Micro-drone autopilot architecture for efficient static scheduling

Gautier Hattenberger*[,1], Fabien Bonneval[1], Matheus Ladeira[2], Emmanuel Grolleau[2], and Yassine Ouhammou[2]

[1]École Nationale de l'Aviation Civile, Université de Toulouse, Toulouse, France
[2]LIAS, ISAE-ENSMA, Université de Poitiers, Poitiers, France

ABSTRACT

This paper presents the internal architecture of a Modifiable Off-the-Shelf open-source autopilot. We show starting from a set of functional and hardware requirements why most autopilots use as a core thread a main loop acting as a non-preemptive static scheduler, reacting to external events, some solicited, some unsolicited (but expected). We explain how the type of bus used to communicate with the sensor impacts the nature of the events received from the sensors (solicited or not). We show that depending on the workload that a main loop iteration has to handle, the execution time of an iteration can be larger than the period, creating potential delays in the attitude correction. Finally, we explore the degrees of freedom that can be used to reduce the impact of these overloads by smoothing the periodic workload.

1 INTRODUCTION

There is a growing interest in open and flexible architecture for UAV system, both for research and industry, aiming at providing standards for reference implementation [1, 2, 3]. This tendency concerns single drone operation, with various level of autonomy, but also swarm and fleet control [4], sometimes based on cloud technology [5]. At a very high level, a UAV (or RPA, Remotely Piloted Aircraft) system is composed of the actual flying object(s) (*airborne segment*), a remote pilot station (*ground segment*) and a command and control link (*communication segment*) as stated by ICAO Concept of Operation [6]. Most of the described architectures, such as in [7], are considering the flight controller as a black-box, with the flight stabilization and basic navigation services provided by the manufacturer or by one of the open-source projects existing on the market, such as *PX4* [8], *ArduPilot* [9] or *Paparazzi* [10, 11]. These autopilots are often seen as modifiable off-the-shelf (MOTS) components by SME manufacturing custom drones, who have to extend these MOTS autopilot to comply with their specific needs. The internal low level timing and execution sequence, while subject to real-time constraints, and interacting with external sensors, actuators and payload control, is rarely presented or studied in the specific case of small UAV autopilots. In order to understand the internal behaviour, one has to rely on the developer documentation or even the source code. It can be a problem when trying to customize MOTS autopilots, especially when the central stabilization functions are to be addressed.

This article aims at presenting the internal software architecture of the low level functions of an open-source autopilot flight stack. We show in this paper how and why Paparazzi's central architecture is based on a central monolithic thread (*main loop*) acting as a static non-preemptive scheduler.After introducing the functions scheduling problem in section 2 and the airborne architecture used as a reference in section 3, the scheduling problem is detailed in section 4 and finally functions durations are evaluated in section 5.

2 AUTOPILOT SCHEDULING PROBLEM

The design of an autopilot consists of executing several interacting functions, each one responsible for a specific part of the complex control system that links the drone's sensors to its actuators. The connections between the functions form a Directed Acyclic Graph (DAG), which would ideally be executed in continuous time, with no delay between input and output.

However, the functions have to be executed on a hardware platform – microcontroller or microprocessor – that executes each thread sequentially, one instruction at a time, hence one function at a time for each thread. The ideally continuous control must therefore be executed in discrete time. Like for any control and command system, a loop is created by the physical process (i.e., the drone) between the actuators and the sensors, since the actuators change the state of the controlled system. Control and aerodynamics experts usually consider that in order to ensure stability, the actuators shall be controlled, given a fresh and good estimation of the state, at a rate between 400 Hz and 3 kHz for rotorcrafts such as helicopters, or multirotors, and between 50 Hz and 400 Hz for a fixed-wing aircraft. As a result, in most autopilots, the function chain starting from the state estimation and ending at setting the actuators should be executed at a rate ranging from 50 Hz to 3 kHz. For a typical rate of 1 kHz, on an embedded platform, this functional chain executed at a period of about 1 millisecond, and including computationally intensive work,

*Corresponding author: gautier.hattenberger@enac.fr

represents a non-negligible load and appears as the core part of any autopilot.

