Flight Plans

From PaparazziUAV
Jump to: navigation, search

A flight plan is a XML document which one can create and store aboard an autopilot. The flight plan will describe how you want your aircraft to travel if released into into the wild blue yonder.

Contents

DTD and Structure

The formal description of the flight plan file is given in the DTD (located in conf/flight_plans/flight_plan.dtd). This DTD must be referenced in the header of your flight plan XML document using the following line:

```
<!DOCTYPE flight_plan SYSTEM "flight_plan.dtd">
```

The flight plans are stored in the conf/flight_plans directory. The flight plan editor can be used to create basic flight plans via the GUI.

Extract from the DTD:

```
<!ELEMENT flight_plan (header?,waypoints,sectors?,include*,exceptions?,blocks)>
```

A flight plan is composed of two mandatory elements: waypoints and blocks

The root flight_plan element is specified with several attributes:

```
<flight_plan name lat0 lon0 ground_alt security_height home_mode_height qfu alt max_dist_from_home>
```

name
: The name of the mission (a text string)

lat0, lon0
: Defines the latitude and longitude coordinates of the reference point {0,0} in WGS84 degree coordinates

ground_alt
: The ground altitude (in meters), Above Sea Level where you are flying. It defines the GROUND_ALT constant value which can be used in combination with a waypoint <height> parameter to define a waypoint height

security_height
: The height (over ground_alt) used by the circle-home failsafe procedure and in other flight procedures such as formation flight and anti-collision avoidance. Warnings are produced if you place a waypoint lower than security_height (usually the case for the landing point)

home_mode_height (optional)
: This optional attribute available since v4.2 allows to override security_height as failsafe height in home mode. If home_mode_height Is set lower than security_height, the later is used. This attribute is useful if you need to return home at a high altitude rather than a low altitude.

qfu (optional)
: defines the global constant QFU. It usually is the magnetic heading in degrees (north=0, east=90) of the runway, the opposite of wind direction. This constant may be used in the mission description. It is also used by the simulator as the original course of the aircraft. So if you want to take off and climb to the West you would use qfu=270.

alt
: The default altitude of waypoints (Above Sea Level). So if your ground altitude is 400 then alt needs to be a value greater than ground altitude and above any obstructions in the flight plan.

max_dist_from_home

A radius representing the maximum allowed distance (in meters) from the HOME waypoint. Exceeding this value (ie flying outside the circle with this radius) will trigger an exception. It is up to you to define the block to be executed (ie what to do) for the exception.

Here is an example of such a line in the top of a flight plan:

```
<flight_plan alt="250" ground_alt="185" lat0="43.46223" lon0="1.27289" name="Example Muret" max_dist_from_home="3
```

Note that a flight plan could also contain optional `include`'s and `exceptions` cases.

In English the above flight plan says the name is Example Muret. The reference coordinates for the 0,0 point is: 43.46223 (lat) and 1.27289 (long). The flying site 0,0 location is 185m above sea level. The security height is 25m above 0,0 point or 210m above sea level. The default (ie if not defined in a waypoint this alt is used) altitude is 250m (above sea level). The home mode block altitude is defined to be 150m above sea level. Also, for security, a circle is defined with a radius that's 300m from 0,0 position. This is the max_dist_from_home value. Fly 301m from 0,0 and an exception is triggered. A useful block is to trigger/go to the home mode block and return to home when the aircraft flies outside the safety circle. Example flight plans are helpful for study before you build your own from scratch.

Waypoints

The waypoints are the geographic locations used to specify the trajectories. A waypoint is specified by it's name and coordinates:

```
<waypoint name wpx wpy [alt] [height]/>
```

where wpx and wpy are real positional coordinates (`lat/lon`) or UTM coordinates (`utm_x0/utm_y0`) or relative coordinates (`x/y`) in meters from your reference point {0,0} . `alt` is an optional parameter and can be used to assign an altitude to a particular waypoint that is different from the globally defined `alt` parameter of the flightplan. The `height` attribute can be used to set the waypoint height relative to the ground altitude (`ground_alt`) of the flight plan for this waypoint.

