UNIVERSIDADE FEDERAL DE MINAS GERAIS

Escola de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica

Arthur Viana Lara

Design of an Embedded System Architecture for a Safety-Critical System

Belo Horizonte

2019

Arthur Viana Lara

DESIGN OF AN EMBEDDED SYSTEM ARCHITECTURE FOR A SAFETY-CRITICAL SYSTEM

Thesis submitted to the Graduate Program in Electrical Engineering of Escola de Engenharia at the Universidade Federal de Minas Gerais, in partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

Advisors: Guilherme Vianna Raffo, Janier Arias Garcia and Leandro Buss Becker

Belo Horizonte
2019

L318d

Lara, Arthur Viana.

Design of an embedded system architecture for a safety-critical system [recurso eletrônico] / Arthur Viana Lara. - 2019.

1 recurso online (130 f. : il., color.) : pdf.

Orientador: Guilherme Vianna Raffo.

Coorientadores: Janier Arias Garcia, Leandro Buss Becker.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Escola de Engenharia.

Apêndices: f. 100-130.

Bibliografia: f. 94-99.

Exigências do sistema: Adobe Acrobat Reader.

1. Engenharia elétrica - Teses. 2. Veículo Aéreo Não Tripulado - Teses. 3. Sistemas embutidos de computador - Teses. 4. Sistemas de segurança - Teses. I. Raffo, Guilherme Vianna. II. Arias García, Janier. III. Becker, Leandro Buss. III. Universidade Federal de Minas Gerais. Escola de Engenharia. IV. Título.

CDU: 621.3(043)

Ficha catalográfica elaborada pela Bibliotecária Roseli Alves de Oliveira CRB/6 2121
Biblioteca Prof. Mário Werneck, Escola de Engenharia da UFMG

UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
Programa de Pós-Graduação em Engenharia Elétrica

ATA DA 1160ª DEFESA DE DISSERTAÇÃO DE MESTRADO DO PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ATA DE DEFESA DE DISSERTAÇÃO DE MESTRADO do aluno **Arthur Viana Lara** - registro de matrícula de número 2017717937. Às 09:00 horas do dia 10 do mês de dezembro de 2019, reuniu-se na Escola de Engenharia da UFMG a Comissão Examinadora da DISSERTAÇÃO DE MESTRADO para julgar, em exame final, o trabalho intitulado **"Design of an Embedded System Architecture For a Safety-critical System"** da Área de Concentração em Sinais e Sistemas, Linha de Pesquisa Controle, Automação e Robótica. O Prof. Guilherme Vianna Raffo, orientador do aluno, abriu a sessão apresentando os membros da Comissão e, dando continuidade aos trabalhos, informou aos presentes que, de acordo com o Regulamento do Programa no seu Art. 8.16, será considerado APROVADO na defesa da Dissertação de Mestrado o candidato que obtiver a aprovação unânime dos membros da Comissão Examinadora. Em seguida deu início à apresentação do trabalho pelo Candidato. Ao final da apresentação seguiu-se a arguição do candidato pelos examinadores. Logo após o término da arguição a Comissão Examinadora se reuniu, sem a presença do Candidato e do público, e elegeu o Prof ....Guilherme V. Raffo.................. para presidir a fase de avaliação do trabalho, constituída de deliberação individual de APROVAÇÃO ou de REPROVAÇÃO e expedição do resultado final. As deliberações individuais de cada membro da Comissão Examinadora foram as seguintes:

Membro da Comissão Examinadora	Instituição de Origem	Deliberação	Assinatura
Prof. Dr. Guilherme Vianna Raffo - Orientador	DELT (UFMG)	APROVADO	[illegible]
Prof. Dr. Janier Arias García	DELT (UFMG)	Aprovado	[illegible]
Prof. Dr. Leandro Buss Becker	DAS (UFSC)	APROVADO	[illegible]
Prof. Dr. Hugo Daniel Hernandez Herrera	DEE (UFMG)	Aprovado.	Hugo Daniel Hernandez
Prof. Dr. Henrique Resende Martins	DEE (UFMG)	Aprovado	[illegible]

Tendo como base as deliberações dos membros da Comissão Examinadora a Dissertação de Mestrado foi ....APROVADA.................. O resultado final de ..APROVAÇÃO.... foi comunicado publicamente ao Candidato pelo Presidente da Comissão, ressaltando que a obtenção do Grau de Mestre em ENGENHARIA ELÉTRICA fica condicionada à entrega do TEXTO FINAL da Dissertação de Mestrado.O Candidato terá um prazo máximo de 30 (trinta) dias, a partir desta data, para fazer as CORREÇÕES DE FORMA e entregar o texto final da Dissertação de Mestrado na secretaria do PPGEE/UFMG. As correções de forma exigidas pelos membros da Comissão Examinadora deverão ser registradas em um exemplar do texto da Dissertação de Mestrado, cuja verificação ficará sob a responsabilidade do Presidente da Banca Examinadora. Nada mais havendo a tratar o Presidente encerrou a reunião e lavrou a presente ATA, que será assinada pelo Presidente da Comissão Examinadora. Belo Horizonte, 10 de dezembro de 2019.

[illegible]

ASSINATURA DO PRESIDENTE DA COMISSÃO EXAMINADORA

"Design of an Embedded System Architecture For a Safety-critical System"

Arthur Viana Lara

Dissertação de Mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Mestre em Engenharia Elétrica.

Aprovada em 10 de dezembro de 2019.

Por:

[illegible]

Prof. Dr. Guilherme Vianna Raffo
DELT (UFMG) - Orientador

[illegible]

Prof. Dr. Janier Arias García
DELT (UFMG)

[illegible]

Prof. Dr. Leandro Buss Becker
DAS (UFSC)

Hugo Daniel Hernandez

Prof. Dr. Hugo Daniel Hernandez Herrera
DEE (UFMG)

[illegible]

Prof. Dr. Henrique Resende Martins
DEE (UFMG)

Acknowledgements

No one knows how complicated it has been these years of work. Many changes and many days of poor mental and physical health, but I did it. First, I have to thank God, because without him nothing would be possible.

I cannot express enough thanks to my advisors for their continued support even through a Skype remote call. I thank Professor Guilherme Vianna Raffo for all these years of guidance and friendship. Today, I do not see the same one just as a teacher, but also as a friend for life.

My completion of this project could not have been accomplished without the support of colleagues from the ProVANT and my job. I express my immense thanks to my great friend Paulo Tujal who welcomed me in the company with open arms and had the patience to pass on his knowledge and experience gained over several years, especially in the area of Safety Instrumented Systems.

Lastly, I thank my family for all these years of concern and care. If they did not exist I would not be able to finish this work. Ismael, Deborah and Diogo, you are essential in my life. Thank you!

Resumo

Os sistemas de segurança crítica consistem em dispositivos que devem funcionar sem falhas, caso contrário, poderão resultar em mortes, danos materiais significativos ou danos ao meio ambiente. Veículos Aéreos Não Tripulados (VANTs) são exemplos de tais sistemas e seu subsistema mais crítico é o sistema de controle de voo. Tendo em vista a sua implementação, existe uma demanda por uma arquitetura embarcada capaz de executar algoritmos de controle de alto custo computacional. Assim, esta dissertação propõem uma arquitetura que utiliza uma plataforma de desenvolvimento composta por uma GPU de propósito geral ao mesmo tempo buscando garantir a dependabilidade e o atendimento a requisitos de tempo real necessários para a sua operação. Inicialmente, os requisitos da arquitetura foram coletados a partir de um estudo sobre normas de segurança de hardware e software para aviação comercial, conjuntamente com a realização de entrevistas com pesquisadores envolvidos no projeto de um VANT. A partir dos requisitos obtidos, uma arquitetura de hardware de dois níveis foi definida composta por um hardware de alto desempenho e um hardware de baixo desempenho, cuja especificação foi guiada segundo os requisitos de comunicação com a instrumentação. O hardware de baixo desempenho utiliza o sistema operacional FreeRTOS e o hardware de alto desempenho, o Ubuntu 18.04. Além disso, tendo em vista o objetivo de agilizar o desenvolvimento, utilizou-se o framework de aplicações robóticas ROS 2 na plataforma de alto desempenho. Com o objetivo de aumentar a confiabilidade da arquitetura, três estratégias foram adotadas: i) implementação de uma lei de controle simples no hardware de baixo desempenho caso haja falha do hardware de alto desempenho; ii) implementação da estratégia de tolerância a falhas denominada “hot standby” na camada de baixo nível, eliminando um ponto singular de falha; e iii) a utilização de bits redundantes para aumentar a confiabilidade de comunicação entre o hardware de baixo desempenho e o hardware de alto desempenho. Por fim, realizamos uma simulação usando injeção de falhas em um ambiente de simulação via Hardware-in-the-loop para analizar: i) funcionamento correto do sistema de controle de voo sem falhas de hardware; ii) comportamento do sistema de controle de voo perante a falhas do hardware de alto desempenho; e iii) comportamento do sistema de controle de voo perante a falhas do hardware de baixo desempenho.

Palavras-chave: VANT; Sistemas Embarcados; Sistema de segurança crítica.

Abstract

Safety-critical systems consist of devices that must operate without failures, otherwise, it may result in death, significant property damage or environmental damage. Unmanned Aerial Vehicles (UAVs) are examples of such systems and their most critical subsystem is the flight controller. In view of its implementation, there is a demand for an embedded architecture capable of executing high computational cost control algorithms. Thus, this dissertation proposes an architecture that uses a development platform composed by a general purpose GPU while seeking to guarantee the dependability and the real-time requirements necessary for its operation. Initially, the architecture requirements were collected from a study of commercial aviation hardware and software safety standards, together with interviews with researchers involved in the design of a UAV. Based on the requirements obtained, a two-level hardware architecture was defined, consisting of a high performance hardware and a low performance hardware, which was specified according to the instrumentation communication requirements. The low performance hardware uses FreeRTOS and the high performance hardware, Ubuntu 18.04. In addition, in order to speed up the development, the ROS 2 was used on the high-performance platform. In order to increase the dependability of the architecture, three strategies were adopted: i) implementation of a simple control law on low performance hardware in case of high performance hardware failure; ii) implementation of the fault tolerance strategy called hot standby in the low level layer, eliminating a single point of failure; and iii) use of redundant bits to increase communication reliability between low performance hardware and high performance hardware. Finally, we perform a simulation using fault injection in a hardwarein-the-loop simulation environment to analyze: i) flight control system’s operation without hardware failure; ii) flight control system’s behavior against high performance hardware failures; and iii) flight control system’s behavior against low performance hardware failures.

Key-words: UAV; Embedded System; Safety-Critical System.

List of figures

List of tables

List of Algorithms

Acronyms

ABFT	Algorithm-Based Fault-Tolerance
AFDS	Autopilot/Flight Director System
API	Application Programming Interface
CAD	Computer Aided Design
CAN	Controller Area Network
CMSIS	Arm’s Cortex Microcontroller Software Interface Standard
COTS	Commercial Off-The-Shelf
CPHA	Clock phase
CPOL	Clock polarity
CPU	Central Process Unit
CSS	Chip Select Slave
DAL	Design Assurance Level
DLQR	Discrete Linear Quadratic Regulator
DWC	Duplication With Comparison
ESC	Electronic Speed Controller
ECC	Error-Correcting Code
FBW	Fly-By-Wire
FIFO	First-In First-Out
FMEA	Failure mode and effects analysis
FMS	Flight Management System
FTA	Fault Tree Analysis
FPGA	Field Programmable Gate Array
GPS	Global Position System
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
HILS	Hardware-In-The-Loop Simulation
HLH	High Level Hardware
IMU	Inertial Measurement Unit
I2C	Inter-Integrated Circuit Bus
LLH	Low Level Hardware
MOSI	Master Data Output, Slave Data Input

MISO	Master Data Input, Slave Data Output
NMR	N-Modular Redundancy
OS	Operating System
PLC	Programmable Logic Controller
PWM	Pulse Width Modulation
ROS	Robot Operating System
RTAI	Real-Time Application Interface
RTK	Real Time Kinematic
RTOS	Real-Time Operating System
SCLK	Serial Clock
SDA	Serial Data SPI Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
UAV	Unmanned Aerial Vehicles
USART	Universal Synchronous Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
USD	United States Dolar
VTOL	Vertical Takeoff and Landing

Notation

General Notation

a	Italic lower case letters denote scalars
$\boldsymbol{a}$	Boldface italic lower case letters denote vectors
$\boldsymbol{A}$	Boldface italic upper case letters denote matrices

Model notation

m_n	Mass of the n-body
$\boldsymbol{r}_1$	First reference
$\boldsymbol{r}_2$	Second reference
$\boldsymbol{\xi}$	Position of the main body with respect to the inertial frame
φ	Roll angle
θ	Pitch angle
ψ	Yaw angle
α_R	Inclination of the right propeller with respect to the main body of the aircraft
α_L	Inclination of the left propeller with respect to the main body of the aircraft
$\boldsymbol{d}_{Ai}^{V}$	Displacement vector from B to Ai , expressed in B
$\boldsymbol{d}_{Ci}^{Ai}$	Displacement vector from Ai to Ci , expressed in Ai

β	Fixed inclination angle of the thrusters towards the aircraft's geometric center
$\boldsymbol{q}$	Generalized coordinates
$\boldsymbol{R}$	Relative weights of input usage to tune a DLQR controller.
$\boldsymbol{Q}$	Relative weights of state deviation to tune a DLQR controller.
f_{eq}^{R}	Magnitude of the thrust generated by the right propeller when the UAV is in equilibrium point
f_{eq}^{L}	Magnitude of the thrust generated by the left propeller when the UAV is in equilibrium point
τ_{eq}^{R}	Magnitude of the torque generated by the right servomotor when the UAV is in equilibrium point
τ_{eq}^{L}	Magnitude of the torque generated by the left servomotor when the UAV is in equilibrium point

Contents

1 Introduction

The daily life of modern society is totally dependent on small programmable electronic systems that are commonly unknown to its users. As can be seen in *Figure 1*, they are present at cars, airplanes, toys, hospitals, markets, among others. They ensure our commodity, safety, leisure, food and supply. They are named embedded systems and consist in devices which perform specific functions (Barr, 1998). For being highly specialized, an embedded system is optimized in relation of energy, code size, execution time, weight and dimensions.

According Oyetoke, 2015, "About 98% of all microprocessors being manufactured are used in embedded systems". Besides, its market size in Europe is expected to exceed USD 258.72 billion by 2023, according to a study conducted in 2016[1] . *Figure 2* shows the growth projection suggested by the study.

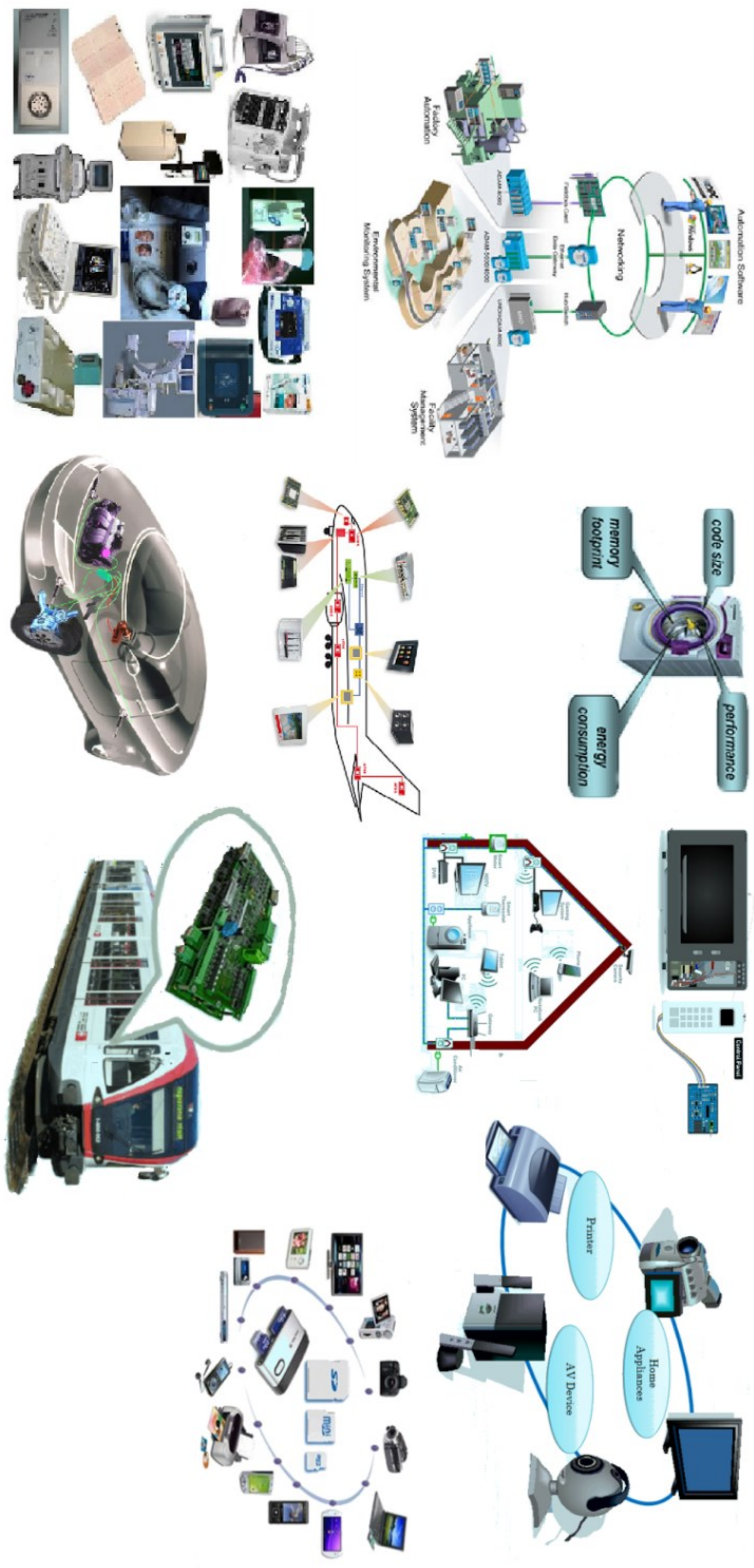
There is an important class of embedded systems that must work without any errors, otherwise it may result in loss of life, significant property damage or environmental damage (Knight, 2002). They are called safety-critical systems. Safety-critical applications demand deterministic behavior during its operation. In order to deal with them, two important features must be explored: the dependability and the temporal determinism.

An example of safety-critical system is the fuel control system used in modern cars. As its name suggests, its purpose is just to control the injection of fuel in the engine (Wang Sujing et al., 2008) and if it fails, it can generate automotive accidents, leading to injuries and deaths. Ford Motor Co. recalled an estimated 1.28 million 2012-2018 Ford Focus compact cars in 2018 because of a fuel system problem[2].

[1] Embedded System Market Size By Application (Automotive, Industrial, Consumer Electronics, Telecommunication, Healthcare, Military and Aerospace), By Product (Software, Hardware) Industry Outlook Report, Regional Analysis, Application Development Potential, Price Trends, Competitive Market Share and Forecast, 2016 – 2023. URL: https://www.gminsights.com/industry-analysis/embedded-system-market. Accessed: 2019-10-07

[2] Ford Recalls 1.2M Focus Cars for Fuel System Defect. URL: https://www.automotive-fleet.com/317645/ford-recalls-1-2m-focus-cars-for-fuel-system-defect, Accessed: 2019-10-07

Figure *1*: Embedded system's applications.



