested in is Copter::failsafe_radio_on_event. This is the function that triggers the RC failsafe. It is being called only by the function Copter::set_failsafe_radio
- All that the function Copter::set_failsafe_radio does is to call the above function if needed. It is being called by one of:
    - Copter::read_radio
    - Copter::set_throttle_and_failsafe - This function is being called only by Copter::read_radio.
- All that the function Copter::set_throttle_and_failsafe does is to **set the radio failsafe on and off** (with the `set_failsafe_radio` function) in case of 3 (nearly) consecutive valid or invalid throttle values. An invalid throttle value is defined to be lower then the parameter `FS_THR_ENABLE`.
- The function Copter::read_radio has multiple stages:
    - At the first stage it tries to get new input with the function RC_Channels::read_input.
        - This function checks if there are new "regular" RC commands or new RC overrides. If not - it returns `False`.
            - If there are new "regular" RC commands, set `_has_had_rc_receiver` to be `True`.
        - Every call of this function will end with the `has_new_overrides` variable set to `False`.

            - The only place that this variable is set back to `True` is in the function RC_Channel::set_override that is being called only at:
                - GCS_MAVLINK::handle_rc_channels_override - that is being called for every `RC_CHANNELS_OVERRIDE (#70)` message received (and only that).
                - GCS_MAVLINK::manual_override - that is being called for every `MANUAL_CONTROL (#69)` message received (and only that).
        - This means that the function RC_Channels::read_input will return `False` if no new overrides were sent (even if the override timeout is still not done).
    - If the function succeeds (There is a new RC input) - It calls the function Copter::set_throttle_and_failsafe (that may trigger RC failsafe, as described above), sets the new RC commands and sets `last_radio_update_ms` to be the time now.
    - If it fails - Then if the time elapsed from the last RC command is bigger then the timeout and the RC failsafes are not disabled via the `FS_THR_ENABLE` parameter.
    - The timeout described above can be either `FS_RADIO_RC_OVERRIDE_TIMEOUT_MS` (=1s) or `FS_RADIO_TIMEOUT_MS` (=0.5s)
        - If the function RC_Channels::has_active_overrides returns True, use `FS_RADIO_RC_OVERRIDE_TIMEOUT_MS`.
            - This function returns True if any of the channels has an override. The meaning of "has an override" is that (see RC_Channel::has_override):
                - The channel has an override value (sent via one of the RC override MAVLink messages with RC_Channel::set_override).
                - RC overrides timeout is not disabled (`RC_OVERRIDE_TIME` not set to 0).
                - The override timeout is not over.
        - Else, use `FS_RADIO_TIMEOUT_MS`.

8.4.2. Conclusions

In case of not disabling RC failsafes via `FS_THR_ENABLE`:

- If there is an active override (has a value and not timed out yet) - Radio failsafe will be triggerd if **no overrides were sent for more then `FS_RADIO_RC_OVERRIDE_TIMEOUT_MS` (=1s)**
- If there is no active override - Radio failsafe will be triggerd if **no "regular" RC commands were sent for more then `FS_RADIO_TIMEOUT_MS` (=0.5s)**

If you do disable RC failsafes using `FS_THR_ENABLE`, none of the above will ever happen, and no RC failsafe will ever be triggerd.

Common sense says that the last RC command will be the one active until a new one will apear, because if not - what will be the active RC command? the only other logical option is a "neutral" option, but this does not make a lot sense because what will happen if one RC command will fail to be send? The drone whould shake in that cirumstance, and lets say we are in `STABILIZE` mode, what is that "neutral" option? The drone does not hold its atlitude so there is no "nuetral" option - very unlikely.

9. Appendices

9.1. The Function `GCS-MAVLINK-Copter::handleMessage`

```
void GCS_MAVLINK_Copter::handleMessage(const mavlink_message_t &msg)
{
    ...

    switch (msg.msgid) {

    case MAVLINK_MSG_ID_MANUAL_CONTROL:
    {
        if (msg.sysid != copter.g.sysid_my_gcs) {
            break; // only accept control from our gcs
        }

        ...


        // a manual control message is considered to be a 'heartbeat'
        // from the ground station for failsafe purposes
        gcs().sysid_myggcs_seen(tnow);
        break;
    }

    ...

    default:
        handle_common_message(msg);
        break;
    }     // end switch
} // end handle mavlink
```

