DOI: 10.15514/ISPRAS-2019-31(4)-3

Designing robust quadcopter software based on a real-time partitioned operating system and formal verification techniques

S.M. Staroletov, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
M.S. Amosov, ORCID: 0000-0002-2056-2794 <faystmax@gmail.com>
K.M. Shulga, ORCID: 0000-0003-1422-4681 <kirsh.ru@yandex.ru>
Polzunov Altai State Technical University,
Lenin avenue 46, Barnaul, 656038, Russia

Abstract. Currently, the creation of reliable unmanned aerial vehicles (drones) is an important task in science and technology because such devices can have a lot of use-cases in digital economy and modern life, so we need to ensure their reliability. In this article, we propose to assemble a quadcopter from low-cost components in order to obtain a hardware prototype, and also to develop a software solution for the flight controller with high-reliability requirements, which will meet avionics software standards, using existing open-source software solutions. We apply the results as a model for teaching courses «Components of operating systems» and «Software verification». In the study, we analyze the structure of quadcopters and flight controllers for them, and present a self-assembly solution. We describe Ardupilot as open-source software for unmanned aerial vehicles, the appropriate APM controller and methods of PID control. Today's avionics standard of reliable software for flight controllers is a real-time partitioned operating system that is capable to respond to events from devices with an expected speed, as well as to share processor time and memory between isolated partitions. A good example of such OS is the open-source POK (Partitioned Operating Kernel). In its repository, it contains an example design of a system for a quadcopter using AADL language for modeling its hardware and software. We apply such a technique with Model-driven engineering to a demo system that runs on real hardware and contains a flight management process with PID control as a partitioned process. Using a partitioned OS brings the reliability of flight system software to the next level. To increase the level of control logic correctness we propose to use formal verification methods. We also provide examples of verifiable properties at the level of code using the deductive approach as well as at the level of the cyber-physical system using Differential dynamic logic to prove the stability.

Keywords: quadcopter; partitioned OS; ARINC 653; formal verification; Cyber-physical system

For citation: Staroletov S.M., Amosov M.S., Shulga K.M. Designing robust quadcopter software based on a real-time partitioned operating system and formal verification techniques. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 39-60. DOI: 10.15514/ISPRAS-2019-31(4)-3

Разработка программного обеспечения квадрокоптера с повышенными требованиями к надёжности на основе партицированной ОС и технологий формальной верификации

С.М. Старолетов, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
М.С. Амосов, ORCID: 0000-0002-2056-2794 <faystmax@gmail.com>
К.М. Шульга, ORCID: 0000-0003-1422-4681 <kirsh.ru@yandex.ru>
Алтайский государственный технический университет им. И.И. Ползунова,
656038 Барнаул, проспект Ленина, 46

Аннотация. Создание надежных беспилотных летательных аппаратов является важной задачей для науки и техники, потому что необходимо обеспечивать надежность таких решений. В этой статье предлагается использование аппаратного прототипа квадрокоптера и разработка программного решения для полетного контроллера с высокими требованиями к надежности, которое будет соответствовать новым стандартам для программного обеспечения авионики и будет использовать существующие программные решения с открытым исходным кодом. В ходе исследования мы анализируем состав квадрокоптеров и полетных контроллеров для них. Мы описываем открытое программное обеспечение Ardupilot для беспилотных летательных аппаратов, контроллер APM и методы ПИД-регулирования. Сегодняшним стандартом надежного программного обеспечения для бортовых контроллеров являются партицированные операционные системы реального времени, которые способны реагировать на события от оборудования с ожидаемой скоростью, а также разделять процессорное время и память между изолированными разделами. Хорошим примером такой ОС с открытым исходным кодом является POK (Partitioned Operating Kernel). В репозитории она содержит пример описания системы для дронов с использованием языка AADL с моделированием аппаратного и программного обеспечения решения. Мы применяем такую технику к демонстрационной системе, которая работает на реальном оборудовании и содержит процесс управления полетом с PID-регулятором в виде изолированного процесса. Использование партицированного ОС выводит надежность программного обеспечения полетного контроллера на новый уровень. Для того, чтобы повысить уровень корректности логики управления, мы предлагаем использовать формальные методы верификации и демонстрируем примеры проверяемых свойств на уровне кода, используя дедуктивный подход, а также проводя моделирование на уровне киберфизической системы с использованием динамической дифференциальной логики для доказательства устойчивости.

Ключевые слова: квадрокоптер; операционная система; партицирование; ARINC 653; формальная верификация

Для цитирования: Старолетов С.М., Амосов М.С., Шульга К.М. Разработка программного обеспечения квадрокоптера с повышенными требованиями надёжности на основе партицированной ОС и технологий формальной верификации. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 39-60 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(4)-3

1. Introduction

Unmanned aerial vehicles (UAVs) also called drones are becoming a big part of our digital life. In this paper, we consider *quadcopters* – vehicles with four software-controllable motors. They can be used for taking nice videos, for delivering parcels from Internet stores to end-customer by air, and some people even proposed to use them on the FIFA World Cup to deliver the balls.

However, in some countries, the use of UAVs is prohibited in public places, mostly because of fears of a poorly functioning machine falling from the air onto people. An example of last year's public testing of Russian Post delivering drone, which crashed after a few seconds from the start, shows us that we need to build highly reliable software for it, which should prevent the drone from failure and fall in the most of cases (breakages of hardware or software, incorrect and contradictory commands from a pilot).

Also, we should take in mind that a drone is a representation of a normal air vehicle. Most of the technologies to build the quadcopter, to create the software for it, to control safety and liveness properties, to model physical properties of flight are the same as for big air vehicles, so it could be a cheap model to design of highly reliable aircraft.

The scope of the work includes the following.

- Assembling a flying quadcopter for the purpose of obtaining a hardware prototype (from components that are mass-marketed);
- Developing a flight controller software solution with high-reliability requirements, which will coincide with the standards of avionics software (to some extent);
- Obtaining a model for teaching courses on the design of operating system components and software verification.

The first task covers selection of existing hardware components including passive (i.e., frame, battery, etc.) and active parts (their state could be read and changed by software, i.e., GPS sensor, accelerometer, controllable motors, etc.) with maximum compatibility; and selection of a basic flight controller, which manages the whole hardware system; we are going to write software for it.

The second task is devoted to creating an operating system for the drone, which will be based on ARINC 653 [1] – an industrial standard interface of applied software for avionics and existing open-source solutions in a free code for flight controllers to produce control logic and interoperation with the hardware.

And the last task is to apply the project achievements in the study process of future software engineers to improve the interest of students in system programming and to offer different tasks as lab works.

2. Related work

The task of building reliable software for UAVs (and also for aircraft) is strongly connected to developing an operating system that is robust according to the initial design. Today's OS for flight machines should be real-time and should offer memory space and time division capabilities. There exists an avionics industry standard for these requirements, created by the Aeronautical Radio, Inc., ARINC 653 [1]. BSD-licensed open-source POK OS, which satisfies this standard with some limitations, was created in France by Julien Delange [2]. It uses Model-Based Engineering approach [3, 4] for describing the system configuration, and its source code is available in [5].

In Russia, JetOS, a certified operating system for aircraft, was created by GosNIIAS and ISP RAS [6, 7] as a fork of POK with advanced debugging capabilities, rewritten scheduler, system partition feature and different platforms support. The code is partially available on GitHub under the GPLv3 licence [8].

Building robust software encompasses not only the design of the OS for the flight controller. It should also include formal verification of developed code and proof of software correctness to satisfy some requirements. ISP RAS is the leader in this field in Russia, the recent results were published in [9, 10, 11]. Their principal result is producing testing and deductive verification techniques and an adequate memory model, mainly used in checking of correctness of Linux modules.

The task of building software and hardware for a drone is being solved in project Crazyflie [12] and the models are given in [13]. It is used in some universities, e.g., at Chalmers University of Technology where students build dynamic models for Crazyflie drone using the modeling language Modelica and then design control algorithms based on them [14].

3. The drone: components and terminology

Consider a drone that is assembled from scratch based on components available on the market. It consists of the following components (see fig. 1):

- **frame** connects all the components and provides electrical routing of high-current energy to the motors (here we use a 450mm frame);

- **brushless motors** (4x for a quadcopter) provide lifting force to the propellers, which allows the drone to fly or rotate;
- **ESC** (Electronic Speed Controller) provides a high-level interface to control a corresponding motor, translates required RPM to electric voltage and controls the results of motors rotation;
- **battery** (usually of the LiPo type) provides electricity to the drone, and the time of drone operation depends on it; a 11.1V, 3000mAh 3S battery provides about 10-15 mins of operation and requires a special balancing charger;
- **power module** connected to the battery provides two electrical circuits: a low-current circuit to the flight controller and a high-current one to the motors, also it provides a controlling signal to the flight controller with the current state of the battery;
- external **GPS/compass** module is used for getting current coordinates to use in waypoint algorithms, it should be separated from the flight controller and frame to minimize the effects of electrical noise;
- **telemetry** is a radio transceiver that works at the frequency of 915Mhz and allows to make a channel to a ground control module to send the current state and receive commands by providing radio-transparent UART (communication port);
- **additional periphery** is the useful load of a drone, in the simplest case it may be a camera, in more complex cases it may be for example a fire extinguishing device; such load can be operated by an additional controller or even by the main flight controller with special isolated processes;
- **flight controller** is responsible for controlling the whole drone state (the states of all internal hardware components) and operating it like the car ECU; it has some sensors on-chip and software for flying either with operator's control and stand-alone. Our goal is to create reliable software for it.