Source: *https://fxdfronteira.blogspot.com*

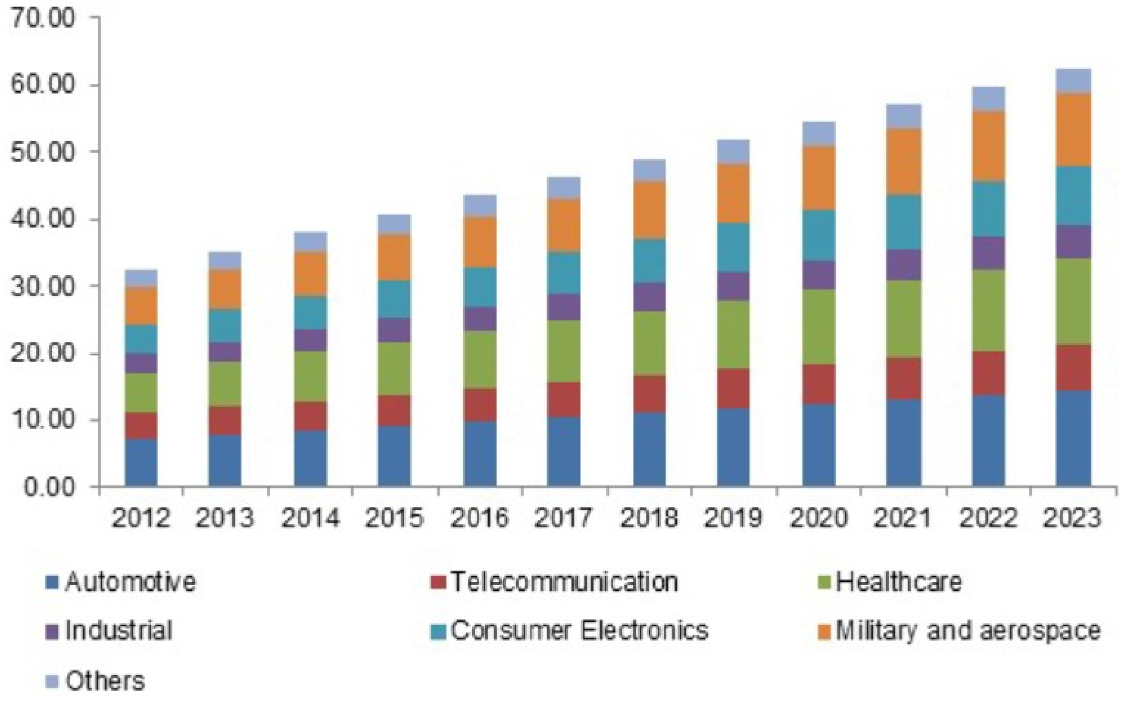
Medical devices are also safety-critical systems. According Alemzadeh et al. (2013), “Medical devices are often subject to a nonnegligible number of failures with potentially catastrophic impacts on patients. Between 2006 and 2011, 5,294 recalls and 1,154,451 adverse events were reported to the US Food and Drug Administration”. An example of medical device is a defibrillator, an important equipment used to restore or beat the heart by applying electrical pulses (Dhurjaty & Atre, 2016), and if it fails, it can generate defective pulse generators that promptly induces deaths.

Other examples are aerospace applications. In case of commercial aircrafts, dozens of passengers can die if there is some human failure or some serious failure in any of the airplane’s safety-critical subsystems. To illustrate, in October 2018[3] and March 2019[4] two units of Boeing 737 Max crashed and 346 people were killed. Another aerospace application that recently has begun to be a concern, is the use of Unmanned Aerial Vehicles (UAVs) in civil airspace. They can injure or kill people on the ground or crash with commercial aircrafts leading to serious accidents.

[3] Boeing’s 737 Max grounded for longer after new flaw discovered, URL: https://www.dw.com/en/boeings-737-max-grounded-for-longer-after-new-flaw-discovered/a-49369628, Accessed: 2019-10-07

[4] Ethiopian Airlines plane crashes shortly after takeoff , URL: https://www.dw.com/en/ethiopian-airlines-plane-crashes-shortly-after-takeoff/a-47841392, Accessed: 2019-10-07

Figure 2: Embedded System's industry trend.

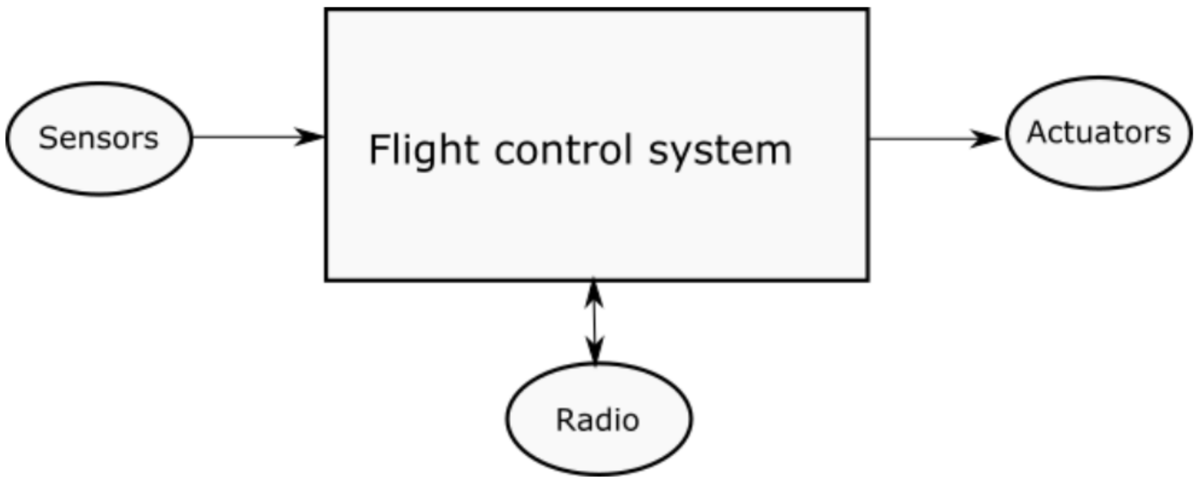


Source: *Global Market Insights*

1.1 Study case System

In the case of commercial airplanes and UAVs, one of the most important safety-critical systems is the flight control system. Basically, it is a subsystem that interfaces with sensors and actuators; and for each sample time, it executes a feedback control law to update the commands of actuator. *Figure 3* illustrates its structure.

Figure *3*: Main idea of embedded system architecture.

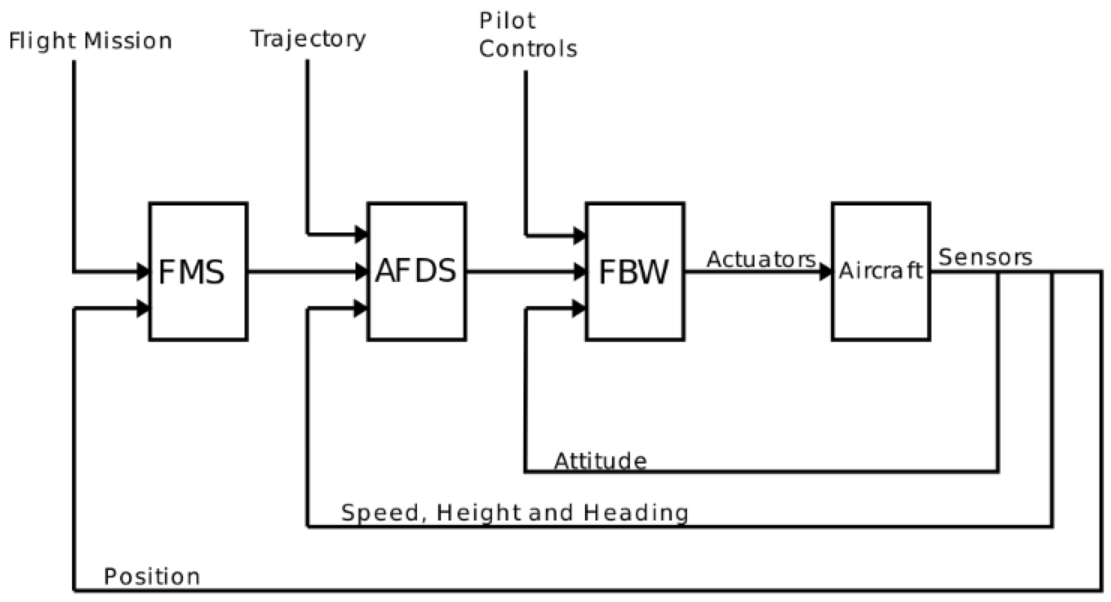


Source: The author

In the context of commercial airplanes, they are normally composed of three flight control functions, Fly-By-Wire (FBW), Autopilot/Flight Director System (AFDS) and Flight Management System (FMS). These functions are interconnected and may be described as three nested control loops, each one with their own distinct purpose. They are shown in *Figure 4*. The FBW is a function that controls the attitude of the aircraft. The AFDS controls the speed, height, and heading. Finally, the FMS performs the navigation or mission function, ensuring that the position aircraft will reach multiple way-points that composes the aircraft route (Hitt, 2006). However, this organization is not a general rule for every aircraft, UAVs can have, for example, just one control loop instead of these three flight loops.

In view of the challenge to design an architecture for a flight control system, this work proposes an embedded system for use by a VTOL UAV called ProVANT 4.0, which is shown in *Figure 5* , where a flight control system will be implemented. It is a tilt-rotor UAV prototype that has been designed in ProVANT. ProVANT is a collaborative research project composed by researchers from the *Universidade Federal de Santa Catarina*, the *Universidade Federal de Minas Gerais* and the *Universidad de Sevilla*.

Figure *4*: Interaction between FBW, AFDS, FMS and instrumentation.

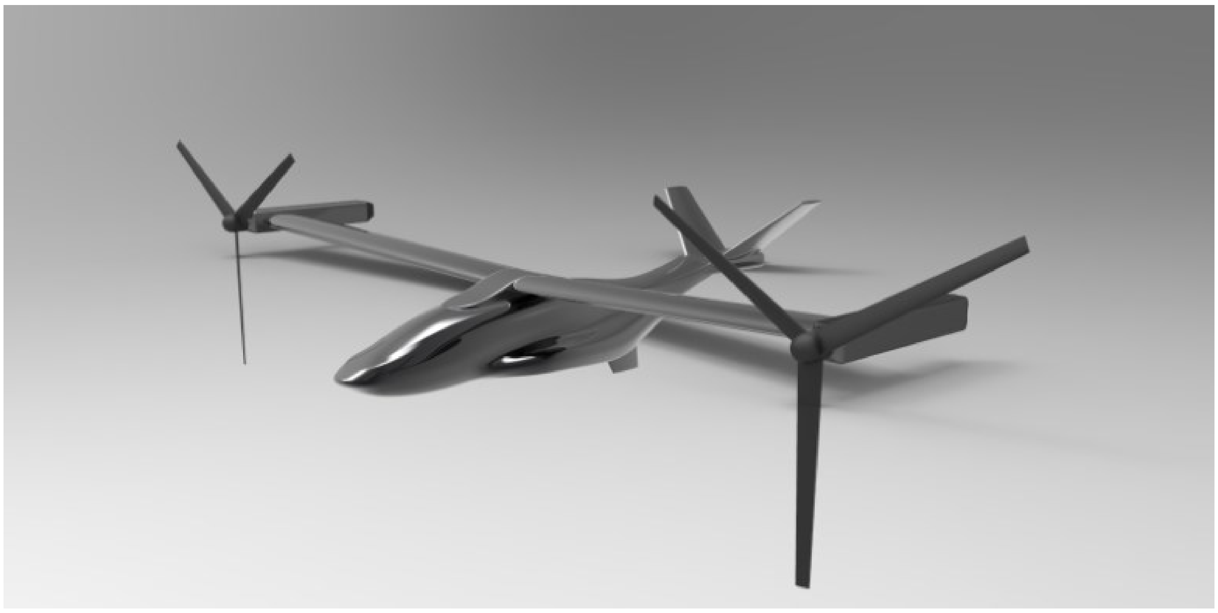


Source: The author

Tilt-rotor UAV is a convertible aircraft which has two propellers and mechanisms responsible for tilting them, located at the ends of fixed wings. The tilt-rotor UAV has two modes of flight, helicopter and cruise, which requires little space for landing and takeoff and acquires elevated speeds. The tilt-rotor UAV, in relation to quadrotors, has a gain in autonomy and, in comparison to airplanes, a gain of mobility in small spaces.

ProVANT 4.0 (del Pino, 2016) has been designed for Search and Rescue operations and, therefore, it must reach the emergency sites within a short time, flying in spaces with a small free area and acting in any environment, whether open or closed. It has reduced dimensions suitable for transporting in a vehicle of rapid intervention and it must carry an automatic defibrillator. Besides, it will use commercial batteries and electric propulsion, which are adapted to the use of renewable resources (da Silva, 2017). Besides, it has a container of 17cm long, 20cm wide and 7cm high for on-board systems.

Figure *5*: ProVANT 4.0 conceptual design.



Source: MACRO research group

It shall perform the following missions:

- Complete two vertical take-off and landing maneuvers. One at the starting point and one at the focus of the emergency, having to travel between them a distance of 20 km to go and another 20 km to return. It must carry a payload of about 3 kg.
- Vertical take-off from the outpost, capacity for a one-hour reconnaissance flight and return to the starting point. The device must carry at least one thermal and visual camera.

1.2 Related Works

This section presents some current literature review about multi-core embedded systems architectures on safety-critical applications, fault-tolerance techniques applied on embedded systems, and small UAV's onboard architectures.

1.2.1 Embedded Systems architectures on safety-critical applications

Nowadays, there is an effort for the adoption of hardwares to host multiple safety-critical functions with mixed-critical levels on a common computing platforms, reducing the number of hardware scattered in the application. Besides, mainly due to the emerging artificial intelligence, high performance platforms have also been demanded for implementation of high computational cost algorithms (Saidi et al., 2015).

Integrated modular avionics (IMA), for aerospace domain, and Advanced Driver Assistance Systems (ADAS), for automotive domain, are some of the main target applications for this trend. GE Aviation has already developed IMA architectures for Boeing 787 Dreamliner, Boeing C-130 combat aircraft, and Boeing KC-767 Tanker (Watkins & Walter, 2007). However, the use of multi-core hardware still generate some reluctance for its use in critical applications, for example, flight control systems (Gaska et al., 2015).

These hardware platforms can reduce the power consumption, the length and weight. Consequently, it can also reduce the operational costs. However, there are many challenges to be solved. According to Saidi et al. (2015), “the main reasons are: a) shared resources imposes a strong timing correlation between concurrently running components in the same chip; and b) standard commercial off-the-shelf (COTS) multicore components are optimized to improve the average case performance and not the worst case”.

Given these issues, there are research projects that focus on developing solutions that meet the requirements for safety-critical systems. An example is the Hercules H2020 Project[5], which aims to implement the first industrial-grade framework to provide real-time guarantees on top of cutting-edge heterogeneous COTS platforms for embedded domains. Other examples are parMERASA, CERTAINTY, P-SOCRATES and EMC 2.

[5] http://hercules2020.eu/, accessed 24/11/2017

1.2.2 Fault-tolerant techniques applied on embedded systems

The dependability is one of the open problems of adopting multi-core platforms. Thus, in order to increase it, fault-tolerant architecture can be adopted. An important fault-tolerant strategy is the Dynamic Reconfiguration. Through detection and replacement of defective components, it increases the availability and reliability of solutions. In this direction, Mozafari & Meyer (2015) present an approach for employing hot spares in multi-core processors. Besides, Rahme & Xu (2017) also apply this concept for multiple software spare components to a cloud computing applications.

For safety-critical applications, Dynamic Reconfiguration becomes essential to implement fail-operational safety architectures. Designers aim to switch the system operation from normal mode to emergency mode when a fault occurs. In this direction, Sari & Reuss (2018) discuss about fail-operational safety architectures for ADAS. Moreover, Vivekanandan et al. (2016) propose a fail-operational safety architecture, composed by two heterogeneous hardware and software platforms with distinct reliability and performance characteristics, for a UAV’s onboard hardware; however, they do not present any reliability data to support their assumptions and justify the use of this architecture in real-life.

Triple Modular Redundancy (TMR) is another fault-tolerant strategy. It masks faults through a voter mechanism. Kahe (2018) proposes an architecture composed by five modules, each one with three multi-core ARM processors in parallel using TMR strategy, obtaining evaluation results that meet aerospace requirements. Janson et al. (2018) also adopt this strategy for a fault-tolerant software architecture without a synchronization mechanism. TMR can also be adopted in FPGA design. Thus, LaMeres et al. (2015) use it to design a dependable space aplication.

An extension of TMR, when considering N redundant components plus a voter, is called N-Modular Redundancy (NMR). There is a trend towards its adoption for multi-core applications; however, it increases the energy consumption of the platform, becoming a limiting factor for embedded systems. To solve this issue, Salehi et al. (2016) propose a two-phase NMR with block-partitioned scheduling and pseudodynamic slack management in order to achieve minimized energy consumption and ensure the deadline requirements.

For GPU-based and FPGA-based applications, there are some recent works that discuss how to deal with faults caused by radiation. Pilla et al. (2014) experimentally demonstrate that the Error-Correction Code (ECC) does not ensure enough reliability as required. Then, they propose an Algorithm-Based Fault-Tolerance (ABFT) technique that provided smaller failure rates and small overhead than ECC. However, d. Santos et al. (2017) demonstrated that the ECC is more efficient than the ABFT from the point of view of Silent Data Corruption rate.

Besides, Oliveira et al. (2014) propose Duplication With Comparison (DWC) strategies and suggest that DWC strategies can be more effective than ECC when the input data are duplicated.

For GPU domain, ECC, ABFT and DWC are methods that deals with faults in memory, registers and logic, but not with faults in the schedulers. Thus, Milluzzi et al. (2017) propose to use TMR with persistent threads to solve this issue; however, as a drawback, this strategy limits the hardware performance and creates an overhead that may be unacceptable depending on the target application.

1.2.3 Small UAV's onboard architecture

According Chao et al. (2010), an autopilot is a system used to guide the UAV without assistance of human operators. There are several autopilots available on market and, in this direction, Zhaolin Yang et al. (2016) present a survey of existing autopilots and compares some of them.

However, due to the lack of information and support for custom implementation of its own algorithms, there are several works that design their own autopilots. In this research line, Ellingson & McLain (2017) propose a fixed wing autopilot code for educational and research purposes built on ROS to speed up and facilitate the implementation of control, estimation and path planning algorithms.

Czerniejewski et al. (2018) port a UAV's autopilot, called Paparazzi UAV, to the Real-Time Specification for Java, which ensures the meeting of the real-time requirements based on the specification given by the developer, allowing them to focus only on the algorithm development.

Besides, Meier et al. (2015) provide a novel middleware and programming environment, used by an autopilot. From the Nutt X OS, they create a software layer called Object Request Broker in order to abstract the communication between threads and to provide connective with other platforms through ROS.

