Crash Course Paparazzi 2019
Developing your own module

Part 5

Tom van Dijk

February 2019

This final part is set up a bit differently than the previous ones: instead of telling you step-by-step what to do, this document will give you a brief overview of the concepts involved in writing your own module and it will show you where to find further information.

The simplest way to start writing your own code is by modifying the example modules: the colored object detector and orange avoider. The rest of this document will often refer to these modules as examples and describe *why* they are written as they are. Once you understand how these example modules work, you can create new or additional modules if you need them.

Contents

1 Module code overview

In the previous part you added the cv_colorfilter and orange_avoider modules to your airframe. In this part, we will take a closer look at how these modules are written and use this as an example for your own module.

1.1 Module files

Modules consist of the following files: a module .xml file, source and header files. The .xml file contains a description of the module that tells Paparazzi which files to include during compilation. These xml files can be found in `paparazzi/conf/modules/`. Take for instance the orange_avoider.xml:

```
<module name="orange_avoider">
  <doc>
    ...
  </doc>
  <depends>...</depends>
  <header>
    <file name="orange_avoider.h"/>
  </header>
  <init fun="orange_avoider_init()"/>
  <periodic fun="orange_avoider_periodic()" freq="4"/>
  <makefile >
    <file name="orange_avoider.c"/>
  </makefile>
</module>
```

The xml file starts with a 'module' element that sets the name of the module (orange_avoider). Optionally, this element can contain a 'dir' attribute as well, to specify the location of the source files relative to `sw/airborne/modules/`. In this case the directory is not provided and the module name is used (i.e. `sw/airborne/modules/orange_avoider/`).

After a documentation and dependency section, the xml contains a 'header' element in which the header files of the module are listed. Typically, you will only see one header file here that provides an easy-to-use access point for other modules.

The header element is followed by an 'init' and 'periodic' element. These specify what functions in your module code should be called by the autopilot, and in case of the periodic function it also specifies its frequency in Hz. Section 1.2 will describe these functions in more detail.

At the end of the xml file is the 'makefile' element. This section describes how your source files should be compiled. Simple modules such as the orange avoider only list one or more source files. More complicated modules such as `cv_opencvdemo` can specify additional compiler flags (to link OpenCV, for example) and can have different makefile sections depending on whether the autopilot is compiled for use on the drone (`target="ap"`) or in simulation (`target="nps"`).

More information on module .xml files can be found here.

The source and header files of your module can be found in `sw/airborne/modules/<your module dir>/`. In the next sections we take a closer look at the content of these files.

When creating your own module, you can use the `create_module` script in the `paparazzi` folder to create stubs for your module xml and the source and header files.

1.2 Module functions

The `orange_avoider.c` file has roughly the following structure:

```
#include ...

#define ...

void orange_avoider_init() {
   ...
}

void orange_avoider_periodic() {
   ...
}

...
```

Note that the module does not have a 'main' function. This is because the module is not a stand-alone program. Instead, the autopilot will regularly call other functions that are part of your module, such as the `orange_avoider_init` and `orange_avoider_periodic` functions in this example. Which functions are called is defined by the module xml file described earlier.

The Paparazzi wiki lists the types of functions you can register in the module xml: `init`, `periodic`, `event` and `datalink`, of which init and periodic are the most relevant for this course.

The *init* function is called once at startup. You can use this function to initialize important variables of your module, or for instance to subscribe to new video frames (section 1.3).

Once the autopilot is fully initialized, it will enter an infinite loop[1] in which it will continuously read new sensor data, feed this to the guidance and stabilization controllers and send new commands to the Bebop's motors[2]. From this loop, the autopilot can also call your module's *periodic* function at a frequency specified in the module xml. Within this function, you can for instance get the drone's state and use this to calculate new setpoints for the guidance controller. In the `orange_avoider_periodic` example, the module reads the amount of orange in the front camera image and uses this to set its next waypoint positions.