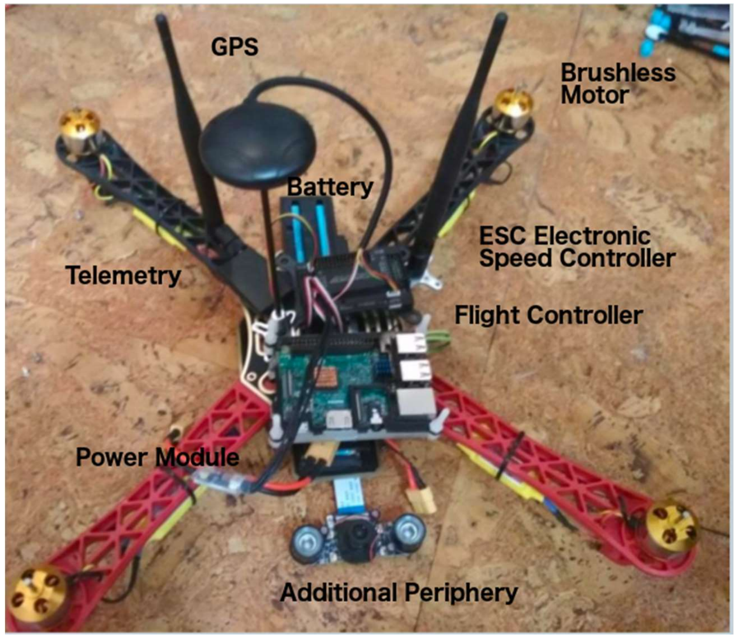


Fig. 1. The parts of the quadcopter

Consider a question, how does the quadcopter fly? According to [15], in the stable state the drone has following properties (fig. 2):

- the equivalence of forces: $\sum T_i = -mg$;
- the equivalence of moments: $\sum M_i = 0$;
- the equivalence of directions: $T_{1,2,3,4} || g$;
- the sum of rotation speeds: $(w_1 + w_3) - (w_2 + w_4) = 0$.

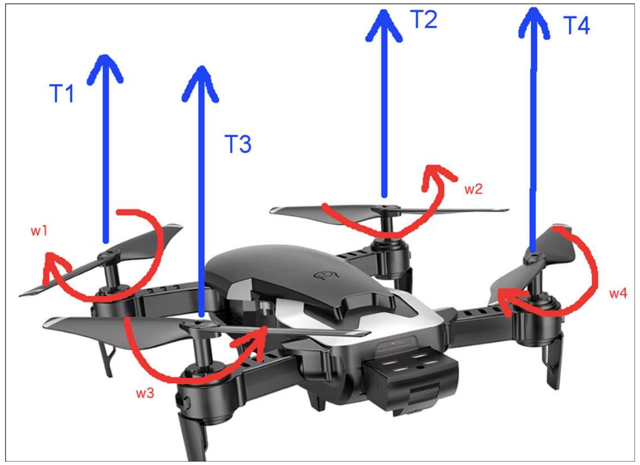


Fig. 2. Quadcopter flight state

Every violation of the properties leads to move of the drone among one of three axes, and so-called Euler's angles are defined (fig. 3):

- pitch;
- roll;
- yaw.

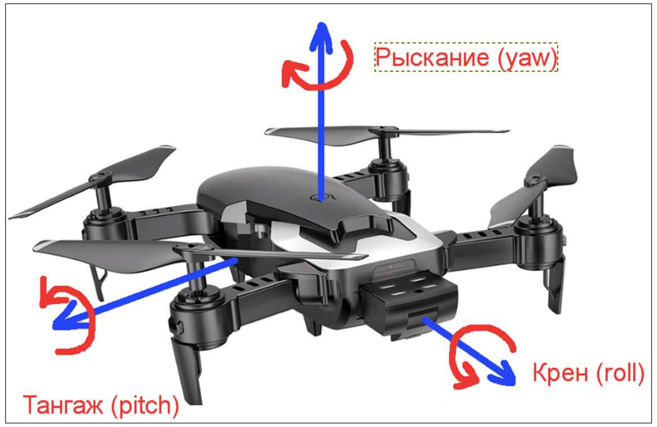


Fig. 3. Quadcopter angles

Movement or rotation of the quadcopter is done by means of different rotation speeds of its propellers controlled by the flight controller.

4. Today's software solution

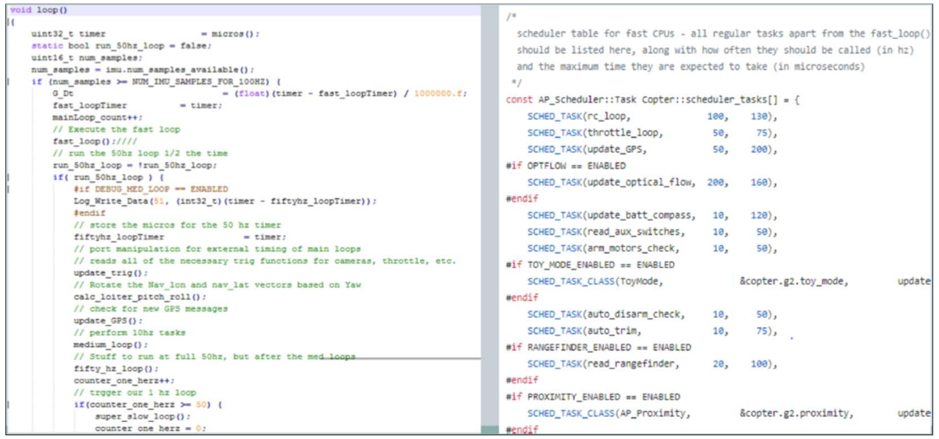
4.1 Ardupilot / Adrucopter

Ardupilot project [16, 17] is the most known open-source project to provide free software to control the drones with its Arducopter codebase (as well as rovers and planes, refer to Ardurover and Arduplane codebases respectively) and high-level abstraction to operate hardware components for them. The source code of control is based on the recent results of the UAV research community, so it is a good base for tracking recent achievements. Inside, it consists of HAL (Hardware Abstraction Layer) that allows to run the system on different hardware platforms, main scheduling loop and a set of different libraries that provide some code to communicate with a particular device (get and store data), mathematical calculations, control algorithms. We are mostly interested in the following libraries:

- AC_PID: a library to implement an algorithm of PID-control;
- AC_AttitudeControl: attitude position control of a drone using PID-control algorithms;
- AC_WPNav: pre-defined trajectory flight using waypoints and PID-control.
- APM_Control: stabilisation on the pitch, roll, yaw axes;
- AP_Math: a module with internal mathematical routines;
- AP_Techs: combined energy, speed and altitude control;
- AP_Arming: check the preconditions of equipment performance before starting drone control;
- AP_Compas: a module to work with compass hardware;
- AP_Baro: a module to work with barometer hardware;
- AP_BattMonitor: a module of work with the battery controller, including the detection of its discharge;
- AP_GPS: a module to work with GPS hardware;
- AP_Motors: a module to work with drone's motors, supporting statuses, control and their testing;
- AP_Frsky_Telem: a module to work with the radio telemetry;
- GCC_MAVLink: support for MAVLink telemetry transmission protocol.

These libraries allow to develop a custom software solution for a drone based on hardware described above.

Consider the main working procedure in the flight controller in Ardupilot software. It has to get current hardware state using HAL-abstraction and libraries to work with devices, then construct a high-level state of the drone (for example, the orientation of it in the space), run algorithms of control based on the current mode, and after then apply some commands to hardware. Due to various types of devices and different protocols of communication (for example, SPI or UART), the code should use a different speed for devices polling or priorities. Fig. 4 shows an evolution of Ardupilot software depending of properties of corresponding hardware: up to branch 3.1 [17], Ardupilot used loops with different frequencies in the main loop code (some of them executed at each iteration, some – at each two iterations, etc.), then the code was upgraded to use the concept of tasks with priorities (depending on hardware platform it can be executed like loops of early versions or run inside processes of a real-time OS).



```
void loop()
{
    uint32_t timer                    = micros();
    static bool run_50hz_loop = false;
    uint16_t num_samples;
    num_samples = imu.num_samples_available();
    if (num_samples >= NUM_IMU_SAMPLES_FOR_100HZ) {
        G_Dt                              = (float)(timer - fast_loopTimer) / 1000000.f;
        fast_loopTimer          = timer;
        mainLoop_count++;
        // Execute the fast loop
        fast_loop();////
        // run the 50hz loop 1/2 the time
        run_50hz_loop = !run_50hz_loop;
        if( run_50hz_loop ) {
            #if DEBUG_MED_LOOP == ENABLED
            Log_Write_Data(51, (int32_t)(timer - fiftyhz_loopTimer));
            #endif
            // store the micros for the 50 hz timer
            fiftyhz_loopTimer                 = timer;
            // port manipulation for external timing of main loops
            // reads all of the necessary trig functions for cameras, throttle, etc.
            update_trig();
            // Rotate the Nav_lon and nav_lat vectors based on Yaw
            calc_loiter_pitch_roll();
            // check for new GPS messages
            update_GPS();
            // perform 10hz tasks
            medium_loop();
            // Stuff to run at full 50hz, but after the med loops
            fifty_hz_loop();
            counter_one_herz++;
            // trgger our 1 hz loop
            if(counter_one_herz >= 50) {
                super_slow_loop();
                counter one herz = 0;
```

```
/*
  scheduler table for fast CPUs - all regular tasks apart from the fast_loop()
  should be listed here, along with how often they should be called (in hz)
  and the maximum time they are expected to take (in microseconds)
 */
const AP_Scheduler::Task Copter::scheduler_tasks[] = {
    SCHED_TASK(rc_loop,              100,    130),
    SCHED_TASK(throttle_loop,         50,     75),
    SCHED_TASK(update_GPS,            50,    200),
#if OPTFLOW == ENABLED
    SCHED_TASK(update_optical_flow,  200,    160),
#endif
    SCHED_TASK(update_batt_compass,   10,    120),
    SCHED_TASK(read_aux_switches,     10,     50),
    SCHED_TASK(arm_motors_check,      10,     50),
#if TOY_MODE_ENABLED == ENABLED
    SCHED_TASK_CLASS(ToyMode,              &copter.g2.toy_mode,         update
#endif
    SCHED_TASK(auto_disarm_check,     10,     50),
    SCHED_TASK(auto_trim,             10,     75),
#if RANGEFINDER_ENABLED == ENABLED
    SCHED_TASK(read_rangefinder,      20,    100),
#endif
#if PROXIMITY_ENABLED == ENABLED
    SCHED_TASK_CLASS(AP_Proximity,         &copter.g2.proximity,        update
#endif
```

Fig. 4. Evolution of the main loop code of flight controller:
from simple loops with given frequency (left) to RT OS with scheduling tasks (right)

In the beginning, Ardupilot / Arducopter software was designed to support APM controller hardware (APM stands for ArduPilot Mega and Ardu stands for Arduino), and the project goal was to use Arduino Mega board as a flight controller. Later, the code was abstracted from hardware and now it can be used in open-source platforms (APM, Pixhawk, Pixhack, see fig. 5) as well in proprietary solutions (Intel Aero, SnapDragon).