By and large, as noticed in these works, the majority of autopilots available are built around a centralized set of microprocessors. However, going in another direction, Louali et al. (2017) propose a distributive architecture for an autopilot through the use of Controller Area Network (CAN) data bus. It was designed for a fixed wing UAV and tested through a hardware-in-the-loop simulation.

1.3 ProVANT's embedded system history

Since the ProVANT was created, it has already designed 5 versions: ProVANT 1.0, ProVANT 2.0, ProVANT 2.1, ProVANT 3.0 and ProVANT 4.0. The ProVANT 1.0, shown in *Figure 6*, is the first prototype and it only has the ability to conduct flight in helicopter mode. Its development was a craft project, since, it did not use any CAD software to help its design. In addition, several mechanical filters were allocated with the focus of the attenuation of the structure's vibration, avoiding interferences in the measurement of the Inertial Measurement. There is a video on internet that shows it in testing phase[6].

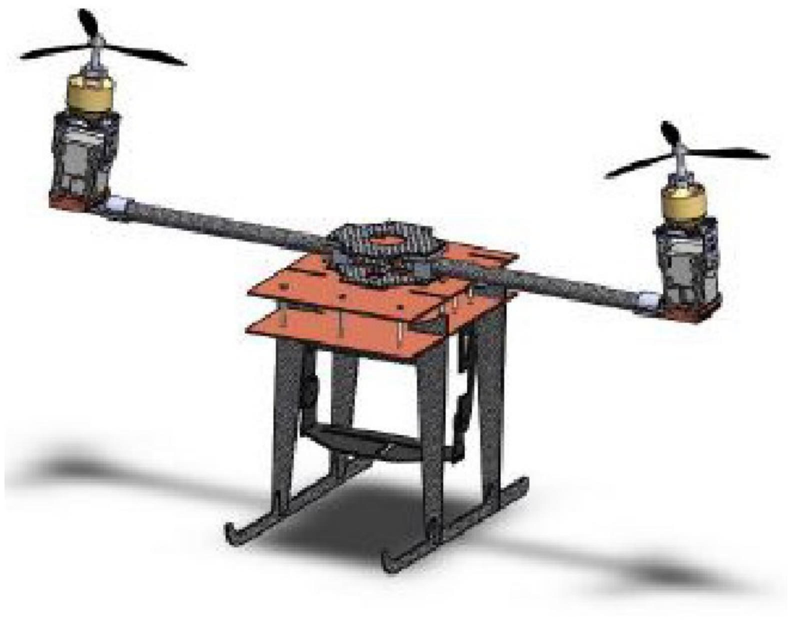
The ProVANT 1.0 has two microprocessors in its architecture: the STM32F4DISCOVERY board that interfaces with the instrumentation, and the beaglebone board that executes computationally expensive control laws. Besides, according to Donadel et al. (2015), ProVANT 1.0 uses the following instrumentation:

[6] https://www.youtube.com/watch?v=f94fbhJjzuo

- 2x brushless motors AXI 2814/20 GOLD LINE
- 2x servomotors Dynamixel RX-24F
- 1x 9-DOF inertial measurement system (IMU) GY-85
- 2x electronic speed controller (ESC) Mikrokopter BL-Ctrl 2.0
- 1x Ultrasonic ranging module HC-SR04
- 1x 2.4 GHz 6 channels radio receiver hobbyking HK-TR6A
- 1x 4-cell LIPO battery (16.8V) Turnigy nanotech 3000mah
- 1x voltage regulator 16.8V/5V
- 1x voltage regulator 16.8V/12V

The ProVANT 1.0 embedded a software architecture in the STM32F4DISCOVERY with the principles of organization presented in Lakos (1996). For this, it was implemented a vertical hierarchical structure of the elements of one level in relation to those of a higher level. Following a verticalization concept, the project elements are organized as show in *Figure 7*. The application level is composed by modules that implement the UAV functions, for example, the control law, navigation, and communication with the instrumentation. The middleware layer is composed by public or private libraries, an Operating System, a State Machine, and the Hardware Abstraction Layer (HAL). It uses FreeRTOS as operating system. Lastly, the Core is composed by CMSIS, a vendor-independent hardware abstraction layer for Arm Cortex microcontrollers.

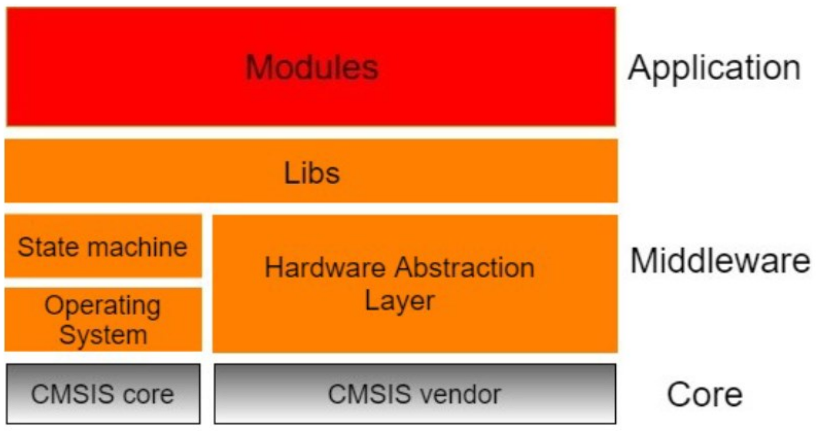
Figure 6: ProVANT 1.0.



Source: MACRO research group

The beaglebone's software is not structured as the STM32F4DISCOVERY. It uses an Ubuntu 14.04.3 LTS and its application is composed by three threads, each one responsible for one different function: communication with STM32F4DISCOVERY, Controller and Data processing.

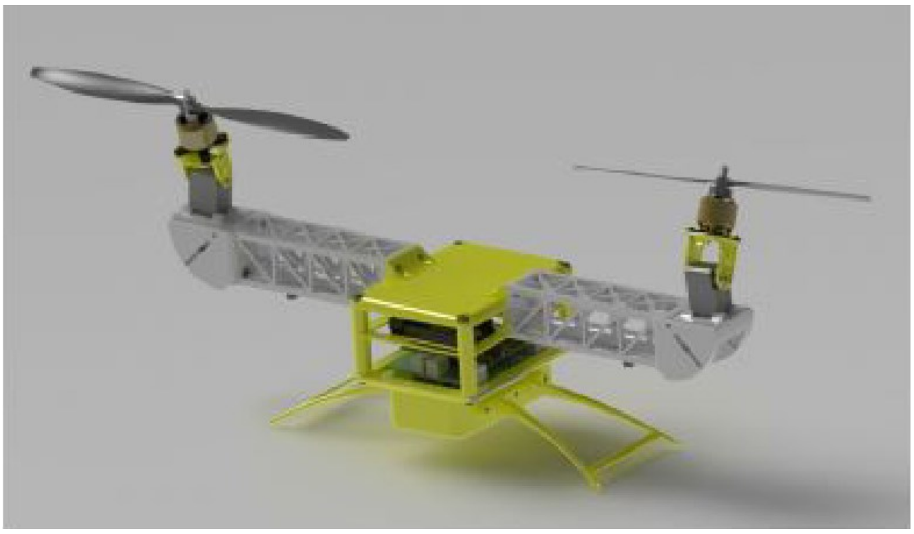
Figure 7: Software Architecture



Source: The author

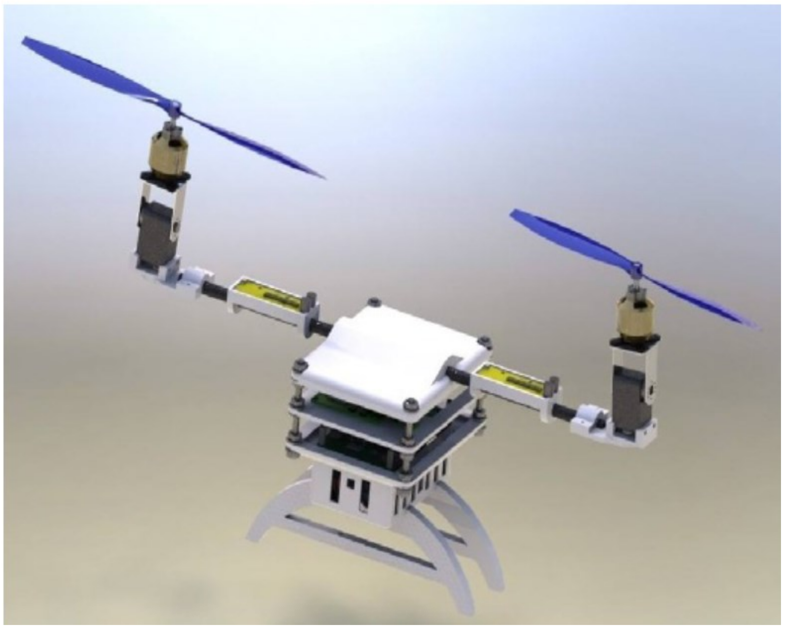
The next version was the ProVANT 2.0, that is shown in *Figure 8*. This version was designed with Solidworks aiming to build it with 3D printer. Its embedded system architecture was similar with the ProVANT 1.0; however, instead of the adoption of the beaglebone board, it uses a Raspberry PI 2 board. A while later, it was designed the ProVANT 2.1, that is shown in *Figure 9*. Although it uses the same embedded system that was chosen for ProVANT 2.0, it has differences in the mechanical design.

Figure *8*: ProVANT 2.0.



Source: MACRO research group

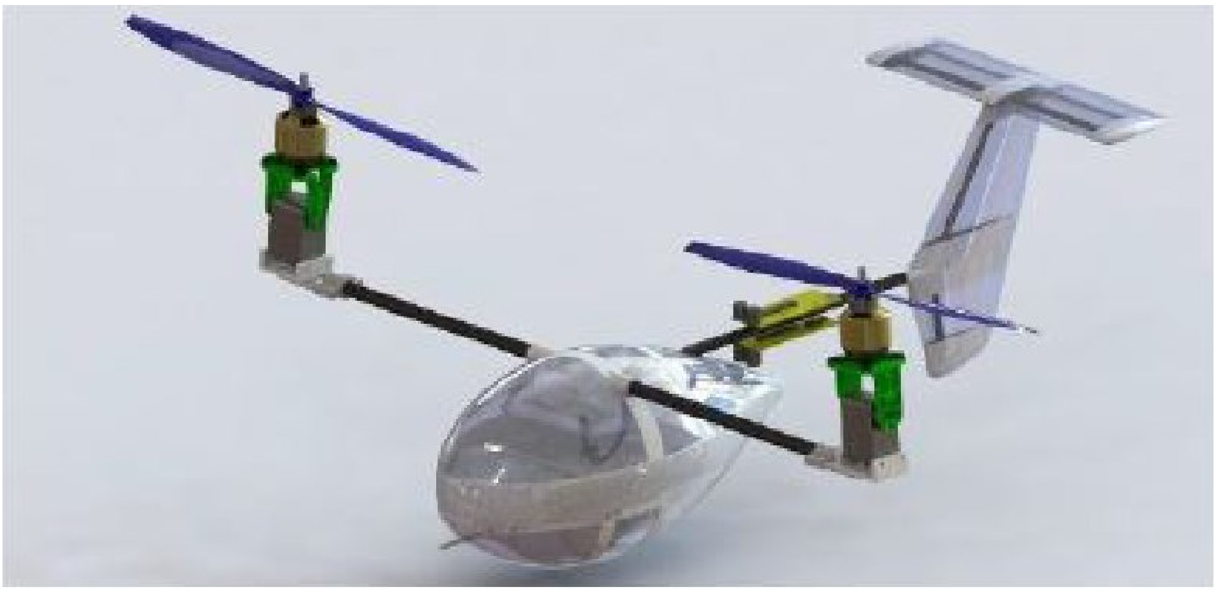
Figure *9*: ProVANT 2.1.



Source: MACRO research group

The ProVANT 3.0 is shown in *Figure 10*. In this version, it was already known that the Beaglebone and Raspberry PI are unsuitable for running high costly computational control algorithms in short sampling periods. ProVANT are designing algorithms of estimation that approximately runs in 500 milliseconds and control laws that approximately runs in 8 seconds in the Simulink Environment Simulation. However, in the UAV’s on board system, these algorithms must be executed at times in order of 10 milliseconds. In this context, Miranda (2017) presents a multi-core software approach developed to improve the time performance of predictive control strategies. DE0-Nano-SoC Kit/Atlas-SoC was used with Debian Jessie 8.5 as operating system, and from the use of Open Multi-Processing API, a control algorithm was implemented with some parallelized regions, decreasing its execution time. This work does not used the FPGA present on the platform.

Figure *10*: ProVANT 3.0.



Source: MACRO research group

Unlike the solution adopted by Miranda (2017), there are some studies that apply FPGAs or GPUs in order to solve this problem. Both alternatives are potentials solutions for using in ProVANT 4.0. This work aims to use the the second option, due to the fast deployment of the solution. The platform chosen is a Jetson TX2 development kit that suggests to be a better solution than Beaglebone and Raspberry PI platforms for execution of high cost control algorithms due to the 64 bits Quad ARM Cortex-A57 processor, the 64 bits Dual NVidia Denver processor and the GPU of 1,3 GHz with Pascal architecture of 256 kernels, when compared with the AM3358 processor of Beaglebone and with the BCM2836 quad core Cortex A7 processor of Raspberry PI 2.

1.4 Justification and Objective

As could be noted before, there is a movement to replace the federative embedded architecture, composed by several specialized mono-cores platforms, with only one multi-core platform in safety-critical applications. This strategy supports greater computational power for executing high cost computational algorithms in short periods. However, dependability is one of the main problems for providing certification to these platforms, due to the existence of a large quantity of shared resources between the cores. Since there is not a container to avoid the influence of faults of a given application to other applications, faults becomes more dangerous than the same ones in federative embedded architecture. In order to deal with this issue, fault-tolerant techniques can be applied to ensure the safety operation of the platform. In the autopilot design domain, only Vivekanandan et al. (2016) propose some fault tolerance technique to ensure the safety of a UAV operation, while the other ones concern with the performance and scalability of the applications. However, the assumptions about the “high assurance platform”, considering that it is reliable enough to provide the safety demanded for UAV operation are not consistent. Besides, only Czerniejewski et al. (2018) make a more depth concern about the ensuring real-time features; however, in no time they make any offline validation method, as required for hard real-time applications. From these gaps described, this thesis proposes a safety-critical embedded system architecture using a Jetson TX2 development kit for a VTOL UAV autopilot.

To reach this goal, some specific objectives are defined:

- Design a hardware-in-the-loop simulation environment for testing the embedded system to be embedded in VTOL UAV’s prototypes;
- Design a hardware and software architecture based on the requirements collected throughout the work;
- Create a prototype of the hardware and software architecture designed previously.
- Test the prototype’s resilience from injection faults in a hardware-in-the-loop simulation.

1.5 Structure of work

This thesis is organized as follows:

- Chapter 2 provides some background. It introduces key concepts about real-time systems, dependability and tools used in this work to understanding the guidelines used to design the hardware and software architecture.
- Chapter 3 describes the design process of the embedded system architecture. Two steps are shown: gathering requirements and design the architecture.
- Chapter 4 presents the experimentation used to validate the proposed architecture. For this porpose, it is designed a Hardware-in-the-loop simulation environment to simulate the system operation in presence of faults.
- Chapter 5 summarizes the contributions and results presented in this dissertation, and suggests possible future research lines.

2 Background

This chapter aims to describe important concepts used in the present dissertation. First of all, some concepts related to dependability are presented. After, it is introduced the concepts about real-time systems. In the end, the Robot Operating System is presented.

2.1 Dependability

The term system is used in several areas of science and engineering, which consists of “an entity that interacts with other entities, i.e., other systems, including hardware, software, humans and physical world with its natural phenomena” (Avizienis et al., 2004). However, for this thesis, we use this term in the context of computing and communication systems. Such systems are artificial elements designed with determined functions, services, and structure. Service is the information or behavior demanded by the users. Function is the internal behavior of a system adopted, in other words, its implementation. Lastly, structure defines how the interaction between its subsystems is performed.

In order to express the ability of a system to deliver its intended level of service to its user, the concept of dependability is used. This concept is described by three features: impairments, attributes and means. Impairments express the threats to dependability. Attributes measure its features. Means consist of techniques used to ensure the meeting of required attributes given the existence of threats during the system’s life cycle.

2.1.1 Impairments

Service failure is a inability of the system providing a service. For example, a service failure of an oven can be an inability of generating enough fire, and a service failure of a program can be a incorrect computation from given inputs. For simplification, from now on, we will just refer to the term failure.

Another important concept is the error, which consists of the deviation of the results due to a failure to the correct output. In the context of the previous examples, it is the difference between the expected fire and the weak fire generated, and the difference between the output of the program and the expected output.

Lastly, the causes of errors are called faults. For an oven, a cause of the weak fire can be the lack of gas, while the fault of a program can be an addition of an incorrect command in the source code during its development phase. In general, faults can be classified into eight different sets that are shown in *Figure 11* (Avizienis et al., 2004).

Note the direct causality relation of these three concepts, but the inverse is not always true. For example, there may be situations where a system would have faults, but they would remain dormant without generating errors and, therefore, also without failures. For example, an unknown bug of software.

