PORTING ARDUPILOT TO ESP32: TOWARDS A UNIVERSAL OPEN-SOURCE ARCHITECTURE FOR AGILE AND EASILY REPLICABLE MULTI-DOMAINS MAPPING ROBOTS

L. Beaudoin[1, 2, *], L. Avanthey[1, 2], C. Villard[1]

[1] SEAL Research Team (Sense, Explore, Analyse and Learn), ÉPITA, 94270 Le Kremlin Bicêtre, France,
[2] Équipe Acquisition et Traitement, IGN, 94165 Saint-Mandé, France
(laurent.beaudoin, loica.avanthey, charles.villard)@epita.fr

KEY WORDS: Ardupilot, ESP32, Multi-domains exploration robots, UAV, UGV, USV, UUV, Close-range remote sensing

ABSTRACT:

In this article, we are interested in the implementation of an open-source low-level architecture (critical system) adapted to agile and easily replicable close-range remote sensing robots operating in multiple evolution domains. After reviewing the existing autopilots responding to these needs, we discuss the available hardware solutions and their limits. Then, we propose an original solution (software and hardware) that we developed to obtain a universal low-level architecture for all our exploration robots, whatever their environment of evolution, and the steps needed to make it run on our chosen family of micro-controllers: the ESP32. Finally, we present the operational results obtained on our different platforms (land, surface, submarine and air), their limits and the envisaged perspectives.

1. INTRODUCTION

In close-range remote sensing, whatever the domain of evolution, the vector which displaces the payload holds a critical place. Indeed, a successful data acquisition campaign, and therefore the quality of the work which results from it, supposes that the study area is explored in a systematic and complete way with control of the redundancy. However, in a natural environment, reaching all these criteria manually is difficult, if not impossible. Automated robotic platforms make it possible to achieve this high level of systematization necessary for the precision of these studies. In this article, we focus in particular on agile and easily replicable platforms, favoring open-source solutions whenever possible, because they are easy to share, evolve or maintain over time and are fully transparent.

The construction of the platform itself (the mechanical part) is simplified by the evolution and accessibility of rapid prototyping tools such as 3D printers or numerically controlled cutting machines (CNC). Concerning the electronic and on-board computing part of the vector, we distinguish two architectures (Catsoulis, 2006, Siciliano, 2008): one low level (close to the hardware) and the other high level. The first one manages the control / command part (including control loops for attitude, direction, position, speed, etc.). Safety and security tasks are also mainly managed at this level. So, this critical level should never fail. That is the reason why its hardware architecture is mainly based on micro-controllers. As for the second, it usually takes care of everything that involves making decisions or processing more complex information to realize the objectives of the mission. To deal with these complex tasks, the hardware architectures of this hight level is close to classical computer (ARM or X86) and frameworks like Robot Operating System (ROS) are used as a software solution at this level. The figure 1 describes this general organization of an exploring robot, whatever its domain of evolution. The separation of tasks managed by one or the other of these architectures is nowadays becoming increasingly blurred because micro-controllers become enough powerful to managed some high-level task and conversely, high-level units natively incorporate more capacities to communicate with low-level sensors. But a secure system need both architectures.

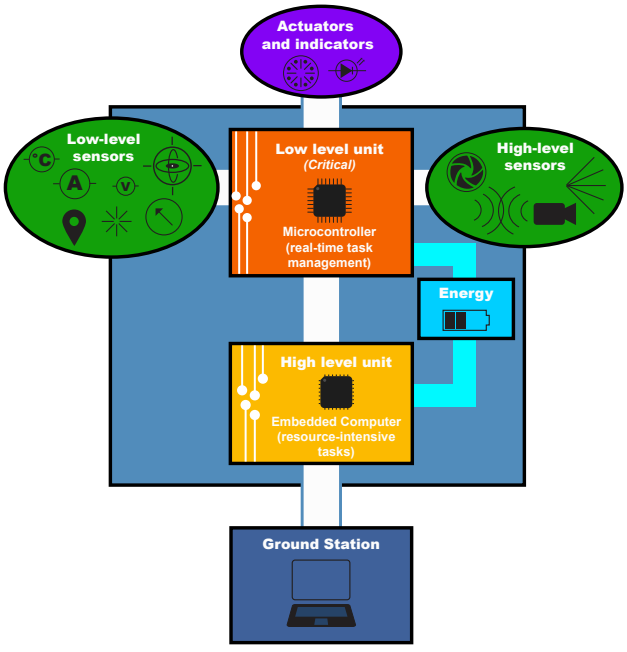

Figure 1. General organization of an exploration robot, whatever its environment of evolution (air, land, sea), with the low-level critical architecture and the high level resource-intensive architecture.