Because the periodic function is called from within the autopilot's control loop, you should take care that the function does not take too much time to run. The autopilot runs at 512 Hz, which means that it has slightly less than 2 ms to run your module code, the code of the other modules and the control loops and estimators. If your periodic function takes too long, the autopilot will run at a lower frequency than intended, which can lead to instability. In practice you have to make things pretty bad before this becomes a problem, but you should be careful when using large or nested loops in your periodic function, and video processing is best performed in the video callback function (section 1.3) as this callback runs in a separate thread.

Together with the periodic function, the module xml can specify a *start* and *stop* function. These are called when the module is started or stopped, respectively. You can find an example of these functions in `sw/airborne/modules/loggers/file_logger.c`, where they are used to open and close the log file.

[1] `sw/airborne/firmwares/rotorcraft/main.c` line 68.
[2] `sw/airborne/firmwares/rotorcraft/main_ap.c handle_periodic_tasks` (line 202).

1.3 Handling video

Video handling in Paparazzi works similar to the module functions described above: the autopilot will call a function in your module when a new video frame becomes available. Consider the following extract from the colorfilter module, `sw/airborne/modules/computer_vision/colorfilter.c`:

```
#include "modules/computer_vision/colorfilter.h" // Includes computer_vision/cv.h!

...

struct image_t *colorfilter_func(struct image_t *img) {
   ...
   return img; // Return modified image for further processing
}

void colorfilter_init(void) {
   listener = cv_add_to_device(&COLORFILTER_CAMERA, colorfilter_func, COLORFILTER_FPS);
}
```

This code fragment contains two functions: a video callback function `colorfilter_func` and an init function `colorfilter_init`. As in the previous section, the init function is specified by the module xml, but this is not possible for video callbacks. Instead, the init function uses `cv_add_to_device` (from `modules/computer_vision/cv.h`) to register the video callback function. The `cv_add_to_device` function requires a video device pointer (here `&COLORFILTER_CAMERA`, which is set to `front_camera` in the airframe file), the name of the callback function (`colorfilter_func`) and the FPS at which the function should be called (`COLORFILTER_FPS`).

After registering, the video callback is called when new frames become available. The video callback function takes one argument: a `struct image_t` pointer to the current camera frame. You can perform further processing of this image inside your video callback function. At the end of the function you need to return a `struct image_t` pointer. This pointer is then used as input for the next module that subscribed to this camera. Typically this pointer is the same as the input pointer.

The video callback function takes a pointer to the latest camera frame as argument. This camera frame is an `image_t` struct, as defined in `sw/airborne/modules/computer_vision/lib/vision/image.h` (line 43). This struct gives you access to the raw pixel values in the `buf` field.

Images in Paparazzi use a YUV encoding, where Y is the luminance ('brightness') of a pixel and U and V specify the color. Pixel colors are stored in a UYVY format, which works as follows: for each pixel, two bytes of color information are used: one byte for the Y value, and one byte for either the U or V value. In other words, each pixel is described by either a (U, Y) or (V, Y) byte pair. This means that *you cannot get both the U and V value for a single pixel*! Instead, you need to read the missing U or V value from a neighboring pixel, which is a reasonable approximation in most natural images.

While it is possible to work directly with the UYVY values (as, for instance, in `image.c`'s `image_yuv422_colorfilt` (line 154)), it is probably easier to transform this image to an OpenCV `Mat` which allows you to use OpenCV to further process the image. An example of this can be found in `sw/airborne/modules/computer_vision/openc`

```
#include "opencv_image_functions.h"
#include <opencv2/...>
using namespace cv;

int opencv_example(char *img, int width, int height) {
   // Transform image buffer img into an OpenCV YUV422 Mat
   Mat M(height, width, CV_8UC2, img);
   // Convert to OpenCV BGR
   Mat image;
   cvtColor(M, image, CV_YUV2BGR_Y422);

   /* Do OpenCV stuff here */
```

```
   ...

   // Convert back to YUV422, and put it in place of the original image
   colorrgb_opencv_to_yuv422(image, img, width, height);
   return 0;
}
```

The function first creates a `Mat` object from the image buffer, then uses OpenCV's `cvtColor` to convert it from Paparazzi's UYVY format to OpenCV's BGR. The image can then be processed as normally by OpenCV. At the end of the function, the image is transformed back to Paparazzi's UYVY format using `colorrgb_opencv_to_yuv422`. Of course, this step is only necessary if you want to pass a modified image to the next module that subscribed to this camera.