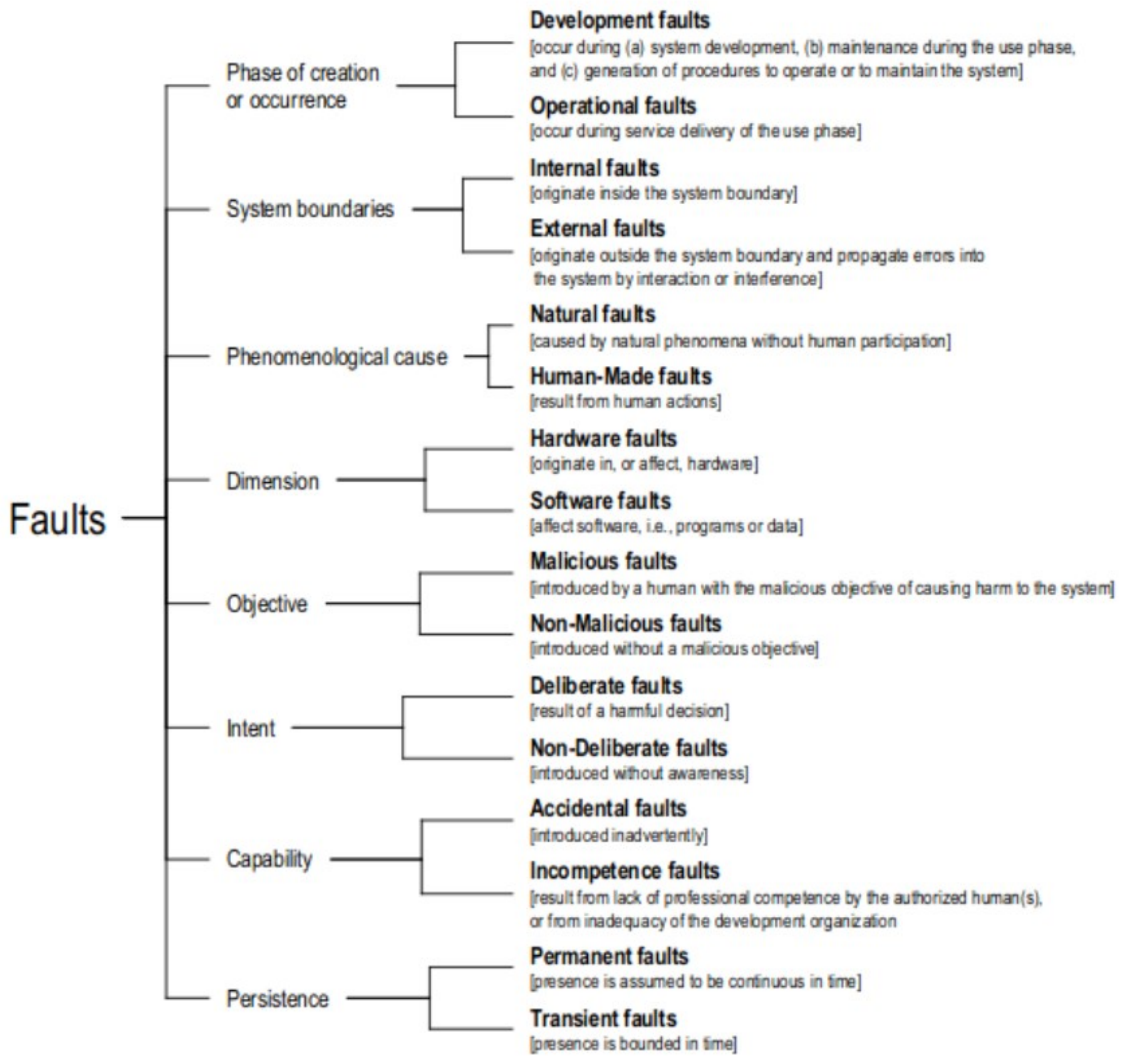
2.1.2 Attributes

In order to translate the main requirements for a given dependable system, there are three attributes: reliability, availability, and safety. They represent a given system behavior related to the existence of faults and according to the application, each one has a different importance. Reliability, R(t), of a system at time, t is the probability that it operates without a failure in the interval [0, t], given that the system was performing correctly at time 0 (Dubrova, 2013). It is a time-dependent metric and measures how much time is expected to a system to operate without problems. Examples of a dependable system, that the reliability is meaningful, are medical devices.

Availability, A(t), of a system at time t is the probability that it is functioning correctly at the instant of time t (Dubrova, 2013). It expresses the fraction of period that the system is in the operational state. An example of a system that requires availability is an Automatic Teller Machine. It is an application where faults are tolerable, but the time for repairing them must be very short.

Lastly, safety, S(t), of a system at time t is the probability that it either performs its function correctly or discontinues its operation in a fail-safe manner in the interval [0, t], given that the system was operating correctly at time 0 (Dubrova, 2013). There is a concern with the absence of catastrophic consequences. An example is a nuclear power plant control system.

Figure *11*: Kind of faults.


Faults
Phase of creation or occurrence
Development faults
[occur during (a) system development, (b) maintenance during the use phase, and (c) generation of procedures to operate or to maintain the system]
Operational faults
[occur during service delivery of the use phase]
System boundaries
Internal faults
[originate inside the system boundary]
External faults
[originate outside the system boundary and propagate errors into the system by interaction or interference]
Phenomenological cause
Natural faults
[caused by natural phenomena without human participation]
Human-Made faults
[result from human actions]
Dimension
Hardware faults
[originate in, or affect, hardware]
Software faults
[affect software, i.e., programs or data]
Objective
Malicious faults
[introduced by a human with the malicious objective of causing harm to the system]
Non-Malicious faults
[introduced without a malicious objective]
Intent
Deliberate faults
[result of a harmful decision]
Non-Deliberate faults
[introduced without awareness]
Capability
Accidental faults
[introduced inadvertently]
Incompetence faults
[result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization
Persistence
Permanent faults
[presence is assumed to be continuous in time]
Transient faults
[presence is bounded in time]

Source: Avizienis et al. (2004)

2.1.3 Means

Means are the ways used to deal with faults in order to provide the dependable attributes required for a given system. In general, there are four ways: fault prevention, fault removal, fault forecasting and fault tolerance. This thesis interests in means to be applied during design phase of an embedded system, then fault removal and fault forecasting is out of the scope of this thesis.

Fault-prevention techniques are the first efforts of an embedded system design to deal with faults. These techniques are applied during the specification and design in order to avoid developer-client communication problems and to abstract the system complexity. In this phase, it can be used mature and formally verified components, standards (for example IEC 61508), formal methods (for example Z methods and model checking), well-established engineering practices and risk assessment techniques, as FMEA (Failure mode and effects analysis) and FTA (Fault tree analysis). in order to identify potential faults (Lala & Harper, 1994).

Fault-tolerant techniques ensures successful operation of a system even if faults occur using redundancy. They are classified according to the type of redundancy in four classes of groups: hardware, software, information and time (Dubrova, 2013). Each one has its advantages and disadvantages, and for each project the designer chooses the best alternatives that solve the problem. In general, all solutions found in the literature use a restricted set of strategies or the hybrid of them. According to Hitt (2006), there are three categories of hardware fault-tolerant architectures: masking, reconfiguration, and hybrid. They are based on the premise that hardware failures occur randomly, caused at most by wear-outs and environmental interference. Moreover, software fault-tolerant techniques are more complex due to the greater software complexity over the hardware. They consist in programming errors. Normally, diversity is adopted in order to avoid common-mode faults. It can use software versions from different programmers, different programming languages, different compilers and so on. Temporal fault-tolerant techniques explore the possibility of repeating the execution of a given algorithm more than twice different times in order to avoid damage by transient failures. Lastly, information fault-tolerant techniques consist in powerful techniques which helps us to avoid unwanted information changes during data storage or transmission. Some examples are parity, Hamming code, and Cyclic Redundancy Check.

Particularly, in this work we intend to use the mature and formally verified components from commercial off-the-shelf (COTS) elements. Thus, given this fact, the present thesis concerns in adding fault-tolerant techniques from the standpoint of the hardware and communication failures in order to design a dependable solution.

Hardware Redundancy

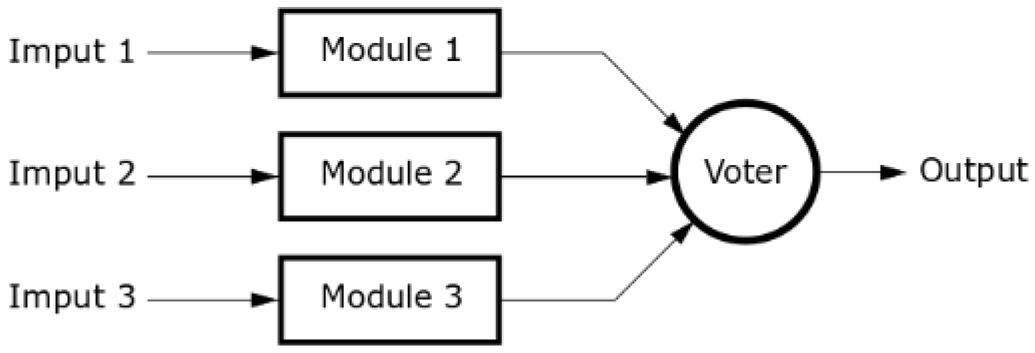
Hardware redundancy consists in put multiple hardwares in parallel in order to tolerate hardware faults. Hardware faults can be permanent, transient and intermittent. Permanent faults are events that remain active until some corrective actions are performed, for example, chip burning. Transient faults are events that happen periodically, mainly due to environmental events such as alpha particles, atmospheric neutrons, electrostatic discharge, electrical power drops, and overheating. Lastly, intermittent faults are due to implementation flaws, ageing, wear-out, and unexpected operating conditions (Dubrova, 2013).

In general, there are three kinds of techniques to add hardware redundancy in a system: fault-masking, reconfiguration, and hybrid configuration. These techniques consider that most of the time, faults happen independently and randomly. Besides, there are the common-mode faults that is not in the scope of this definition. In order to tolerate them, the designer usually chooses different hardware technologies from different manufacturers to create redundancies.

Fault masking is a technique to tolerate faults without detecting them. It consists in an architecture that has multiple hardwares operating in parallel, an synchronization mechanism to ensure algorithms run on all hardwares at the same time, and a voter mechanism that, from the outputs of them, can infer the correct value according to the common response of the majority. Fault masking technique is usually applied on high-reliability applications in which short downtime are unacceptable and can interfere with system dynamics, for example, flight control systems.

The most famous architecture is the Triple Modular Redundancy that is shown in *Figure 12*. It consists of three hardware in parallel. However, a more general architecture is called n-modular redundancy, and as its name suggests, its architecture use any number of parallel hardware plus a voting mechanism.

Figure *12*: Triple Modular Redundancy.



Source: The author

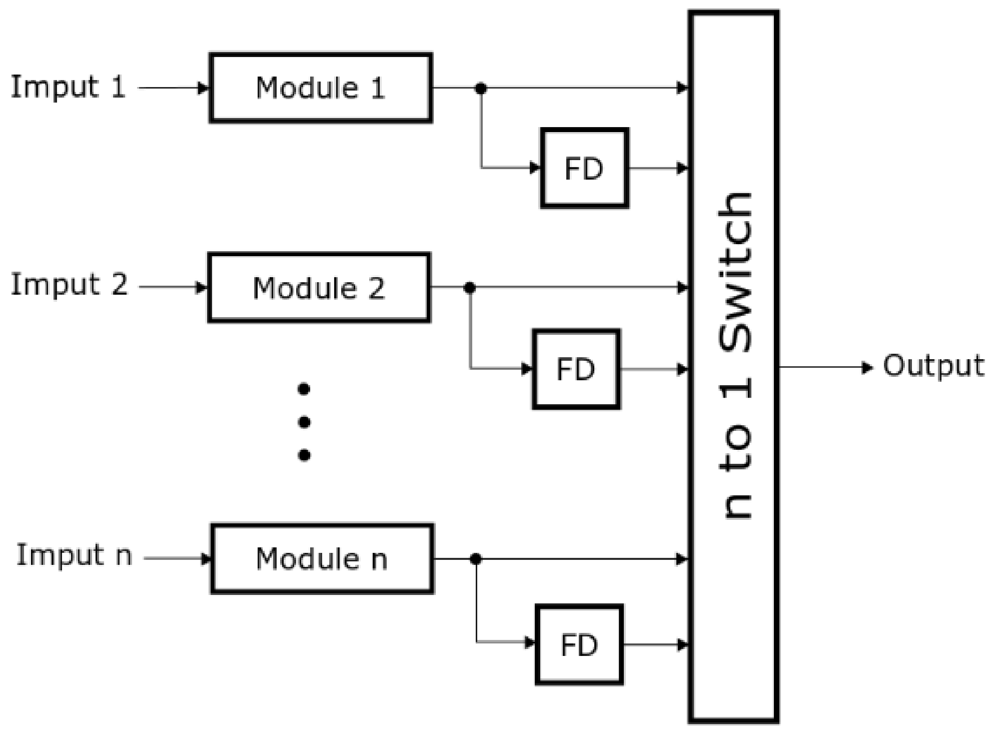
However, the number of redundancies does not consist of an arbitrary choice of designer. According to Lamport et al. (1982), it is proved that a solution must have 3n+1 redundant hardwares to cope with n failed hardwares.

Regarding the reconfiguration technique, it reconfigures the whole solution from a detection mechanism of fault to remove the influence of faults in the system, ensuring the return of the system to an operational state. This technique is usually applied for an application that needs high availability.

Standby Redundancy is the most famous reconfiguration technique. It consists of an architecture of n parallel modules and n fault-detectors, but just one of them is in operation, while the others are spare components. Such a solution can tolerate n–1 module faults. Its architecture is shown in *Figure 13*.

According to the implementation of a spare, there are two kinds of standby redundancy: hot standby and cold standby. The former consists of an implementation that has a shorter time of reconfiguration than the latter due to the spare has already power on, but at the same time it is likely to have failures, different of the cold standby that due to the spare be power off, it will not be influenced by any external environment stimulus.

Figure *13*: Standyby Redundancy.



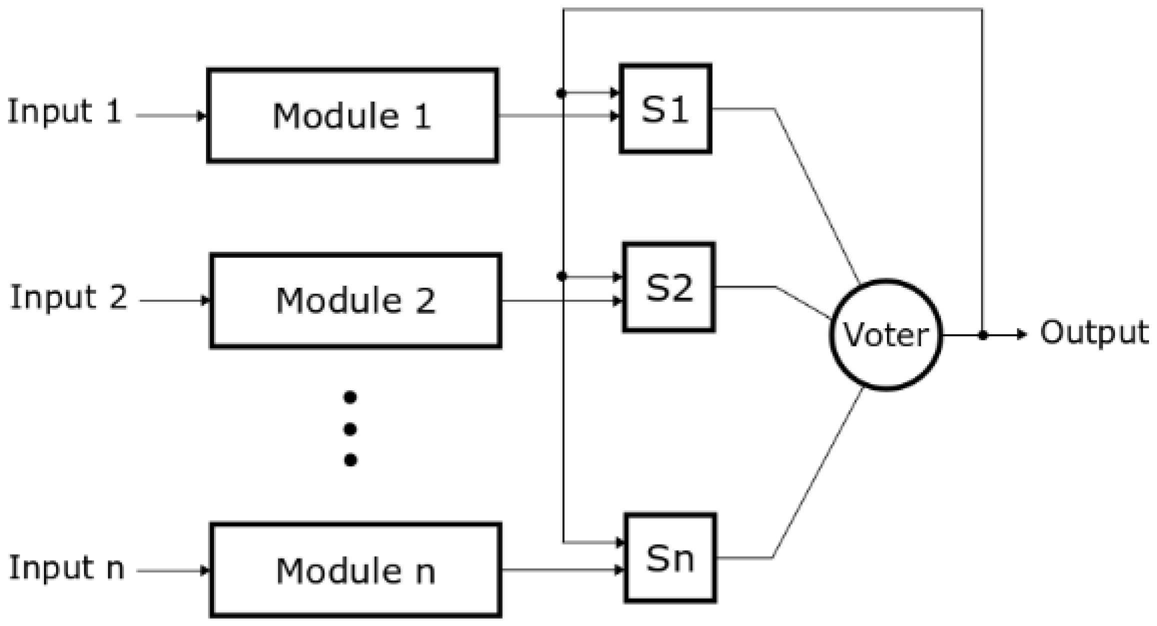
Source: The author

An example of a fault detector used in this architecture is the watchdog timer. It consists of an electronic device that is used in order to detect software problems and reset the processor if any happens (Murphy & Barr, 2001). Another kind of fault detector, that is applied in PLCs, checks if the main processor is alive, sending an information to the hardware and waits for answers.

Lastly, hybrid redundancies combine the advantages of fault-masking and reconfiguration techniques. Fault-masking avoids instantaneous effects of fault, while reconfiguration immediately acts on the system by repairing the failed module with a spare. Since these solutions are very robust, it is well used in safety-critical systems

An example of hybrid architecture is that Self-Purging Redundancy. In this architecture, a voter mechanism decides the correct output and then it is compared with the result of each module. The module, whose result is different of the correct value, is removed from the structure and is replaced by a spare.

Figure *14*: Self-Purging Redundancy.



Source: The author

Information redundancy

By a paradox of the “Two Generals” Gmytrasiewicz & Durfee (1992), it is known that no one can guarantee state consistency of two entities in a communication that happens in an unreliable channel. However, we can improve the communication reliability using several strategies, for example, using the re-submission of information. Another strategy consists in the insertion of redundant bits computed before the transmission and their recomputation after the information has reached its destination. If the bits are the same, we have high confidence that the transmission is correct. Otherwise, we have detected an error.

There are several algorithms used to compute redundant bits. In this work, the Fletcher’s checksum (Fletcher, 1982) algorithm is used. This algorithm detects multiple errors, swapping of data blocks, and insertion of random numbers, using lower computational effort compared to other coding techniques. Its logic is described in Algorithm 1.

Algorithm 1: Fletcher's checksum algorithm

```
Divide the data word into a sequence of equally-sized blocks, b1 b2 ... bn
Define two checksums, starting at C1 = 0 and C2 = 0
i = 0
while i < n {
C1 = C1 + $b_i$
C2 = C2 + C1
i=i+1
} C1 = C1 mod 255
C2 = C2 mod 255
Return (C1 and C2)
```

2.2 Real-Time Systems

A real-time system consists of a computer system that must answer a stimulus correctly and before a given time constraint, otherwise undesirable consequences may occur (Stankovic, 1988). The most important feature of this system is the required temporal determinism and, according to the consequence of its failure, it can be classified into two type of systems: hard real-time and soft real-time. The former can generate injuries, deaths and patrimonial/environmental damage, while the latter can generate at most loss of performance of the application. One example of real-time is the flight guidance system of Apollo 11, which was a priority-interrupt system capable of handling several jobs at once time.[7]

Safety-critical systems are considered hard real-time systems because of their criticality, and time constraints could have similar importance to the dependability requirements. The design process of these systems demands not only correct and deterministic operation, but also a previous offline validation (Liu, 2000).

According to Walls (2012), there are four ways to implement a real-time systems: a simple processing loop, a background processing loop with interrupt service routines, a multitasking system using a scheduler, and a multitasking system using a Real-Time Operating System (RTOS). The first and second alternatives are suitable for simple applications. When the application becomes complex with several

[7] https://history.nasa.gov/computers/Ch2-6.html, Accessed: 2019-10-07

tasks and there is a great risk undesirable consequences, the third and fourth alternatives should be used. Task is a set of commands that implements its behavior in the context of CPU-based systems. The fourth one is more appropriate than the third one when the application demands some services like multiple task priorities and mailbox communication. For this work, it is used the FreeRTOS.[8]

2.2.1 FreeRTOS

FreeRTOS is a free kernel that is suitable for embedded real-time applications. It can be used with microcontrollers or small microprocessors. FreeRTOS was originally developed by Richard Barry in 2003 and was later developed and maintained by Richard's company, Real Time Engineers Ltd. FreeRTOS was a runaway success, and in 2017 Real Time Engineers Ltd. passed stewardship of the FreeRTOS project to Amazon Web Services.[9]

It uses the C language, schedules the processor with preemption for tasks with different priorities and uses the round robin scheduler with time slicing for tasks with the same priority. As process synchronization mechanism, it uses mutexes with priority inheritance, recursive mutexes, binary and counting semaphores. Lastly, it also uses streams, message buffers and queues for the task communication.

2.3 Robot Operating System

ROS is a framework that provides libraries and tools to speed up the development of robotic systems. It provides facilities for the implementation of communication between processes and between different computers. It is also an open source code project and as a large user community, providing a significant collection of algorithms and drives.[10]

[8] https://www.freertos.org, Accessed: 2019-10-07
[9] https://www.freertos.org/RTOS.html
[10] http://wiki.ros.org/pt

2.3.1 Concepts

Package:

The unit of the ROS organization is called Package. A Package can contain some source code, software libraries with tested algorithms, ready-to-use algorithms, a set of configuration files, and files describing the constitution of messages and services used by nodes. A Package allows easy reuse of libraries and executables. It is made up of a directory whose name is the same as the Package and consists of at least two files: CMakeLists.txt and package.xml.