An example:

```
<waypoints>
  <waypoint name="HOME" x="0.0" y="30.0"/>
  <waypoint name="BRIDGEOVERRIVER" x="-100.0" y="60.0" alt="270."/>
  <waypoint name="MyBarn" x="-130.0" y="217.5" alt="3000."/>
  <waypoint name="3" x="-30.0" y="50" height="50."/>
  <waypoint name="4" x="-30.0" y="50." alt="ground_alt + 50"/>
  <waypoint name="_MYHELPERSPOT" x="-30.0" y="60" height="50."/>
  <waypoint name="_MYOTHERHELPERSPOT" x="-70.0" y="90" height="70."/>
  <waypoint name="TOWER" lat="48.858249" lon="2.294494" height="324."/>
  <waypoint name="MountainCAFE" utm_x0="360284.8" utm_y0="4813595.5" alt="1965."/>
</waypoints>
```

Tips

- Waypoints are easily adjusted with the flight plan editor.
- If a waypoint name starts with an underscore (_), the waypoint is not displayed in the GCS, except in editor mode.

- The maximum number of waypoints is 254.
- A waypoint named HOME is required if the failsafe HOME mode procedure is used.
- A waypoints index/reference pointer is derived by prefixing the waypoint name with "WP_". Useful when a call function uses the waypoints reference index vs. it's name.

Sectors

Static sectors (default)

Flat *Sectors* can be described as an area defined by list of waypoint corners. Such an area will be displayed in the Ground Control Station (GCS) by colored lines connecting the cornerpoints. A function is generated to check if a point, usually the aircraft itself, is *inside* this sector. Currently, this feature requires that the polygon is convex and described in a clockwise order. For a sector named MyBigGarden the generated function for the example here would be bool_t InsideMyBigGarden(float x, float y); where x and y are east and north coordinated, in meters, relative to the geographic reference of the flight plan. If the flight plan is dynamically relocated, such a sector will be relocated but the display is currently not updated on the GCS. It would be great if one would help improving that part of the source code. Note that sector names are not allowed to contain spaces.

For example, with the following element in a flight plan.

```
<sectors>
  <sector name="MyBigGarden" color="red">
    <corner name="_1"/>
    <corner name="_2"/>
    <corner name="_3"/>
    <corner name="_4"/>
  </sector>
</sectors>
```

It is then possible to add an exception clause to your flightplan. For example if the aircraft for some reason flies outside this, defined by us, sector the airframe will fly to a standby waypoint. The exclamation mark (!) means the boolean operator NOT in this example. In regular language one would describe "If my airframe is NOT inside the MyBigGarden sector anymore then deroute it to the standby waypoint. In Flightplan "Speak" this is written like:

```
<exception cond="! InsideMyBigGarden(GetPosX(), GetPosY())" deroute="standby"/>
```

NOTE: editing of the waypoints of the sector during the flight will not dynamically update the inside function. It will always check if the position is inside the original sector.

Tips

- A nice option in the corner notation is that one can add an underscore (_) in front of the name; a corner or waypoint name that starts with an underscore is not displayed in the GCS. Only in editor mode it is visible. It is visible in editor mode, because if you the could not see it, it also would be not possible to edit or drag the corner or waypoint to another position.
- The color indicating the sector borders is not fixed but can be defined by oneself if wished for via the color attribute.

Dynamic sectors

With the latest version (v5.5-devel-628), it is possible to create dynamic sectors. The procedure to create the sector is the same than for the static version with an extra attribute type="dynamic":

```
<sectors>
  <sector name="MyBigGarden" color="red" type="dynamic">
    <corner name="C1"/>
    <corner name="C2"/>
    <corner name="C3"/>
    <corner name="C4"/>
  </sector>
</sectors>
```

It is also recommended to avoid using hidden waypoints (no _ prefix), so you can move the corner of your sector from the GCS. The polygon is updated on the 2D map to reflect the new waypoints positions. Beside the possibility to change the shape of the area in flight, one of the main benefit is that the algorithm behind allows concave hulls. The only restriction is that the edges of the polygon should not cross each other.