Actually, controllers may use the following types of system software:

- firmware without OS;
- real-time OS;
- Linux.

Fig. 5. Ardupilot/Arducopter compatible hardware on the market

The difference of controllers is based on their purposes: for amateur or commercial use, operate with slow or high speed, the necessity of having additional logic that implies using special sensors. Modern flight controllers act like ECU (Electronic Control Unit) for cars and even contain CAN (Controller Area Network) bus for the periphery. Some controllers also include a «safety co-processor» [18] to ensure the robustness of the flight state.

Note that when we move from the simple loop with device polling to real-time scheduling algorithms, we should take in account that process switching can add some delays to normal handling of real physical process control data, so such algorithms must be formally verified.

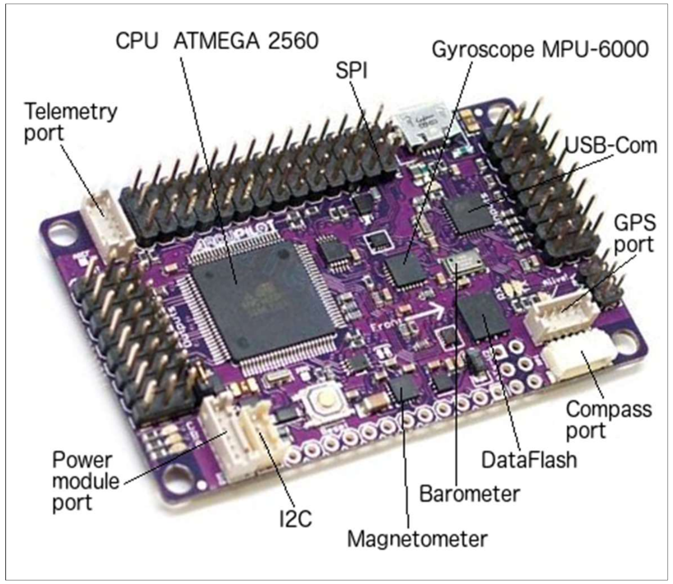


Fig. 6. Components of APM 2.6

The APM controller was built in 2011-2012 [18] as a result of initial Ardupilot project development by 3D Robotics company. Later, the non-profit organisation ardupilot.org was established. Today the controller is slightly outdated but it is cheap and available on market. The components of the controller are shown in fig.6:

- ATMEGA 2560 processor (8bit);
- barometer MS5611, SPI connection;
- 3-axes magnetometer (compass) HMC5843, I2C connection;
- 6-axes gyroscope and accelerometer MPU-6000, SPI connection;
- GPS: UART external interface;
- FrSky telemetry: UART external interface;
- external SPI interface.

In our current research work, we learned to work with this hardware and provided a transfer of necessary data to another controller to execute control algorithms. For us following control algorithms are most interesting:

- altitude (position) control;
- waypoint flight or curve-based flight;
- smoothing pilot's commands.

All algorithms are based on the PID-controller approach based on the Control Theory.

4.2 PID control

The PID (Proportional, Integral, Differential) controller is a feedback system for correcting the state of the control object (fig. 7). When controlling the object we calculate an error between current and desired state (for example, between current and set drone altitude), then based on current error we calculate the impact based on three parts with given coefficients.

- The proportional part (P) is responsible for the proportional reduction of the error (present).
- The integral part (I) is a statistical change of the error (past).
- The differential part (D) is the change of the error, its tendency to 0 (future).

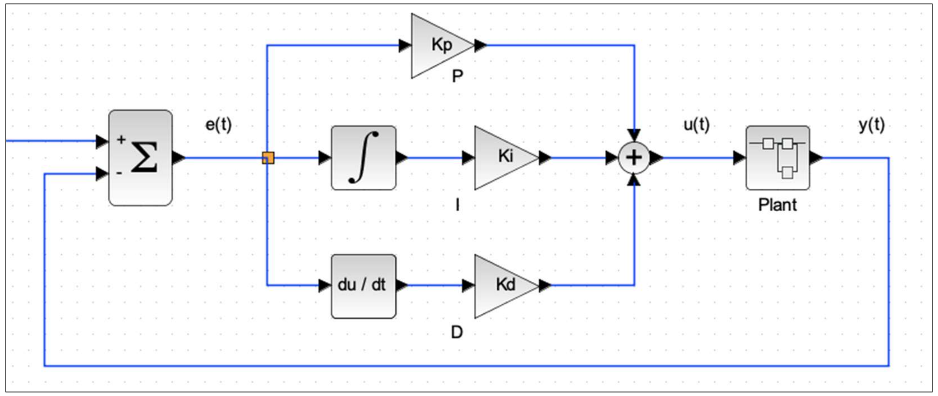


Fig. 7. PID-control scheme.

PID-control is some kind of abstraction when the control process is primarily based on the difference (error) but not on attempts to describe the exact physical model of the system, because during flight a variable wind can blow, the rotation of the propellers may be unstable, the center of mass is shifted and so on, but in such situations PID controller will detect the deviation and will try to make an impact to change it.

The analytic equation for the PID-control scheme:

$$u(t) = P + I + D = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de}{dt} \quad (1)$$

where K_p, K_i and K_d are the PID coefficients.

Presently, the coefficients are set up in the user interface of ground control software, for example, using Mission Planner [20]. Some guidance on coefficients' choice exists, and their values are known for standard drone assemblies but for custom quadcopter (for example, a heavyweight one) they should be identified as a result of experiments.

If a value of some coefficient is zero the controlled system may lose stability or smoothness of behaviour and the controller can be called P-controller, PD-controller or PI-controller.

4.3 PID control implementation in Ardupilot

According to [21], the current implementation of Ardupilot/Arducopter for altitude control uses a combined scheme of P- and PID-controller. The orientation for each axis is controlled with a special P-controller to convert the angle error (the difference between the specified angle and the actual one) to the desired rotational speed. Then a PID-controller converts the rotational speed error to a high-level motor command. A special part of the P-controller – «square root controller» – at first represents the angle function linearly and then uses its square root approximation. [22] provides a discussion of the overall control scheme. This scheme and its current refactoring are shown in fig. 8.

The scheme demonstrates that all the axes have an influence on each other and the construction of the whole analytical equation like (1) is a very challenging task. Also, it shows some input transformation blocks that help to control smoothly [22].

- User's actions to control the drone (for example, commands to move it) are not translated to drone motors directly because they are not regular, may be conflicting and have different

duration. They are buffered and divided into small pieces of actions, and an internal scheduler sends them with some constant frequency.

- Based on the collected set of user's actions and parameters, a final action may be calculated in advance and then posted to the scheduler.

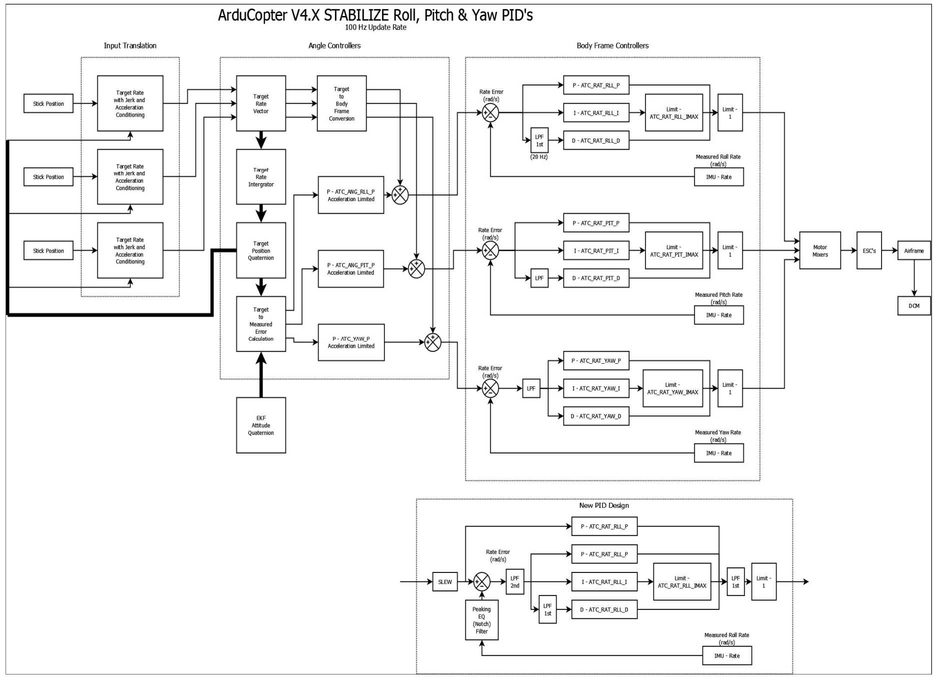


Fig. 8. Overall PID-control scheme (taken from the ArduCopter project)

To provide flight along a given trajectory it is necessary to carry out maneuvers in advance, otherwise the drone will have to inefficiently turn in sharp corners. The approach of constructing smooth curves for unmanned air vehicles is called L1 [23]. For drones, this method was changed because of other physical characteristics and implemented in the AC_WPNav library.