CMakeLists.txt:

CMakeLists.txt is a file whose contents are input to CMake. It is a system that automatically builds executable files, speeding up the software design. Despite the difficulty with programming in Linux environment with CMake commands, its use facilitates the learning of ROS beginners and facilitates the development of large projects.

Package.xml:

Package.xml is a file whose function is to describe the Package that owns it. This file defines properties such as the Package name, the version of the existing code in the Package, its authors and maintainers. It also spells out all dependencies, allowing the usability of other Packages.

Node:

Node is a name given to the process that performs some computation and is created and executed on top of ROS. ROS combines nodes through a graph, facilitating communication between them. There are three ways to perform communication: via remote procedure call (RPC), via topic media flow, and lastly through parameter server. The goal of building applications using node concepts is to avoid the monolithic implementation of robots by decreasing complexity and increasing the reusability of source code.

Publisher-receiver communication model:

The publisher-receiver communication model is the most widely used form of information transmission between nodes in ROS. Publishers is the name given to nodes whose function is to send information to nodes whose function is to receive information. This is an abstraction of TCP/IP communication via sockets that ensures packet delivery and sequencing regardless of the route taken by the information flow. However, thanks to the ROS API, the developer does not need to perform low level configurations such as defining communication ports. He just uses the API and indicates where the information should flow. This model allows a node to be both a publisher and subscriber, and a publisher may send to multiple recipients through the same communication channel, or through different channels. In addition, the same idea is also true of recipients. The channels of communication in question are called Topics and each Topic only receives one type of information. These types of information are called messages, which in turn are defined in header files through data structures, similar to what is used in structural programming languages.

Client-Server communication model:

The Client-Server communication model, performed through remote procedure calls, is a different way of interaction between nodes. It allows us to request services from other nodes via a single communication channel and let the former know of the success or otherwise of their request. This is a communication model widely used in information systems, especially in internet applications. Similarly to messages, services are described in archives. However, it is divided into two parts, the necessary information that the customer must choose as the communication entry and the information returned by the service. The latter is often a boolean value, reporting success or failure of the service requested.

2.3.2 Versions

There are two versions of ROS: the standard ROS and ROS 2. The standard ROS was created to provide a development environment for the Willow Garage PR2 robot and has the following features:

- Environment created for a single robot;
- Workstation-class computational resources on board;
- No support for real-time requirements (or any real-time requirements would be met in a special-purpose manner);
- It demands excellent network connectivity (either wired or close-proximity high- bandwidth wireless);
- Environment created for applications in research, mostly academia;

However, along the development of robotic systems, the researchers and developers began to explore the development of other types of applications that have more restrictive requirements. To meet the requirements of these new applications, it was proposed to design the ROS 2, that has the following features

- Teams of multiple robots.

- Small embedded platforms.
- Support real-time control directly in ROS, including inter-process and inter-machine communication (assuming appropriate operating system and/or hardware support).
- Behave as well as is possible when network connectivity degrades due to loss and/or delay, from poor-quality WiFi to ground-to-space communication links.

2.4 Final remarks

This chapter introduced some concepts of the most important requirements in a safety-critical system design: dependability and real-time systems. They are necessary to ensure the correct and safe operation. Besides, the concepts of masking, standby, and hybrid hardware architecture were described. Fletcher's checksum was described and it has the purpose of improving the reliability of a communication process. In the end, ROS was presented.

These concepts are essential for understanding the subsequent chapters. The next chapter will describe the whole system design process. Then, in the Chapter 4, some tests will be performed to analize the operation, the dependability and the real-time features of the solution proposed in the Chapter 3.

3 Embedded system architecture

This work proposes a safe embedded system architecture for a safety-critical system and, as a case study, it is applied for a flight control system of a UAV. First of all, this chapter defines the requirements of the UAV's flight control system. Then, the hardware and software architecture to meet these requirements is proposed and, in the end, the prototype is presented.

3.1 Requirements

Requirements elicitation is an important step in a project. It guides the main decisions that must be made to meet all the demands related to a project. The designer must interview the people involved in the project and research the regulations and standards related to the system to be designed.

Regarding the aim of dealing with a UAV's flight control system, some aviation safety standard were consulted. DO-178C and DO-254 are international commercial aviation standards that provide the best practices for design an on-board system with an acceptable level of confidence to comply with the airworthiness requirements. They distinguish five levels of safety requirements called Design Assurance Levels (DALs), as shown in Table 1, that are classified according to the consequence in case of a failure of the system, where A is the most stringent and E is the least. (Fulton & Vandermolen, 2017). A market research has revealed the supply of DAL B certified UAV's autopilot[11]. Indeed, a failure of a UAV can cause at most some injuries or even some fatalities of people on the ground. Thus, we can consider as requirement of the project a target failure rate of 107 chance of hour failure/flight hour.

[11]https://www.embention.com/news/autopilot-uav-certification/
https://www.embention.com/projects/eko-custom-control-system/
https://www.embention.com/news/autopilot-uav-certification/

Table 1 Design assurance levels (DALs).

Design Assurance Level (DAL)	Description	Target Failure Rate
A (Catastrophic)	Failure causes crash, deaths	$< 10^9$ chance of failure/flight hour
B (Hazardous)	Failure may cause crash, deaths	$< 10^7$ chance of failure/flight hour
C (Major)	Failure may cause stress, injuries	$< 10^5$ chance of failure/flight hour
D (Minor)	Failure may cause inconvenience	No safety metric
E (No effect)	No safety effect on passengers/crew	No safety metric

The interview was conducted with some of the main researchers involved in ProVANT project. It was interviewed 6 PhD and Masters students, and 13 general questions to extract their demands were made. These questions and answers can be found in Appendix A.

Table 2: Requirements of ProVANT embedded system architecture to be met.

Number	Description
Req. 1	It shall use a platform with GPU
Req. 2	It shall have instrumentation to make automatic or manual operations
Req. 3	It shall communicate with a ground station
Req. 4	It shall not use proprietary software
Req. 5	It shall have temporal determinism: The control signals must be applied at the right time with a maximum sample time of 10 ms.
Req. 6	Target failure rate of 10^{-7} chance of hour failure/flight hour

3.2 Design Process

During a normal cycle of project, after the collection of requirements, the architecture of a system is designed. In the first step, the whole system is described without much detail in terms of schematics, diagrams and layouts of the project. In view of this context, the instrumentation required by the ProVANT 4.0 is described. Next, the hardware architecture is proposed. Lastly, the whole software architecture used in this work is presented.

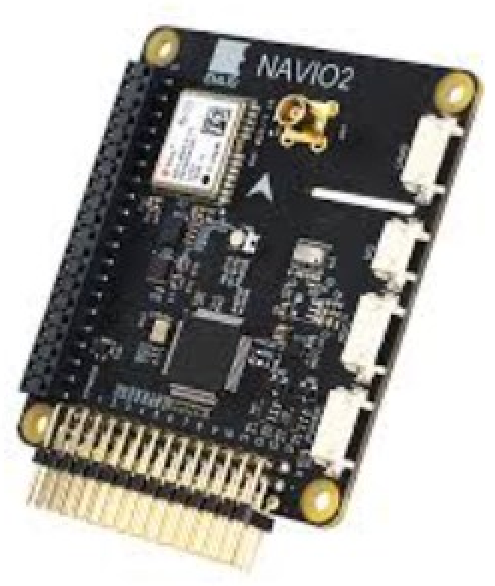
3.2.1 Instrumentation

As demanded by the requirements, the ProVANT 4.0 must operate manually or automatically and must communicate with a ground station (Req. 2 and Req. 3). In order to meet them, its flight control system should be composed by an Inertial Measurement Unit (IMU), an air data sensor, a radar altimeter, a Global Position System (GPS), servo motors, brushless motors, Electronic Speed Controllers (ESCs), Radios, and a power system manager.

IMUs are devices that infer the motion in a non-earth referenced frame. Inertial sensors combine gyroscopes, accelerometers, and sometimes magnetic sensors, depending on the device model. Accelerometer measures the linear acceleration, gyroscope measures the angular velocity, and the magnetic sensor identifies the Earth’s magnetic field to establish the direction of magnetic north. ProVANT 4.0 will use redundant IMUs for providing fault-tolerance features: two of them are provided by Navio2, and the third one is the IMU ADIS16480.

Navio2, that is shown in Figure 15, is an instrument board created to be used with the Raspberry PI 3. It is composed for 14x PWM channels, 2x IMU chips, 1x barometer and 1x GPS. The IMUs included in it are: MPU9250 and LSM9DSI. The former communicates by SPI with until 1 MHz and provides data of 3-Axis gyroscope, 3-Axis accelerometer and 3-Axis magnetometer. The latter communicates by SPI with until 10 MHz and provides data of 3-Axis gyroscope, the 3-Axis accelerometer and 3-Axis magnetometer.

Figure 15: Navio 2.



Source: Emlid

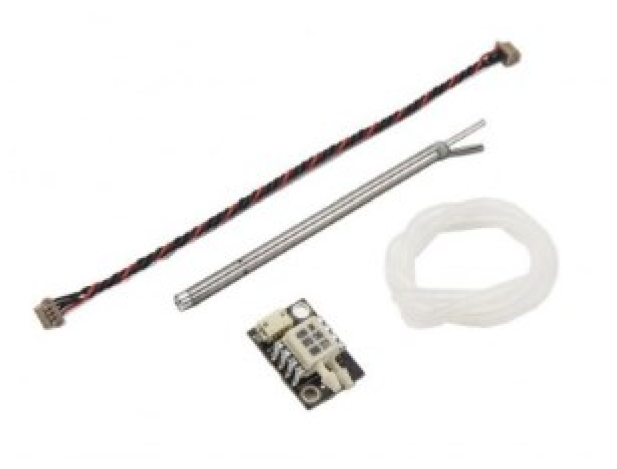
The IMU ADIS16480, that is shown in Figure 16, features a 3-Axis gyroscope, a 3-Axis accelerometer, 3-Axis magnetometer, pressure sensor, and an Extended Kalman Filter for dynamic orientation sensing. Its outputs are stable quaternions, Euler angles, and rotation matrix in the local navigation frame. As protocol of communication, it uses SPI with until 15 MHZ.

Figure 16: ADIS16480.

Source: Analog Devices

Air data system senses the wind flow through which the aircraft is flying, measuring the dynamic pressure, static pressure and temperature. From these data, it can infer the barometric altitude, the airspeed, the vertical speed, the Mach value, the air temperature, the true airspeed, and the angle of attack. The Navio2 provides a barometer, MS5611, which gives one float of static pressure and communicates by I2C. Besides, the IMU ADIS16480 also supply one float of static pressure. Lastly, the pitot tube 3DR Pixhawk Airspeed Sensor Kit is considered that, from I2C communication protocol, provides the dynamic pressure resulted by the air movement. It is presented in Figure 17.

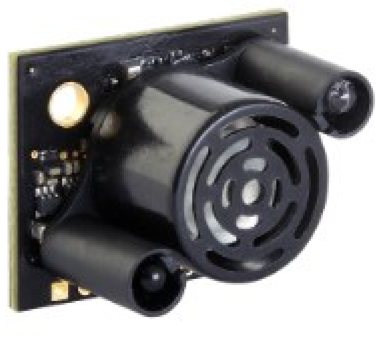
Figure 17: 3DR Pixhawk Airspeed Sensor Kit.



Source: Gaba Hobby Center.

The sonar uses sonic transmissions to reflect off a surface immediately below the aircraft. The sonar provides an absolute distance above the surface. This contrasts with the air data system, where the altitude allows to generate a warming that the aircraft is close to the ground and needs to take corrective action. ProVANT 4.0 uses an MB2530 IRXL-MaxSonar-CS3 that communicates by PWM, RS232 or analog voltage, which is shown in Figure 18.

Figure 18:MB2530 IRXL-MaxSonar-CS3.



Source: Max Botix

GPS is a satellite radio navigation system that provides a highly accurate position and a highly velocity to an unlimited number of properly equipped users spread all over the world. It provides a worldwide common grid reference system based on the Earth-fixed coordinate system. ProVANT 4.0 uses redundant GPS, one that is mounted in Navio2, a UBLOX NEO-M8N which provides the latitude, the longitude, the height, and communicate by SPI with until 5.5 MHz. Besides, a UBLOX NEO-M8T, as shown in Figure 19, is part of the GPS RTK module. This module provides a high accuracy latitude, longitude, height, and communicates by USB (Grigulo & Becker, 2018).

Figure 19: UBLOX NEO-M8T.

Source: GNSS OEM

ProVANT 4.0 uses two fast servomotor to tilt the propellers. They are composed by FLAT MAXON MOTOR BRUSHLESS EC 45 flat Ø42.8 mm of 50 Watt (shown in Figure 3.6), MAXON CONTROLLER ESCON 36/3 EC, and MAXON Sensor Encoder MILE, 512 CPT. The encoder communicates by RS422 and the controller by PWM. Moreover, four slow servomotor command the control surfaces of the UAV. They are Hitec D145SW Digital HV devices (see Figure 20), and each one communicates by PWM through the the Navio2 shield.

Figure 20: Hitec D145SW Digital HV devices.

Source: Modelflight RC

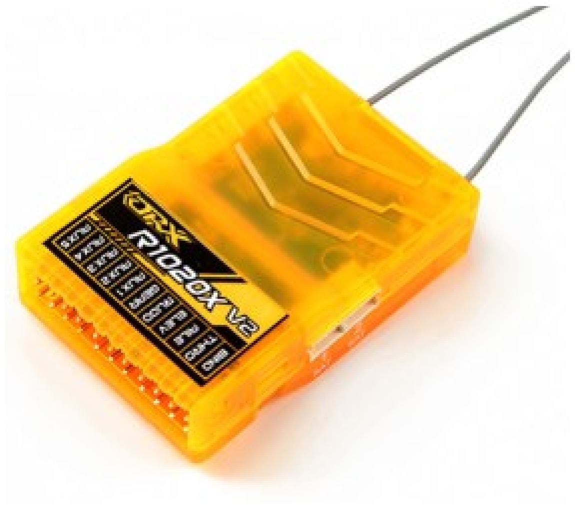
Brushless DC motors are synchronous electric motors powered by a hardware called Electronic Speed Controller (ESC), that is an electronic circuit in charge of, from DC power, controlling and regulating the motor speed. It may also provide reversing of the motor and dynamic braking depending on the model. In ProVANT 4.0, AXI 5345/14 HD 3D Extreme V2 brushless motors rotates the propeller (see Figure 21) and two Mezon 160 ESC command them. These ESCs communicate by PWM.

Figure 21: AXI 5345/14 HD 3D Extreme V2

Source: Modelmotors

As radio for telemetry, it will be used a OrangeRx R1020X (see Figure 22), which communicates by PPM. Lastly, the energy management system was designed by da Silva (2017) and communicates by UART.

Figure 22: OrangeRx R1020X.



Source: Hobbyking.

A table with more details of the instrumentation and embedded systems used in ProVANT 4.0 can be found in Appendix D.

3.2.2 Embedded hardwares

In order to execute high cost algorithms using GPU's resources, the Jetson TX2 development toolkit (see Figure 23, is chosen. It is a platform designed specially for running artificial intelligence algorithms. Jetson TX2 is composed by a Tegra X2, which is a system-on-ship with a 64 bits Quad ARM Cortex-A57 processor, a 64 bits Dual Nvidia Denver processor, and a GPU of 1,3 GHz (Req. 1) with Pascal architecture of 256 kernels.

Figure 23: Jetson TX2.

Source: Amazon

Moreover, it has 8 GB of RAM and 32GB eMMC storage capacity. As integration feature, it possesses:

- 1x HDMI 2.0;

- 1x 802.11a/bg/n/ac 2x2 867Mbps WiFi;

- 1x Bluetooth 4.1;

- 1x USB3 + 1x USB2;

- 1x 10/100/1000 BASE-T Ethernet;

- 12 lanes MIPI CSI 2.0, 2.5 Gb/sec per lane;

- PCIe gen 2.0, 1x4 + 1x1 or 2x1 + 1x2;

- 1x SATA;

- 1x SDcard;

- 1x dual CAN bus;

- 2x UART;

- 1x SPI;

- 3x I2C;

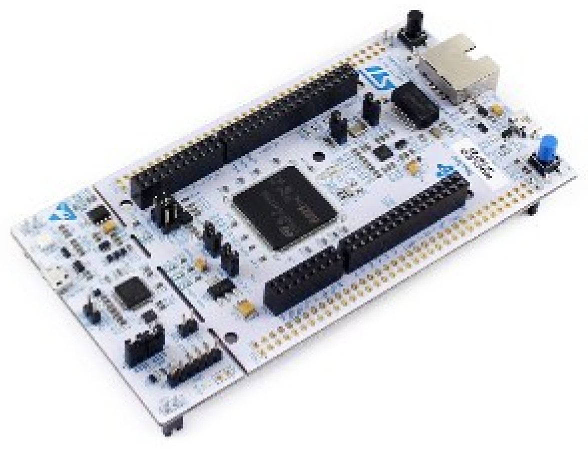
However, Jetson TX2 does not provide the necessary UART, SPI, and I2C interfaces demanded by the ProVANT 4.0 instrumentation, defined previously. In total, it is needed eighteen peripherals of communication. Thus, another platform is added with enough peripheral devices to establish communication with sensors and actuators, similar to the ones used in the previous UAV versions. From now on, to facilitate the understanding, Jetson TX2 will be called high level hardware (HLH) and the other one, the low level hardware (LLH).

The chosen LLH is a Nucleo-f767zi (see Figure 3.11), which is composed by Arm Cortex-M7 with 216 MHz, 2048 kB of Flash memory, and 512 kB of RAM. As integration feature, it possesses:

- 4x USARTs;
- 4x UARTs;
- 6x SPIs;
- 4x I2C
- 3x CAN
- 1x USB 2.0,
- 1x Ethernet
- 4x PWM

By using both hardwares, a two-layer architecture is proposed, where the Jetson TX2 is responsible to execute algorithms and the Nucleo board to interface the instrumentation. By and large, for each sample time, the instrumentation data are read by LLH and after sent to the HLH in order to execute the control law. After the control law is computed, the control signals are transmitted to LLH that commands the actuators.