1.3.1 Thread safety

Unlike the module's periodic function, the video callback functions run in a separate thread from the autopilot. This means that the video callback runs *in parallel* to the autopilot loop, and as a result the processing time in your callback function can exceed 2 ms without slowing down the rest of the autopilot.

Threading, however, makes it tricky to send information from the video callback to the module's periodic function, as both functions could be running in parallel and reading/writing the same variable at the same time. Threading problems are notoriously hard to debug as they often depend on the relative timing between threads, which is more-or-less random. To prevent threading issues, variables accessed by the video thread should be protected by a *mutex*. Before accessing shared variables, a thread will attempt to lock this mutex. A mutex can only be locked once, other threads that try to lock the same mutex will wait until the mutex is unlocked again. Proper use of a mutex ensures that shared variables are accessed by only one thread at a time. A full-on introduction of mutexes is far beyond the scope of this course, instead you are recommended to follow the example from `sw/airborne/modules/computer_vision/cv_detect_color_object.c`:

```
#include "pthread.h"

static pthread_mutex_t mutex;
struct color_object_t global_filter; // Variable shared by video and ap thread
                                     // Note: in the real code this is an array,
                                     // here just a variable to simplify this example.

...

// Video callback, executed in video thread
static struct image_t *object_detector(...) {
   /* Calculate object detection results */
   ...

   pthread_mutex_lock(&mutex);
   global_filter = ... // Store results in global_filter
   pthread_mutex_unlock(&mutex);
   ...
}

...

// Module init function
void color_object_detector_init(void) {
   ...
   pthread_mutex_init(&mutex, NULL);
   ...
}
```

```c
// Module periodic function
// Executed in autopilot thread
void color_object_detector_periodic(void) {
   static struct color_object_t local_filter;
   pthread_mutex_lock(&mutex);
   local_filter = global_filter; // Copy results from global_filter for processing
   pthread_mutex_unlock(&mutex);

   /* Do stuff with local_filter */
   ...
}
```

The example works as follows: this module contains two functions that need to communicate with each other: a video callback function `object_detector` that runs on the video thread and performs computationally intensive calculations, and a periodic function `color_object_detector_periodic` that runs on the autopilot thread and further processes the result.

The threads communicate through the `global_filter` struct, which holds the object detection results that should be processed in the periodic function. Access to this struct is protected by the `mutex` object. The mutex object is initialized in the module's init function using `pthread_mutex_init(...)`.

When a new video frame arrives, this is processed in `object_detector`. After processing, it will store the results in `global_filter`. To prevent threading problems, the function will first lock the mutex using `pthread_mutex_lock` before it writes to `global_filter`. After writing, it immediately releases the mutex using `pthread_mutex_unlock`.

A similar mechanism is used in `color_object_detector_periodic` to read the `global_filter` struct: the function first locks the mutex, then copies global_filter to a local variable and immediately releases the lock. Note that the periodic function does not operate directly on the `global_filter` variable. The reason for this is that the video thread will hang as long as the periodic function has locked the mutex. After copying the results to a local variable, the `global_filter` is no longer necessary and can be overwritten again, which means that the mutex can be unlocked *before* further processing occurs and therefore the video thread does not need to wait as long.

In summary:

1. The video callback operates on a different thread than the rest of the autopilot. Variables that are shared between video callbacks and the rest of the code should be protected by a mutex.
2. Add a `pthread_mutex_t` object to your module and initialize this in your module's init function using `pthread_mutex_init`.
3. Before reading/writing a shared variable, lock the mutex using `pthread_mutex_lock`.
4. As soon as possible, release the mutex using `pthread_mutex_unlock`. Make a local copy of the shared variable(s) if you need more time to process them.

