LUCAS Team Corporation
Kevin Delmas - kdelmas31140@gmail.com
Benoît Gayraud - gayraudbenoit@gmail.com
Susanna Polino - susanna.polino@gmail.com
Rodrigo Rivera - rodrigoariverac191@gmail.com
Yingqing Yu - shirley.lapin9032@gmail.com

USER GUIDE

In-flight diagnoser
for Parrot AR.drone

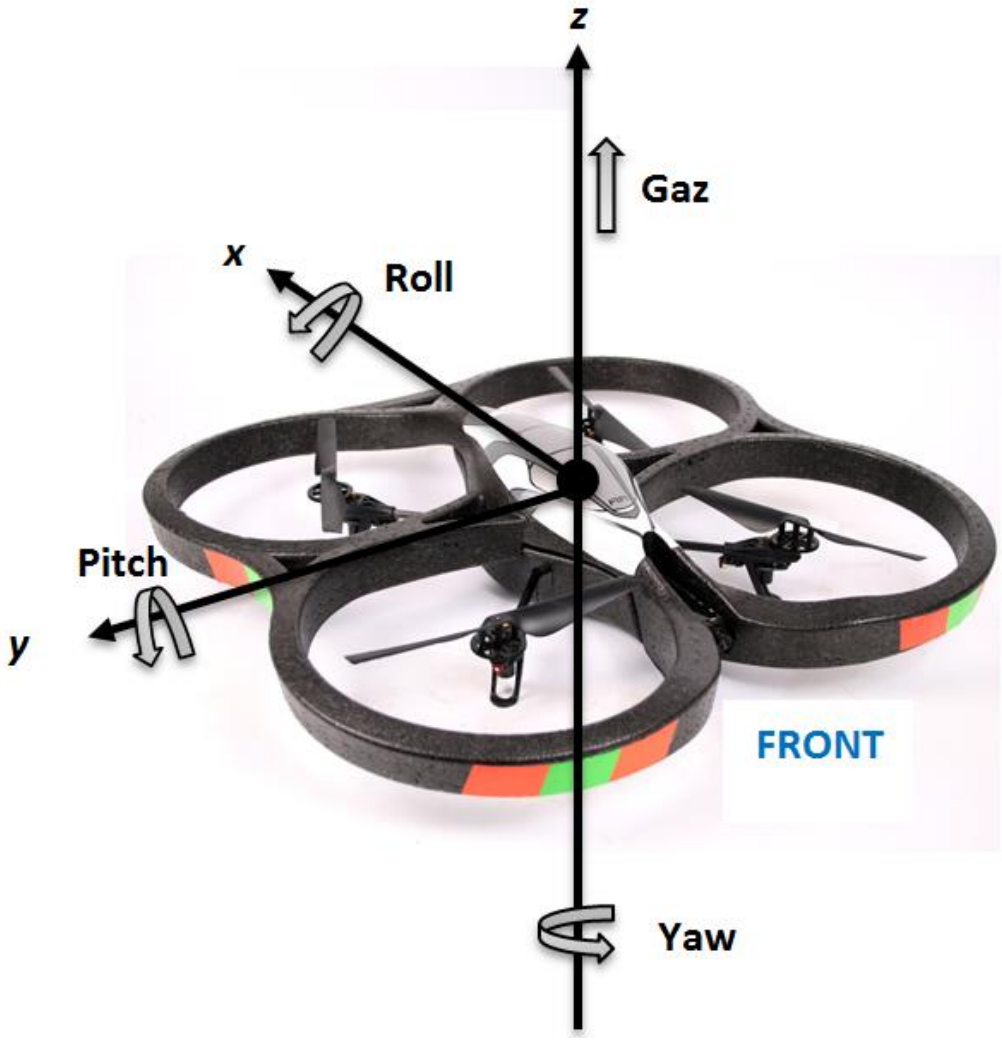
smartfox

Contents

1. Getting started

1.1. Terminology

Before doing anything, it is necessary to know briefly the right terminology concerning the drone motion to use Smartfox.



1.2. Set up

Please execute each following step in the same order.

- Platform :

The version supplied is only runnable on **Linux 32 bits platform** (it is not working on Mac OS). The navigation application is based on the keyboard. Its access depends on the OS and has been done for Linux.

If you have another operating system different from Linux, please use a virtual machine (set up detailed in the next part).