5. Towards a partitioned Ardupilot code

5.1 POK

POK (Partitioned Operating System Kernel) is a name of OS created in France by Julien Delange during his PhD research.

The main features of this work are the following:

- MDE (Model-Driven Engineering) approach: initial OS kernel configuration was defined with the AADL language, which allowed to generate code and represent the configuration graphically;
- it is a good proof-of-concept of working models and examples;
- the system partially conforms to the ARINC 653 standard for real-time onboard aviation systems;
- the system uses protected partitions with time and memory space resources isolation;

- two types of real-time processes schedulers with different strategies exist – the partition planner and the process planner in each partition;
- controllable message exchange between processes or using BlackBoard concept.

The use of OS designed according to avionics standards and providing isolation of processes and verifiable interprocess communications increases the robustness of the solution at the system level. By browsing source code [5], we have created a scheme of internal POK architecture (see fig. 9). It consists of three layers: Arch with platform-dependent code (open-source repository includes implementations for three platforms: x86-qemu, PowerPC, and Spark), Core as the internal code of kernel and syscalls, and libpok that can be used as an API.

The ARINC 653-compatible API provides a possibility to work with partitions, processes, locks, ports, queries, and messages in a standardized way.

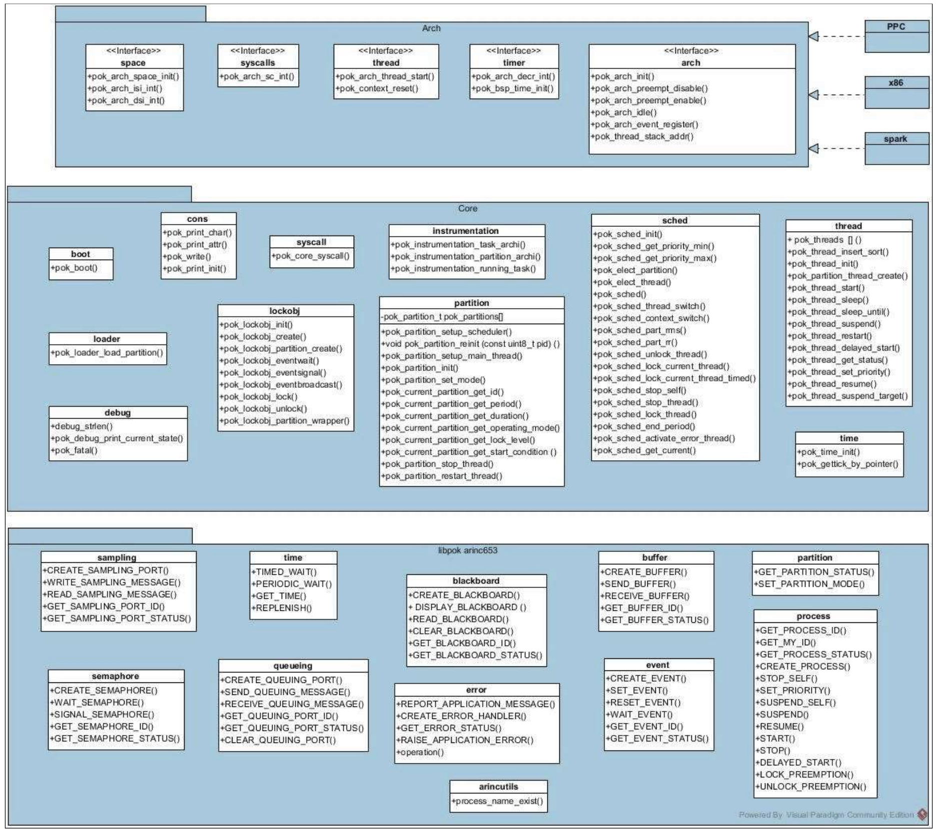
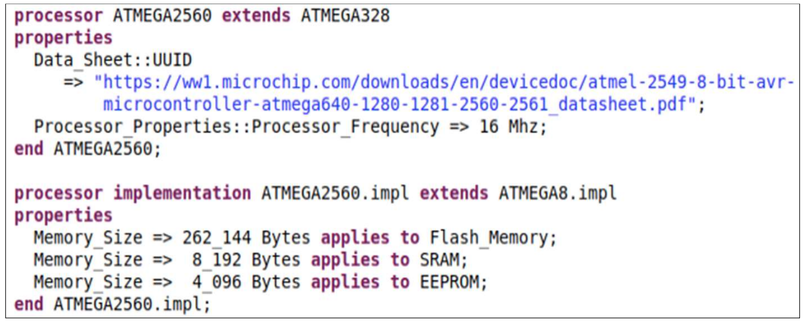


Fig. 9. Overall scheme of POK architecture



```
processor ATMEGA2560 extends ATMEGA328
properties
  Data_Sheet::UUID
     => "https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-
         microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf";
  Processor_Properties::Processor_Frequency => 16 Mhz;
end ATMEGA2560;

processor implementation ATMEGA2560.impl extends ATMEGA8.impl
properties
  Memory_Size => 262_144 Bytes applies to Flash_Memory;
  Memory_Size =>  8_192 Bytes applies to SRAM;
  Memory_Size =>  4_096 Bytes applies to EEPROM;
end ATMEGA2560.impl;
```

Fig. 10. Defining a simple processor model as an extension in AADL

5.2 AADL

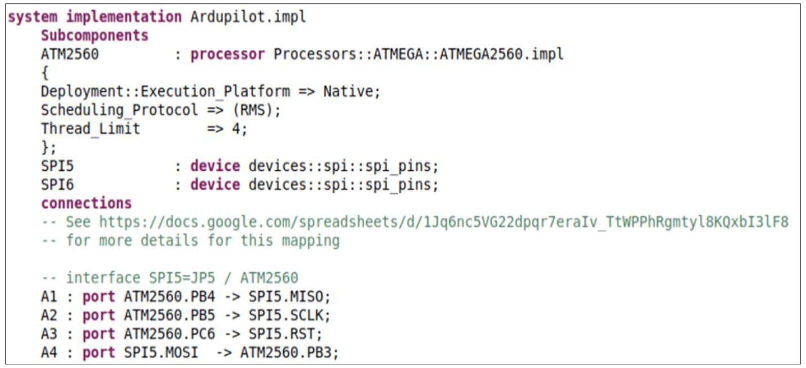
AADL (Architecture Analysis & Design Language) [24] provides system modeling engineers with following capabilities:

- to develop a top-level system design;
- to think in abstractions and then write implementations;
- to see a graphical representation of the code;
- to generate code using integrated tools such as Ocarina;
- to validate properties of developed systems automatically;
- to create certified solutions.

In essence, AADL acts like «executable UML» but at the top level and it is designed not for expressing algorithms, but for describing systems with additional safety requirements.

The language allows to describe systems and components as sets of extensible properties to model both hardware and software parts. For example, if we need to define a model for the APM board (already introduced in fig. 6) we may first define a model for ATMEGA 2560 processor as an extension of existing processor ATMEGA 328 (fig. 10) with another frequency and memory sizes and then use this model in a board model definition (fig. 11) additionally specifying connections by ports with use of SPI buses.

In our work, AADL is used to model the whole structure of the partitioned OS for the flight controller with the possibility to generate and validate its kernel configuration.



```
system implementation Ardupilot.impl
  Subcomponents
  ATM2560         : processor Processors::ATMEGA::ATMEGA2560.impl
  {
  Deployment::Execution_Platform => Native;
  Scheduling_Protocol => (RMS);
  Thread_Limit        => 4;
  };
  SPI5            : device devices::spi::spi_pins;
  SPI6            : device devices::spi::spi_pins;
  connections
  -- See https://docs.google.com/spreadsheets/d/1Jq6nc5VG22dpqr7eraIv_TtWPPhRgmtyl8KQxbI3lF8
  -- for more details for this mapping

  -- interface SPI5=JP5 / ATM2560
  A1 : port ATM2560.PB4 -> SPI5.MISO;
  A2 : port ATM2560.PB5 -> SPI5.SCLK;
  A3 : port ATM2560.PC6 -> SPI5.RST;
  A4 : port SPI5.MOSI  -> ATM2560.PB3;
```

Fig. 11. Defining an implementation of APM board with a processor and SPI buses in AADL

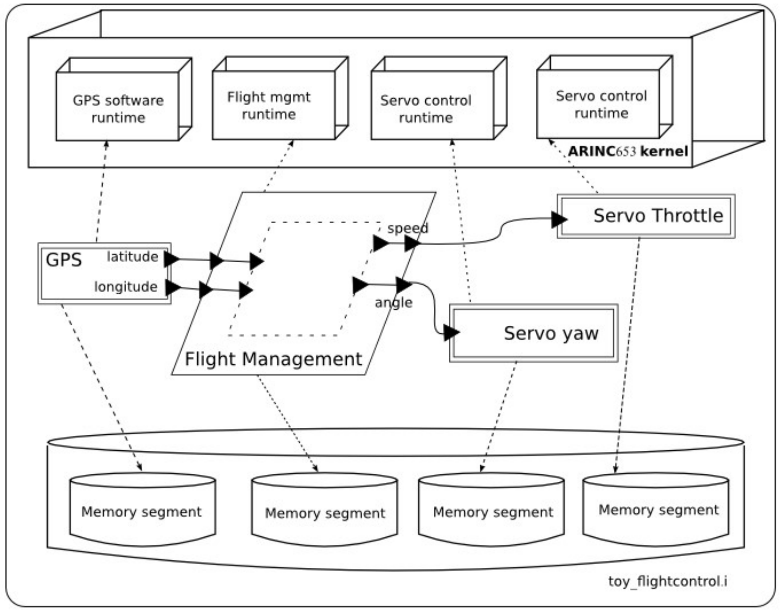


Fig. 12. Flight controller case study in a POK way [25]