The generated function is the same than the static version and can be used the same way.

Includes

include is used to add some flight plan elements defined in an external procedure. It’s useful to include pre-written procedures with only few arguments and then clarify the flight plan. Here is the structure:

```
<include name procedure> [<arg name value />]*[<with from to />]*</include>
```

where name attribute of the include element will be used in this flight plan to prefix the blocks of the procedure, the XML referenced file. Named arguments may be given with their value in the arg elements. The with tag allows to link labels (e.g. attribute of a deroute instruction or of an exception) from the procedure to blocks of the main flight plan. Then, each block of the procedure is like any block of the flight plan and is designated with a dotted identifier: block b of a procedure named p is named b.p .

Here is an example:

```
<includes>
  <include name="landing" procedure="landing.xml"/>
</includes>
```

Blocks

Block elements are the main part of a flight plan: they describe each unit of the mission. They are made of various primitives, called stages and exceptions, you can put one after the other. When a stage (or a block) is finished, the autopilot goes to the next one. The behaviour after the last stage of the last block is undefined.

As described in the DTD, the blocks element is composed of block elements which are sequence of *stages*:

```
<!ELEMENT blocks (block+)>
```

```
<!ELEMENT block  (exception|while|heading|attitude|go|xyz|set|call|circle|deroute|stay|follow|survey_rectangle|for|
```

Example:

```
<block name="circlehome">
  <circle radius="75" wp="HOME"/>
</block>
```

You can add a button in the strip of the aircraft with the attribute `strip_button`:

```
<block name="descent" strip_button="Descent">
  <circle wp="HOME" throttle="0.0" pitch="-15" vmode="throttle"/>
</block>
```

This button will activate the block. If the attribute `group` is specified, all strip buttons of the same group will be placed vertically on top of each other.

In the same way, a key shortcut can be specified:

```
<block key="D" name="descent" strip_button="Descent">
  <circle wp="HOME" throttle="0.0" pitch="-15" vmode="throttle"/>
</block>
```

Modifiers are allowed, using the syntax of GTK accelerators.

An icon can be specified to display the button. The `strip_button` label then is a tooltip for the icon. The icon must be an image file available in the directory `data/pictures/gcs_icons`:

```
<block name="Takeoff" strip_icon="takeoff.png" strip_button="Takeoff">
```

You can call functions before or after each execution of the block:

```
<block name="circlehome" pre_call="function_to_call_before_circle()" post_call="function_to_call_after_circle()">
  <circle wp="HOME"/>
</block>
```

Expressions

Most of the numeric attributes in stages are analyzed as C expressions. The syntax of this C expression is restricted to

- numeric constants
- some internal autopilot variables (not fully documented, see examples)
- Some binary operators: <, >, <=, >=, <>, ==, +, -, /, *
- Some utility functions

Some examples of usable expressions are given in the next sections.

Initialization Blocks

Most flight plans will have three blocks of flight plan initialization blocks. It is good practice to follow this example below if you first start learing to create flightplans

The first block waits until the GPS fix has been established, as shown below.

```
 <blocks>
  <block name="Wait GPS">
   <set value="1" var="kill_throttle"/>
   <while cond="!GpsFixValid()"/>
  </block>
```

The second block updates the local waypoints with respect to the UAV.

```
  <block name="Geo init">
   <while cond="LessThan(NavBlockTime(), 10)"/>
   <call fun="NavSetGroundReferenceHere()"/>
  </block>
```

This next block prevents the UAV from starting the engine and taking off.

```
  <block name="Holding point">
   <!--set var="nav_mode" value="NAV_MODE_ROLL"/-->
   <set value="1" var="kill_throttle"/>
   <attitude roll="0" throttle="0" vmode="throttle"/>
  </block>
```

Exceptions

The flight manager can handle exceptions. They consist in conditions checked periodically (at the same pace as the navigation control), allowing the control to jump to a given block. Here is the syntax of exceptions:

```
<exception cond="..." deroute="...">
```

where `cond` is an expression and `deroute` is the name of the block we want to switch to as soon as the condition is true.