1.4 ABI: publish/subscribe messaging between modules

The simplest way to communicate *between different modules* is to share global variables between them. In more complex cases, however, the ABI messaging system provides an alternative to global variables. The ABI system allows modules to publish or subscribe to messages. Compared to global variables, this has the advantage that the modules do not need to know about each other – only about the message type – which simplifies code maintenance and makes it easier to swap out different modules, for instance different IMU or sonar drivers.

An example of ABI messaging can be found in the colored object detector (`sw/airborne/modules/computer_vision/cv_de` and the orange avoider (`sw/airborne/modules/orange_avoider/orange_avoider.c`). In this example, the colored object detector acts as publisher and the orange avoider as subscriber. The ABI message used in this example (`VISUAL_DETECTION`) is defined in `conf/abi.xml`, the sender ID (`COLOR_OBJECT_DETECTION1_ID`) in `sw/airborne/subsystems/abi_sender_ids.h`:

`conf/abi.xml`:

```
<protocol>
  <msg_class name="airborne">
    ...
    <message name="VISUAL_DETECTION" id="27">
      <field name="pixel_x"      type="int16_t">Center pixel X</field>
      <field name="pixel_y"      type="int16_t">Center pixel Y</field>
      <field name="pixel_width"  type="int16_t">Width in pixels</field>
      <field name="pixel_height" type="int16_t">Height in pixels</field>
      <field name="quality"      type="int32_t">Detection quality</field>
      <field name="extra"        type="int16_t">Extra field for options ...</field>
    </message>
    ...
  </msg_class>
</protocol>
```

`sw/airborne/subsystems/abi_sender_ids.h`:

```
...

/*
 * VISUAL_DETECTION communication (message 27)
*/
#ifndef COLOR_OBJECT_DETECTION1_ID
#define COLOR_OBJECT_DETECTION1_ID 1
#endif

#ifndef COLOR_OBJECT_DETECTION2_ID
#define COLOR_OBJECT_DETECTION2_ID 2
#endif

...
```

`sw/airborne/modules/computer_vision/cv_detect_color_object.c`:

```
#include "subsystems/abi.h"

...

void color_object_detector_periodic(void) {
   ...
   AbiSendMsgVISUAL_DETECTION(COLOR_OBJECT_DETECTION1_ID,
         local_filters[0].x_c, local_filters[0].y_c,
         0, 0, local_filters[0].color_count, 0);
   ...
}
```

`sw/airborne/modules/orange_avoider/orange_avoider.c`

```
#include "subsystems/abi.h"

...

#ifndef ORANGE_AVOIDER_VISUAL_DETECTION_ID
#define ORANGE_AVOIDER_VISUAL_DETECTION_ID ABI_BROADCAST
#endif
static abi_event color_detection_ev;
static void color_detection_cb(uint8_t sender_id, ...) { ... }
```

```
void orange_avoider_init(void) {
   ...
   AbiBindMsgVISUAL_DETECTION(ORANGE_AVOIDER_VISUAL_DETECTION_ID,
         &color_detection_ev, color_detection_cb);
}
```

To set up communication over ABI, you should first define your message in `conf/abi.xml`. Look at the existing messages to see how you should formulate your own, then make sure you set the `id` to a value that is not in use yet. You should also modify `sw/airborne/subsystems/abi_sender_ids.h` to add one or more sender ID numbers, these are used to identify the source of the messages. Sender ID's should be unique within a message type.

Publishing messages is pretty straightforward: include the `subsystems/abi.h` header, then use `AbiSendMsg<YOUR_MESSAGE ...)` to publish a message. The first argument of this function is the sender ID, the other arguments are the fields of your message as defined in `conf/abi.xml`.

Subscribing to messages is similar to subscribing to video frames. Include the `subsystems/abi.h` header to get access to the subscribe functions. Then, use `AbiBindMsg<YOUR_MESSAGE>(sender_id, ev, cb)` to subscribe to this message class. AbiBindMsg takes the following arguments: a `sender_id`, which is used to filter messages by their source ID. Set this sender_id to `ABI_BROADCAST` to accept messages from all senders. The second argument is a pointer to an `abi_event` that you declare somewhere in your code (here `color_detection_ev`). Finally, the third argument is a pointer to your callback function which gets called whenever a new message is published. The callback function takes as arguments the `sender_id`, followed by the fields defined for this message class.