5.3 Flight controller system in a POK way

In this subsection, we define a structure of real-time OS for the flight controller with increased reliability requirements using POK way. In [25], POK authors give some tips on such structure definition (see fig. 12).

```
system implementation ardupilot.i
subcomponents
   kernel         : processor poklib::pok_kernel.demo_four_partitions;
   mem            : memory    ardupilot_platform::mem.i;
   prs_gps        : process   ardupilot_software::process_gps.i;
   prs_mgmt       : process   ardupilot_software::process_mgmt.i;
   prs_throttle   : process   ardupilot_software::process_throttle.i;
   prs_yaw        : process   ardupilot_software::process_yaw.i;
connections
  c1: port prs_gps.altitude   -> prs_mgmt.altitude;
  c2: port prs_gps.latitude   -> prs_mgmt.latitude;
  c3: port prs_gps.longitude  -> prs_mgmt.longitude;
  c4: port prs_mgmt.speed     -> prs_throttle.speed;
  c5: port prs_mgmt.angle     -> prs_yaw.angle;
properties
   Actual_Memory_Binding => (reference (mem.segment1)) applies to prs_gps;
   Actual_Memory_Binding => (reference (mem.segment2)) applies to prs_mgmt;
   Actual_Memory_Binding => (reference (mem.segment3)) applies to prs_throttle;
   Actual_Memory_Binding => (reference (mem.segment4)) applies to prs_yaw;

   Actual_Processor_Binding => (reference (kernel.partition1)) applies to prs_gps;
   Actual_Processor_Binding => (reference (kernel.partition2)) applies to prs_mgmt;
   Actual_Processor_Binding => (reference (kernel.partition3)) applies to prs_throttle;
   Actual_Processor_Binding => (reference (kernel.partition4)) applies to prs_yaw;

   POK::Additional_Features => (console, libc_stdio, libc_stdlib) applies to kernel.partition1;
   POK::Additional_Features => (libmath, console, libc_stdio, libc_stdlib) applies to kernel.partition2;
   POK::Additional_Features => (console, libc_stdio, libc_stdlib) applies to kernel.partition3;
   POK::Additional_Features => (console, libc_stdio, libc_stdlib) applies to kernel.partition4;
end ardupilot.i;
```

Fig. 13. System structure definition for flight controller software in AADL [25]

```
process process_mgmt
features
   altitude    : in data port poklib::integer;
   latitude    : in data port poklib::float;
   longitude   : in data port poklib::float;
   speed       : out data port poklib::integer;
   angle       : out data port poklib::integer;
end process_mgmt;

process implementation process_mgmt.i
subcomponents
   thr : thread thr_mgmt.i;
connections
  p1: port altitude     -> thr.altitude;
  p2: port latitude     -> thr.latitude;
  p3: port longitude    -> thr.longitude;
  p4: port thr.speed    -> speed;
  p5: port thr.angle    -> angle;
end process_mgmt.i;

thread thr_mgmt extends poklib::thr_periodic
features
   altitude    : in data port poklib::integer;
   latitude    : in data port poklib::float;
   longitude   : in data port poklib::float;
   speed       : out data port poklib::integer;
   angle       : out data port poklib::integer;
properties
   Initialize_Entrypoint => classifier (ardupilot_software::spg_flt_mgmt_init);
end thr_mgmt;

thread implementation thr_mgmt.i
calls
   call1 : { pspg : subprogram spg_flt_mgmt_simulation;};
connections
   c1: parameter altitude   -> pspg.altitude;
   c2: parameter latitude   -> pspg.latitude;
   c3: parameter longitude  -> pspg.longitude;
   c4: parameter pspg.speed -> speed;
   c5: parameter pspg.angle -> angle;
end thr_mgmt.i;

subprogram spg_flt_mgmt_simulation extends poklib::spg_c
features
   altitude    : in parameter poklib::integer;
   latitude    : in parameter poklib::float;
   longitude   : in parameter poklib::float;
   speed       : out parameter poklib::integer;
   angle       : out parameter poklib::integer;
properties
   Source_Name => "flt_mgmt_simulation";
   Source_Text      => ("../../../flt-mgmt.o");
end spg_flt_mgmt_simulation;

subprogram spg_flt_mgmt_init extends poklib::spg_c
properties
   Source_Name => "flt_mgmt_init";
   Source_Text      => ("../../../flt-mgmt.o");
end spg_flt_mgmt_init;
```

Fig. 14. Linking a process with variables and ports to a thread and a compiled C-code [25]

The scheme is generated from AADL code and represents the partitioning: four partitions and four memory segments for GPS interoperation, runtime support for throttle servo control, and runtime support for yaw and flight management to provide a PID-style control.

Initial AADL code is shown in fig. 13 where *demo_four_partitions* defines a processor with four partitions, a major time frame, time slices for the partitions, and scheduling policy; *partitions 1..4* define partitions with given individual schedulers and additional user libraries; *segments 1..4* define memory segments of given types and sizes; *connections* section defines ports to support interprocess communication between parts of the system; *prs_gps*, *prs_mgmt*, *prs_throttle* and *prs_yaw* define actually working threads within partitions.

Fig. 14 illustrates the AADL approach to forwarding system variables to processes through ports and then to features in threads inside these processes, as well as to binding those to external object code that will run in these threads. We see that the stabilizing PID controller gets a tuple (altitude, latitude, longitude) and adjusts speed and angle of a quadcopter.

So, configuring OS using this approach provides a Model-Driving approach to design software for the flight controller. The process interoperations are clear and the configuration is verifiable, therefore this increases the reliability of the solution.

5.4 Current state of our solution

To obtain a model for developing OS and testing PID controllers we propose an architecture shown in fig. 15.

We took the 3.2.1 branch of Ardupilot software and modified its code to send current state data of a quadcopter through SPI from the APM controller to a different controller that executes a partitioned OS and provides a flight control based a stabilizing PID controller. Fig.16 shows the current hardware connection.

This solution is the first step to migrate the whole flight controller logic to a fully-partitioned code – we start from using the APM controller as a gateway to quadcopter hardware. The patched code of the APM transfers input data through SPI to a partitioned process (prc_gps in fig.12 and 13) that provides flight-control and sends control data through SPI back to APM. The CRC (cyclic redundancy check) algorithm is used to ensure the correctness of the transaction. We plan to add partitions with additional monitoring processes to ensure dynamic properties of safety and stability.

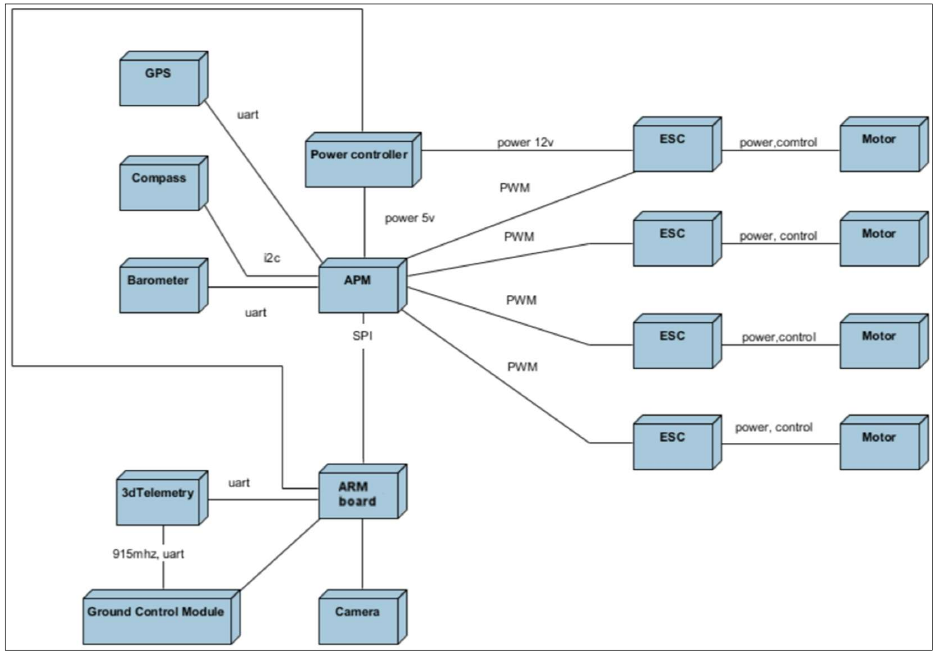


Fig. 15. Our intermediate architecture: APM controller, its sensors, motors and ARM board connection using SPI bus