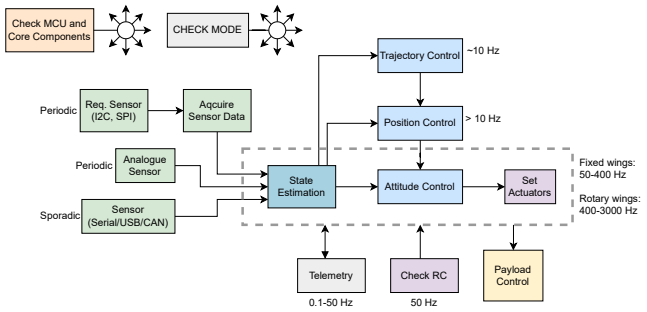

Figure 1: Representation of functions in an autopilot. The two functions in the top left corner are broadcasting to all other functions.

Figure 1 shows an extract of functions which are part of any autopilot nowadays. We can see in a common frame the chain from state estimation to actuation, which will dictate its rhythm to the control. For closely related autopilots, it can be noticed that depending on the hardware interface used to communicate, three types of constraint can appear. (1) If the sensor is analogue, then it is usually read when necessary, at a time chosen by the application (a sensor reading take less than 5 microseconds). Such a sensor can thus be read during a polling phase initiated by the application. (2) If the sensor is connected to a bus (serial, USB, CAN, etc.), then it is usually sending its readings to the autopilot periodically, but using its own clock that can drift compared to the clock used by the autopilot. We can therefore establish a minimal interval between two successive frames coming from a sensor, but we cannot choose when exactly the frame will be received. As a result, an incoming frame from such a sensor can only be considered sporadic. (3) If the bus used to communicate with the sensor is of type Master/Slave (e.g., I2C, SPI), then a delay has to be considered between the request from the Master to the response from the sensor. Since such delay can be long (dozens up to hundreds of microseconds), such functions can be decomposed in two parts: (i) the request from the Master, (ii) handling the response, where a sufficient amount of time has to elapse between the successive executions of the two functions.

Other functional chains expressed on Figure 1 have a smaller rate than the core chain state estimation to actuation. For example, the desired rate for position and trajectory control is in the order of magnitude of 10 Hz. Inputs received through legacy Radio Command (RC) is typically 40 Hz, while Datalink received from, and Telemetry transmitted to the ground station have a frequency typically ranging from 0.1 Hz to 50 Hz. The custom payload may require any frequency depending on its nature: from even higher than the core autopilot frequency, down to less than one hertz.

Note that, on each functional chain, the current state (Fly-by-Wire, Stop, Take off, etc.) shall be the same, otherwise the behaviour of the system may be inconsistent. As a result, a mode change shall occur either between two executions of the chain: at the beginning or at the end of the execution of the DAG.

Finally, MOTS autopilots usually target several types of hardware platforms, from small low-cost microcontrollers to modern heterogeneous multiprocessor on a chip boards, some barely able to host an Operating System (OS), others able to host a full POSIX PSE 54 Linux OS. As a result, the choice for Paparazzi was to be executable bare metal as well as on an OS. The same choice was made for Ardupilot, while PX4 can only be executed on a POSIX compliant OS. Regarding these functional and hardware constraints, the next section shows the internal software architecture of Paparazzi.

3 AIRBORNE ARCHITECTURE FOR THE PAPARAZZI SYSTEM

3.1 Components organization

The general architecture for any UAV system aims at providing a closed loop system with perception, decision and action. If each implementation will differ, a general good practice is to design a loosely coupled architecture, usually based on modules exchanging data through dedicated components, for example middlewares or blackboards. Such organization is found in most robotics platforms, as with the ROS/ROS2 framework [12], or in UAV specific systems like PX4 or ArduPilot. In this case study, based on the Paparazzi system, the architecture presented on Figure 2 follows the same principles.

In particular, the sensors data collected by the modules at the top right are sent to the state estimation filters (INS / AHRS blocks) through a software bus. This bus is using the publish / subscribe scheme, where data is pushed when available by the producers to the subscribing consumers. Only the common definition of messages is required to connect the elements. Note that other elements can use the same bus to create interactions between payload components for instance.

The result of the state estimation is pushed by INS/AHRS filters to a blackboard type structure, referred as *state interface* on Figure 2. The main characteristic of this interface is that new data can be pushed in any supported format (e.g. Euler angles, rotation matrix or quaternion in the case of attitude representation), while they can be retrieved from any other components within AP process and in any format as well. If a format transformation is required (e.g. from quaternion to Euler), the conversion is performed on the fly, only once, until a new update is available for the state. The available data in this interface represents position, velocity, acceleration, orientation (attitude), rotation speed, and air and wind speed. The mathematical library involved in these conversions and other algorithms have been formally verified against runtime errors [13].