Figure 24: Nucleo-f767zi



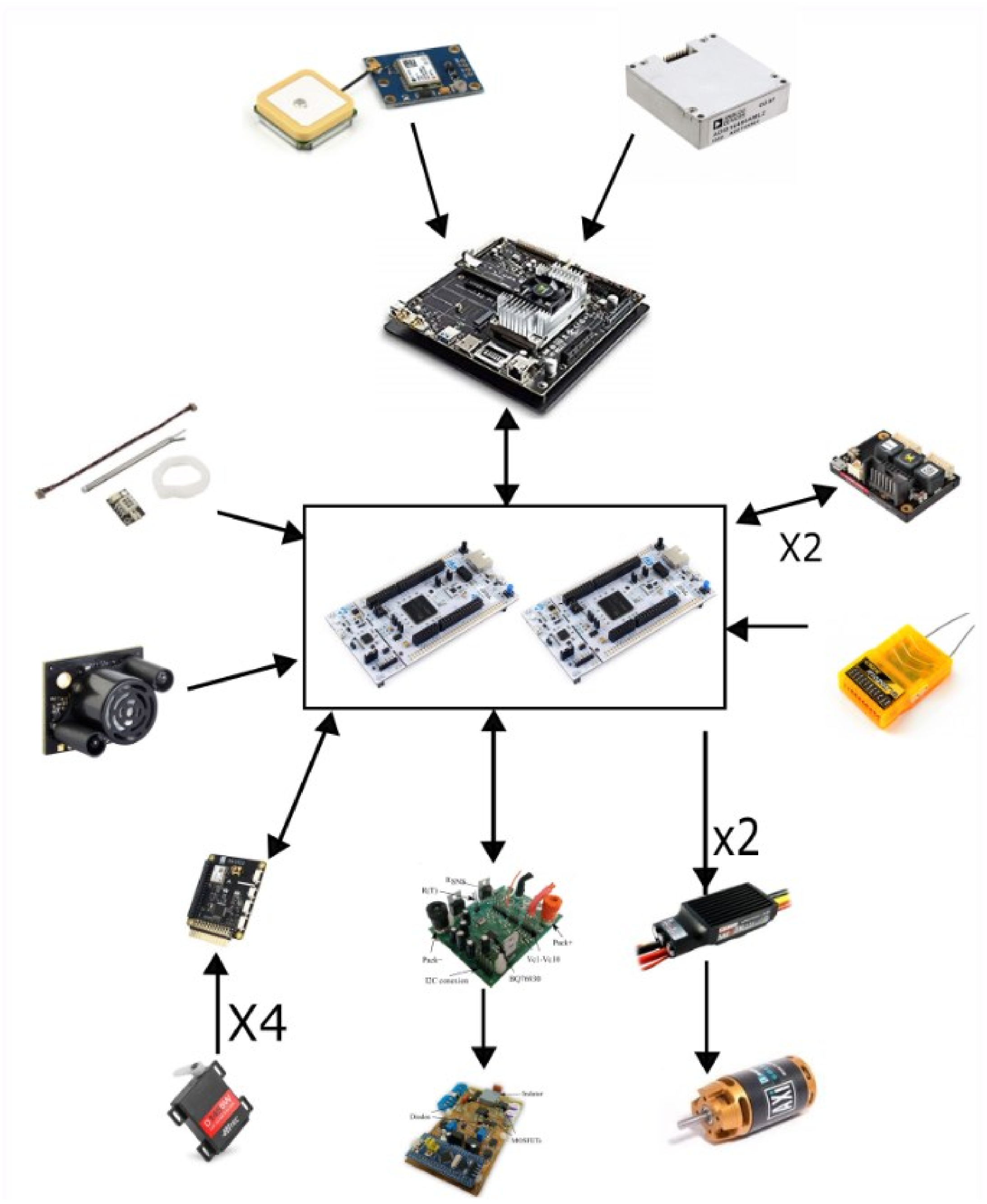
Source: Amazon

3.2.3 Hardware Architecture

The whole architecture proposed by this work is shown in Figure 25. The Jetson TX2 communicates with Nucleo boards from a UART channel. Besides, the most of sensors communicate with LLH, while only the GPS RTK and the high accuracy IMU ADIS16480 communicate directly with HLH in order to decrease the latency of the communication between HLH and LLH. During the work, the communication process proved to be a system bottleneck.

Aiming to increase the reliability of the communication channel between Jetson TX2 and Nucleo boards, and to detect of unwanted information changes during data transmission between LLH and HLH, a Fletcher's checksum was implemented. If any frame arrives at the destination with error, the receiver will render the same.

Figure 25: Hardware architecture.



Source: The author

Next, it is proposed an implementation in the LHL of a fail-operational control law, similar to Vivekanandan et al. (2016), in order to make the architecture be fault tolerant related to HLH faults. However, Vivekanandan et al. (2016) assumes that the

LLH is reliable enough to ensure safe operation of UAVs without any reliability data or experience. As the platforms chosen also has not detailed reliability information available by the manufacturers, two Nucleo boards are put in parallel, using the hot standby strategy and improving the solution proposed by Vivekanandan et al. (2016).

These strategies are justified since a UAV is a safety-critical application, which are essential to concern with dependability. However, in this thesis, we only deals with the HLH and LLH failures. Thus, it is not in the scope of this work to deal with instrumentation faults and the communication faults with ground station/radio controller, which are being dealt by other works of ProVANT's project.

As no detailed reliability information is available by the manufacturers, we collect some failures modes and faults for a general embedded system.

A) Software Failure Modes:

- Buffer overflow;
- Dangling pointers;
- Resource leaks;
- Race conditions;
- Semantic design;

B) Software Faults:

- Deadlock;
- Resource starvation;
- Too small memory;
- Bugs;

C) Hardware Failure Modes:

- Electrical failure;
- Mechanical failure;
- Temperature effects;
- Material failure;

D) Hardware Faults:

- Radiation;

- Hostile environments;

- Aging;

- Choosing the wrong dimensions;

- Manufacturing/assembly process deficiencies;

- Energy loss.

This work is dealing with permanent software and hardware faults in both platforms, for example, aging, communication failure, deadlock, energy loss and crash.

3.2.4 Software Architecture

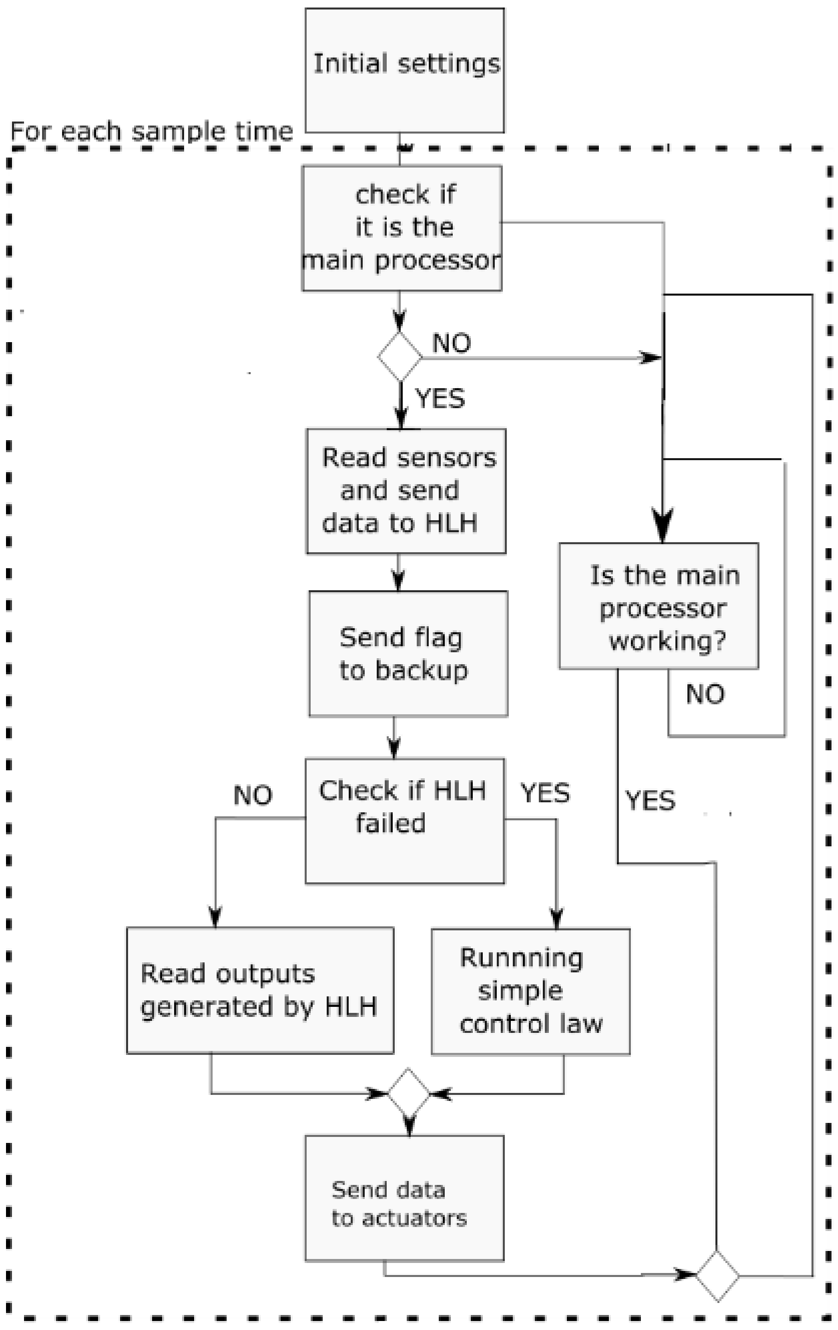
Logic description

Aiming to allow remote and automatic operation, the LLH have three tasks, each one with different priorities: Controller task, Radio task, and Groundstation task. The first one has the biggest priority and the last one, the smallest priority. Controller task is responsible for the communication with sensors, actuators and HLH, besides the execution of a fail-operational control law in case of HLH failure. Radio task communicates with radio, allowing remote operation of the UAV. Lastly, Groundstation task communicates with an external computer, passing data such as internal system information or obeyig commands.

The Controller task is periodic with period of 10 ms. Its logic is resumed by Figure 26. Note that this system has two modes of working: main LLH mode and backup LLH mode. Initially, the device checks which mode is configured. By default, every device are set up as a backup LLH mode. But as soon as the first LLH device is executed, it will not find any other LLH and it immediately turns to the main LLH mode.

Next, the second LLH will find the main LLH working and it will remain in the backup LLH mode.

Figure 26: New control task of LLH's processors.


Initial settings
For each sample time
check if it is the main processor
NO
YES
Read sensors and send data to HLH
Send flag to backup
Is the main processor working?
NO
Check if HLH failed
NO
YES
YES
Read outputs generated by HLH
Runnning simple control law
Send data to actuators

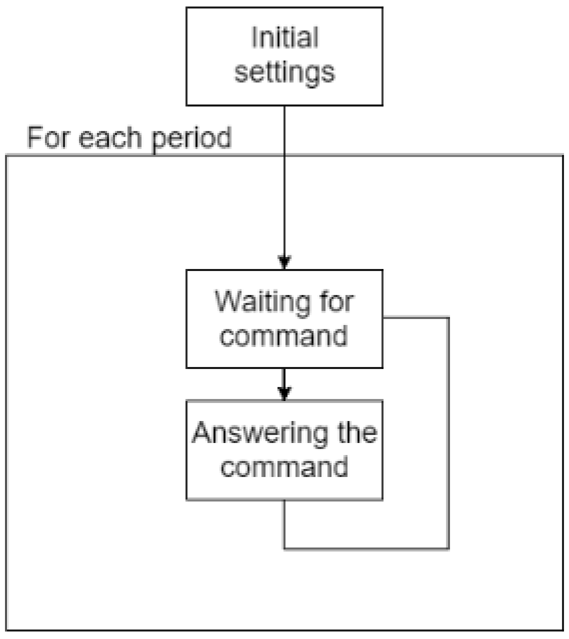
Source: The Author.

In the main board mode, LLH reads the sensors and sends the obtained data to HLH. Next, it sends a flag for the backup LLH and waits for output data from the HLH. If no data arrives in two sample times, it stores the information that the HLH is in failure mode and takes control of the UAV, running a simple control law. Otherwise, LLH obtains the output of HLH and commands the actuators with this data.

In the backup mode, LLH only waits for the flag sent by the main LLH. If no data arrives for three sample times, it will change its mode for main LLH mode and resets the other board.

The Groundstation task is periodic with the period of 500ms. The task checks if there is some data sent from ground station and, if it arrives, it answers. The logic of this task is shown in Figure 27.

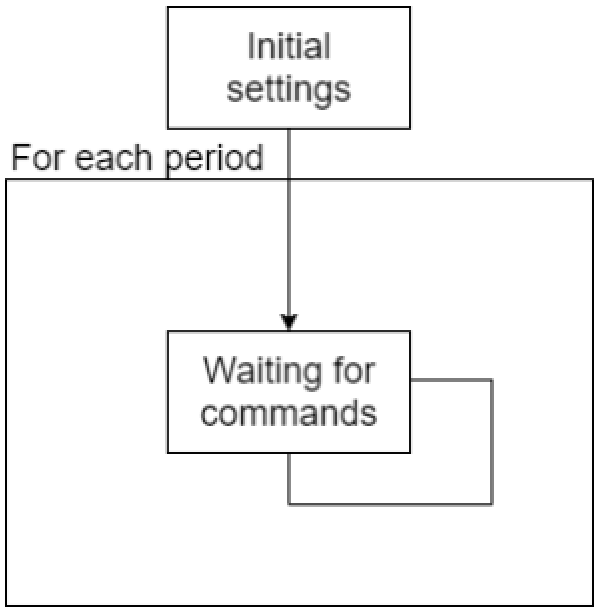
Figure 27: LLH’s Groundstation task.



Source: The Author

The Radio task is also periodic with the period of 100ms. The task only checks if there is some data sent from a radio. For example, if the received command is to switch the control law between remote/automatic, it changes a global variable to switch the operating mode. The logic of this task is shown inFigure 28.

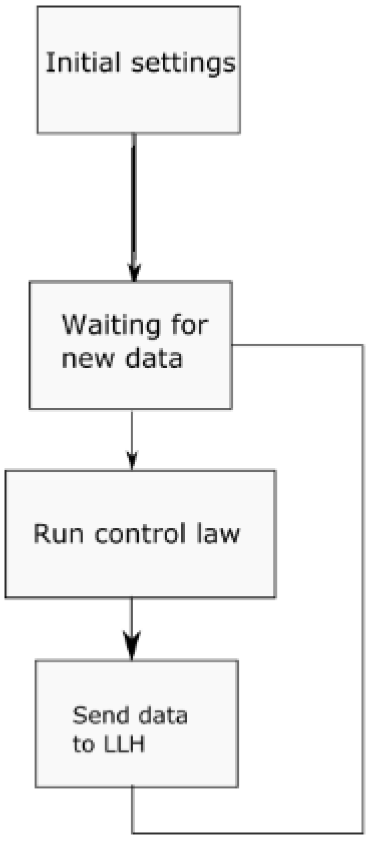
Figure 28: LLH’s Radio task.



Source: The Author

The logic of HLH is summarized in Figure 29. The HLH waits for LLH data to run the control law, and in parallel it has two periodic routines for reading data from an IMU and a GPS. After the execution of the control law, it sent the output to the LLH.

Figure 29: HLH’s Control task.



Source: The Author

Physical description

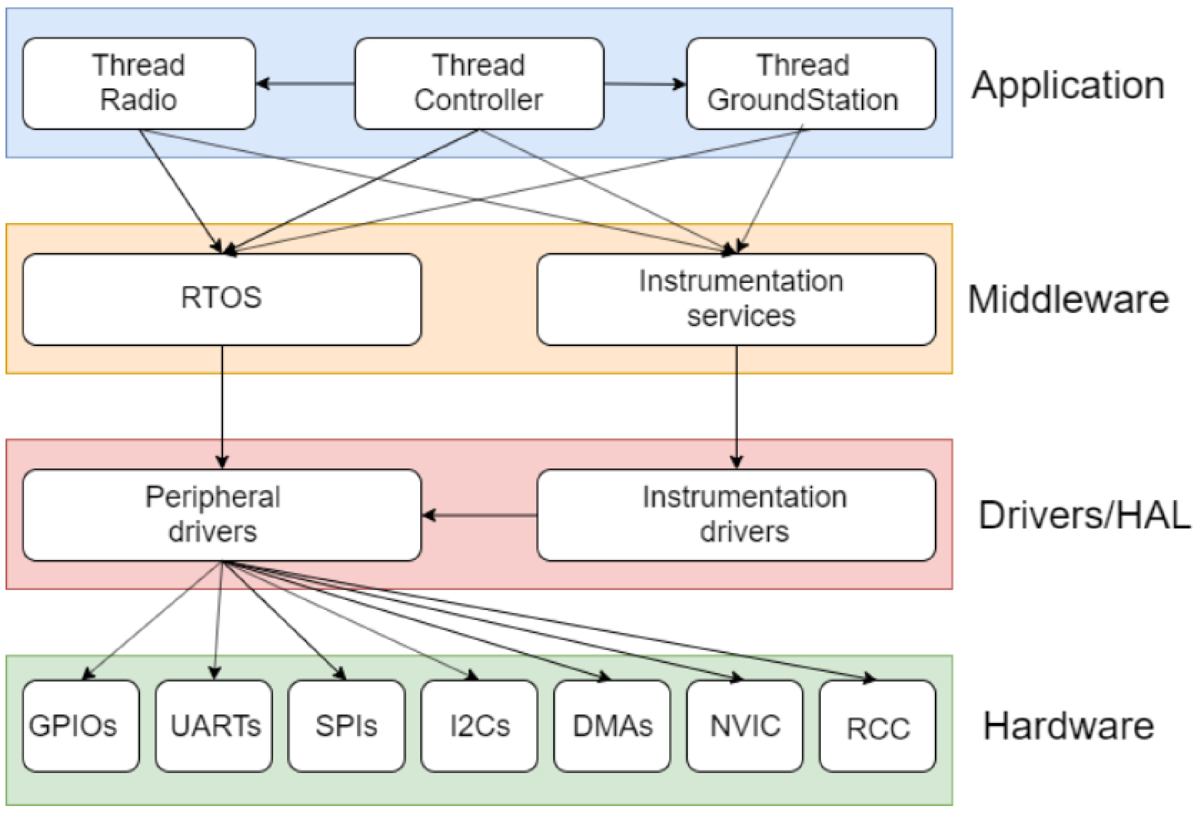
The LLH software is implemented using C language, and its architecture is designed in a vertical hierarchical structure of the elements of one level in relation to those of a higher level. The proposed architecture is shown in Figure 30. The software interacts with peripherals through peripheral driver supplied by the STM32CubeMX. It is a graphical tool that allows an easy configuration of STM32 microcontrollers and supplies a project with peripherals drives ready for use.

FreeRTOS at version 10 is the RTOS adopted because ProVANT research group members are already familiar with it. Besides, it is not a proprietary software (Req. 4) and it is also already available fully configured by STM32CubeMX.

The HLH architecture is structured as depicted in Figure 31. The platform uses Ubuntu 18.04 since it is the standard operating system provided by the manufacturer, and ROS 2, since it provides tools for speeding up the development of future applications and is especially designed for embedded applications, unlike of the ROS Standard (Req. 4).