9.2. The Function `GCS-MAVLINK::handle-common-message`

```
/*
  handle messages which don't require vehicle specific data
 */
void GCS_MAVLINK::handle_common_message(const mavlink_message_t &msg)
{
    switch (msg.msgid) {

    case MAVLINK_MSG_ID_HEARTBEAT: {
        handle_heartbeat(msg);
        break;
    }

    ...

    case MAVLINK_MSG_ID_RC_CHANNELS_OVERRIDE:
        handle_rc_channels_override(msg);
        break;

    ...
```

```
    }

}
```

9.3. The Function `GCS-MAVLINK::handle-rc-channels-override`

```
// allow override of RC channel values for complete GCS
// control of switch position and RC PWM values.
void GCS_MAVLINK::handle_rc_channels_override(const mavlink_message_t &msg)
{
    if(msg.sysid != sysid_my_gcs()) {
        return; // Only accept control from our gcs
    }

    const uint32_t tnow = AP_HAL::millis();

    mavlink_rc_channels_override_t packet;
    mavlink_msg_rc_channels_override_decode(&msg, &packet);

    const uint16_t override_data[] = {
        packet.chan1_raw,
        ...
        packet.chan16_raw
    };

    for (uint8_t i=0; i<8; i++) {
        // Per MAVLink spec a value of UINT16_MAX means to ignore this
field.
        if (override_data[i] != UINT16_MAX) {
            RC_Channels::set_override(i, override_data[i], tnow);
        }
    }
    for (uint8_t i=8; i<ARRAY_SIZE(override_data); i++) {
        // Per MAVLink spec a value of zero or UINT16_MAX means to
        // ignore this field.
        if (override_data[i] != 0 && override_data[i] != UINT16_MAX) {
            // per the mavlink spec, a value of UINT16_MAX-1 means
            // return the field to RC radio values:
            const uint16_t value = override_data[i] == (UINT16_MAX-1) ? 0 :
override_data[i];
            RC_Channels::set_override(i, value, tnow);
        }
    }

    gcs().sysid_myggcs_seen(tnow);

}
```

9.4. The Function `Copter::three-hz-loop`

```
// three_hz_loop - 3.3hz loop
void Copter::three_hz_loop()
{
    // check if we've lost contact with the ground station
    failsafe_gcs_check();

    // check if we've lost terrain data
    failsafe_terrain_check();

#if AC_FENCE == ENABLED
    // check if we have breached a fence
    fence_check();
#endif // AC_FENCE_ENABLED


    // update ch6 in flight tuning
    tuning();

    // check if avoidance should be enabled based on alt
    low_alt_avoidance();
}
```

9.5. The Function `Copter::failsafe-gcs-on-event`

```
// failsafe_gcs_on_event - actions to take when GCS contact is lost
void Copter::failsafe_gcs_on_event(void)
{
    AP::logger().Write_Error(LogErrorSubsystem::FAILSAFE_GCS,
LogErrorCode::FAILSAFE_OCCURRED);
    RC_Channels::clear_overrides();

    // convert the desired failsafe response to the Failsafe_Action enum
    Failsafe_Action desired_action;
    switch (g.failsafe_gcs) {
        case FS_GCS_DISABLED:
            desired_action = Failsafe_Action_None;
            break;
        case FS_GCS_ENABLED_ALWAYS_RTL:
        case FS_GCS_ENABLED_CONTINUE_MISSION:
            desired_action = Failsafe_Action_RTL;
            break;
        case FS_GCS_ENABLED_ALWAYS_SMARTRTL_OR_RTL:
            desired_action = Failsafe_Action_SmartRTL;
            break;
        case FS_GCS_ENABLED_ALWAYS_SMARTRTL_OR_LAND:
            desired_action = Failsafe_Action_SmartRTL_Land;
            break;
        case FS_GCS_ENABLED_ALWAYS_LAND:
            desired_action = Failsafe_Action_Land;
            break;
```

```
        default: // if an invalid parameter value is set, the fallback is
RTL
            desired_action = Failsafe_Action_RTL;
    }

    // Conditions to deviate from FS_GCS_ENABLE parameter setting

    ...

    } else if
(failsafe_option(FailsafeOption::GCS_CONTINUE_IF_PILOT_CONTROL) &&
!flightmode->is_autopilot()) {
        // should continue when in a pilot controlled mode because
FS_OPTIONS is set to continue in pilot controlled modes
        gcs().send_text(MAV_SEVERITY_WARNING, "GCS Failsafe - Continuing
Pilot Control");
        desired_action = Failsafe_Action_None;
    }

    ...

    // Call the failsafe action handler
    do_failsafe_action(desired_action, ModeReason::GCS_FAILSAFE);
}
```