Here are some example of exceptions:

```
 <exception cond="10 > PowerVoltage()" deroute="go_down"/>
 <exception cond="(ground_alt+10 > GetPosAlt())" deroute="go_up"/>
 <exception cond="(autopilot_flight_time > 840)" deroute="quick_land"/>
```

Exceptions can be local to a block or global to the flight plan, in the `<exceptions>` element. In the following example, time since last reception of a message from the ground station is monitored and the navigation is switched to the `Standby` block if no message have been received for 22s. This exception is valid for all the blocks.

```
 <flight_plan ...>
  <waypoints> ... </waypoints>
  <exceptions>
   <exception cond="datalink_time > 22" deroute="Standby"/>
  </exceptions>
 <blocks> ...
```

Deroute

The `deroute` is the *goto* directive of the flight plan; it switches the navigation to the given block:

```
<deroute block="landing"/>
```

Note that this primitive should not be used to execute loops which are provided by the following elements.

Return

The return is also a *goto* directive that brings you back to the last block (and last stage). It has no argument.

```
<return/>
```

Loops

Unbounded loops are written with while elements whose cond attribute is a boolean expression. Children of while are stages:

```
 <while cond="TRUE">
   <go wp="A"/>
   <go wp="B"/>
   <go wp="C"/>
   <while cond="5 > stage_time"/>
  </while>
```

In this example, we run an infinite loop, lettin the aircraft try to go via waypoints A, B and C and waiting for 5 seconds before repeating.

Bounded loops are written with the for tag:

```
 <for var="i" from="0" to="3">
   ...
 </for>
```

where the body of the loop will be run four times.

The variable of a for loop can be used inside expressions appearing as attributes of the stages:

```
 <for var="i" from="1" to="5">
   <circle wp="HOME" radius="75" alt="ground_alt+50*$i" until="stage_time>10" />
 </for>
```

In this example, the aircraft will circle around waypoint HOME for 10 seconds at and altitude above ground of 50m, 10 seconds at altitude 100 meter (50+50), ... until 250m (5x +50).

Note: Two bounded loops using the same control variable are not allowed in the same block.

Navigation modes

Navigation modes give the description of the desired trajectory in 3D. While the horizontal mode is specified through *stages*, the vertical control is specified with various attributes of these stages. The current available navigation stages are

- attitude : just keep a fixed attitude;
- heading : keep a given course;
- go : go to a given waypoint;

- path : list of waypoints linked by *go*
- circle : circle around a waypoint;
- oval : two half circles with a straight between two nav points
- eight : fly a figure of eight through a waypoint and around another
- stay : hold the position (hard to realize for a fixed-wing aircraft);
- follow : follow another aircraft;
- xyz : circle around a point moveable with the RC transmitter stick (obsolete with the datalink).

The vertical control is achieved using the vmode attribute of these stages. The possible values are

- alt (the default) : the autopilot keeps the desired altitude which is the altitude of the waypoint (if any) or the altitude specified with the alt attribute;
- climb : the autopilot keeps the desired vertical speed specified with the climb attribute (in m/s);
- throttle : the autopilots sets the desired throttle specified with the throttle attribute (between 0 and 1);
- glide : the autopilot keeps the desired slope between two waypoints

The default control is done with the throttle. However, setting the pitch attribute to auto and the throttle attribute to a constant allows a vertical control only by controlling the attitude of the A/C. The pitch attribute also can be set to any value (in degrees) while the throttle control is in use: it usually affects the airspeed of the aircraft.

The different navigation modes are detailed in the next sections.