There are open-source solutions for these two architectures, but they are not always easy to implement depending on the evolution environment (air / land / sea) and the specificity of the mission to achieve. The task is even more complex when there are strong physical constraints for the robot. Indeed a light and

* Corresponding author

agile vector implies a very compact size and limited resources both in computing power and in energy. Meanwhile an easy replicability implies strong constraints on the cost of the solution.

In our field, we are interested in the exploration of various environments from the air, the ground, the surface of the water or underwater. The diversity of the required platforms poses several problems. Among them: how to make robots more reliable while limiting the time dedicated to maintenance, or ensuring that there is no hardware or system incompatibility for communication and interaction between these heterogeneous robots for future collaborative missions? Therefore, developing universal high and low level architectures for all these platforms seems to be an adequate solution to solve these problems.

Due to its strong dependency with the hardware layer, the low level architecture is the most impacted by the specificity of the platform. This low-level architecture was initially called in the literature a "flight controller" and is now known as an "autopilot" with the addition of functionalities allowing different levels of autonomy. It is on this low level unit that we will focus in this article.

1.1 Open-source autopilots for diverse environments

The choice of open-source is important, because the autopilots thus designed can benefit from all the support of the community to integrate and propose the most efficient algorithms from the literature. It is therefore the ideal choice for research work.

Within this category, many autopilots are exclusively developed for the air domain (Chao et al., 2010, Colomina, Molina, 2014), providing a diversified offer for the different types of flying vehicles (rotary wings like multicopters or helicopters, fixed wings like aircrafts, delta wings or gliders, etc.). To give a brief historical review of these main projects, one of the oldest autopilots is Paparazzi, created in 2003. OpenPilot is a competing project born in 2009 which gave birth to TauLabs (since 2012, which is rather intended for professionals and researchers), LibrePilot (since 2015) and dRonin (since 2015). On the other hand, BaseFlight / MultiWii (2013) gave birth to CleanFlight (since 2014) and then BetaFlight (since 2016), these latter projects being more oriented to perform drone races and acrobatics. Finally, INav, created in 2016 from BetaFlight, is more oriented towards the autonomous navigation of drones.

However, we are also interested in terrestrial and aquatic environments. So, most of the previous solutions are too much specialized in aerial evolution to be used in these other environments. In this regard, essentially two open-source autopilots offer this diversity and are also the most popular in the community among all those mentioned above: Ardupilot (since 2009) and PX4 (since 2012). These two autopilots have a similar architecture and some developers are working on both projects at the same time, although we can note that Ardupilot is slightly more advanced than PX4 on certain functionalities. In 2013, they joined the DroneCode initiative, a nonprofit organization governed by the Linux Foundation which aims to encourage open-source development for UAVs (Unmanned Aerial Vehicles). This allowed these two autopilots to take full advantage of other projects that can be found in the fold of DroneCode such as MAVLink (Micro Air Vehicle Link), a communication protocol (Koubaa et al., 2019) and QGroundControl, a mission planning and monitoring software (Ramirez-Atencia, Camacho, 2018). From 2016, Ardupilot remained in the spirit of free code by keeping the GPL license, while PX4 and the other projects federated by DroneCode went into BSD license in order to market their solutions (source codes can then become proprietary codes).

A summary of all these autopilots, classified by number of contributors (which can be seen as a support indicator by the opensource community) is shown at table 1. In relation to our problems and our constraints, we choose to focus on the solution which is the most open-source and our choice therefore fall on Ardupilot. This autopilot has also proven operational in different environments as we find it in application on, of course, aerial vehicles (UAV) (Wardihani et al., 2018, Fawcett et al., 2019, Melo et al., 2017, Carlson, Rysgaard, 2018, Cucho-Padin et al., 2019, Washburn et al., 2017), but also on ground vehicles (UGV) (Velaskar et al., 2014) or surface vehicles (USV) (Sinisterra et al., 2017, Moulton et al., 2018, Raber, Schill, 2019) and even on underwater vehicles (UUV) (Schillaci et al., 2017, Luo et al., 2019, Sani et al., 2019).