- Set up a virtual machine :

1. Download a Linux image (Ubuntu desktop advised: http://www.ubuntu.com/download/desktop). Please select the **32 bits** version.
2. Download a virtual machine like Virtual Box (www.virtualbox.org) or VMware (https://my.vmware.com only for Windows).
3. Set up Linux on the virtual machine.

- Download the sources files :

2 options:

- Clone the repository on your local computer : *git clone https://yourmail@code.google.com/p/drone-diagnostic-insa-toulouse-2014/*

- Download the zip file containing the source files on the Google drive directory or on the Smartfox website

- Compile the project :

1. Go to *smartfox/Build*
2. Type *make* in the terminal
3. The system will probably invite you to install other packages: you must accept everything. The build may take several minutes, it is normal.
4. When the build is completed the executable file is in *smartfox/* and is simply called *smartfox.*

- Configure the user application (keyboard) :

1. Open the file *UserApp/user_app.conf* where you can see the parameter keyboard_file. The path of the keyboard file depends on your computer (the path set by default is probably not the same on your computer).
2. Open the terminal and go to */dev/input/by-path*. You may have a file containing *keyboard* or *kbd* like in the default path in *user_app.conf (platform-i8042-serio-0-event-kbd)*.
3. Update the file name in *user_app.conf* with this one just found before

- Run smartfox

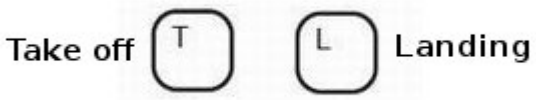
You need to be in super user to run the application.
Go to *smartfox/* and type ***sudo smartfox*** in the terminal.

2. Demonstration Application

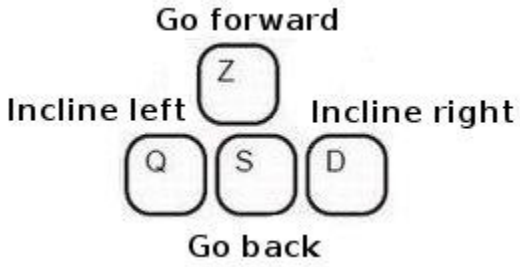
2.1. Keyboard controlling app

In order to test the diagnosis layer, a basic navigation application is supplied. It enables to control easily the drone with the keyboard.

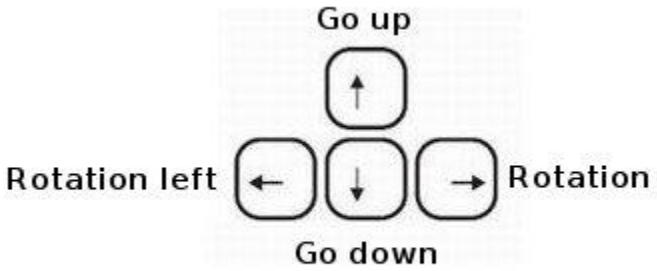
- **Take-off and landing :**



- **Pitch and roll angles :**



- **Gaz and yaw control :**



NOTE :

The application is configured to be used with a standard Linux (Debian). If you use another operating system, you must update the *UserApp/user_app.conf* file.

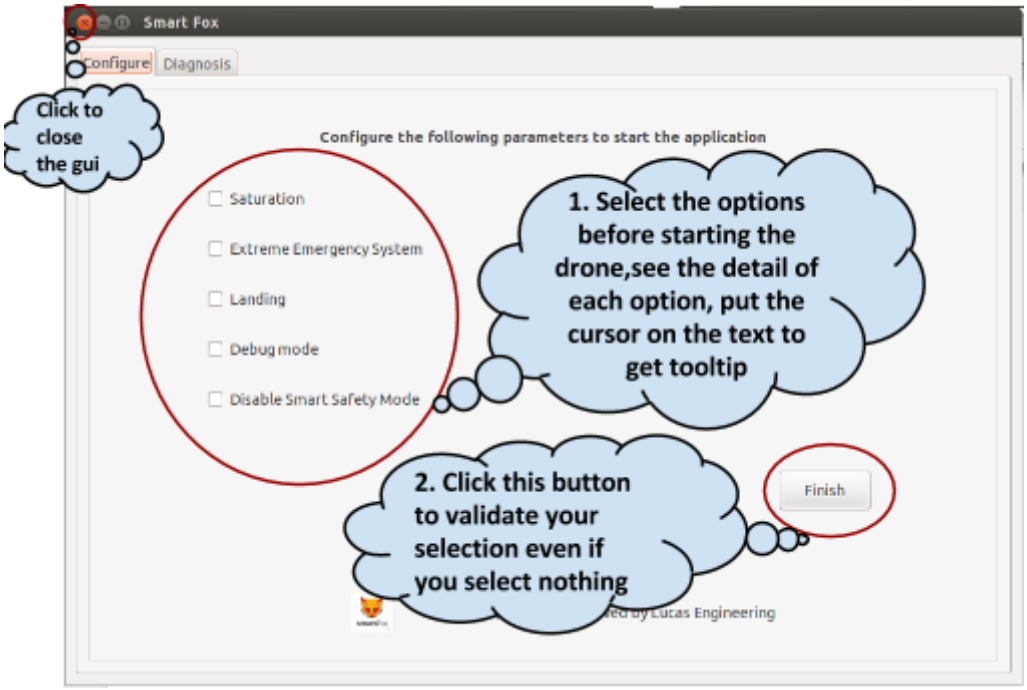
This file contains the path of the *keyboard file* on your machine.