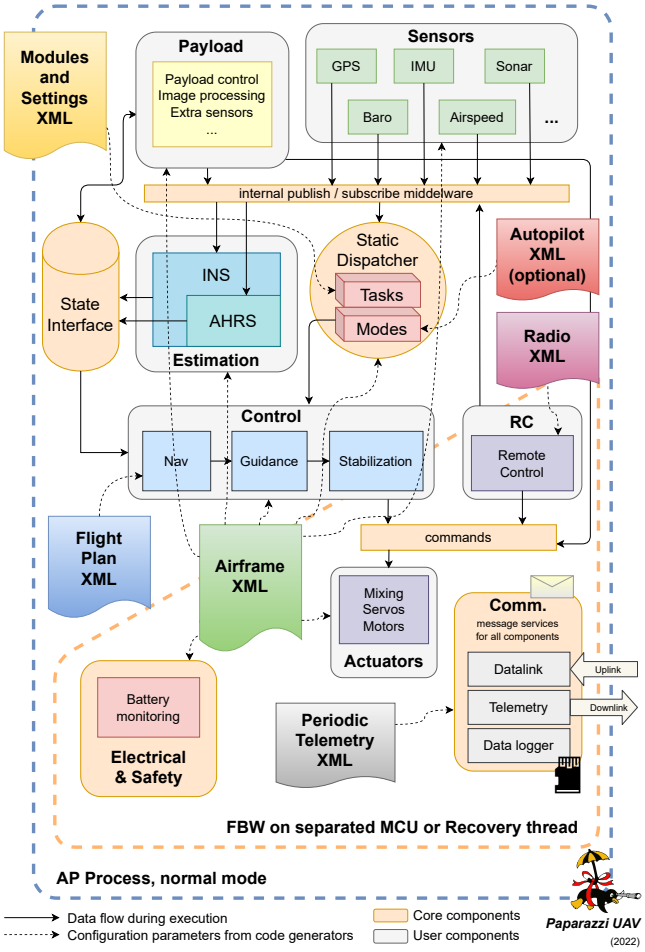

Figure 2: Airborne architecture of the *Paparazzi* system

Some other core components and services are available within the system. A *commands* array stores the normalized control vector computed by the navigation, guidance and control loops. They are ultimately applied to the physical actuators with a mixing defined in an airframe configuration file. The communication services allow sending and receiving messages from the ground, from another aircraft or from an embedded companion computer. Data logging on SD card is also possible on supported hardware. Electrical and safety monitoring are checking the energy source to apply appropriate actions in case of voltage drop.

Finally, the dispatcher is in charge of calling all the components with the correct sequence. The scheduling is statically defined during compilation based on the characteristics of the different modules. The section 4 is presenting the scheduling strategy in details.

3.2 *Configuration of the system*

The system presented Figure 2 is configured with a set of XML files, describing the airframe, the flight plan, the messages used for telemetry, the remote control and the settings (list of variables accessible from the ground control station). In particular, the airframe and the flight plan provide a list of *modules*. These modules correspond to XML files containing the following information:

- a name and a task group (note that in the document, functions can be referred to as tasks, not to be confused with threads),
- a documentation,
- a list dependencies with required or conflicting modules or functionalities, as well as functionalities provided by the module itself,
- a list of settings,
- a list of header files with public elements (functions and variables),
- a list of initialization functions,
- a list of periodic functions with the associated frequency,
- a list of event function (polling functions to check and process sporadic events),
- a list of callbacks to trigger on incoming datalink messages,
- compilation instructions with file names, compilation flags, includes, filtering options (e.g. restrict to certain firmware), etc.

The build process consists in parsing all the configuration files, solving the dependencies to find the list of required modules and generate a code corresponding to the different components, flight plan state machine and makefile to build the final binary that is finally flashed to the flight controller board.

4 Static Scheduling Approach

The scheduling approach is to divide the problem into three complementary steps:

1. For each module, resolve the dependency graph and provide an ordered list of components to load.
2. Generate initialization, periodic or event function calls, based on the task group associated to each module. Call the task groups in a predefined order to respect the closed-loop control scheme and some temporal constraints.
3. Each periodic task group consists in a static scheduler, which offset is chosen to limit the number of periodic functions being called at each iteration.