9.6. The Function `Copter::do-failsafe-action`

```
void Copter::do_failsafe_action(Failsafe_Action action, ModeReason reason){

    // Execute the specified desired_action
    switch (action) {
        case Failsafe_Action_None:
            return;
        case Failsafe_Action_Land:
            set_mode_land_with_pause(reason);
            break;
        case Failsafe_Action_RTL:
            set_mode_RTL_or_land_with_pause(reason);
            break;
        case Failsafe_Action_SmartRTL:
            set_mode_SmartRTL_or_RTL(reason);
            break;
        case Failsafe_Action_SmartRTL_Land:
            set_mode_SmartRTL_or_land_with_pause(reason);
            break;
        case Failsafe_Action_Terminate: {
#if ADVANCED_FAILSAFE == ENABLED
            g2.afs.gcs_terminate(true, "Failsafe");
#else
            arming.disarm(AP_Arming::Method::FAILSAFE_ACTION_TERMINATE);
#endif
```

```
        }
        break;
    }

#if GRIPPER_ENABLED == ENABLED
    if (failsafe_option(FailsafeOption::RELEASE_GRIPPER)) {
        copter.g2.gripper.release();
    }
#endif
}
```

9.7. The Function `Copter::set-mode-RTL-or-land-with-pause`

```
// set_mode_RTL_or_land_with_pause - sets mode to RTL if possible or LAND
with 4 second delay before descent starts
//  this is always called from a failsafe so we trigger notification to
pilot
void Copter::set_mode_RTL_or_land_with_pause(ModeReason reason)
{
    // attempt to switch to RTL, if this fails then switch to Land
    if (!set_mode(Mode::Number::RTL, reason)) {
        // set mode to land will trigger mode change notification to pilot
        set_mode_land_with_pause(reason);
    } else {
        // alert pilot to mode change
        AP_Notify::events.failsafe_mode_change = 1;
    }
}
```

9.8. The Function `Copter::set-mode-land-with-pause`

```
// set_mode_land_with_pause - sets mode to LAND and triggers 4 second delay
before descent starts
//  this is always called from a failsafe so we trigger notification to
pilot
void Copter::set_mode_land_with_pause(ModeReason reason)
{
    set_mode(Mode::Number::LAND, reason);
    mode_land.set_land_pause(true);

    // alert pilot to mode change
    AP_Notify::events.failsafe_mode_change = 1;
}
```

9.9. GCS Timout Parameter

```
    // @Param: FS_GCS_TIMEOUT
    // @DisplayName: GCS failsafe timeout
    // @Description: Timeout before triggering the GCS failsafe
    // @Units: s
    // @Range: 2 120
    // @Increment: 1
    // @User: Standard
    AP_GROUPINFO("FS_GCS_TIMEOUT", 42, ParametersG2, fs_gcs_timeout, 5)
```

9.10. The Function `NavEKF2-core::healthy`

```
// Check basic filter health metrics and return a consolidated health
status
bool NavEKF2_core::healthy(void) const
{
    uint16_t faultInt;
    getFilterFaults(faultInt);
    if (faultInt > 0) {
        return false;
    }
    if (velTestRatio > 1 && posTestRatio > 1 && hgtTestRatio > 1) {
        // all three metrics being above 1 means the filter is
        // extremely unhealthy.
        return false;
    }
    // Give the filter a second to settle before use
    if ((imuSampleTime_ms - ekfStartTime_ms) < 1000 ) {
        return false;
    }
    // position and height innovations must be within limits when on-ground
and in a static mode of operation
    ftype horizErrSq = sq(innovVelPos[3]) + sq(innovVelPos[4]);
    if (onGround && (PV_AidingMode == AID_NONE) && ((horizErrSq > 1.0f) ||
(fabsF(hgtInnovFiltState) > 1.0f))) {
        return false;
    }

    // all OK
    return true;
}
```