Auto-pilot	Contri.	Date	Domain
(Ardupilot, Source Code)	474	2009	air+land+sea
(PX4, Source Code)	375	2012	air+land+sea
(BetaFlight, Source Code)	373	2016	air (races)
(CleanFlight, Source Code)	308	2014	air (races)
(iNav, Source Code)	246	2016	air
(dRonin, Source Code)	108	2015	air (races)
(Paparazzi, Source Code)	101	2003	air
(TauLabs, Source Code)	90	2012	air
(LibrePilot, Source Code)	72	2015	air

Table 1. Popular open-source autopilots ranked by number of contributors as a support indicator by the opensource community.

1.2 Hardware solutions for Ardupilot

The most common off-the-shelf hardware solution for running Ardupilot is the Pixhawk. It comes in the form of a box with dedicated connectors for peripherals (sensors, motors, etc.). Its main advantage is that it is open-hardware. However, when we consider the case of agile and easily replicable robots for multi-domain mapping, it has some disadvantages. Indeed, with regard to these criteria, its size and cost are not negligible (for the Pixhawk 4: $44 \times 84 \times 12$mm and an average price of 180-220$) and its modularity is not complete (impossible to change the integrated IMU or barometer in the event of failure or if more efficient sensors are available).

To overcome these drawbacks, making a new complete custom board like the Pixhawk one is not satisfactory, since the cost remains high despite everything and the maintenance would be huge. A more appropriate solution is to start with ready-made development modules. Those boards, available at a very low cost (around 10$), provide some utilities around a micro-controller (USB to UART interface, CPU pin, reset buttons, etc.) beside providing some voltage peaks protection. Then it only remains to design an electronic support board that distributes the inputs / outputs to take full advantage of all the resources of the micro-controller and obtain functional hardware for the low-level unit. The two main families of micro-controllers powerful enough to run Ardupilot are the STM32 and the ESP32.

The STM32 family of micro-controllers is the one used on the Pixhawk and on other hardwares for Ardupilot as it is the one which is natively supported by the autopilot. They are powerfull

and efficient micro-controllers. On the other hand, for the same price, the ESP32 modules are more efficient and, since they are optimized for IoT, they consume much less energy. They also have a built-in WiFi which is an interesting feature to easily communicate directly with the low-level. And, as they have various built-in additional calculation units, it can be usefull to introduce some security encryption over MAVLink protocol to secure the communication link with the ground station (a functionality under development) without adding an additional electronic module.

Our choice therefore fall on the ESP32. All the sensors can be changed and upgraded easily, the module itself can be quickly changed in the event of failure at minimal cost, including during a field mission, and the simplicity of the support board will limit possible failures. However, the ESP32 family of micro-controllers, unlike the STM32 family, is not supported by Ardupilot, and this will require porting the code of the autopilot.

2. PORTING ARDUPILOT TO ESP32

2.1 Ardupilot: overview

The Ardupilot embedded software suite offers a set of solutions to be able to use the chosen vector with a variable degree of autonomy, from assisted steering to automatic navigation, provided if sensors capable of acquiring the necessary information for the different algorithms to operate (attitude, altitude / depth, absolute position, speed, distance, etc.) are available (Audronis, 2017). To allow its portability on several types of boards (hardware) and to adapt to a great diversity of robotic architectures (vectors), Ardupilot is based on different layers of abstraction. As shown in figure 2, there are three particularly important layers of abstraction: one for the hardware, one for the sensors and the last one for the kind of vehicle (and therefore the robot dynamics).

The hardware abstraction allows independence from specifics of the board on which Ardupilot runs. The link between physical material and abstractions is described in a module: the HAL (Hardware Abstraction Layer). Each type of board therefore has its own HAL. But what makes Ardupilot so convenient is that this layer of abstraction makes all the other layers completely independent of physical material. In fact, they exclusively use the abstractions defined by the HAL to interact with the peripherals and memories. In other words, this mode of operation is similar to a request system. For example, to recover the data measured by a sensor, the driver implements the sensor logic and the way of using it. Then it sends the commands to the HAL which implement the underlying protocols like I^2C, SPI or UART using hardware module specific to the board. So, only the HAL knows how to communicate with the hardware and the other higher modules of Ardupilot can continue to work the same way whatever the board used.