The following subsections provide details for each step.

```
<dep>
 <depends>
   module1,module2|module3,@func1
 </depends>
 <provides>func2</provides>
 <conflicts>module4,@func3</conflicts>
</dep>
```

Figure 3: Example of module's dependency, functionalities are prefixed with `@` in `depends` and `conflicts` nodes, `|` corresponds to a logical *or* operator

4.1 Modules dependencies resolution

As mentioned in section 3.2, every component to be integrated to the airborne code is described in a XML module file. This file describes, for each component, its interface as the list of provided functions (see Figure 3), dependencies and conflicts as the list of required and conflicting modules or functions.

An ordered list of modules is obtained with the two-step topological sort described by Algorithm 1, and based on the depth-first search from [14]. During the first step, each time an explicit module name is specified in the `depends` node, this module is included in the recursive tree search presented in Algorithm 2. However, when it is a functionality or a Boolean expression (with `|`), it is stored for a later validation. The list of provided functionalities and potential conflicts is also stored. At the end of the first sorting algorithm, a list of modules is available, either selected by the user or required by dependency. Then the list of required functionalities is checked against the list of provided functionalities, same for logical expression and conflicts. If this check is successful, the list of modules is complete, but not properly ordered: during the first pass, functionalities are not yet instantiated with a real module. This is why a simpler second pass of the topological sorting algorithm is necessary. This time, no new modules are added and there is no need to check for functionalities and conflicts. The final result of the second pass is a fully ordered list of modules. Note that this order is not unique, and the first valid solution is kept, but it is guaranteed that a given module will always be placed after the modules it is depending on.

4.2 Temporal execution sequence

The second aspect for the correct scheduling of function calls is to guarantee a correct temporal sequence. The approach in Paparazzi autopilot is to split the functions into a predefined list of task groups, which is shown on Figure 4, in the *Init* box.

The selection and ordering of these tasks is the result of an expert-based functional analysis. Considering the *Init* phase, the initialization of the microcontroller unit (MCU) and its peripherals should be executed prior to all other tasks. Then core components of the autopilot system, and finally other

Algorithm 1 Topological sort

Require: airframe and flight plan XML files
procedure SORT MODULES
Extract list of modules selected by user from input files
Add extracted modules to a $root$ module
RESOLVE($root$)
if at least a module from resolved in conflict list **then**
fail
end if
if at least a functionality from required in conflict list **then**
fail
end if
if not all required modules and functionalities are in provided list **then**
fail
end if
Substitute functionalities or logical expression with proper modules from resolved list
RESOLVE($root$ (updated)) ▷ second pass
Return resolved list of fully ordered modules
end procedure

Algorithm 2 Dependency resolution

resolved list ← []
unresolved list ← []
conflict list ← []
provided list ← []
function RESOLVE(m) ▷ provide module as input
if m is valid for firmware and target **then**
add m to unresolved
for each dependency d in m **do**
if d is a module **then**
if d is not in resolved **then**
if d is in unresolved **then**
fail, cyclic dependency detected
else
RESOLVE(d)
end if
end if
else ▷ functionality or logical expression
add d to required list
end if
end for
add conflict to list
add provided to list
add m to resolved list
remove m from unresolved list
end if
end function

components such as inputs/outputs (sensors, radio control, actuators, datalink), estimation and control, and finally the payload. Within each group, the order of execution respects the order resulting from Algorithm 1.

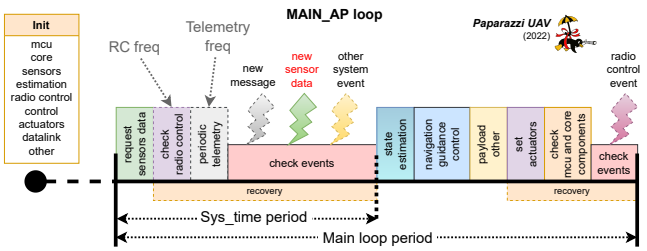

Figure 4: Main loop temporal sequence

After the initialization phase, an infinite loop with a fixed desired frequency is started. This frequency corresponds to the *Main loop period* on Figure 4. Each time the main periodic function is called, the periodic functions from each task group are called in the order presented on the figure, mostly the same as during initialization phase, but with some modifications and timing constraints. An important consideration to have is that the sensors used for state estimation on modern MAVs have digital interfaces, such as I2C or SPI, especially the Inertial Measurements Units (IMU). When called, the periodic functions for these sensors initiate a transaction to get new data. Obviously, it is not necessary to run the state estimation filters and the control stack before data availability.