9.11. The Function `Copter::failsafe-gcs-check`

```
// failsafe_gcs_check - check for ground station failsafe
void Copter::failsafe_gcs_check()
{
    // Bypass GCS failsafe checks if disabled or GCS never connected
    if (g.failsafe_gcs == FS_GCS_DISABLED) {
```

```
        return;
    }

    const uint32_t gcs_last_seen_ms =
gcs().sysid_myggcs_last_seen_time_ms();
    if (gcs_last_seen_ms == 0) {
        return;
    }

    // calc time since last gcs update
    // note: this only looks at the heartbeat from the device id set by
g.sysid_my_gcs
    const uint32_t last_gcs_update_ms = millis() - gcs_last_seen_ms;
    const uint32_t gcs_timeout_ms =
uint32_t(constrain_float(g2.fs_gcs_timeout * 1000.0f, 0.0f, UINT32_MAX));

    // Determine which event to trigger
    if (last_gcs_update_ms < gcs_timeout_ms && failsafe.gcs) {
        // Recovery from a GCS failsafe
        set_failsafe_gcs(false);
        failsafe_gcs_off_event();

    } else if (last_gcs_update_ms < gcs_timeout_ms && !failsafe.gcs) {
        // No problem, do nothing

    } else if (last_gcs_update_ms > gcs_timeout_ms && failsafe.gcs) {
        // Already in failsafe, do nothing

    } else if (last_gcs_update_ms > gcs_timeout_ms && !failsafe.gcs) {
        // New GCS failsafe event, trigger events
        set_failsafe_gcs(true);
        failsafe_gcs_on_event();
    }
}
```

9.12. The Function `sysid-myggcs-last-seen-time-ms`

```
class GCS
{
    ...

    // last time traffic was seen from my designated GCS.  traffic
    // includes heartbeats and some manual control messages.
    uint32_t sysid_myggcs_last_seen_time_ms() const {
        return _sysid_mygcs_last_seen_time_ms;
    }

    ...
}
```

9.13. The Function sysid-myggcs-seen

```
class GCS
{
    ...

    // called when valid traffic has been seen from our GCS
    void sysid_myggcs_seen(uint32_t seen_time_ms) {
        _sysid_mygcs_last_seen_time_ms = seen_time_ms;
    }

    ...
}
```

9.14. The Function GCS-MAVLINK::handle-heartbeat

```
void GCS_MAVLINK::handle_heartbeat(const mavlink_message_t &msg) const
{
    // if the heartbeat is from our GCS then we don't failsafe for
    // now...
    if (msg.sysid == sysid_my_gcs()) {
        gcs().sysid_myggcs_seen(AP_HAL::millis());
    }
}
```

9.15. The Function AP-Arming-Copter::mandatory-gps-checks

```
// performs mandatory gps checks.  returns true if passed
bool AP_Arming_Copter::mandatory_gps_checks(bool display_failure)
{
    ...

    if (mode_requires_gps) {
            if (!copter.position_ok()) {
                // vehicle level position estimate checks
                check_failed(display_failure, "Need Position Estimate");
                return false;
            }
        }

    ...

    // check EKF's compass, position and velocity variances are below
failsafe threshold
    if (copter.g.fs_ekf_thresh > 0.0f) {
        float vel_variance, pos_variance, hgt_variance, tas_variance;
        Vector3f mag_variance;
```

```
        ahrs.get_variances(vel_variance, pos_variance, hgt_variance,
mag_variance, tas_variance);
        if (mag_variance.length() >= copter.g.fs_ekf_thresh) {
            check_failed(display_failure, "EKF compass variance");
            return false;
        }
        if (pos_variance >= copter.g.fs_ekf_thresh) {
            check_failed(display_failure, "EKF position variance");
            return false;
        }
        if (vel_variance >= copter.g.fs_ekf_thresh) {
            check_failed(display_failure, "EKF velocity variance");
            return false;
        }
    }

    ...
}
```