2.2. GUI options and informations

We have developed a user interface to give users more details about the faults detected and the actions taken in the emergency situation. All the files concerning to the GUI is located in the UI directory.

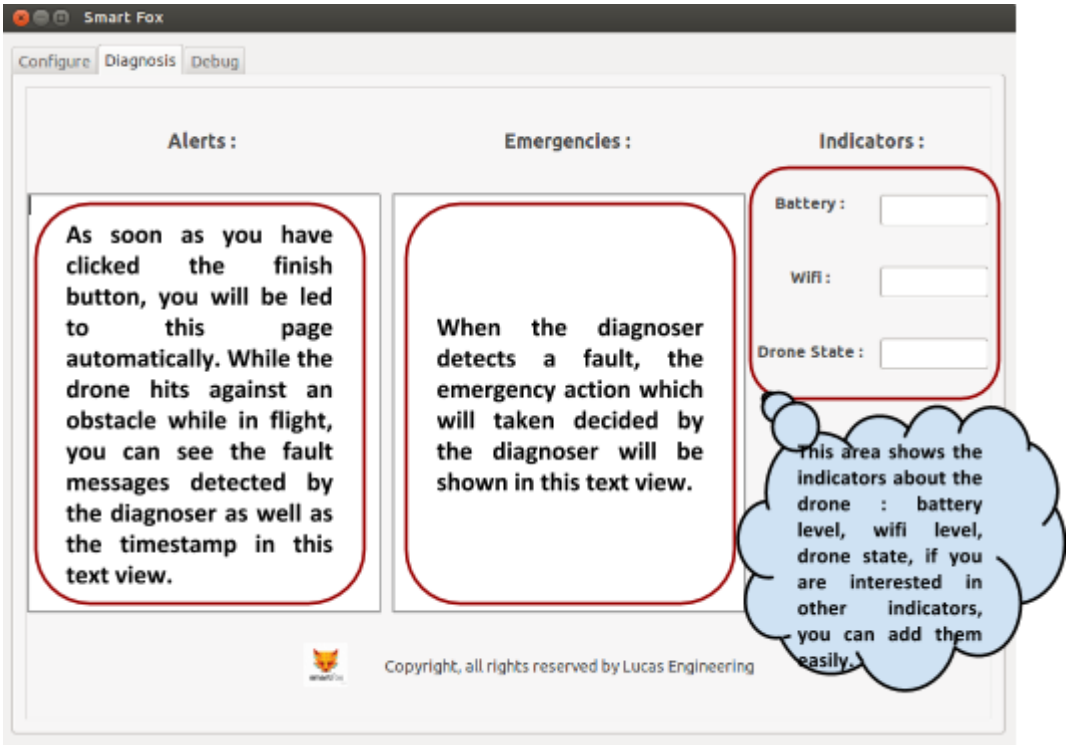
The GUI has 3 pages:

- Configure Page :