ABI callbacks run in the thread that called `AbiSendMsg`, which in practice should be the autopilot thread (do *not* call AbiSendMsg from the video thread!). This means that you do not need to use mutexes to get/set global variables from the callback, but also that you should take care that your callback does not take too much time.

1.5 Getting the drone's state

At some point in your module, you may want to get the current state of the drone. An example can be found in `sw/airborne/modules/orange_avoider/orange_avoider.c`, where the drone's position and heading are used to update the location of the next waypoint:

```
#include "state.h"

...

static uint8_t calculateForwards(struct EnuCoor_i *new_coor, float distanceMeters)
{
  float heading  = stateGetNedToBodyEulers_f()->psi;

  // Now determine where to place the waypoint you want to go to
  new_coor->x = stateGetPositionEnu_i()->x +
      POS_BFP_OF_REAL(sinf(heading) * (distanceMeters));
  new_coor->y = stateGetPositionEnu_i()->y +
      POS_BFP_OF_REAL(cosf(heading) * (distanceMeters));
  VERBOSE_PRINT("Calculated %f m forward position. x: %f  y: %f "
                "based on pos(%f, %f) and heading(%f)\n", distanceMeters,
                POS_FLOAT_OF_BFP(new_coor->x), POS_FLOAT_OF_BFP(new_coor->y),
                stateGetPositionEnu_f()->x, stateGetPositionEnu_f()->y,
                DegOfRad(heading));
  return false;
}

...
```

To get the drone's current state, include the `state.h` header. Then, the state can be accessed using the `stateGet...` functions. Scroll through the `state.h` header to get an idea of the values you can access.

State.h uses the following naming conventions: functions ending in `_f` return the drone's state in a floating-point format (recommended). These are the easiest to work with, but can be more computationally intensive on small microcontrollers. Functions ending in `_i` return the current state in a fixed-point format. The `_BFP_OF_REAL` and `_FLOAT_OF_BFP` macros in `sw/airborne/math/pprz_algebra_int.h` allow you to convert data between these formats.

The drone's state can be expressed in different coordinate frames. For this course, the North-East-Down (`Ned`) and East-North-Up (`Enu`) frames are likely the most relevant. Positions in these frames are expressed relative to the flight plan origin, which for the example flight plans lies in the center of the Cyberzoo. Angular rates and Euler angles are expressed in the body frame, which follows a Front-Right-Down convention.

1.6 Navigation and guidance commands

To avoid obstacles, your module will at some point need to control where the drone is going. Depending on the level of control you want there are multiple ways to achieve this.

1.6.1 NAV mode

The flight plan lets you describe an entire flight or mission and lets you give high-level commands to the drone, such as waypoint sequences to follow. The flight plan by itself does not give you fine-grained enough control to avoid obstacles, and will therefore not be discussed in further detail here.

You can, however, interact with the flight plan and waypoints from within your own module, which is the method used in `sw/airborne/modules/orange_avoider/orange_avoider.c`. Consider the following flight plan and code extracts:

`conf/flight_plans/tudelft/course2019_orangeavoid_cyberzoo.xml`:

```
<flight_plan ...>
   ...
   <waypoints>
      ...
      <waypoint name="GOAL" x="1.9" y="1.0"/>
      ...
   </waypoints>
   ...
   <blocks>
      ...
      <block name="START" ...>
         <call_once fun="NavSetWaypointHere(WP_GOAL)"/>
         <stay wp="GOAL"/>
      </block>
   </blocks>
</flight_plan>
```