Several options are available to decide when to start the rest of the flight control stack. In order to keep the estimation and control at a stable frequency, independent of the transaction time jitter, the design choice in Paparazzi is to run them after half of the main frequency. In practice, it corresponds to setting the *Sys_time period* to half of the main loop period and alternate sensor reading and control on this base frequency.

To evaluate the compatibility of this choice with existing sensors, let's consider two types of IMUs used on existing boards: the *InvenSense MPU9250* connected other I2C at 400 kHz and the *InvenSense ICM-20600* connected over SPI at 1 MHz. Each sensor sends on request 14 bytes of data and a status byte to be retrieved at each transaction, with different protocols overhead. The transaction time is presented in Table 1 and is less than 0.5 ms for these two sensors. Considering the half-period timing, it means that the main loop frequency can be set up to 1kHz in all cases, even more with fast sensors, which is compatible with typical values presented in section 2.

4.3 *Static scheduling of periodic function calls*

Scheduling the autopilot functions in this scenario requires analyzing some characteristics of the functions to be scheduled. Once the dependency has been resolved and the internal order has been defined (Algorithm 1), assuming a main loop frequency of 1kHz, functions with a period of 1ms

model	# bytes (bits/byte)	baudrate	time (ms)	max AP freq
MPU9250	15+3 (9)	400k	0.4	$\approx$ 1 kHz
ICM-20600	15+1 (8)	1M	0.13	$\approx$ 3 kHz

Table 1: IMU transaction time and maximum recommended main loop frequency

are already completely scheduled.

However, in functions that have a greater period than 1ms (a lower frequency than 1kHz), there is a degree of freedom relative to the choice of which loops will host the execution of these functions. For example, neglecting any precedence constraints, two 500 Hz functions can be executed in alternating loops such that they will never be executed in the same loop. This strategy can better distribute the workload, helping to avoid the main loop taking more than its defined period to execute.

The choice of the loop where to execute each function is not trivial. This problem is equivalent to choosing an offset for periodic functions, which has been treated in the literature and identified as closely related to a NP-hard problem [15]. Some efficient heuristic methods have already been proposed to deal with the problem for independent functions [16, 17, 18, 15] but they cannot be trivially adapted to functions subject to precedence constraints.

The current implementation is a naive algorithm that shift each periodic functions by a fraction of its period, unless user-specific instructions. The Figure 5 is showing this principle on a simplified case with three tasks running at three different frequencies. Without offset, the tasks are all called in the same loop periodically (every 6 time units in this case). With the offsets, the result is a smooth execution but without guarantee nor optimization. Also, execution time is not considered.

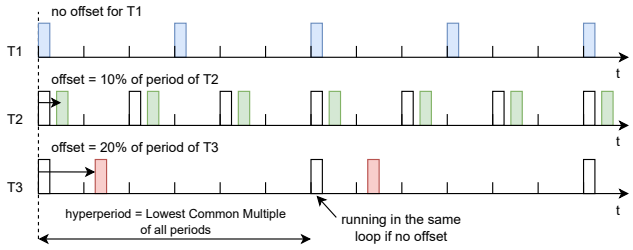

Figure 5: Task scheduling example of 3 tasks with an hyperperiod of 6 units. With no offset (white boxes), the 3 tasks are executed sometimes in the same loop. The offset of a task is computed as x times its period where $x \in [0, 1[$ is incremented by 0.1 at each new task.

5 PERFORMANCE EVALUATION

The performance of the autopilot has been measured by recording the execution time of the sub-functions of the autopilot during a flight. The conditions and experimental setup are as follow. The flight has been performed with a standard quadrotor frame, running the "rotorcraft" firmware of Paparazzi at 500 Hz for the main loop frequency, on the *Tawaki* board equipped with a STM32F777 at 216 MHz. In order to capture the time required by different functions, there are three main phases during the flight, that have been explicitly labeled with background colors on Figure 8:

1. The aircraft is in *KILL* mode, i.e., with no control activity. After 100 seconds, position data from a motion capture (indoor flight) are received through the datalink (red background).
2. After 120 seconds, the aircraft mode switches to *ATTITUDE* control with stabilization control activated (orange background).
3. After 210 seconds, it switches to *NAV* mode and performs a fully autonomous flight from flight plan navigation (green background).