Attitude

Element attitude is the navigation mode which corresponds to the current lowest control loop for horizontal mode. The autopilot then keeps a constant attitude. The roll attribute is required (in degrees, positive to put right wing low).

To fly away, at constant airspeed:

```
<attitude roll="0" vmode="throttle", throttle="0.5"/>
```

To fly around, holding a given altitude:

```
<attitude roll="30" alt="ground_alt+50"/>
```

Note that it is not a *safe* navigation mode since the geographic position of the plane is not controlled. However, this mode is useful to tune the roll attitude control loop.

Heading

heading primitive is relative to the second level loop for horizontal mode in the autopilot which will keep the given course, a required attribute (in degrees, clockwise, north=0, east=90).

One example to takeoff, following the QFU, 80% throttle, nose up (15 degrees) until height of 30m is reached:

```
<heading course="QFU" vmode="throttle" throttle="0.8" pitch="15" until="(GetPosAlt() > ground_alt+30)"/>
```

Go

The go primitive is probably the most useful one. Basically, the autopilot will try to join a given waypoint (wp, the only required attribute). So the simplest thing you can ask for is

```
<go wp="HOME"/>
```

which will set the HOME waypoint as the desired target position. Note than since vmode="alt" is the default, the altitude of the target waypoint is also taken into account. The navigation will switch to the next stage as soon as the target is reached.

It is usually not a good idea to try to join a waypoint without asking for a precise trajectory, i.e. a given line. Setting the hmode attribute to route, the navigation will go over a segment joining two waypoints:

```
<go from="wp1" wp="wp2" hmode="route"/>
```

The target altitude is the altitude of the target waypoint; it can also be set with the alt attribute. The following example keeps an altitude with fixed throttle:

```
<go from="wp2" wp="wp3" hmode="route" pitch="auto" throttle="0.75" alt="ground_alt+100"/>
```

The attributes related to the vertical control can also be set to replace the default altitude mode:

```
<go from="wp1" wp="wp2" hmode="route" vmode="climb" climb="1.5"/>
```

Finally, the approaching_time (in seconds) attribute helps to decide when the target is *reached*. It can be set to 0 to go over the target waypoint (default value is the CARROT time, set in the airframe configuration file).

```
<go from="wp1" wp="wp2" hmode="route" approaching_time="1"/>
```

Path

The path primitive is just a shorthand expression for a set of go primitives. A list of waypoints defined with the wpts attribute is pre-processed into a set of go primitives with the hmode attribute. For example:

```
<path wpts="wp1, wp2, wp3"/>
```

Other attributes are optional:

```
<path wpts="wp3, wp1, wp2" approaching_time="1" pitch="auto" throttle="0.5"/>
```

Circle

The circle primitive is the second main navigation mode: the trajectory is defined as a circle around a given waypoint with a given radius:

```
<circle wp="HOME" radius="75"/>
```

A positive radius makes the UAS move clockwise, a negative counter-clockwise.

The until attribute may be used to control the end of the stage. The following example defines an ascending trajectory at constant throttle, nose up (15 degrees), over growing circles, until the battery level is low:

```
<circle wp="wp1" radius="50+(GetPosAlt()-ground_alt)/2" vmode="throttle" throttle="0.75" pitch="15" until="10>PowerVolta
```

Oval

The oval consists of two half circles that are connected with two straight lines. This flight path is usefull when a IMU is used because the straights allow for level flight.

```
<oval p1="1" p2="2" radius="nav_radius"/>
```

Eight

Fly a figure of eight that consists of two straight legs that pass though the center and the center of the half circle at the end of the two legs is in the turn around waypoint. The altitude of the center waypoint is used for the entire figure. The turn around waypoint is moved to match radius given.

```
<eight center="1" radius="nav_radius" turn_around="2"/>
```

Survey rectangle

Fly a survey rectangle defined by two waypoints. The distance between the legs of the grid (in meter) and the orientation of the grid (NS or WE) can be set by the operator. The plane will turn outside of the border of the rectangle before starting a new leg.

```
<survey_rectangle wp1="1" wp2="2" grid="200" orientation="NS"/>
```

Follow

The follow is a special primitive which makes the UAS follow another UAS (real or simulated, named with its ac_id) at a given distance (in meters) behind and at a given height (in meters) above.