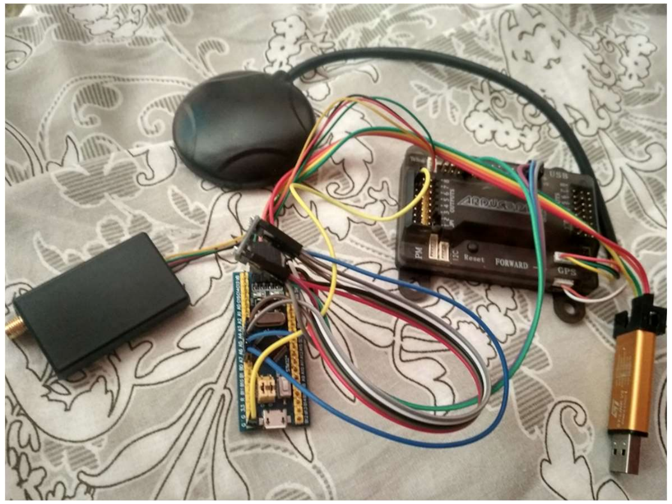
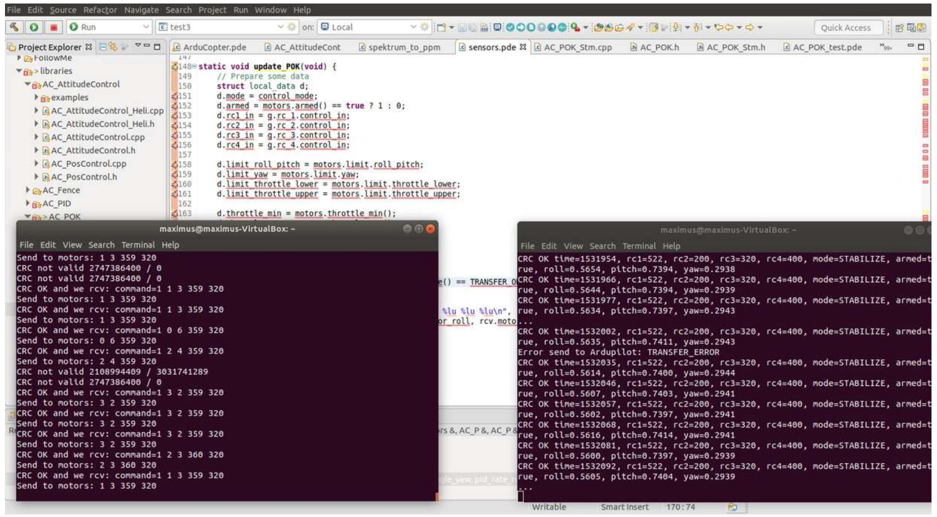


Fig. 16. Connection of the controllers using SPI bus

Fig.17 shows two windows of USB-Com port devices monitoring the APM and the new controller with partitioned OS that provides PID-control based on real sensors data.

We used the ARM M3-based board STM32f103 as the controller hardware. The source code of FreeRTOS was used to study some platform-related stuff. The ARM M7-based board STM32F746 was used to work with ARM MPU regions [26] with debugging process based on OpenOCD and STM HAL library to deal with periphery. We are working to deploy the solution on the Raspberry Pi board as some open source programs for it [27] allow to work with internal Broadcom hardware. We are planning to describe POK porting issues in further papers.

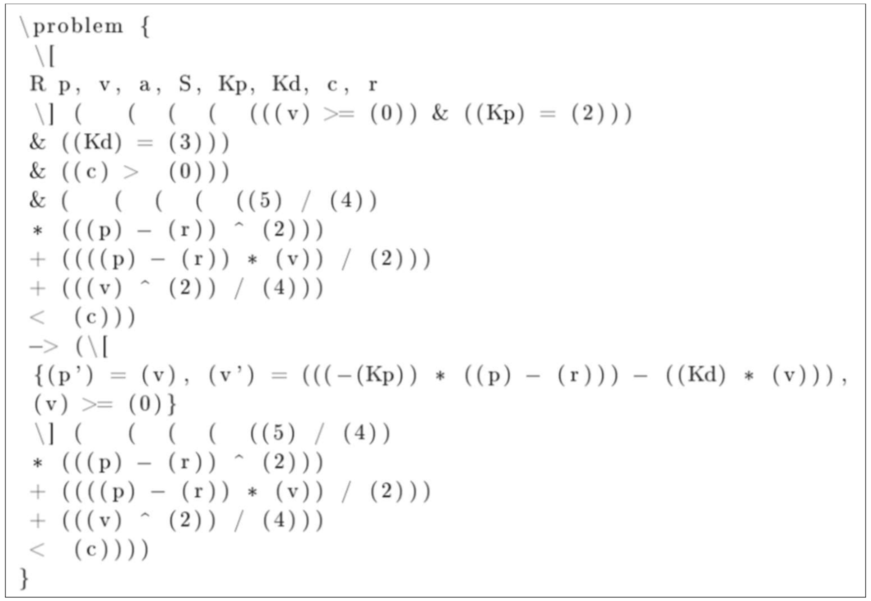


```
static void update_POK(void) {
    // Prepare some data
    struct local_data d;
    d.mode = control_mode;
    d.armed = motors.armed() == true ? 1 : 0;
    d.rc1_in = g.rc_1.control_in;
    d.rc2_in = g.rc_2.control_in;
    d.rc3_in = g.rc_3.control_in;
    d.rc4_in = g.rc_4.control_in;

    d.limit_roll_pitch = motors.limit.roll_pitch;
    d.limit_yaw = motors.limit.yaw;
    d.limit_throttle_lower = motors.limit.throttle_lower;
    d.limit_throttle_upper = motors.limit.throttle_upper;

    d.throttle_min = motors.throttle_min();
```

```
Send to motors: 1 3 359 320
CRC not valid 2747386400 / 0
CRC not valid 2747386400 / 0
CRC OK and we rcv: command=1 1 3 359 320
Send to motors: 1 3 359 320
CRC OK and we rcv: command=1 1 3 359 320
Send to motors: 1 3 359 320
CRC OK and we rcv: command=1 0 6 359 320
Send to motors: 0 6 359 320
CRC OK and we rcv: command=1 2 4 359 320
Send to motors: 2 4 359 320
CRC not valid 2108994409 / 3031741289
CRC not valid 2747386400 / 0
CRC OK and we rcv: command=1 3 2 359 320
Send to motors: 3 2 359 320
CRC OK and we rcv: command=1 3 2 359 320
Send to motors: 3 2 359 320
CRC OK and we rcv: command=1 3 2 359 320
Send to motors: 3 2 359 320
CRC OK and we rcv: command=1 2 3 360 320
Send to motors: 2 3 360 320
CRC OK and we rcv: command=1 1 3 359 320
Send to motors: 1 3 359 320
```

```
CRC OK time=1531954, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5654, pitch=0.7394, yaw=0.2938
CRC OK time=1531966, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5644, pitch=0.7394, yaw=0.2939
CRC OK time=1531977, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5634, pitch=0.7397, yaw=0.2943
...
CRC OK time=1532002, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5635, pitch=0.7411, yaw=0.2943
Error send to Ardupilot: TRANSFER_ERROR
CRC OK time=1532035, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5614, pitch=0.7400, yaw=0.2944
CRC OK time=1532046, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5607, pitch=0.7403, yaw=0.2941
CRC OK time=1532057, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5602, pitch=0.7397, yaw=0.2941
CRC OK time=1532068, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5616, pitch=0.7414, yaw=0.2941
CRC OK time=1532081, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5600, pitch=0.7397, yaw=0.2939
CRC OK time=1532092, rc1=522, rc2=200, rc3=320, rc4=400, mode=STABILIZE, armed=t
rue, roll=0.5605, pitch=0.7404, yaw=0.2939
...
```

Fig. 17. USB-Com port monitoring of two interconnected controllers



```
\problem {
 \[
R p, v, a, S, Kp, Kd, c, r
 \] (  (  (  (  (((v) >= (0)) & ((Kp) = (2)))
& ((Kd) = (3)))
& ((c) >  (0)))
& (  (  (  (  ((5) / (4))
* (((p) - (r)) ^ (2)))
+ ((((p) - (r)) * (v)) / (2)))
+ (((v) ^ (2)) / (4)))
<  (c)))
-> (\[
{(p') = (v), (v') = (((-(Kp)) * ((p) - (r))) - ((Kd) * (v))),
(v) >= (0)}
 \] (  (  (  (  ((5) / (4))
* (((p) - (r)) ^ (2)))
+ ((((p) - (r)) * (v)) / (2)))
+ (((v) ^ (2)) / (4)))
<  (c))))
}
```

Fig. 18. Hybrid program in KeYmaera syntax to check the stability

6. Formal verification

6.1 Verification at the cyber-physical system level

Cyber-physical system is a computer science abstraction of controllable physical process. It consists of two parts:

- a Cyber part – discrete controller;
- a Physical part – continuous model of the system usually expressed in ordinary differential equations.

These systems can be modeled as Hybrid automata with discrete-time and continuous-time transitions. Such models are known as Hybrid ones and are specified using the Dynamic Differential Logic. The approach is a kind of sliding state changing [28]. According to A. Platzer [29], the syntax of hybrid programs is defined as follows:

$$\alpha ::= x := e \,|\, ?Q \,|\, x' = f(x)Q \,|\, \alpha \cup \alpha \,|\, \alpha;\alpha \,|\, \alpha^* \tag{2}$$

where α is a meta-variable for a hybrid program; x is a meta-variable for the program variables; e is a meta-variable for first-order real-valued terms; f is a meta-variable for continuous real functions; and Q is a meta-variable for first-order formulas over real numbers. The construct «;» means sequential composition, «∪» is non-deterministic choice, «?» is the condition operator, and «α^*» is non-deterministic iteration (like Kleene-star) [30].

Here we discuss a formal verification of a simple PD-controller (the simplification of PID-controller) given from example [31]. The model of the system is represented as a Hoare's triple:

$$init \Longrightarrow [controller](req) \tag{3}$$

where *init* – a precondition; *controller* – a hybrid model; *req* – requirements that are invariants.

Then, we decompose the system into precondition, continuous PD-controller and requirements. Precondition:

$$init :== v \geq 0 \wedge c > 0 \wedge K_p = 2 \wedge K_d = 3 \wedge V(p, p_r, v) < c \tag{4}$$

where v is the velocity; c – a number greater than zero; K_p – a proportional part coefficient; K_d – a differential part coefficient; $V(p, p_r, v)$ – is a Lyapunov function.

The continuous state:

$$controller: == p' = v, v' = -K_p \cdot (p - p_r) - K_d \cdot v \tag{5}$$

where p is a current position; p_r – a resulting position.