The Figures 6, 7, 8 and 9 are all showing the average execution time (thick blue line), as well as the min and max execution time, where each data point corresponds to 1000 calls of the corresponding group. Data is stored on an on-board SD card. We measured and stored data related to 1000 calls in order to decrease the interference of the measuring time compared to the execution of the observed system. The Table 2 is summarizing the call frequency over the complete flight.

task	freq (Hz)	std
event	10007.575	0.047
sensors	500.496	0.154
radio	58.882	0.436
gnc	500.489	0.155
core	500.491	0.154
telemetry	500.483	0.154

Table 2: Execution frequency and standard deviation (std) of the periodic tasks and event functions

The polling of the event functions is running at 10 kHz, which is the resolution of the Real-Time scheduler used at low level. It can be seen on Figure 6 that the functions always takes less than 100 us, which means that it doesn't exceed the allocated time for event polling. The extra activity due to datalink incoming messages is visible at 100 seconds.

The sensor task (Figure 7) should not take a lot of time since it is mostly only requesting new data from digital sensors. The average duty time is very low, however accessing the I2C or SPI peripherals may take time in comparison depending on the underlying RTOS activity, where each peripherals' driver is running in a dedicated thread.

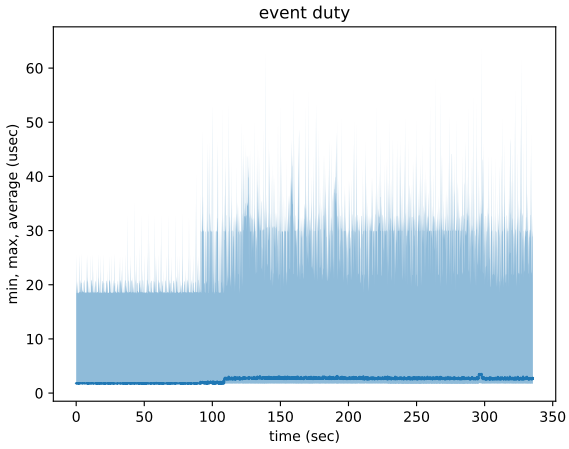

Figure 6: Duty time (average, min and max of 1000 calls in us) of the event functions during test flight. The 100 us threshold is never exceeded, which means that next event calls are not delayed. Activity is increasing with incoming data from datalink at 100 seconds.

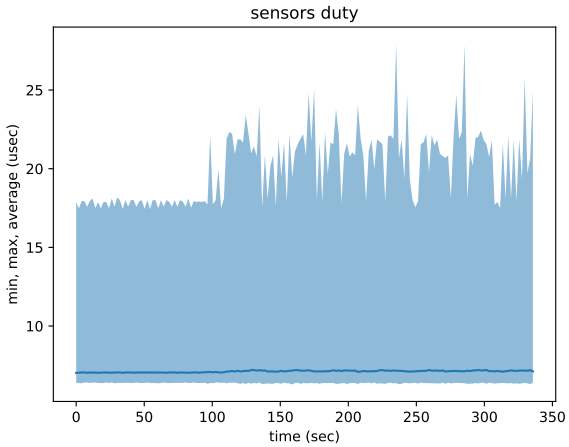

Figure 7: Duty time (average, min and max of 1000 calls in us) of the sensors periodic functions during test flight. Requesting new data takes less than 25 us.

The activity of the GNC (grouping estimation, navigation, control and payload tasks) is the most interesting, as Figure 8 shows that the execution time is varying a lot according to the flight phase. This setup is using an advanced controller that computes dynamically a control allocation with an optimization algorithm for the attitude control [19]. Therefore, the control activity is rising from 20 us to almost 80 us when the aircraft is flying from radio control (orange part), and even more when guidance loop is activated for navigation (green part). The maximum execution time still remains low (less than 120 us) compared to the main loop period of 2000 us (500 Hz).