In this example, the autopilot will try to follow A/C number 4, staying 50m behind and 20m above.

```
<follow ac_id="4" distance="50" height="20"/>
```

Stay

The stay is a mode for UAS's able to hover:

```
<stay wp="HOME" alt="10"/>
```

XYZ

xyz is a special mode where the UAS circles around a user moveable waypoint. This waypoint is moved with the RC sticks:

- YAW channel controls the point over the west-east axis;
- PITCH channel controls the point over the south-north axis;
- ROLL channel controls the altitude.

Example (default radius is 100):

```
<xyz radius="40"/>
```

Set

The set element is a dangerous one which should be used only by expert users: it is used to directly set an internal variable of the autopilot. For example, you can change the value of the default ground altitude, a variable used by the home mode failsafe procedure (and maybe by your own flight plan):

```
<set var="ground_alt" value="ground_alt+50"/>
```

This directive is extremely powerful and has great potential for error - use with caution.

Call

The call allows the user to define its own navigation procedures in C. The value must be a call to a boolean function which must return TRUE as long as the stage is not completed (a function which should be called only once would then return immediately FALSE). This feature is illustrated with the line pattern:

```
 <call fun="nav_line_setup()"/>
 <call fun="nav_line_run(WP_1, WP_2, nav_radius)"/>
```

where nav_line_setup() returns FALSE and nav_line_run() always returns TRUE (this stage never ends). Note that a waypoints index is derived/denoted by prefixing the waypoint name with WP_(i.e.: 1 --> WP_1, 2 --> WP_2)

To call *any* function exactly once regardless of the return value (e.g. call a void function), add loop="FALSE"

```
 <call fun="viewvideo_take_shot(TRUE)" loop="FALSE"/>
```

Such extra navigation functions are usually written as a Module and the header files are included automatically.

If you want to call functions that are not part of a module, you need to include the header file which contains the function declaration:or supplementary C file which must be specified in the

```
 <header>
#include "path/to/header.h"
 </header>
```

where the path is relative to the sw/airborne directory.

You can also call functions before or after each execution of the block (this means continuously on each iteration of each stage of the block, not just when entering o exiting the block).

```
 <block name="circlehome" pre_call="function_to_call_before_circle()" post_call="function_to_call_after_circle()">
  <circle wp="HOME"/>
 </block>
```

Pre Call

```
<block name="Standby" strip_button="Standby" strip_icon="home.png" pre_call="if(!InsideKill(GetPosX(), GetPosY())) NavKil
```

Post Call

```
<block name="traj" pre_call="formation_pre_call()" post_call="formation_flight()"> <!-- formation flight is call after all other
```

Procedures

Procedures are libraries which can be included in flight plans. They are composed of waypoints, sectors and blocks. The header of a procedure may contain some parameters which are replaced by arguments when the procedure is included.

Extract of the DTD: a procedure is a sequence of parameters, waypoints, ...:

```
<!ELEMENT procedure (param*,header?,waypoints?,sectors?,exceptions?,blocks?)>
```

A parameter is just a name. A parameter is optional if it is declared with a default value. An example with a required and an optional parameter:

```
 <param name="alt"/>
 <param name="radius" default_value="75"/>
```

Procedures are called with the include element in a flight plan. A procedure cannot be included twice or by another procedure. A procedure call requires:

- the name of the procedure file, the name given to this inclusion;
- values for the parameters;
- backlinks for block name exits of the procedure.