For the requirement, it is proposed to try a stability check using the Lyapunov method in the form:

$$req :== V(p, p_r, v) < c. \tag{6}$$

The stability means here that the UAV during control will be stabilized around a given point in space. The Lyapunov function is defined in a quadratic form as follows:

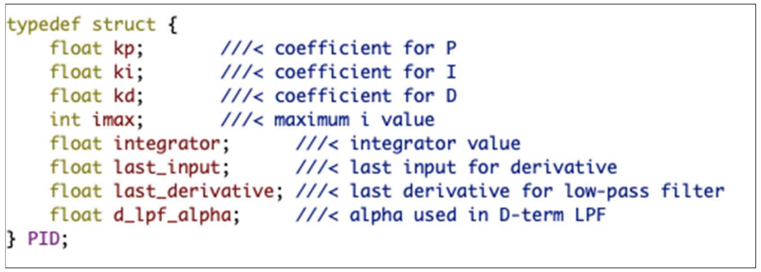
$$V(p, p_r, v) = 5/4 \cdot (p - p_r)^2 + (p - p_r) \cdot v/2 + v^2/4 \tag{7}$$

Using KeYmaera tool (see fig.18 for the system (2)-(7) representation in code in the form of Hybrid program) it is possible to automatically verify the stability of the CPS that modeling the PD-controller. For the real PID-controller of the drone (see fig. 8) it is hard to obtain an analytical form of the whole model and to find a Lyapunov function to prove the stability. Possibly, a special kind of linearisation is required here. So it is a very challenging task and a subject of further research. Moreover, additional research is required to find ways to generate proof obligations with preconditions, postconditions and invariants in hybrid automaton states to help automatically prove stability of a system (our initial results are desrcribed in [32]).

6.2 Verification at the code level

Let's discuss a verification technique for a PID controller using the Weakest Precondition (WP) method and adding ACSL annotations [33] into C code. The WP approach as an extension of the Hoare's logic was proposed by Dijkstra. It requires a precondition to be as simple as possible to surely reach the corresponding postcondition. In this case, the further program verification will be as follows: first, calculate $W = wp(f, Q)$, go from the end of Q to the start of the function, and then post a task to prove $P => W$ to a theorem prover (we use Frama-C tool with WP plugin and its internal Alt-Ergo prover).

For example, below we deductively prove several functions from the code that performs the PID control based on the Ardupilot code. In this code, the intermediate values are not calculated every time but are stored in the PID structure (fig. 19).

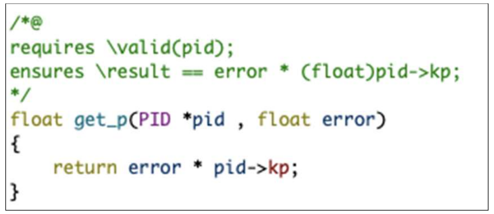


```
typedef struct {
    float kp;       ///< coefficient for P
    float ki;       ///< coefficient for I
    float kd;       ///< coefficient for D
    int imax;       ///< maximum i value
    float integrator;      ///< integrator value
    float last_input;      ///< last input for derivative
    float last_derivative; ///< last derivative for low-pass filter
    float d_lpf_alpha;     ///< alpha used in D-term LPF
} PID;
```

Fig. 19. PID structure (from Ardupilot code)

To prove the code, we write annotations for postconditions, preconditions, and side effects of functions in ISO-standardized ACSL language. Consider first the simplest code with annotations for the function *float get_p (PID * pid, float error*) (fig. 20).

This specification is fairly obvious: *\valid* defines the requirement of a non-NULL pointer to the PID structure, *\result* – the return value, *requires* means the precondition, *ensures* – the postcondition. However, even such a simple function cannot be proven because multiplying of two real numbers can cause overflow with an unexpected result. Therefore. the verification tool cannot guarantee the correctness of this code without explicitly using «infinite» Real logical type. To do this, we run the proof tool with the parameter *"-wp-model typed+real"* and (for this moment) we abandon possible floating-point errors with loss of precision.



```
/*@
requires \valid(pid);
ensures \result == error * (float)pid->kp;
*/
float get_p(PID *pid , float error)
{
    return error * pid->kp;
}
```

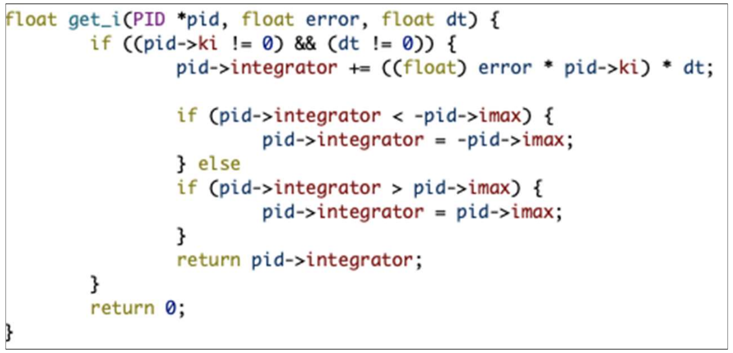
Fig. 20. A simple function with annotations

To prove the code of function *float get_i (PID * pid, float error, float dt),* it is necessary to construct lemmas describing the verifying code in terms of logic, similar to the examples in [34].

Firstly, we note that the function can change the value of *pid->integrator* and there are three cases:

- pid-> integrator <-pid->imax: it is limited to -pid-> imax;
- pid-> integrator> pid->imax: it is limited to pid-> imax;
- otherwise, that is, (integrator> = -max) and (integrator <= max): no change of pid-> integrator.

At the same time, there must first be a change of *pid-> integrator* to (*error * pid-> ki) * dt*. Therefore, the solution is to create a set of lemmas in ACSL terminology and a logic that will be used as a function in the *ensures* section.

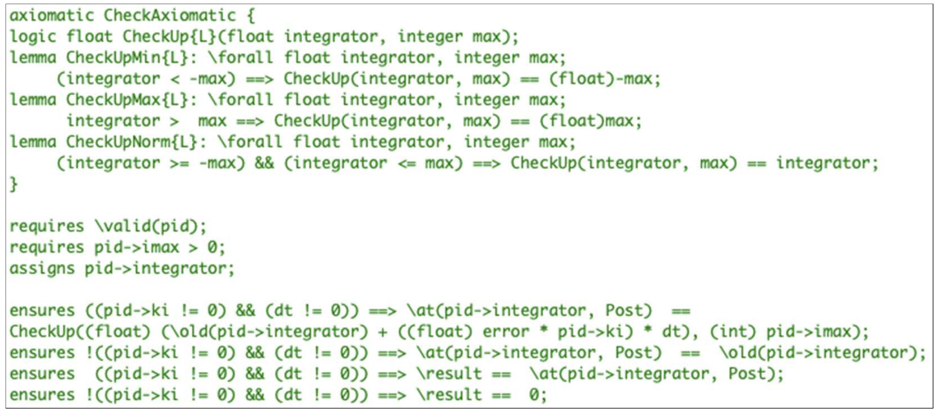


```
float get_i(PID *pid, float error, float dt) {
        if ((pid->ki != 0) && (dt != 0)) {
                pid->integrator += ((float) error * pid->ki) * dt;

                if (pid->integrator < -pid->imax) {
                        pid->integrator = -pid->imax;
                } else
                if (pid->integrator > pid->imax) {
                        pid->integrator = pid->imax;
                }
                return pid->integrator;
        }
        return 0;
}
```

Fig. 21. A function to prove

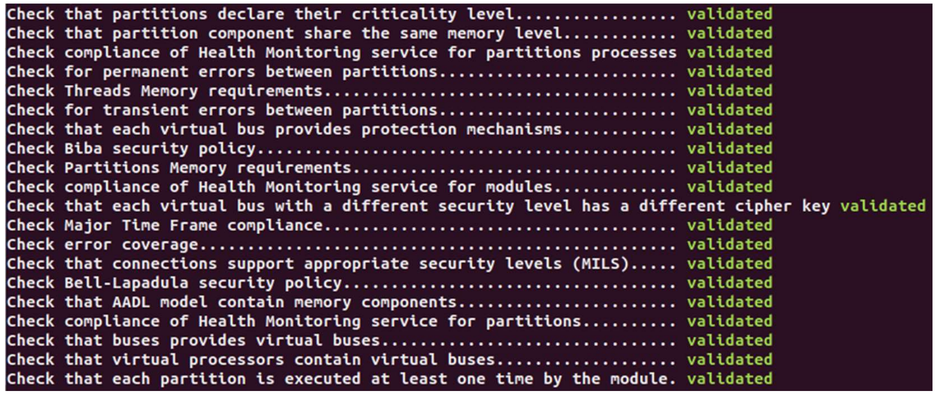
Secondly, we note that the function returns 0 if the first condition does not hold and it does not change the value of *pid->integrator*. To describe the postcondition, we provide a description of the guard conditions in a form of implications. In fig. 22 we show a specification for the function in fig. 21. Here *\old* is the memory state before calling the function and *\at(..., Post)* – after calling it.