The variability of the sensors that can be connected to specific hardware is also managed at the HAL layer. We can distinguish two types of sensors: those that are integrated into the board and those that can be added. The former are critical: without them there is no possible action for the robot. For example, to make a drone flying, at least an AHRS (Attitude and Heading Reference System) and a barometer are needed. These critical sensors are meant to be natively implemented on the board. The user can calibrate and configure them but doesn't need to care about how they are connected to the board. The other sensors that the user can choose to connect or not are considered external: Ardupilot scans the different communication ports at startup to establish the list of sensors actually available by recognizing them with their identifiers. A lot of sensors are natively supported by Ardupilot.

The second level of abstraction concerns the sensors. It allows within the control algorithms to focus on the information rather than on the way in which this was acquired. For example, the EKF (Extended Kalman Filter) uses the concept of absolute position, whether it comes from a GPS, or from a SLAM algorithm or based on a LIDAR or a camera sensor. This organization helps to give great flexibility in relation to the different domains of use, because many different sensors may be used to obtain the same information whether we are in air or in water for example.

The third level of abstraction relates to the frame and makes it possible to define the physical characteristics of the platform according to its predefined type (fixed wing, rotary wing, holonomic or non-holonomic surface vehicle, underwater vehicle, etc.), to describe its specificities (number of motors, their position relative to the center of gravity, their direction of action, etc.) and offers a chain of basic modules adapted to this set as well as the corresponding safety and security procedures (returning to the base or landing when the signal from the ground station is lost, for example).

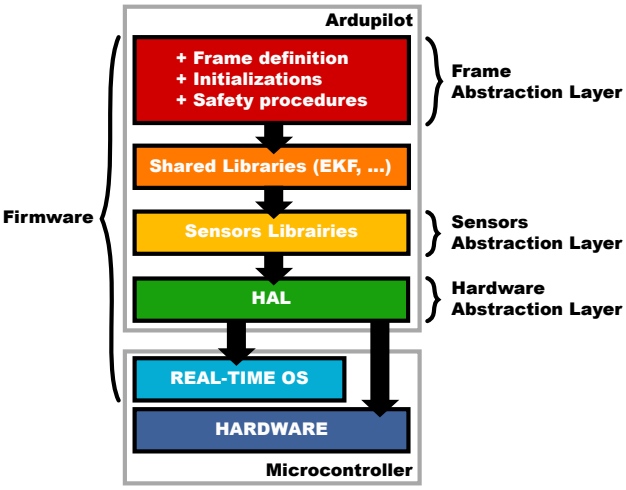

Figure 2. General overview of Ardupilot and its three main layers of abstraction.

Finally, to complete this overview, Ardupilot operates on top of the minimalist on-board operating system (OS) which is responsible for accessing the functionalities of the micro-controller and managing its operation. The OS of micro-controllers are real time OS, that is to say that they manage time very strictly: indeed, the maximum time to perform an action from its launch (*jitter*) and the scheduling of these actions are precisely determined. Ardupilot and the real-time OS are compiled in a single firmware which forms the program integrated into the micro-controller. This process and the changes needed for it to work on ESP32 is described in section 2.3.

2.2 Porting HAL on ESP32

We saw in the introduction that Ardupilot was developed for the family of STM32 micro-controllers. Those embed the ChibiOS real-time operating system. Our hardware solution is based on

the ESP32 family of micro-controllers that embeds the FreeRTOS real-time operating system. It will therefore be necessary to port the Ardupilot code on this new family of micro-controllers to benefit from all its functionalities. But as we saw in the previous section, thanks to the levels of abstraction, the modifications required for porting Ardupilot are all exclusively located in the HAL. In this section, besides of all the problems involving porting a program to a new micro-controller unit, we will present the two main changes that need to be addressed: the differences in hardware layout and the differences in scheduling on simple or multiple cores.

The familly of ESP32 micro-controllers is optimized for IoT applications, which explains the major architectural differences with the STM32 familly. Indeed, the needs of IoT focus on energy saving and ease of communication more than on processing speed performance. This is why the ESP32 does not have DMA (Direct Memory Access) unlike the STM32. When this device exists, the data passing through a peripheral is managed by a dedicated controller which redirects it to the main memory without the intervention of the CPU (Central Processing Unit) apart from the start and end orders of the action. On the contrary, in its absence, the entire action is taken over by the CPU, so it is therefore necessary to review all the memory accesses in the HAL to make these modifications. It increases the overall load of CPU, but as the processor of the ESP32 is powerful, that does not pose any problem concerning the speed of execution in the end.