9.16. The Function `Copter::ekf-check`

```
// ekf_check - detects if ekf variance are out of tolerance and triggers
failsafe
// should be called at 10hz
void Copter::ekf_check()
{
    ...

    // return immediately if ekf check is disabled
    if (g.fs_ekf_thresh <= 0.0f) {
        ekf_check_state.fail_count = 0;
        ekf_check_state.bad_variance = false;
        AP_Notify::flags.ekf_bad = ekf_check_state.bad_variance;
        failsafe_ekf_off_event();   // clear failsafe
        return;
    }

    ...

    // take action based on fs_ekf_action parameter
    switch (g.fs_ekf_action) {
        case FS_EKF_ACTION_ALTHOLD:
            // AltHold
            if (failsafe.radio || !set_mode(Mode::Number::ALT_HOLD,
ModeReason::EKF_FAILSAFE)) {
                set_mode_land_with_pause(ModeReason::EKF_FAILSAFE);
            }
            break;
        case FS_EKF_ACTION_LAND:
        case FS_EKF_ACTION_LAND_EVEN_STABILIZE:
        default:
```

```
            set_mode_land_with_pause(ModeReason::EKF_FAILSAFE);
            break;
    }

    ...
}
```

9.17. The Function `Copter::ekf-over-threshold`

```
// ekf_over_threshold - returns true if the ekf's variance are over the
tolerance
bool Copter::ekf_over_threshold()
{
    // return false immediately if disabled
    if (g.fs_ekf_thresh <= 0.0f) {
        return false;
    }

    ...
}
```

9.18. The Function `Copter::read-radio`

```
void Copter::read_radio()
{
    const uint32_t tnow_ms = millis();

    if (rc().read_input()) {
        ap.new_radio_frame = true;

        set_throttle_and_failsafe(channel_throttle->get_radio_in());
        set_throttle_zero_flag(channel_throttle->get_control_in());

        // RC receiver must be attached if we've just got input
        ap.rc_receiver_present = true;

        // pass pilot input through to motors (used to allow wiggling
servos while disarmed on heli, single, coax copters)
        radio_passthrough_to_motors();

        const float dt = (tnow_ms - last_radio_update_ms)*1.0e-3f;
        rc_throttle_control_in_filter.apply(channel_throttle-
>get_control_in(), dt);
        last_radio_update_ms = tnow_ms;
        return;
    }

    // No radio input this time
    if (failsafe.radio) {
```

```
        // already in failsafe!
        return;
    }

    const uint32_t elapsed = tnow_ms - last_radio_update_ms;
    // turn on throttle failsafe if no update from the RC Radio for 500ms
or 2000ms if we are using RC_OVERRIDE
    const uint32_t timeout = RC_Channels::has_active_overrides() ?
FS_RADIO_RC_OVERRIDE_TIMEOUT_MS : FS_RADIO_TIMEOUT_MS;
    if (elapsed < timeout) {
        // not timed out yet
        return;
    }
    if (!g.failsafe_throttle) {
        // throttle failsafe not enabled
        return;
    }
    if (!ap.rc_receiver_present && !motors->armed()) {
        // we only failsafe if we are armed OR we have ever seen an RC
receiver
        return;
    }

    // Nobody ever talks to us.  Log an error and enter failsafe.
    AP::logger().Write_Error(LogErrorSubsystem::RADIO,
LogErrorCode::RADIO_LATE_FRAME);
    set_failsafe_radio(true);
}
```

9.19. The Function `Copter::failsafe-radio-on-event`

```
void Copter::failsafe_radio_on_event()
{
    ...

    // set desired action based on FS_THR_ENABLE parameter
    Failsafe_Action desired_action;
    switch (g.failsafe_throttle) {
        case FS_THR_DISABLED:
            desired_action = Failsafe_Action_None;
            break;
        case FS_THR_ENABLED_ALWAYS_RTL:
        case FS_THR_ENABLED_CONTINUE_MISSION:
            desired_action = Failsafe_Action_RTL;
            break;
        case FS_THR_ENABLED_ALWAYS_SMARTRTL_OR_RTL:
            desired_action = Failsafe_Action_SmartRTL;
            break;
        case FS_THR_ENABLED_ALWAYS_SMARTRTL_OR_LAND:
            desired_action = Failsafe_Action_SmartRTL_Land;
            break;
```

```
        case FS_THR_ENABLED_ALWAYS_LAND:
            desired_action = Failsafe_Action_Land;
            break;
        default:
            desired_action = Failsafe_Action_Land;
    }

    // Conditions to deviate from FS_THR_ENABLE selection and send specific
GCS warning

    ...

    // Call the failsafe action handler
    do_failsafe_action(desired_action, ModeReason::RADIO_FAILSAFE);
}
```