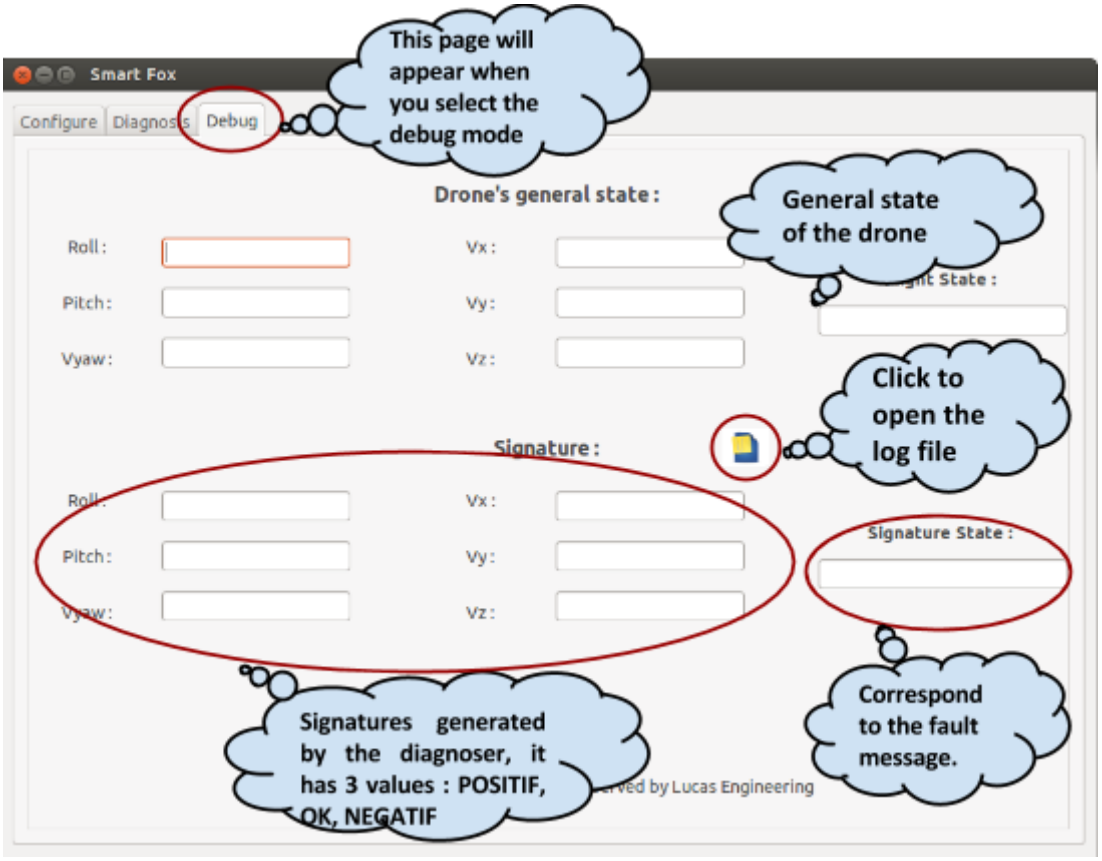
The configure page will appear just after launching the application. It is defined in the file *configurePage.h* and *configurePage.c*. The function *configPage()* is used for defining all the widgets in this page : 5 check buttons, 5 tooltips, 1 button etc. When click the finish button, the function *check_button_callback(GtkWidget *widget, gpointer data*) will be called as a callback function. In this function, we have a infinity loop in order to receive navigation data and different messages all the time.

- Diagnosis Page :



The diagnosis page will be shown when the user clicks the finish button. This page corresponds to the file *diagnosisPage.h* and *diagnosisPage.c*. The user will see different messages while the drone is in flight, as well as some indicators about the drone (battery level in percentage, Wi-Fi level and drone state). If the drone is in a nominal state, the fault messages will be displayed in black, otherwise, they will be displayed in red. Idem for the emergency message.

- Debug Page :



The use will see the debug page when the debug mode option is selected. *debugPage.h* and *debugPage.c* contain useful functions to draw this page. In this page, the user can find different parameters about drone's general state: roll , pitch in degree, Vyaw in degree/s, Vx, Vy,Vz in m/s and the flight state (Landing, Landed, Take off, Flying, Unknown State). The diagnosis results are also in this page: signatures (Positif, Ok, Negatif) and the signature state which is equal to the fault message. We display the signatures only when we detect a fault, so we ensure the correspondence between the signatures of parameters and the signature state.

The user can close the GUI at any moment by clicking the red-cross on the left bottom of the application.

2.3. Log files

If you choose the debug Mode option, log files are generated in *$(Current Directory)/DataModel*:

- The logSFM records the complete state of diagnosis and smart safety mode modules each 5ms (timestamp is printed).