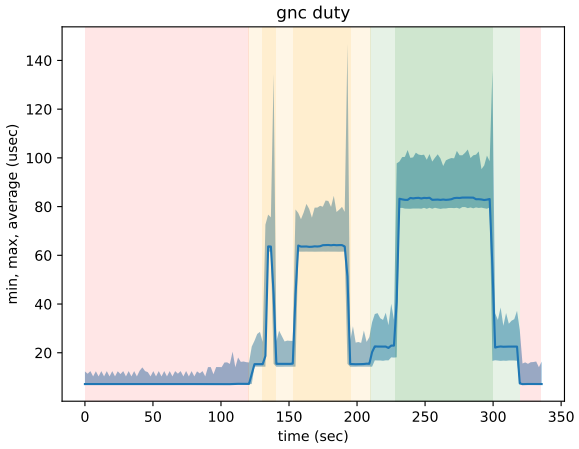

Figure 8: Duty time (average, min and max of 1000 calls in us) of the estimation, navigation, control and payload functions (GNC) during test flight. Background colors indicates the flight mode: red for kill mode (no activity), orange for direct attitude control (stabilization loop), green for flight plan navigation (full guidance and stabilization). Darker color indicates that the quadrotor is flying, with higher CPU duty.

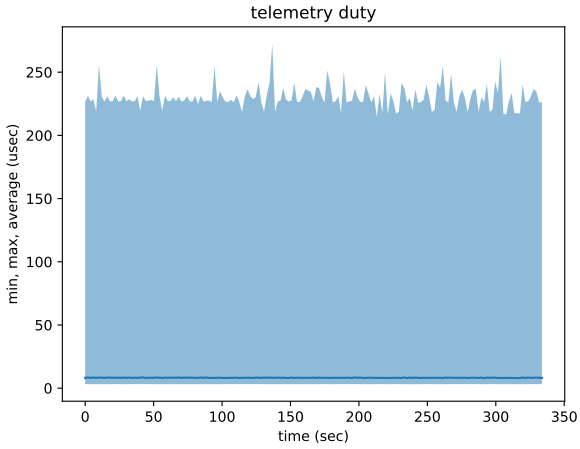

Figure 9: Duty time (average, min and max of 1000 calls in us) of the event functions during test flight. The average duty stays very low ($<$ 10 us) compare to the maximum values (around 225 us), this is due to the sporadic nature of message sending in each interval. When a message is sent, most of the duty time is spend in waiting access to the UART buffer, shared with the peripheral thread.

Finally, the last set of functions presented here is the telemetry group (see Figure 9). For similar reasons than with the sensors functions, the average duty time is very low ($<$ 10 us). These functions are called at 500 Hz, but the messages themselves are scheduled at much lower frequency, typically between 0.1 and 50 Hz (see Figure 1). As mentioned before, each MCU peripheral, including the UART connected to the modem, is running in a dedicated thread, and accessing the shared memory buffer, protected by a mutex, involves calls to the scheduling functions of the underlying RTOS. As a result, the maximum execution time of this group is the longest, going up to 270 us, although no processing related to navigation is involved. The other functions of the autopilot that are not presented on these figures are showing similar trend than the sensors or telemetry task and are not much interesting to be discussed.

The overall MCU load, including the autopilot thread and all other peripheral threads are measured around 10 % by the RTOS.

6 Conclusion

Nowadays autopilots are MOTS, meaning that personalized modules can be added to an existing autopilot. Moreover, experimentation shows that, while an autopilot control loop running at 500 Hz should mainly support the basic functions of the autopilot, there are two situations where some interference between execution of functions may postpone the execution of some core functions. First, if a specific module given by the user of the autopilot as a MOTS is computationally intensive, the additional functions may postpone some functions of the autopilot. Moreover, as told in introduction, some controller may be executed at a higher rate. In this case, the execution of some loops may postpone the execution of the next loop, creating delays and jitters, which usually have a negative impact on quality of control [20].

We have identified, as a way to avoid this type of interference, a means which consists in adjusting offsets of periodic functions with a period higher than the main loop frequency. Nevertheless, the current naive implementation doesn't other guarantees and some academic work has to be done for offset assignment methods to take all constraints into account, especially precedence constraints with a minimal inter-release interval.

Finally, the measurement framework is in place, and can be adapted to all the systems that drone manufacturers would like to base on Paparazzi. As an autopilot, Paparazzi is highly configurable.The perspective is to provide a framework allowing drone manufacturers to (1) measure the performances of the functions during flight, and (2) generate a static scheduler able to balance the load by adjusting offsets, while showing that at least in average, the main controller loop is able to be fully executed within its allocated window. For the purpose of being adaptable to every possible use, we are developing a bridge between files generated by the compilation of Paparazzi, and a Domain Specific Language (DSL), based on an extension of AADL proposed in the COMP4DRONES European project. This DSL will allow custom autopilots based on Paparazzi to be scheduled, and analyzed by the framework discussed in this paper.