For example:

```
<include name="landing" procedure="landing.xml"/>
```

Here is the corresponding procedure landing.xml:

```
<!DOCTYPE procedure SYSTEM "flight_plan.dtd">
 <procedure>
  <waypoints>
   <waypoint name="AF" x="177.4" y="45.1" alt="30"/>
   <waypoint name="TD" x="28.8" y="57.0" alt="0"/>
   <waypoint name="_BASELEG" x="168.8" y="-13.8"/>
  </waypoints>
  <blocks>
   ...
   <block name="land">
    <call fun="nav_compute_baseleg(WP_AF, WP_TD, WP__BASELEG, nav_radius)"/>
    <circle radius="nav_radius" until="NavCircleCount() > 0.5" wp="_BASELEG"/>
    <circle radius="nav_radius" until="And(NavQdrCloseTo(DegOfRad(baseleg_out_qdr)-10), 10 > fabs(GetPosAlt()- Waypoin
   </block>
   ...
  </blocks>
 </procedure>
```

Note that the name of procedure land block will be renamed into landing.land:

```
<deroute block="landing.land"/>
```

will jump to this procedure block.

Suppose you have a go-around condition in your landing procedure. You would write it

```
<exception cond="..." deroute="go-around"/>
```

then you must link this block exit with one of your block (e.g. Standby). So you would include the procedure as follows:

```
 <include name="landing" procedure="landing.xml">
   <with from="go-around" to="Standby"/>
 </include>
```

Advanced Examples

Parameters used in a flight plan can be computed expressions. In this example, the plane is asked to perform 5 circles at progressively increasing altitudes for exactly one minute at each altitude:

```
 <for var="i" from="1" to="5">
   <circle wp="HOME" radius="75"
         alt="ground_alt + 50*$i"
         until="stage_time > 60" />
 </for>
```

Below you find some random examples of the posibilities. This is only the tip of the iceberg, use your imagination and go wild with new creative ideas for your flightplan

Gains on the fly

It is very well possible to set specific gain for an airframe if it reaches e.g a certain block.

Dynacmically adjustable maximum speed

Immobilize Actuators

h_ctl setpoints variable are set by the h_ctl_attitude_loop() (from fw_h_ctl.c) loop) which can be disabled with the h_ctl_disabled flag:

```
 <set var="h_ctl_disabled" value="TRUE"/>
 <set var="h_ctl_aileron_setpoint" value="0"/>
 <set var="h_ctl_elevator_setpoint" value="MAX_PPRZ/2"/>
 .... waiting for a condition ...
 <set var="h_ctl_disabled" value="FALSE"/>
```

Tips and Tricks

There are many ways to skin a cat just as there are many ways to craft your flight plan. Following the best practices tips can save you from a lot of frustration and mishap.

- Simulate your flight plan before taking it to the sky. Flight plans should always be carefully tested

prior to flight, take a look at the simulation page for details on how to simulate your plan.
- Make an subdirectory in the Flight_plan directory with your own name and add your flight plans there. Make sure that the location of the DTD is correct, e.g by using relative directory double dots as in <!DOCTYPE flight_plan SYSTEM "../flight_plan.dtd">
- Take a good look at other flight plans included with Paparazzi. To learn from example flight plans please visit the flight plan examples page
- There are several option to build failsafe features into you flightplan, for some examples visit the Failsafe page.
- Some flight plan examples define waypoint locations using relative coordinates. These are relative positions from the fixed lat and lon in the header of the flight plan. When simulating your flight plan the aircraft always use the lat/lon as defined in the flight plan since a regular simulation has no notion of you current position of you local PC where you simulate on. This is something to keep in mind if you test your flight plan in real flights.

Navigation

- Home
- Hardware
- Software
- FAQ
- Downloads
- Remembering Hecto

Communication

- Mailing list
- Gitter
- Contact

Development

- How to contribute
- Developer Guide
- Doxygen docs
- Git repository
- Build tests

Wiki tools

- Recent changes
- Random page
- Editing Help

Print/export

- Create a book
- Download as PDF
- Printable version

Tools

- What links here
- Related changes
- Special pages
- Permanent link
- Page information

- This page was last modified on 10 July 2015, at 07:36.
- This page has been accessed 99,083 times.
-