  [residue Analysis] R:1, P:1, Y:1, Vx:1, Vy:1, Vz:1
  Give the current signature of diagnoser (0 is negative residue, 1 residue OK, 2 positive residue)

  CurrentFault : 24
  Give the current fault state of the drone, the enumeration is defined in *residue.h*

  [sfm]: state:0 R:0,000000,P:0,000000,Y:0,000000,G:0,000000
  Give the current state of SFM and the SFM commands

  drone state: alt:0,708000 pitch:0,535038 roll:-0,249036
  Vyaw:0,000960 Vx:0,000000 Vy:0,000000 Vz:0,000000
  Give the current drone state

  time:5
  Timestamp

- The *Fdata*.m* files record the data after each processing, in a MATLAB compliant format.
  - *FdataReal*: data directly received from the drone
  - *FdataSelected*: data converted in SI units and with derivative computing for Vyaw and Vz
  - *FdataFiltered*: data after filtering block
  - *FResidue*: residues values
  - *FdataModel*: data computed by drone model

  These files may help you to understand more accurately the functioning of SmartFox diagnosis layer.

3. Use SmartFox API

The Smartfox layer supplies an Application Programming Interface that enables to program your **own drone control application** and also to obtain the **diagnosis data**.

All the functions provided by the API are in the file : ***Sources/UserApp/smartfox_api.h***

3.1. Drone control application

To send motion commands to the drone, you need to call the following function:

```
void set_command (Inputs_t * newCommand , commandType_t type);
```

- `newCommand` is a structure that contains the command parameters (pitch, roll, yaw, gas).

  This is the `Inputs_t` structure definition:

```
typedef struct Inputs {
  float pitch; // negative = go forward / positive = go backward
  float roll;  // negative = bending leftward / positive = bend rightward
  float Vyaw;  // negative = rotation leftward / positive = rotation rightward
  float gas;   // negative = go down / positive = go up
} Inputs_t;
```

All the parameters are floating point values that **must be between -1.0 and 1.0** (otherwise the command is not taken into account). For pitch and roll, the more the value is near from 1.0 or -1.0, the more the angle will be high (left-right bending angle and front-rear bending angle). For gas and Vyaw, the more the value is near from 1.0 or -1.0 the more the velocity will be high (rotational velocity and vertical speed).
If all the parameters are set to 0.0, the drone will try not to move (hovering).

- `type` enables to specify what kind of command you need to send to the drone.

  All these types are defined in the `commandType_t` enumeration:

```
typedef enum commandType {
  TAKEOFF_REQUEST,
  LANDING_REQUEST,
  FLYING_REQUEST,   // to fly normally following the newCommand parameters
} commandType_t;
```

If type is set to `TAKEOFF_REQUEST` or `LANDING_REQUEST` the `newCommand` parameter is ignored and can be set to `NULL`.

NOTE : Interaction with diagnosis layer :
When the Smart Safety Mode is switched on, all the orders sent by this function are not taken into account.

3.2. SmartFox layer API

In your application, you can obtain all the information given by the diagnoser, just calling the next three functions:

- `getFault(fault_t * fault)`
- `getSignature(alarm_t * signature)`
- `getResidues(Navdata_t * model, Navdata_t * real, Residue_t * residue)`

See the section 4.1, to get more details about each ones.

3.3. Add threads to the user app using AR.drone API

In the Smarfox API provided, only one thread is available for the user application and you may need to add other threads. To do that, you have to follow precisely the following step. **You must not create on your own a thread with the pthread library.**

Note : In all the files you have tags ## USER APPLICATION ## that help you to locate where you have to insert code.

1. In *UserApp/* create 2 files for your thread : *mythread.c* and *mythread.h*

- In mythread.h set these lines :
```
#include <VP_API/vp_api_thread_helper.h> // api pour les threads
PROTO_THREAD_ROUTINE(my_thread, data);
```

- In *mythread.c* set these lines :
```
DEFINE_THREAD_ROUTINE(my_thread, data) {
    // code of your thread
    return 0;
}
```

2. Go to *smartfox_main.c* :

- Include UserApp/mythread.h in the section.
- In `ardrone_tool_init_custom()`, add `START_THREAD (my_thread, NULL);`
- In `ardrone_tool_shutdown_custom()`, add `JOIN_THREAD(th_user_app);`
- Finally, go to the end of the file and insert
  `THREAD_TABLE_ENTRY( my_thread ,custom_priority)`
  Keep in mind that if you set a too high priority to your thread, you may disrupt the diagnoser.

3. Go to *Build/Makefile* :

Find the variable `GENERIC_BINARIES_COMMON_SOURCE_FILES` and add the c files of your thread.