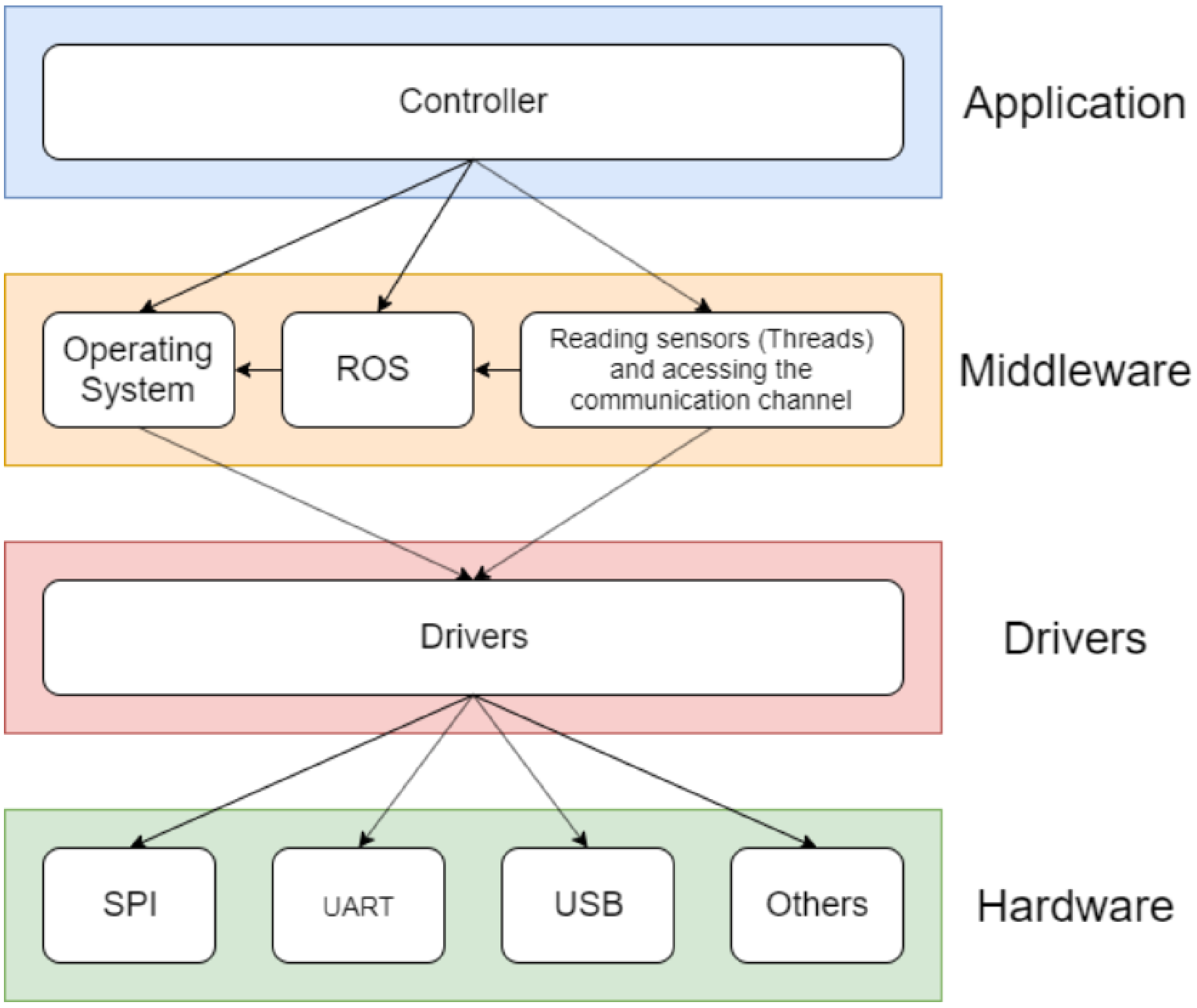
The HLH's drivers are available by default with the operating system, and the services of reading sensors and communicating with the communication channel must be implemented by the designer using the available drivers. At the moment, the HLH only executes the control task. This task has been developed using the C++ language. The control law must be implemented inside class method, whose interface has already used in a simulation software of ProVANT project (Lara et al., 2017), providing development integration between both tools. Its implementation is explained in the Appendix C.

Figure 30: Description of LLH's software architecture.



Source: The Author

Figure 31: Description of HLH's software architecture.



Source: The Author

3.3 Final Remarks

This chapter describe the design process of the hardware and software architecture to be applied in ProVANT 4.0. First, requirements were collected to guide the design process,which consisted of a survey by international standards and interviews with researchers involved in the ProVANT project. After, it was proposed an embedded system architecture for ProVANT 4.0. Lastly, it was shown the prototype built from the proposed architecture. The next chapter will describe the experimental results obtained with the prototype. The dependable and real-time features will be analyzed.

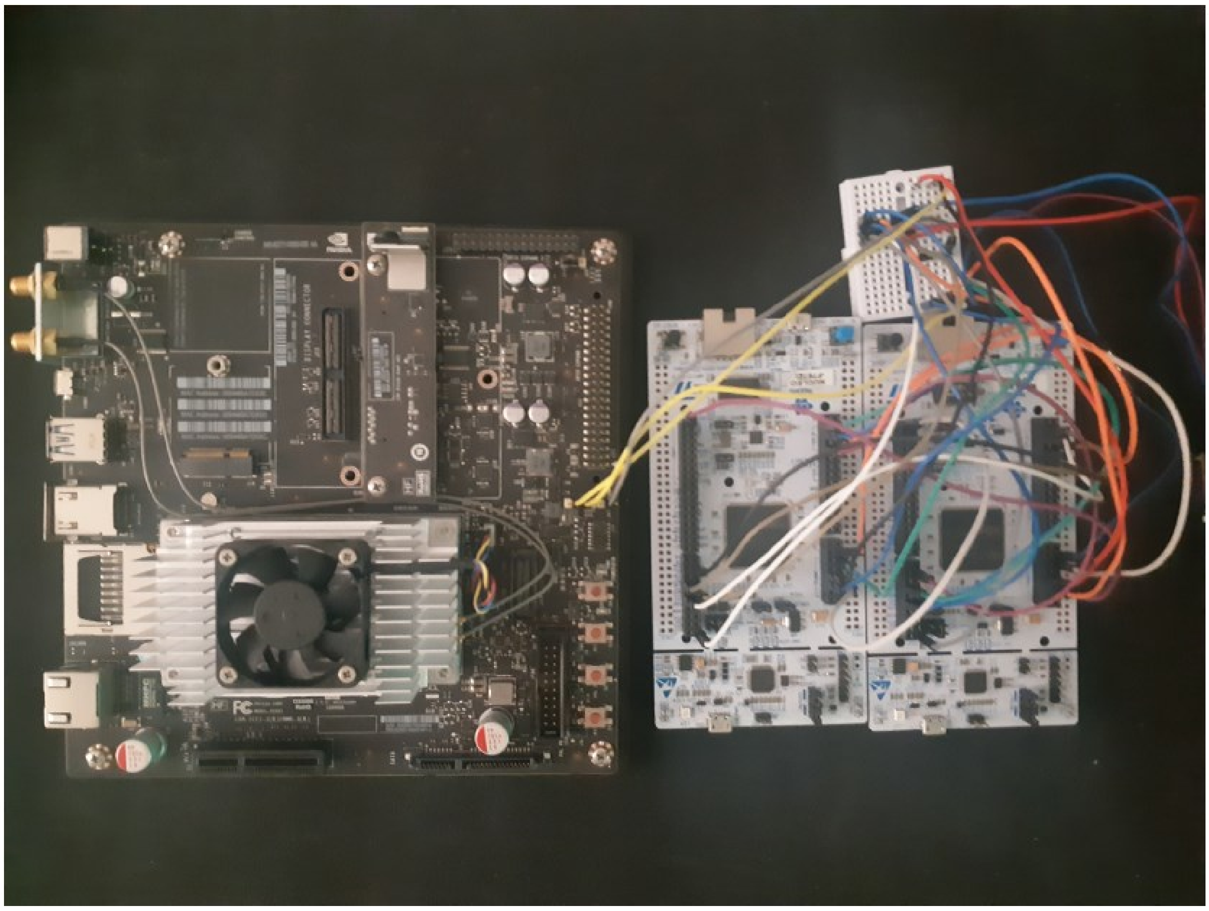
4 Experimental results

This chapter describes the experiments performed with the prototype presented in Chapter 3. In the first section, the HIL simulation environment is presented, which was developed to validate the proposed solution. After, a simulation is performed in order to test the flight of the UAV without faults, to test the flight with faults in the HLH, and with faults in the main LLH. From the experiments two characteristics are analyzed: the hard real-time capabilities and the dependability.

4.1 Prototype

Based on the architecture described in Chapter 3, a prototype was built to analyze the communication between the three hardwares. It is shown in theFigure 32. The schematic of the prototype is found on Appendix D.

Figure 32: Prototype built.



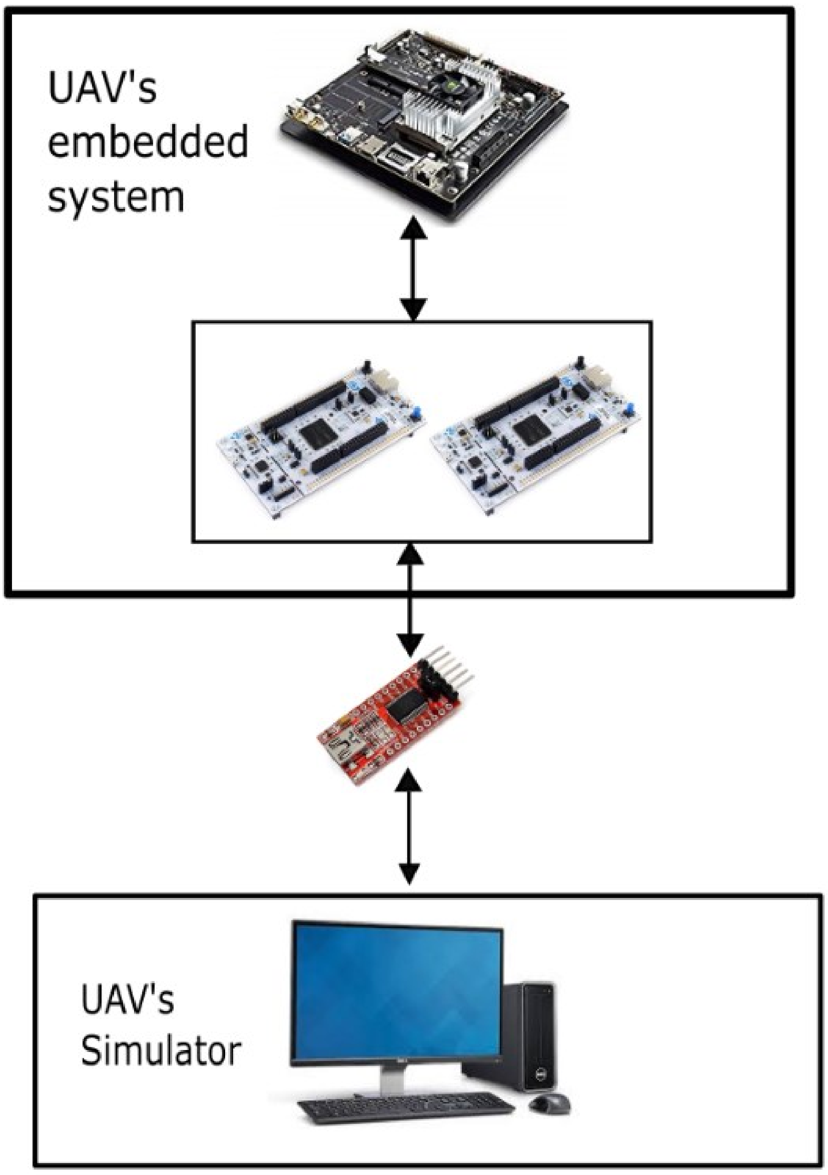
Source: The Author.

4.2 Hardware-in-the-loop simulation for the UAV embedded system

A simulation is any study where many aspects of a phenomenon are translated into mathematical models and executed in computer programs to mimic the outcomes that happen in the natural world. In the context of control systems, simulation is a process that conducts experiments with computational or mathematical models of a system in order to test the efficiency of a control strategy.

However, rather than testing the control algorithm purely in a simulation environment, such as Matlab/Simulink, there is a specific kind of simulation that provides a way of testing of hardware and software performance, besides the integrity of embedded systems, it is called the hardware-in-the-loop simulation (HILS). Thus, in order to validate the embedded system proposed in this thesis, a HILS for the UAV embedded system is designed as shown in Figure 33. While a UAV dynamic model is being simulated on a desktop, the embedded system runs in parallel controlling the simulation model.

Figure 33: Hardware-in-the-loop environment.



Source: The Author.

There are some approaches in the literature that look into the small UAV control design using HILS. Gans et al. (2005) developed a HILS environment for airplane UAV control tests. This work uses virtual reality software to produce real-world scenarios and a wind tunnel for aerodynamic simulation of the aerial vehicle. Trilaksono et al. (2011) designed a HILS for visual target tracking of an octorotor UAV with onboard computer vision. In Cheon et al. (2016), a HILS platform was designed for verifying the image-based object tracking method used in a UAV, composed by image processing, scene generation, and flight control modules.

The HILS used in this thesis is an improvement of the ProVANT Simulator (Lara et al. (2017), Lara et al. (2018)). ProVANT Simulator is a simulation environment based on 3D CAD (Computer Aided Design), the Gazebo Simulator and ROS, with the purpose of validation and implementation of control strategies, being a previous stage of flight testing.

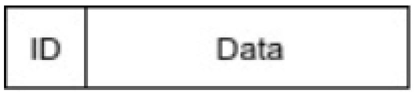
4.2.1 Communication

To perform the communication between the general purpose computer and the embedded system, it has been chosen the UART protocol in the embedded system side and the USB protocol in the general purpose computer side. Thus, to make the translation of these serial protocols, two FT232RL converters are used: one is responsable for sensors and actuators data transfer, and the other for reference data transfer.

The serial communication uses a baud rate of 921600 bps in order to provide fast communication with a worthless time lag. Besides, this application uses a two layer communication protocol. The low level layer tackles data corruption and the entanglement of packages by sending the data with redundant information to solve these problems by using the Fletcher’s Checksum, and the high level protocol handles the flow of communication by implementing a client/server protocol.

The high level protocol consists in a client/server communication where the embedded system is the client of the application and asks to the general purpose computer for services identified through their ID and some information, if necessary, as showed in Figure 34. The following services are provided: i) Simulation start; ii) Reading of sensors data; iii) Transmission of data to actuators.

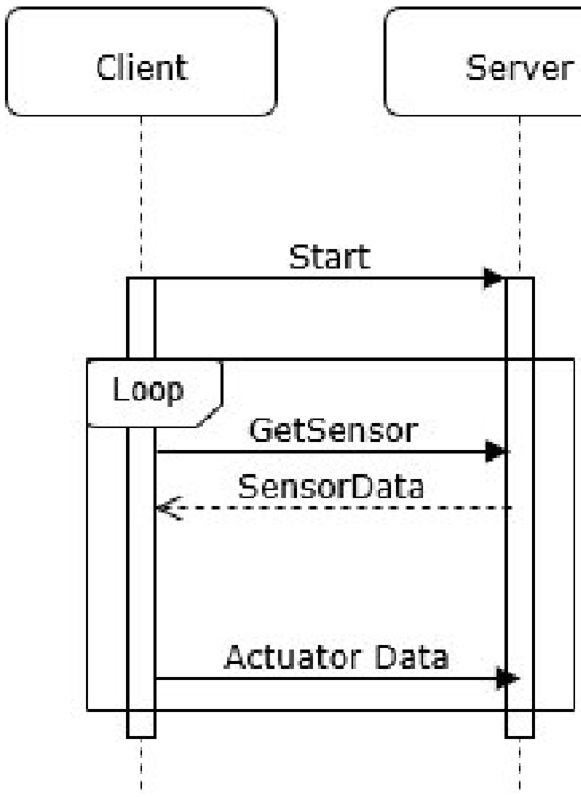
Figure 34: High level protocol's package.



Source: The Author.

The data flow in the communication process works as depicted in Figure 35. First of all, the client starts the application, sending a message with ID 1 without waiting for any response. From this moment, a periodic cycle of requesting sensor data and sending actuator commands is started. In the sensor data request, the client sends a message with ID 2 and waits for a server's answer, which consists of one float array of 16 elements. However, when it sends actuator commands, it uses a message with ID 3 plus our float numbers corresponding to the control signals provided by the control law. As a client/server protocol, this periodic cycle is set on the client-side, while the server just waits for requests.

Figure 35: Communication data flow between server and client.



Source: The Author.

4.2.2 Simulator settings

The general purpose computer uses the linux distribution Ubuntu 18.04 LTS (Long Term Support) as operating system. Besides, the Gazebo Simulator is employed to simulate the dynamic behavior of the Tilt-rotor-UAV.

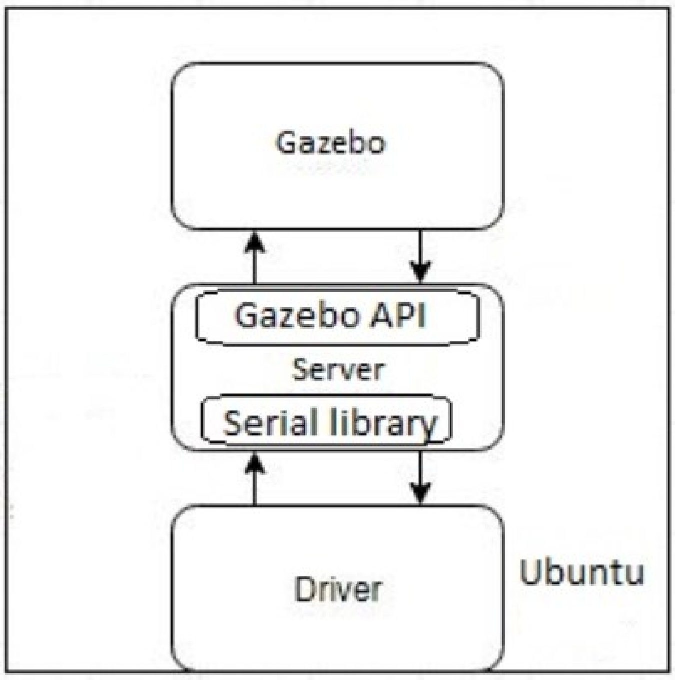
The simulation step of Gazebo simulator was adjusted for 4 ms, which is the period when the simulator obtains the data of control signals and computes the actual states of the system. The simulation step influences the accuracy and execution time of the simulation, the shorter the period, the greater the computational effort. This configuration was chosen because it is accurate enough for flight simulation and fast enough to perform it in real time.

In addition, the Gazebo Simulator is configured with real-time_factor setting equals to 1, making the simulator tries to keep the simulation in real time according to the system clock.

In order to interact with the simulation environment, either by acquiring data and applying control signals, or by changing simulation configurations, it was created a dynamic library called Plugin to be responsible for getting requests from the serial communication. The Boost[12] ASIO API (Application Program Interface) is used, which is a cross-platform C++ library for networking and low-level I/O programming, while the Boost Thread API is used for creating and managing threads.

Plugin works as shown in Figure 37, which is composed for one thread that waits for external requests and one callback that is called in every simulation step. For each type of request, specific reaction occurs and the verification process is performed in the following sequence: 1) start simulation, 2) send actuator data to the simulator, and 3) obtain sensor data.

Figure 36: Server’s software architecture.