Fig 23 shows that the specification often takes even more space than the code itself, and writing it significantly changes the way of development; it ensures the quality of the code by coding the algorithms twice: in programming and logic languages.



```
axiomatic CheckAxiomatic {
logic float CheckUp{L}(float integrator, integer max);
lemma CheckUpMin{L}: \forall float integrator, integer max;
     (integrator < -max) ==> CheckUp(integrator, max) == (float)-max;
lemma CheckUpMax{L}: \forall float integrator, integer max;
      integrator >  max ==> CheckUp(integrator, max) == (float)max;
lemma CheckUpNorm{L}: \forall float integrator, integer max;
     (integrator >= -max) && (integrator <= max) ==> CheckUp(integrator, max) == integrator;
}

requires \valid(pid);
requires pid->imax > 0;
assigns pid->integrator;

ensures ((pid->ki != 0) && (dt != 0)) ==> \at(pid->integrator, Post)  ==
CheckUp((float) (\old(pid->integrator) + ((float) error * pid->ki) * dt), (int) pid->imax);
ensures !((pid->ki != 0) && (dt != 0)) ==> \at(pid->integrator, Post)  ==  \old(pid->integrator);
ensures  ((pid->ki != 0) && (dt != 0)) ==> \result ==  \at(pid->integrator, Post);
ensures !((pid->ki != 0) && (dt != 0)) ==> \result ==  0;
```

Fig. 22. A specification for the last function



```
Check that partitions declare their criticality level................ validated
Check that partition component share the same memory level........... validated
Check compliance of Health Monitoring service for partitions processes validated
Check for permanent errors between partitions........................ validated
Check Threads Memory requirements.................................... validated
Check for transient errors between partitions........................ validated
Check that each virtual bus provides protection mechanisms........... validated
Check Biba security policy........................................... validated
Check Partitions Memory requirements................................. validated
Check compliance of Health Monitoring service for modules............ validated
Check that each virtual bus with a different security level has a different cipher key validated
Check Major Time Frame compliance.................................... validated
Check error coverage................................................. validated
Check that connections support appropriate security levels (MILS)..... validated
Check Bell-Lapadula security policy.................................. validated
Check that AADL model contain memory components...................... validated
Check compliance of Health Monitoring service for partitions......... validated
Check that buses provides virtual buses.............................. validated
Check that virtual processors contain virtual buses.................. validated
Check that each partition is executed at least one time by the module. validated
```

Fig. 23. Validation of AADL code

6.3 Validation of OS config

Thanks to AADL, a language with formal semantics, it is possible to analyze the code in it, create a formal model and then validate it. Some ideas of AADL code checking are given in [4] and [24].

In fig. 23 we demonstrate the result of automatic code validation for the model presented in Section 5 at the phase of the code generation process.

7. Conclusion

During the research, we studied information on modern drones with open-source software and commodity hardware. We did a detailed analysis of the Ardupilot software and the APM board. We touched some modern approaches to the organization of operating systems for such devices using partitions, AADL language and MDE applied to the OS configuration, code generation and validation.

We have developed a demo system, in which a quadcopter state is transferred from the APM board to a different ARM-based board with the control logic implemented in parallel partitioned processes using ports for interprocess communication.

As a result, we propose a design of software solution for UAVs with enhanced reliability requirements. This solution should ensure robustness at the following five levels.

- OS Level. Using a partitioned real-time OS will provide low-level scheduling and process isolation.
- Validation level. The level of interaction between the processes through ports and messages will be described in AADL. Additional model checking is available.
- Code level. Source code annotations and its deductive proof.
- Level of processes-monitors (dynamic testing). Monitoring processes can ensure that the safety conditions of the running system are maintained.
- Level of cyber-physical system (static verification). Proof of correctness of mathematical models of physical processes (safety and stability).

References / Список литературы

[1]. Avionics application software standard interface, part 1 – required services, ARINC specification 653P1-3, November 15, 2010. Aeronautical radio, inc. (ARINC).

[2]. Delange J., Lec L. POK, an ARINC653-compliant operating system released under the BSD license. In Proc. of the 13th Real-Time Linux Workshop, 2011.

[3]. Delange J., Gilles O., Hugues J., and Pautet L. Model-Based Engineering for the Development of ARINC653 Architectures. SAE International Journal of Aerospace, vol. 3, no. 1, 2010, pp. 79-86.

[4]. Hugues J., Delange J. Model-based design and automated validation of ARINC653 architectures using the AADL. In Cyber-Physical System Design from an Architecture Analysis Viewpoint, Springer, 2017, pp. 33-52.

[5]. POK kernel repository. Available at: https://github.com/pok-kernel/pok.

[6]. Mallachiev K.M., Pakulin N.V., Khoroshilov A.V. Design and architecture of real-time operating system. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 181-192. DOI: 10.15514/ISPRAS-2016-28(2)-12.

[7]. Solodelov Yu.A., Gorelits N.K. Certifiable onboard real-time operation system JetOS for Russian aircrafts design. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017, pp. 171-178 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-10 / Солоделов Ю.А., Горелиц Н.К. Сертифицируемая бортовая операционная система реального времени JetOS для российских проектов воздушных судов. Труды ИСП РАН, том 29, вып. 3, 2017 г.. стр. 171-178.

[8]. JetOS. Available at: https://github.com/yoogx/forge.ispras.ru-git-chpok.

[9]. Khoroshilov A.V. On formalization of operating systems behaviour verification. In Proc. of the Eleventh International Conference on Computer Science and Information Technologies, Revised Selected Papers, 2017, pp. 168-172.

[10]. Kulyamin V.V., Lavrischeva E.M., Mutilin V.S., Petrenko A.K. Verification and analysis of variable operating systems, Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 3, 2016, pp. 189–208 (in Russian). DOI: 10.15514/ISPRAS-2016-28(3)-12 / Кулямин В.В., Лаврищева Е.М., Мутилин В.С., Петренко А.К. Верификация и анализ вариабельных операционных систем. Труды ИСП РАН, том 28, вып. 3, 2016 г., стр. 189-208.

[11]. Khoroshilov A.V., Kuliamin V.V., Petrenko A.K. Verification of Operating System Components. System Informatics, No. 10, 2017, pp. 11-22.

[12]. Klein G. Operating system verification – An overview. Sadhana, vol. 34, no. 1, 2009, pp. 27-69

[13]. Giernacki W. et al. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In Proc. of the 22nd International Conference on Methods and Models in Automation and Robotics (MMAR), 2017, pp. 37-42.

[14]. Crazyflie AADL Case Study. Available at: https://github.com/OpenAADL/Crazyflie.

[15]. Santoro C. How does a Quadrotor fly? A journey from physics. Available at: https://www.slideshare.net/corradosantoro/quadcopter-31045379.

[16]. Ardupulot project. Available at: http://ardupilot.org/copter.

[17]. Ardupulot project on Github. Available at: https://github.com/ArduPilot/ardupilot.

[18]. The CUAV Pixhack V3 flight controller board. Available at: https://docs.px4.io/en/flight_controller/pixhack_v3.html.

[19]. History of Ardupilot. Available at: http://ardupilot.org/planner2/docs/common-history-of-ardupilot.html.

[20]. Ardupilot. Advanced Tuning. Available at: http://ardupilot.org/copter/docs/tuning.html.

[21]. Copter Attitude Control. Available at: http://ardupilot.org/dev/docs/apmcopter-programming-attitude-control-2.html.

[22]. Hall Leonard. Practical PID implementation and the new Position Controller. ArduPilot UnConference, 2018. Available at: https://www.youtube.com/watch?v=-PC69jcMizA.

[23]. Park S., Deyst J., How J. A new nonlinear guidance logic for trajectory tracking. In Proc. of the AIAA guidance, navigation, and control conference and exhibit, 2004.

[24]. Delange J. AADL in Practice: Become an expert in software architecture modeling and analysis. Reblochon Development Company, 2017, 252 p.

[25]. POK. Examples. Case Study Ardupilot. Available at: https://github.com/pok-kernel/pok/tree/master/examples/case-study-ardupilot

[26]. Joseph Yiu. The Definitive Guide to the ARM Cortex-M3, 2nd Edition. Newnes, 2009, 479 p. / Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство. Пер. с англ. АВ Евстифеева. Додэка-XXI, 2012, 552 стр.

[27]. RaspberryPi-FreeRTOS. Demo. Drivers. Available at: https://github.com/jameswalmsley/RaspberryPi-FreeRTOS/tree/master/Demo/Drivers

[28]. Brandtstädter H. Sliding mode control of electromechanical systems. PhD Thesis, Technische Universität München, 2009.

[29]. Platzer A. Logical foundations of cyber-physical systems. Springer, 2018, 639 p.

[30]. Sergey Staroletov, Nikolay Shilov et al. Model-Driven Methods to Design of Reliable Multiagent Cyber-Physical Systems. In Proc. of the Conference on Modeling and Analysis of Complex Systems and Processes (MACSPro 2019), 2019.

[31]. Jan-David Quesel, Stefan Mitsch et al. How to model and prove hybrid systems with KeYmaera: a tutorial on safety. International Journal on Software Tools for Technology Transfer, vol. 18, №. 1, 2016, pp. 67-91.

[32]. Baar T., Staroletov S.M. A control flow graph based approach to make the verification of cyber-physical systems using KeYmaera easier. Modeling and Analysis of Information Systems, vol. 25, №. 5, 2018, pp. 465-480.

[33]. Patrick Baudin, Jean-Christophe Filliâtre et al. ACSL: ANSI C Specification Language. Frama-C, 2008, 81 p.

[34]. Jochen Burghardt, Andreas Carben et al. ACSL by Example. DEVICE-SOFT project publication. Fraunhofer FIRST Institute, 2010, 134 p.

Информация об авторах / Information about authors

Сергей Михайлович СТАРОЛЕТОВ – кандидат физико-математических наук, доцент кафедры прикладной математики. Сфера научных интересов: формальная верификация, Model checking, киберфизические системы, операционные системы.

Sergey Michailovich STAROLETOV – Candidate of Physical-Mathematical Sciences (PhD), Associate Professor at the department of Applied Mathematics. Research interests: formal verification, Model checking, cyber-physical systems, operating systems.

Максим Станиславович АМОСОВ – магистрант по специальности «Программная инженерия». Сфера научных интересов: программирование, формальная верификация, системы контроля версий.

Maxim Stanislavovich AMOSOV – Master degree student of Software Engineering. Research interests: programming, formal verification, version control systems.

Кирилл Михайлович ШУЛЬГА – бакалавр по специальности «Программная инженерия». Сфера научных интересов: программирование, квадрокоптеры, блокчейн.

Kirill Mikhailovich SHULGA – Bachelor of Software Engineering. Research interests: programming, quadcopters, blockchain.