In addition, performance in communication speed is not a necessity in IoT either, all the protocols (I2C, UART, SPI, etc.) provided by the API (Application Programming Interface) of the ESP32, are minimalist and absolutely not optimized. It is therefore necessary to re-implement them in the HAL rather than using those provided by the API. This point is particularly fundamental for flying UAVs with rotary wing where the frequency of communication with the AHRS must be very high to allow the control loop on attitude to counter their natively very unstable frames.

On the other hand, going from one to two cores involves managing the distribution of tasks via the scheduler, taking into account the workload of each cores and the priorities of each task. This part must also be recoded in the HAL. We choose to assigned by default each of the two most critical task categories, communications and main loop, to a fixed core. The other tasks are distributed on the fly according to the current load of the cores.

2.3 Compiling and building the firmware

To finalize the porting of Ardupilot, it must be compiled together with FreeRTOS to form a single firmware. The first problem comes from the different management of RAM between the two families of micro-controllers.

On a micro-controller, the Ardupilot program can not fit entirely into the RAM. Therefore, when a part of code is needed, it is loaded in a RAM cache and then executed. This cache is overwrited when it is full and another part of code needs to be executed. Those cache spaces have different access speeds. Therefore, this diversity has to be taken into account to be effective and to perform cache optimization. This operation is strongly dependant to the family of micro-controller used and the distribution on the RAM spaces of the important part of the program is specified in the binary files.

The second problem comes from the fact that the compilation tools for each of these two entities are different (*waf* for ardupilot, *cmake* for FreeRTOS). On STM32, Ardupilot code reimplements the compilation of ChibiOS in an integrated manner. But that implies maintenance every time there is an update on ChibiOS. The alternative we have chosen is to compile the two by their respective toolchain in static libraries and then link them together in a single final binary to create the firmware to flash on the ESP32.

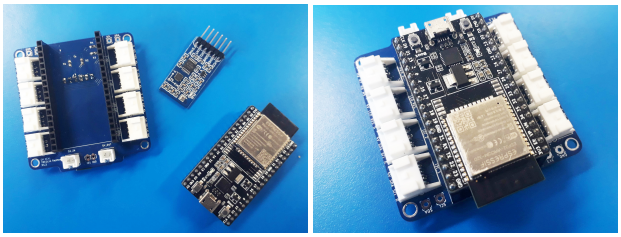

Figure 3. On the left, the internal elements of our low-level unit (the custom support PCB, the integrated IMU and the ESP32 board). On the right: our low-level unit assembled.

3. RESULTS

The previous section presented how to port the Ardupilot software to a new range of micro-controllers and what problems are posed by this port. It is recalled that the objective is to obtain a universal solution for the low level critical architecture suitable for all types of platforms dedicated to close-range remote sensing, whatever the environment of evolution (aerial, terrestrial, surface or underwater).

To demonstrate the universality of our solution, we therefore worked on different exploration robots developed by our team to cover all of the evolution environments. All these robots have the same low level unit (Ardupilot on ESP32, see figure 3), the particularities of each robot (the number and orientation of motors, their orientation, the friction of the frame in the environment, the available sensors, etc.) being managed at the platform abstraction layer as we saw in section 2.1. First, we will present the different platforms tested, then the first results obtained in terms of autonomy for each of these robots and finally the current limits of our solution and the perspectives envisaged to push them back.

3.1 The different close-range remote sensing platforms used

3.1.1 The Mabouya land vehicle This terrestrial robot was developed as part of an international robotics competition, the European Robotics League competition, and more specifically for the Emergency round which takes place on the NATO base in La Spezia in Italy. The scenario of this round is based on an industrial disaster: a boat crashed into port facilities, causing an explosion. Two fully autonomous robots are then responsible for collaborating to establish a map of underwater and land damages. The Mabouya vehicle (figure 4) is dedicated to exploring the land domain. It is a 6-wheeled vehicle, rather compact ($35 \times 35 \times 60$cm), which gives it the ability to squeeze in everywhere, but penalizes it for crossing some obstacles (such as steps for example). This robot can carry 3Kg of payload for an empty weight of 4Kg and reach a mean speed of 20Km/h.