`sw/airborne/modules/orange_avoider/orange_avoider.c`:

```
#include "firmwares/rotorcraft/navigation.h"

...

uint8_t moveWaypointForward(uint8_t waypoint, float distanceMeters);
uint8_t moveWaypoint(uint8_t waypoint, struct EnuCoor_i *new_coor);
```

```
...

void orange_avoider_periodic(void) {
   /* State machine logic left out for this example */
   moveWaypointForward(WP_GOAL, moveDistance);
   ...
}

uint8_t moveWaypoint(uint8_t waypoint, struct EnuCoor_i *new_coor)
{
  waypoint_set_xy_i(waypoint, new_coor->x, new_coor->y);
  return false;
}

uint8_t moveWaypointForward(uint8_t waypoint, float distanceMeters)
{
  struct EnuCoor_i new_coor;
  calculateForwards(&new_coor, distanceMeters);
  moveWaypoint(waypoint, &new_coor);
  return false;
}
```

In this example, the flight plan does not seem to do much: in the `START` block, it first places the `GOAL` waypoint at the drone's current position, then stays at this waypoint. Indeed, without the module, the drone would just stay in place. The avoidance behavior in this case comes from the orange_avoider module. In its periodic function, it will move the GOAL waypoint to a new location. Because the drone is commanded to stay at this waypoint, it will move to follow it.

Using the flight plan and waypoints works, but has its limitations. While you can control where the drone is going, it is not possible to control the drone's velocities. Furthermore, this method assumes that the drone's position is fully known, which may not always be the case such as when flying without Optitrack.

1.6.2 GUIDED mode

As an alternative, the drone can be controlled in GUIDED mode. In this mode, your module can directly send setpoints to the guidance controller, including setpoints for the drone's velocity. An example is provided in `sw/airborne/modules/orange_avoider/orange_avoider_guided.c` and the corresponding flight plan:

`conf/flight_plans/tudelft/course2019_orangeavoid_cyberzoo_guided.xml`:

```
<flight_plan ...>
   <header>
      ...
      inline void setNav(void) {
         autopilot_mode_auto2 = AP_MODE_NAV;
         autopilot_static_set_mode(AP_MODE_NAV);
      }
      inline void setGuided(void) {
         autopilot_mode_auto2 = AP_MODE_GUIDED;
         autopilot_static_set_mode(AP_MODE_GUIDED);
      }
   </header>
   ...
   <blocks>
      ...
      <block name="START" ...>
         <call_once fun="setGuided()"/>
         <stay wp="STDBY"/>
```

```
      </block>
      <block name="STOP">
         <call_once fun="NavSetWaypointHere(WP_STDBY)"/>
         <call_once fun="setNav()"/>
         <stay wp="STBDY"/>
      </block>
      ...
      <block name="Land here" ...>
         <call_once fun="setNav()"/>
         ...
      </block>
   </blocks>
</flight_plan>
```

`sw/airborne/modules/orange_avoider/orange_avoider_guided.c`:

```
#include "firmwares/rotorcraft/guidance/guidance_h.h"

...

void orange_avoider_guided_periodic(void) {
   if (guidance_h.mode != GUIDANCE_H_MODE_GUIDED) {
      return;
   }

   /* State machine logic left out for this example */
   guidance_h_set_guided_body_vel(speed_sp, 0);
   ...
}
```

In this example, the module uses `guidance_h_set_guided_body_vel` to send a velocity setpoint to the guidance controller. However, when the drone is in NAV mode, it will ignore these commands and follow the navigation commands from the flight plan instead.

To switch the autopilot to GUIDED mode, two extra functions were added in the flight plan header: `setNav` and `setGuided`. Calling these functions sets the autopilot mode to NAV or GUIDED, respectively. This is exactly what happens in the `START` block: the drone is commanded to stay at the STDBY waypoint, but just before that the autopilot is switched to GUIDED mode by calling `setGuided`. As a result, the drone will ignore the stay command and follow the module's setpoints instead. In the `STOP` block, the autopilot is switched back to NAV mode and stays at the STDBY waypoint.

You should take special care that the drone is in the correct mode at the start of each block. For instance, forgetting to switch the drone back from GUIDED mode to NAV in the `Land here` block would cause it to ignore the landing command, even when the landing was triggered by a safety exception! It is therefore best practice to call the `setNav` or `setGuided` function at the start of each block to ensure that the autopilot is in the correct mode.