For more details about this, please report to the Parrot's SDK documentation.

4. Advanced Mode

4.1. Understands diagnosis indicators

The layer offers the opportunity to build your own safety application if the Smart Safety Mode does not meet your needs. With **`getSignature()`**, **`getFault()`**, **`getResidues`**(), you can use our diagnoser and residue generator without use the SFM. See section IV.2.1 for more information on return types.

Firstly, this is a quick explanation of the function meaning:

- **getFault(fault_t * fault)** return the current fault of type `fault_t`, it has been established by the diagnoser after a confirmation time of 300ms if the fault is consistent with the commands (i.e. rear obstacle with a move forward command is considered as inconsistent). If the fault is not consistent a 600ms confirmation time is necessary.

- **`getSignature(alarm_t * signature)`** returns an array of `alarm_t [6]` with the current classification of the residue on each data
    - This vector is the signature in diagnosis jargon, so you can freely integrate your signature matrix in order to analyse this signature.
    - Notice that the thresholds used to classify in three categories are exponential envelopes (cf. Diagnosis module requirement in SPRINT2 folder). Here an example of threshold parameters (defined in *residue.c*):
        - `INIT_PERCENT_PITCH*PITCH_DCGAIN`: initial value
        - `END_PERCENT_PITCH*PITCH_DCGAIN` : end value
        - `PITCH_TO` : time constant

- **`getResidues(Navdata_t * model, Navdata_t * real, Residue_t * residue)`** returns the raw values of residues which are computed by a simple difference between real data and model data (real-model).

4.2. Constants and types of diagnosis indicators

1. Typedef in *residue.h*

    - `alarm_t`:
        - Type: `enum`
        - Name: `alarm_t`
        - Values:
            - `ALARM_N  0`
            - `ALARM_Z  1`
            - `ALARM_P  2`

- residue_t:
    - Type: structure
    - Name: Residue_t
    - Attributes:
        - float32_t r_roll
        - float32_t r_pitch
        - float32_t r_Vyaw
        - float32_t r_Vx
        - float32_t r_Vy
        - float32_t r_Vz

- fault_t:
    - Type: enum
    - Name: fault_t
    - Values (used):
        - OBSTACLE_DEVANT 0
        - OBSTACLE_ARRIERE 1
        - OBSTACLE_DROITE 2
        - OBSTACLE_GAUCHE 3
        - OBSTACLE_HAUT 4
        - OBSTACLE_BAS 5
        - NO_FAULT 24
        - UNKNOWN_FAULT 25

2. Global variable in *residue.c*

- Signature matrix:
    - Type: alarm_t [SIGN_TAB_NB_FAULT][6]
    - Name: SIGN_MAT
    - Description: Record of all signatures caused by known fault. Each signature is directly linked to ONE fault.
    - Organisation:

	Roll	Pitch	Vyaw	Vx	Vy	Vz
sign 0	ALARM_Z	ALARM_Z	ALARM_Z	ALARM_P	ALARM_Z	ALARM_Z
...	...	...	...	...	..	...
sign N	ALARM_N	ALARM_Z	ALARM_Z	ALARM_N	ALARM_P	ALARM_N

- Correspondence table:
    - Type: fault_t [SIGN_TAB_NB_FAULT]

- Name: FAULT_TAB
- Description: Link Signature Matrix rows with corresponding fault
- Organisation:

Index	0: sign 0	...	N: sign N
fault	OBSTACLE_DEVANT	...	OBSTACLE_HAUT

- Signature:
  - Type: `alarm_t [6]`
  - Name: `alarm`
  - Description: Is the current signature recorded in the diagnosis module
  - Organisation:

Index	0: Roll	1: Pitch	2: VYaw	3: Vx	4: Vy	5: Vz
value	ALARM_Z	ALARM_Z	ALARM_Z	ALARM_P	ALARM_Z	ALARM_Z

- Fault:
  - Type: `fault_t`
  - Name: `currentFault`
  - Description: Is the current fault detected by diagnosis module

4.3. Process Faults with your application

For thread management, please reports to section 3. Now you know the indicators function you can use them to implement your own safety mode. So as we said in 4.1 you can call the indicators functions in order to get information on drone state and then process them as you wish. Then you will certainly want to send emergency commands to the drone. In order to do so some steps are needed:

- 1: activate option “Disable Safety Mode” in the GUI
- 2: then call `set_command()` and the drone will follow your orders.