3.1.2 The Ryujin underwater vehicle Ryujin (figure 5) is a hybrid underwater micro-robot, that is to say either controllable

Figure 4. Our land vehicle: Mabouya.

(ROV) or completely autonomous (AUV). It was developed from mid-2010 as part of a research work to map the shallow seabed (<100 m) in three dimensions. It measures $20 \times 20 \times 30$cm for a weight of 10Kg. Very reactive, it is particularly agile. We changed its internal architecture by the new one described in this article. It participated with the land robot Mabouya in the ERL competition for the underwater part in autonomy.

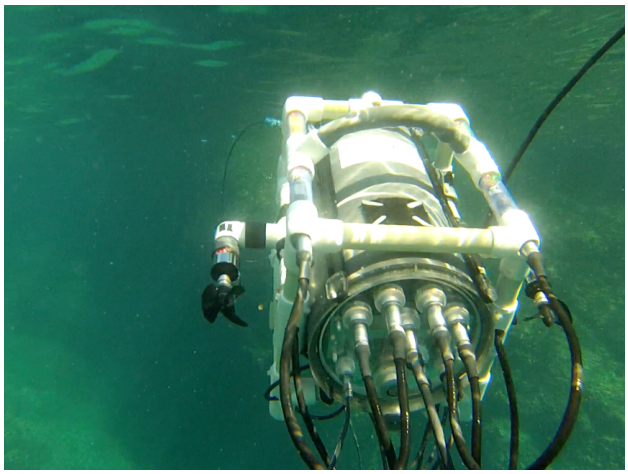

Figure 5. Our underwater vehicle: Ryujin.

3.1.3 The Kraken surface vehicle This surface vehicle is used to map the very shallow seabed (<5m) and to serve as a surface relay for the underwater robot. This latter ability significantly increases the operational reach of Ryujin. Kraken (figure 7) is a catamaran type and measures $100 \times 80 \times 40$cm for a weight of 8Kg. This relative compact size and low weight makes it very easy to transport and deploy in the field.

3.1.4 The Kiwi aerial vehicle The goal of this project under development is to specialize a quadricopter with current technologies to perform mapping tasks (mosaics or 3D models) of emerged land as part of various studies (biodiversity, archeology, etc.). The objective is to move towards a solution as autonomous as possible using light means, both on the vector and on the sensor. Its size is 50cm in diameter for a weight of around 2Kg with its payload included (figure 7).

Figure 6. Our surface vehicle: Kraken.

Figure 7. Our aerial vehicle: Kiwi.

3.2 Autonomy achieved by the solution for each platform

The qualification of the results obtained is done by measuring the maximum level of operational autonomy achieved on the heterogeneous platforms presented in the previous section. Whatever the environment of evolution, (Avanthey, 2016) classifies the operational capacities of autonomy of an exploration robot in five levels (see figure 8): assisted piloting for level I, instructions holding (the pilot gives high level orders) for level II, trajectories planning with mission plans defined a priori for level III, possibility of automatically avoiding unforeseen obstacles during the completion of the course (tactical decision-making) for level IV and re-planning the mission plan on the fly (strategic decision-making: choice of exploration paths, change of priority of objectives, management of data completeness, management of failures, etc.) for level V.

Here is a summary of the first operational results achieved. The land robot and the surface catamaran were tested up to level III (navigator) without difficulty. There is no particular reason why these platforms should not quickly reach level IV (responsive), once equipped with the appropriate sensors. However, the situation is more complicated for aerial and underwater robots.

For the aerial drone, we have so far not been able to exceed level I (assisted control). Indeed, to reach the following levels,

it is essential to have fast, reliable and fluid communication with critical sensors such as the AHRS. However, as we saw in section 2.2, it is necessary to recode the communication protocols for the ESP32 but the current version of our implementation is not yet sufficiently optimized for the frequencies of use required by a flying drone (of the order of several hundred Hz). Once this problem is over, as with terrestrial and surface robots, there is no reason not to easily gain access to level IV autonomy on this type of platform.