1.6.3 MODULE mode

For completeness, it should be mentioned that your module can also *replace* the entire guidance or even stabilization controller. This can be achieved by switching the autopilot to MODULE mode. In module mode, the autopilot will call the `guidance_h_module_run` and `guidance_v_module_run` functions during its main loop, your module should provide these functions.

Two examples of MODULE mode controllers are provided in `sw/airborne/modules/ctrl/ctrl_module_innerloop_demo.` and `_outerloop_demo.c`. In MODULE mode *you* are responsible for the drone's stability, therefore we recommend you to stick to NAV or GUIDED mode during the course.

1.7 Defines and settings

Your module will most likely contain tunable parameters, such as the minimum and maximum YUV values for which a pixel is considered an obstacle. While you can write these numbers directly in your code, this will make it difficult to tune them later. Paparazzi provides two systems to simplify parameter tuning: defines and settings.

Defines allow you to set constant values from the airframe file. See, for example, the following abstract of the `bebop_course2019_orangeavoid.xml` airframe:

```
<airframe ...>
   <firmware ...>
      <target name="ap" board="bebop">
         <define name="COLOR_OBJECT_DETECTOR_LUM_MIN1" value="40"/>
         ...
      </target>
      ...
      <define name="ARRIVED_AT_WAYPOINT" value="0.5"/>
      ...
      <module name="cv_detect_color_object">
         <define name="COLOR_OBJECT_DETECTOR_CAMERA1" value="front_camera"/>
         ...
      </module>
   </firmware>
   ...
   <section name="GUIDANCE_H" prefix="GUIDANCE_H_">
      <define name="CLIMB_VSPEED" value="1.0"/>
   </section>
   ...
</airframe>
```

As you can see, defines can be set at multiple places in the airframe file. The behavior is mostly the same in these cases, with the following exceptions:

- Defines placed in the `<target>` elements are only set when the autopilot is built for that target, i.e. `"ap"` for the real drone and `"nps"` for the simulator. This allows you to, for instance, use different color filter settings on the real and simulated drone.
- Placing a define inside a `<module>` element has no special effect! The define is also visible in other modules, so be sure to use a unique name. Typically, defines are prefixed with the name of the module (e.g. `COLOR_OBJECT_DETECTOR_`) to make them unique. The only reason these defines are placed inside the module element is to improve readability.
- `<section>` elements allow you to specify a `prefix`, this prefix is placed in front of all define names inside this section. In the example, the `CLIMB_VSPEED` define is available in the code as `GUIDANCE_H_CLIMB_VSPEED`.

During compilation, these defines are turned into preprocessor macros and can be referred to directly from your code.

Airframe defines allow you to set constant parameters at compile-time, but in some cases it would be easier if you could change these values during the flight. This is possible with the 'settings' system. Settings are defined in the module xml file. Take for example `conf/modules/cv_detect_color_orange.xml`:

```
<module name="cv_detect_color_object" ...>
   ...
   <settings>
      <dl_settings name="ColorObjectDetector">
         <dl_setting var="cod_lum_min1" min="0" step="1" max="255" shortname="y_min1"/>
         ...
      </dl_settings>
   </settings>
```

```
</module>
```

Settings listed in the module xml can be tuned from the Ground Control station by going to the 'Settings' tab and then selecting the tab belonging to your module, as defined in the `dl_settings` element (here `ColorObjectDetector`). To read the current value of a parameter from the drone, click its value (the number) in the GCS. Te set a value on the drone, adjust the slider, *then click the green checkmark* to upload this new value to the drone. Click the value number again to make sure the setting was updated.

Use the `dl_setting` element in your module xml to add a setting to your module. The `var` attribute specifies the variable this setting should be written to; this variable should be globally accessible. The `min`, `step` and `max` attributes let you specify a range of possible values for this setting. Finally, using `shortname` you can control the name under which this setting is listed in the GCS.

It is possible to combine the define and settings mechanisms, where the define provides a default value that can be adjusted later using settings. This often uses the following pattern:

```
#ifndef MY_DEFINE
#define MY_DEFINE 0
#endif
int my_setting = MY_DEFINE;
```

In this example, `MY_DEFINE` provides the initial value of `my_setting`. MY_DEFINE can be set from the airframe file, but if it is not defined there this code will give it a default value of 0. The actual parameter is stored in `my_setting`, for which a `<dl_setting>` element is included in the module's xml file.