ACKNOWLEDGEMENTS

This work was partially funded by ECSEL JU program as a part of the Comp4Drones project with grant number "826610".

REFERENCES

[1] Jose Luis Sanchez-Lopez, Ramón A. Suárez Fernández, Hriday Bavle, Carlos Sampedro, Martin Molina, Jesus Pestana, and Pascual Campoy. Aerostack: An architecture and open-source software framework for aerial robotics. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 332–341, 2016.

[2] Matheus Ladeira, Yassine Ouhammou, and Emmanuel Grolleau. Towards a modular and customisable model-based architecture for autonomous drones. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1127–1128, 2020.

[3] Mahmoud Hussein and Reda Nouacer. Towards an architecture for customizable drones. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 67–72, 2020.

[4] Juan A. Besada, Ana M. Bernardos, Luca Bergesio, Diego Vaquero, Iván Campaña, and José R. Casar. Drones-as-a-service: A management architecture to provide mission planning, resource brokerage and operation support for fleets of drones. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 931–936, 2019.

[5] Chen Hong and Dianxi Shi. A control system architecture with cloud platform for multi-uav surveillance. In *2018 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1095–1097, 2018.

[6] ICAO RPAS Panel. ICAO RPAS Concept of Operations, 2021.

[7] Georgios Kakamoukas, Panagiotis Sarigiannidis, and Ioannis Moscholios. High level drone application enabler: An open source architecture. In *2020 12th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, pages 1–4, 2020.

[8] PX4. https://docs.px4.io/master/. Accessed: 2022-04-04.

[9] ArduPilot. https://ardupilot.org/. Accessed: 2022-04-04.

[10] Pascal Brisset, Antoine Drouin, Michel Gorraz, Pierre-Selim Huard, and Jeremy Tyler. The Paparazzi Solution. In *MAV 2006, 2nd US-European Competition and Workshop on Micro Air Vehicles*, page pp xxxx, Sandestin, United States, October 2006.

[11] Gautier Hattenberger, Murat Bronz, and Michel Gorraz. Using the Paparazzi UAV System for Scientific Research. In *IMAV 2014, International Micro Air Vehicle Conference and Competition 2014*, pages pp 247–252, Delft, Netherlands, August 2014.

[12] Ros/ros2. https://docs.ros.org/. Accessed: 2022-04-04.

[13] Baptiste Pollien, Christophe Garion, Gautier Hattenberger, Pierre Roux, and Xavier Thirioux. Verifying the Mathematical Library of an UAV Autopilot with Frama-C. In *26th International Conference on Formal Methods for Industrial Critical Systems - FMICS 2021*, Paris, France, August 2021.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[15] Matheus Ladeira, Emmanuel Grolleau, Fabien Bonneval, Gautier Hattenberger, Yassine Ouhammou, and Yuri Hérouard. Scheduling offset-free systems under fifo priority protocol. In *34th Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl, 2022.

[16] Joël Goossens. Scheduling of offset free systems. *Real-Time Systems*, 24(2):239–258, 2003.

[17] Mathieu Grenier, Joël Goossens, and Nicolas Navet. Near-optimal fixed priority preemptive scheduling of offset free systems. In *14th International Conference on Real-Time and Networks Systems (RTNS'06)*, pages 35–42, 2006.

[18] Mathieu Grenier, Lionel Havet, and Nicolas Navet. Pushing the Limits of CAN - Scheduling Frames with Offsets Provides a Major Performance Boost. *4th European Congress on Embedded Real Time Software (ERTS 2008)*, 2008.

[19] Ewoud J. J. Smeur, Guido C. H. E. de Croon, and Qiping Chu. Cascaded incremental nonlinear dynamic inversion control for MAV disturbance rejection. *CoRR*, abs/1701.07254, 2017.

[20] Zakaria Sahraoui, Emmanuel Grolleau, Driss Mehdi, Mohamed Ahmed-Nacer, and Abdenour Labed. Predictive-delay control based on real-time feedback scheduling. *Simulation Modelling Practice and Theory*, 66:16–35, 2016.