For the underwater drone, and we can expect this to be the case for all indoor drones, level II (holding of instructions) is easily reached. But to reach level III (trajectory planning), it would be necessary to be able to obtain an absolute position so that the algorithms can follow the trajectory defined a priori. However, underwater or indoor, there is no GPS information available. Another solution must therefore be found to obtain equivalent information. This could be obtained, for example, from higher-level solutions such as those from SLAM. However, these algorithms can only run on more sophisticated hardware (computer) and software (like ROS) architectures offering more resources. We reach here the limits of the low level. The problem that can then arise is that of the interface between the high and low level layer. To facilitate this interface, the community encourages the use of the MAVLink protocol between these two layers.

For the same reasons, level IV (obstacle avoidance) will be difficult to reach for underwater robots because it is complicated to measure distances to objects underwater without going through high-level algorithms. On the other hand, indoor robots will not have these kind of problems because there are many easily interfaceable aerial low-level sensors specialized for this use.

Finally, for all platforms, level V (decision making and rescheduling) cannot be provided by Ardupilot. These are again resource-intensive algorithms and therefore must be implemented on the high level part. And as for level III underwater and indoor robots, a solution using ROS and MAVLink to facilitate the exchange of information between the two decision-making layers is recommended.

4. CONCLUSION

We have proposed in this article an original solution for a universal open-source low-level architecture for agile and easily replicable close-range remote sensing robots that perform in various environnements based on the strengths of existing resources. We explained why Ardupilot was a wise autopilot choice for this type of problem and we proposed a new hardware solution, the ESP32, which seems to us more suitable for modularity, price and energy consumption. We have seen what were the key points to port Ardupilot to this new hardware and what were the main difficulties posed.

Finally, we presented the results obtained on different robotic platforms covering most of the environments. We were able to show that the levels of autonomy until the maintenance of high-level instructions is easily achievable for all platforms, except for flying devices for which the optimization of the implementation of communication protocols must be further improved.

Tracking is easily obtained as soon as there is a GPS information (land and air domains) but becomes complex in other ways,

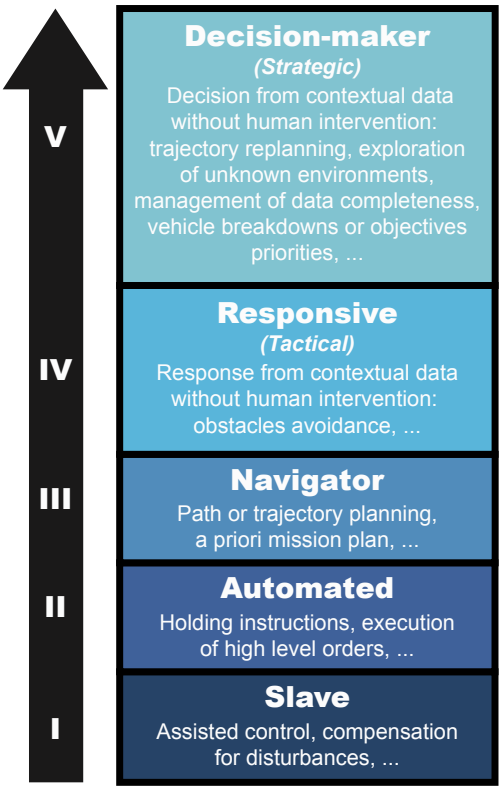

Figure 8. Levels of automation of an exploration robot.

especially for underwater robots and generally all indoor robots. However, native communication with high-level architecture would allow the problem to be circumvented by using SLAM-type algorithms.

Similarly, obstacle avoidance can be easily implemented for air and land environments, but not in the underwater environment where distance measurement is complex.

Finally, decision-making for trajectory re-planning is not included in Ardupilot and must also be managed independently on a high-level architecture.

REFERENCES

Ardupilot, Source Code. `https://github.com/ArduPilot/ardupilot`.

Audronis, T., 2017. *Designing Purpose-Built Drones for Ardupilot Pixhawk 2:1: Build drones with Ardupilot*. Packt.

Avanthey, L., 2016. Acquisition and reconstruction of dense underwater 3D data by exploration robots in shallow water. PhD thesis, Télécom ParisTech.

BetaFlight, Source Code. `https://github.com/betaflight/betaflight`.

Carlson, D. F., Rysgaard, S., 2018. Adapting Open-Source Drone Autopilots for Real-Time Iceberg Observations. *MethodsX*, 5, 1059–1072.

Catsoulis, J., 2006. *Designing Embedded Hardware*. O'Reilly.