2 Logging

2.1 Printing to terminal

2.2 Telemetry

2.3 Video streaming

2.4 CSV file logging

2.5 Video capture

2.6 Downloading log files through FTP

3 Testing

3.1 Offline development with python and cv2

OpenCV available in Paparazzi and in Python. Python – especially IPython (Interactive Python) – might be easier for prototyping. Spyder3.

Bebop: opencv 3.2.0.

3.2 Testing in simulation

World and model files.

Keep in mind: simulation not perfect. Camera colors will be different, camera calibration too.

3.3 Testing on the real drone

4 Collaboration using git

5 Where to find more information

For a *high-level overview* of Paparazzi and descriptions of file types such as the *flight plan xml* or *module xml*'s, refer to the Paparazzi wiki at `https://wiki.paparazziuav.org`.

For more in-depth descriptions of *existing modules* and their defines, check out the auto-generated documentation at `http://docs.paparazziuav.org/latest/`. The content of these pages is generated directly from the module .xml files, which can be found in the `paparazzi/conf/modules` directory.

If you are experiencing *issues with Paparazzi that are not caused by your own code*, check out the issue pages on Github (`https://github.com/paparazzi/paparazzi/issues`, `https://github.com/tudelft/paparazzi/issues`) to see if someone else is running into the same problem and if there is a temporary solution or fix. **If you want to create a new issue, please do so on the tudelft fork of Paparazzi** (`https://github.com/tudelft/paparazzi/issues`).

If you really want to know *what goes on inside the autopilot*, you can't get more detailed information than the source code itself! All of the autopilot code can be found in the `paparazzi/sw/airborne/` directory. The following files and directories are most relevant for the course:

- `paparazzi/sw/airborne/`
  - `firmwares/rotorcraft/` - *This folder contains the autopilot code that is shared between all types of rotorcraft. This includes guidance and stabilization code.*
    - `guidance/` - *Here you can find the guidance controllers. The controller is split into a horizontal part (guidance_h) and a vertical part (guidance_v).*
    - `stabilization/` - *This folder contains the stabilization controllers. The example airframe uses the stabilization_indi_simple controller.*
    - `autopilot_guided.h` - *This header allows you to set position and velocity setpoints while flying in GUIDED mode.* **TODO should probably be guidance_h.h!**
    - `main.c` - *If you were wondering if Paparazzi also has a 'main' function, here it is. After initializing the autopilot, the main function will loop indefinitely while performing its periodic tasks.*
    - `navigation.h` - *These functions are commonly called from the flight plan.*
  - `math/` - *Here you can find math functions for matrix and vector operations in Paparazzi. These are documented in paparazzi/doc/pprz_algebra/headfile.pdf. Pprz_algebra_float.h might be a good starting point for general-purpose floating point calculations.*
  - `modules/` - *Here you find the code for modules that other people have written. Be sure to check out how these modules are written if you're looking for inspiration.*

    * `computer_vision/` - *This folder contains the computer vision code. Use the functions in cv.h to register your video callback.*

  - `state.h` - *Use this header to read the current state of the UAV. Use the stateGet... functions.*

The autopilot code is spread over a large number of files. Use Eclipse to navigate these files effectively. Hovering over a variable or function will show its definition and hopefully some descriptive comments. Ctrl+clicking on a variable will take you to its definition or declaration. Ctrl+clicking will also allow you to quickly open header files, then use Ctrl+Tab to open the corresponding source file. If 'Mark Occurrences' (Shift+Alt+O) is turned on, placing your text cursor on the variable name will highlight all occurrences in the file you have opened. To see how a variable is used across files, right click it and choose 'References → Project'. Finally, 'Search' (Ctrl+H) allows you to search arbitrary text in all files. Set 'Scope' to 'Selected resources' to limit your search to the folder you have selected in the Project Explorer.

If your question is not answered by these resources or if you don't know where to look, feel free to ask the TA's for advice.