9.20. The Function `Copter::set-failsafe-radio`

```
void Copter::set_failsafe_radio(bool b)
{
    // only act on changes
    // -------------------
    if(failsafe.radio != b) {

        // store the value so we don't trip the gate twice
        // -----------------------------------------------
        failsafe.radio = b;

        if (failsafe.radio == false) {
            // We've regained radio contact
            // ----------------------------
            failsafe_radio_off_event();
        }else{
            // We've lost radio contact
            // ------------------------
            failsafe_radio_on_event();
        }

        // update AP_Notify
        AP_Notify::flags.failsafe_radio = b;
    }
}
```

9.21. The Function `Copter::set-throttle-and-failsafe`

```
#define FS_COUNTER 3        // radio failsafe kicks in after 3 consecutive
throttle values below failsafe_throttle_value
void Copter::set_throttle_and_failsafe(uint16_t throttle_pwm)
{
```

```
    // if failsafe not enabled pass through throttle and exit
    if(g.failsafe_throttle == FS_THR_DISABLED) {
        return;
    }

    //check for low throttle value
    if (throttle_pwm < (uint16_t)g.failsafe_throttle_value) {

        // if we are already in failsafe or motors not armed pass through throttle and exit
        if (failsafe.radio || !(ap.rc_receiver_present || motors->armed())) {
            return;
        }

        // check for 3 low throttle values
        // Note: we do not pass through the low throttle until 3 low throttle values are received
        failsafe.radio_counter++;
        if( failsafe.radio_counter >= FS_COUNTER ) {
            failsafe.radio_counter = FS_COUNTER;  // check to ensure we don't overflow the counter
            set_failsafe_radio(true);
        }
    }else{
        // we have a good throttle so reduce failsafe counter
        failsafe.radio_counter--;
        if( failsafe.radio_counter <= 0 ) {
            failsafe.radio_counter = 0;   // check to ensure we don't underflow the counter

            // disengage failsafe after three (nearly) consecutive valid throttle values
            if (failsafe.radio) {
                set_failsafe_radio(false);
            }
        }
        // pass through throttle
    }
}
```

9.22. The Function `RC-Channels::read-input`

```
// update all the input channels
bool RC_Channels::read_input(void)
{
    if (hal.rcin->new_input()) {
        _has_had_rc_receiver = true;
    } else if (!has_new_overrides) {
        return false;
    }
```

```
    has_new_overrides = false;

    last_update_ms = AP_HAL::millis();

    bool success = false;
    for (uint8_t i=0; i<NUM_RC_CHANNELS; i++) {
        success |= channel(i)->update();
    }

    return success;
}
```

9.23. The Function `RC-Channel::set-override`

```
void RC_Channel::set_override(const uint16_t v, const uint32_t
timestamp_ms)
{
    if (!rc().gcs_overrides_enabled()) {
        return;
    }

    last_override_time = timestamp_ms != 0 ? timestamp_ms :
AP_HAL::millis();
    override_value = v;
    rc().new_override_received();
}
```

9.24. The Funciton `GCS-MAVLINK::manual-override`

```
void GCS_MAVLINK::manual_override(RC_Channel *c, int16_t value_in, const
uint16_t offset, const float scaler, const uint32_t tnow, const bool
reversed)
{
    if (c == nullptr) {
        return;
    }
    int16_t override_value = 0;
    if (value_in != INT16_MAX) {
        const int16_t radio_min = c->get_radio_min();
        const int16_t radio_max = c->get_radio_max();
        if (reversed) {
            value_in *= -1;
        }
        override_value = radio_min + (radio_max - radio_min) * (value_in +
offset) / scaler;
    }
    c->set_override(override_value, tnow);
}
```

9.25. The Function `RC-Channels::has-active-overrides`

```
bool RC_Channels::has_active_overrides()
{
    RC_Channels &_rc = rc();
    for (uint8_t i = 0; i < NUM_RC_CHANNELS; i++) {
        if (_rc.channel(i)->has_override()) {
            return true;
        }
    }

    return false;
}
```

9.26. The Function `RC-Channel::has-override`

```
bool RC_Channel::has_override() const
{
    if (override_value == 0) {
        return false;
    }

    uint32_t override_timeout_ms;
    if (!rc().get_override_timeout_ms(override_timeout_ms)) {
        // timeouts are disabled
        return true;
    }

    if (override_timeout_ms == 0) {
        // overrides are explicitly disabled by a zero value
        return false;
    }

    return (AP_HAL::millis() - last_override_time < override_timeout_ms);
}
```