Chao, H., Cao, Y., Chen, Y., 2010. Autopilots for Small Unmanned Aerial Vehicles: A Survey. *International Journal of Control, Automation and Systems*, 8, 36–44.

CleanFlight, Source Code. https://github.com/cleanflight/cleanflight.

Colomina, I., Molina, P., 2014. Unmanned aerial systems for photogrammetry and remote sensing: A review. *ISPRS Journal of Photogrammetry and Remote Sensing*, 92, 79–97.

Cucho-Padin, G., Loayza, H., Palacios, S., Balcazar, M., Carbajal, M., Quiroz, R., 2019. Development of Low-Cost Remote Sensing Tools and Methods for Supporting Smallholder Agriculture. *Applied Geomatics*.

dRonin, Source Code. https://github.com/d-ronin/dRonin.

Fawcett, D., Azlan, B., Hill, T. C. abd Khoon Kho, L., Bennie, J., Anderson, K., 2019. Unmanned Aerial Vehicle (UAV) derived Structure-from-Motion Photogrammetry Point Clouds for Oil Palm (Elaeis guineensis) Canopy Segmentation and Height Estimation. *International Journal of Remote Sensing*, 40(19), 7538-7560.

iNav, Source Code. https://github.com/iNavFlight/inav.

Koubaa, A., Allouch, A., Alajlan, M., Javed, Y., A., B., Khalgui, M., 2019. Micro Air Vehicle Link (MAVlink) in aNutshell: A Survey. *IEEE Access*, 7, 87658–87680.

LibrePilot, Source Code. https://github.com/librepilot/LibrePilot.

Luo, Z., Xiang, X., Zhang, Q., 2019. Autopilot System of remotely operated vehicle based on Ardupilot. *Intelligent Robotics and Applications*, 3, 206–217.

Melo, J. C., Constantino, R. G., Santos, S. G., Nascimento, T. P., Brito, A. V., 2017. A System Embedded in Small Unmanned Aerial Vehicle for Vigo Analysis of Vegetation. *GeoInfo*, XVIII, 310–321.

Moulton, J., Karapetyan, N., Bukhsbaum, S., McKinney, C., Melebary, S., Sophocleous, G., Quattrini Li, A., Rekleitis, I., 2018. An Autonomous Surface Vehicle for Long Term Operations. *OCEANS*.

Paparazzi, Source Code. https://github.com/paparazzi/paparazzi.

PX4, Source Code. https://github.com/PX4/Firmware.

Raber, G. T., Schill, S. R., 2019. Reef Rover: A Low-Cost Small Autonomous Unmanned Surface Vehicle (USV) for Mapping and Monitoring Coral Reefs. *Drones*, 3(38).

Ramirez-Atencia, C., Camacho, D., 2018. Extending QGroundControl for Automated MissionPlanning of UAVs. *Sensors*, 18(7), 2339.

Sani, A. Y. M., He, T., Zhao, W., Yao, T., 2019. Hybrid Underwater Robot System Based on ROS. *International Conference on Robotics, Intelligent Control and Artificial Intelligence*, 396–400.

Schillaci, G., Schillaci, F., Hafner, V. V., 2017. A Customisable Underwater Robot. *ArXiv*, abs/1707.06564.

Siciliano, B., 2008. *Springer Handbook of Robotics*. Springer.

Sinisterra, A., Dhanak, M., Kouvaras, N., 2017. A USV Platform for Surface Autonomy. *OCEANS*.

TauLabs, Source Code. https://github.com/TauLabs/TauLabs.

Velaskar, P., Vargas-Clara, A., Jameel, O., Redkar, S., 2014. Guided Navigation Control of an Unmanned Ground Vehicle using Global Positioning Systems and Intertial Navigation Systems. *International Journal of Electrical and Computer Engineering*, 4(3), 329–342.

Wardihani, E. D., Ramdhani, M., Suharjono, A., A., S. T., Hidayat, S. S., Helmy, Widodo, S., Triyono, E., Saifullah, F., 2018. Real-time Forest Fire Monitoring System using Unmanned Aerial Vehicle. *Journal of Engineering Science and Technology*, 13(6), 1587–1594.

Washburn, L., Romero, E., Johnson, C., Emery, B., Gotschalk, C., 2017. Measurement of Antenna Patterns for Oceanographic Radars Using Aerial Drones. *Journal of Atmospheric and Oceanic Technology*, 34(5), 971–981.