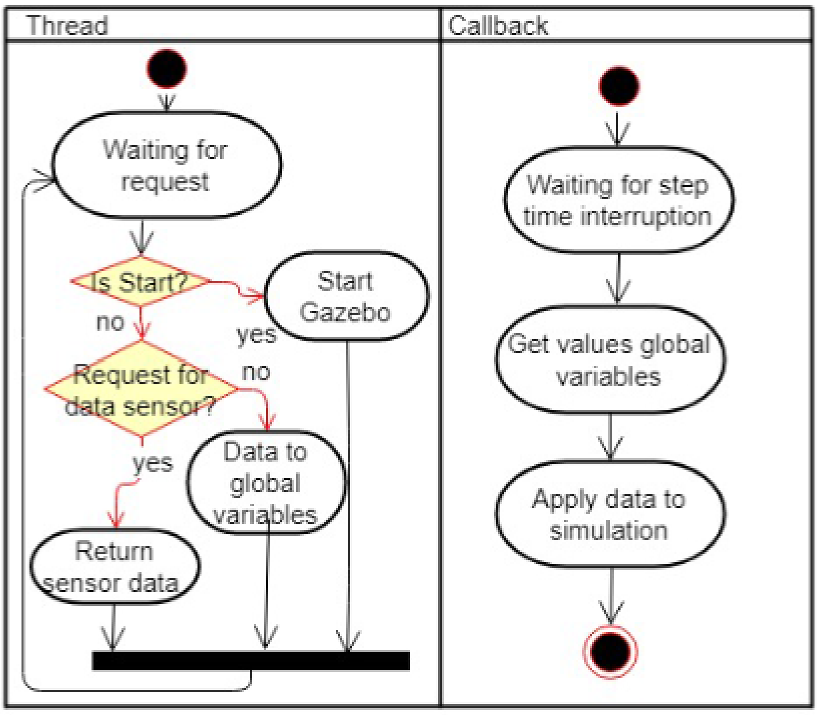


Source: The Author

[12] http://www.boost.org

Source: The Author.

Figure 37: Logic of server software.



Source: The Author.

4.3 Numerical experiments

Numerical experiments were performed to conduct the proof of concept of the architecture rather than to analyze the performance of the proposed solution, while executing high computational cost control algorithms. In these experiments, both HLH and LLH embedded systems were configured with a Discrete Linear Quadratic Regulator (DLQR) control technique proposed by Rego & Raffo (2016) without integral terms. Besides, the implementation of the control law in the HLH does not use the GPU, since the purpose of this experiment is only to validate the proposed architecture.

The control design was based on the physical model presented in Cardoso et al. (2019). In this experiment the ProVANT 4.0 (see Figure 4.7) model must keep in a reference position that changes periodically in two set-points: $\boldsymbol{r_1} = [0, 0, 2]^T$; and $\boldsymbol{r_2} = [0, 0, 2.2]^T$.

The kinematic description of the system is performed according to Figure 38, where $\boldsymbol{\xi} \triangleq [x, y, z]^T$ corresponds to the position of the main body with respect to the inertial frame; φ, θ and ψ describe the orientation of the main body with respect to the inertial frame through the Z-Y-X convention on local axes; α_R and α_L describe the inclination of the propellers with respect to the main body of the aircraft. The vectors $\boldsymbol{d_{Ai}^{B}}$ and $\boldsymbol{d_{Ci}^{Ai}}$, with i ∈ {1, 2, 3}, and the angle of inclination β correspond to design parameters of the aircraft. Table 3 presents the physical parameters of the UAV model used to tune the DLQR controller.

The control strategy is based on the linearization and discretization of the system state equations, obtained through the Euler-Lagrange formulation, around the reference using sampling time of 10 ms. The generalized coordinates are **q** = (x, y, z, φ, θ, ψ, α_R, α_L) and this information is read from the simulator with their derivatives. The parameters used for control design are

$$\boldsymbol{R} = \mathrm{diag}(\frac{1}{15}, \frac{1}{15}, \frac{1}{0.1}, \frac{1}{0.1}),$$

$$\boldsymbol{Q} = \mathrm{diag}(\frac{1}{0.01}, \frac{1}{0.01}, \frac{1}{0.1}, \frac{1}{0.1}, \frac{1}{0.1}, \frac{1}{0.00005}, \frac{1}{0.00005}, \frac{1}{0.00005}, \frac{1}{0.01}, \frac{1}{0.01}, \frac{1}{0.1}, \frac{1}{0.1}, \frac{1}{0.1},$$

$$\frac{1}{0.005}, \frac{1}{0.005}, \frac{1}{0.005}),$$

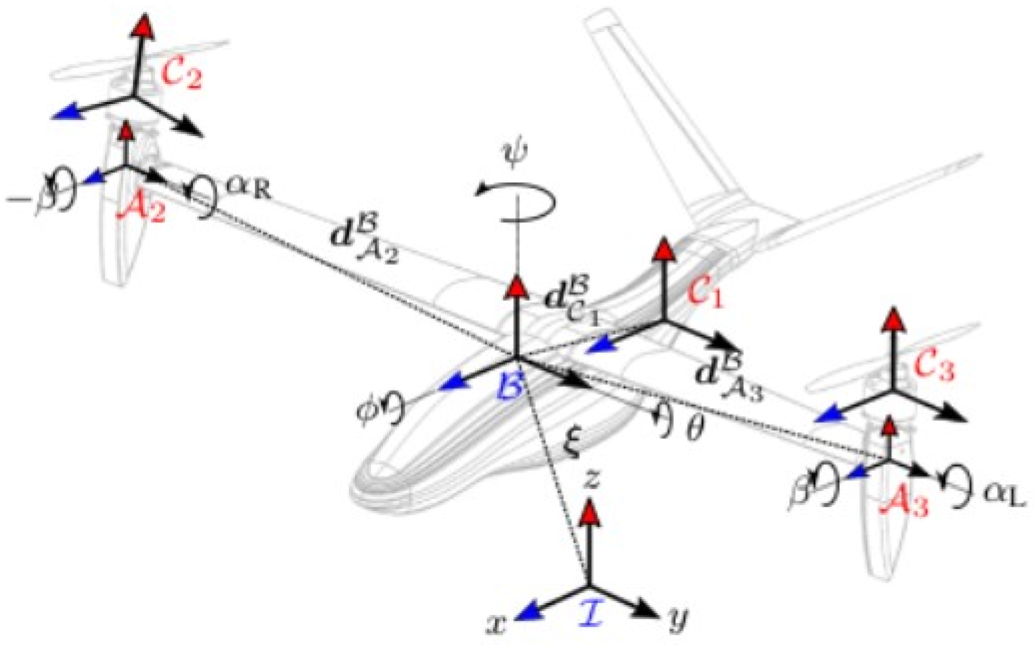
with f_{eq}^{R} = 37.3 N, f_{eq}^{L} = 37.3 N, τ_{eq}^{R} = 0 N.m, τ_{eq}^{L}= 0 N.m.

The proposed architecture was validated with the HIL simulation environment described before, using the designed DLQR control law and considering fault injection. In the following experiments, a energy loss is induced on the boards and represents any of the permanent faults described on Chapter 3.

Table 3: System physical parameters.

Parameter	Value
m_1	1.92 Kg
m_2, m_3	0.08 Kg
d^{B}_{C1}	$[-0.0609\ 0\ -0.0634]^T$ m
d^{B}_{A2}	$[0.0083\ -0.6073\ 0.0406]^T$ m
d^{A2}_{C2}	$[0\ 0\ 0.031]^T$ m
d^{B}_{A3}	$[0.0083\ 0.6073\ 0.0406]^T$ m
d^{A3}_{C3}	$[0\ 0\ 0.031]^T$ m
$I_i 1$	$\begin{bmatrix} 0.1489 & 0 & -0.0189 \\ * & 0.1789 & 0 \\ * & * & 0.3011 \end{bmatrix}$ Kg·m²
$I_i 2$	$\begin{bmatrix} 7.103 & 0 & 0 \\ * & 7.1045 & 0 \\ * & * & 0.2133 \end{bmatrix} \cdot 10^{-3}$ Kg·m²
$I_i 3$	$\begin{bmatrix} 7.103 & 0 & 0 \\ * & 7.1045 & 0 \\ * & * & 0.2133 \end{bmatrix} \cdot 10^{-3}$ Kg·m²
$\hat{\boldsymbol{g}}$	$[0\ \ 0\ \ -9.81]^T$ m/s²
k_τ	$1.7 \cdot 10^{-7}$ N·m·s²
b	$9.5 \cdot 10^{-6}$ N·s²
β	0°

Figure 38: Reference coordinate systems



Source: Cardoso et al. (2019)

4.4 Results

Figures Figure 39, Figure 40, Figure 41 illustrate the results of position in X, Y and Z axis of the ProVANT 4.0 obtained with the HILS. The variable Mode expresses the situation of the embedded system in a given instant.

• Mode 0: represents the embedded system in full operation;

• Mode 2: represents a communication failure between HLH and LLH (such a failure was not caused by an injection fault);

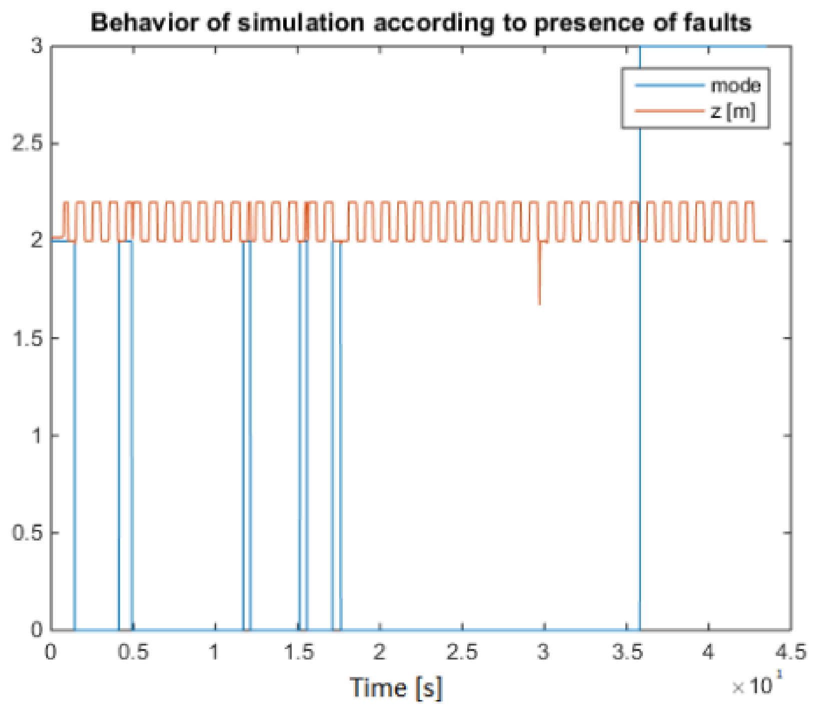
• Mode 3: represents the embedded system with HLH in permanent fault.

Unfortunately, the embedded system was in the failure mode 2 very often. The information from the HLH arrived corrupted to the LLH, but the Fletcher’s code did not detect them. This is acceptable because the system under test is a prototype, the presence of poor wire contact was inevitable and the Fletcher’s code like any fault tolerance technique is not 100% effective. This failure was detected using a comparison with the output computed in the LLH’s control law. Moreover, the transfer behavior from Mode 2 to Mode 0 was not performed by any automatic feature. This transfer has been done manually, as we were investigating at that moment the reliability of the entire system with all components working perfectly.

At t=29.7 s and t=30.1 s, a fault was injected in the LLH main processor. In Figure 42 this issue is highlighted. The system does not change its mode because the UAV was still under control by HLH. The UAV lost the control for some period because the hot standby strategy needed this period to switch the hardware. Quickly, the UAV stabilized again as expected.

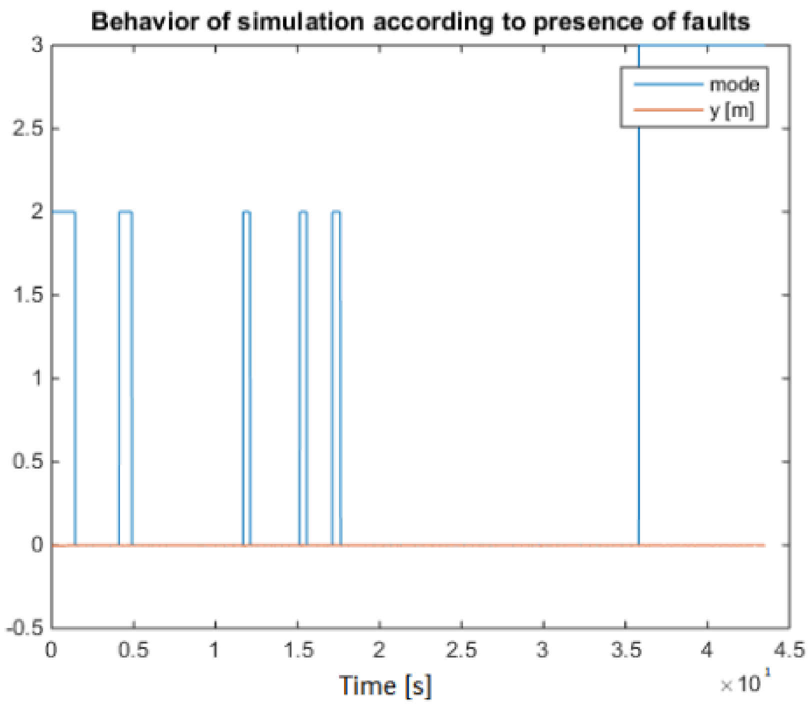
Lastly, at 35.9 s, a fault was injected in the HLH and the embedded system get in Mode 3 as expected. In Figure 43, this part of the simulation was zoomed. At this time, the LLH immediately takes the UAV flight control.

Figure 39: Behavior of Z-axis position in the simulation.



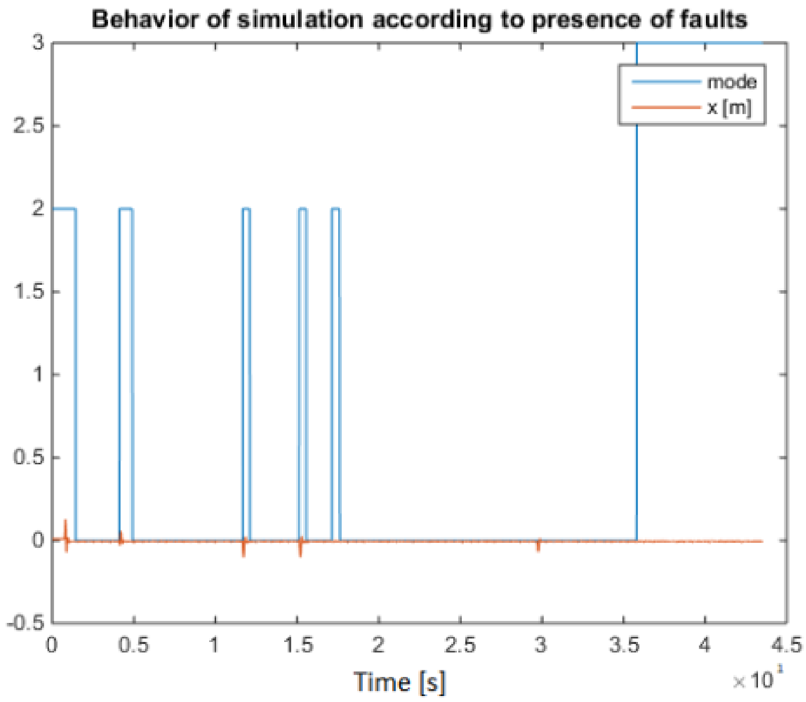
Source: The Author

Figure 40: Behavior of Y-axis position in the simulation.



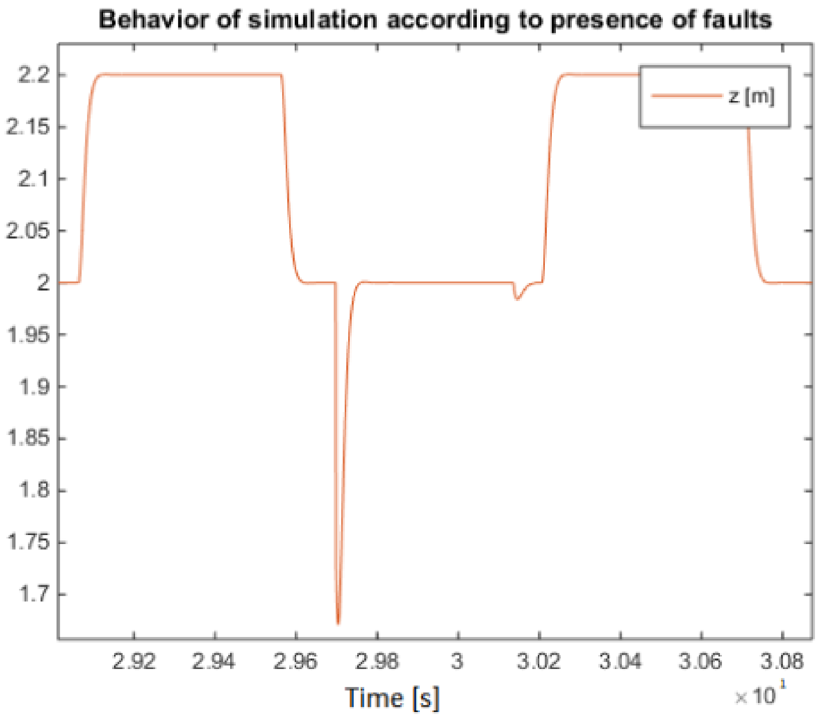
Source: The Author

Figure 41: Behavior of X-axis position in the simulation.



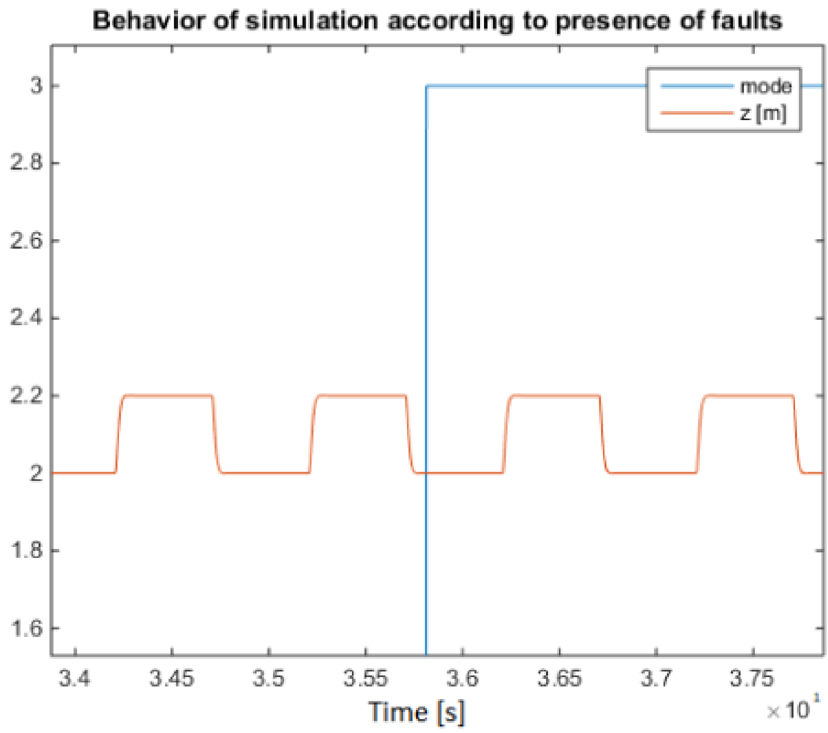
Source: The Author

Figure 42: Highlight the moment occurred LLH's failures.



Source: The Author

Figure 43: Highlight the moment occurred HLH's failure.

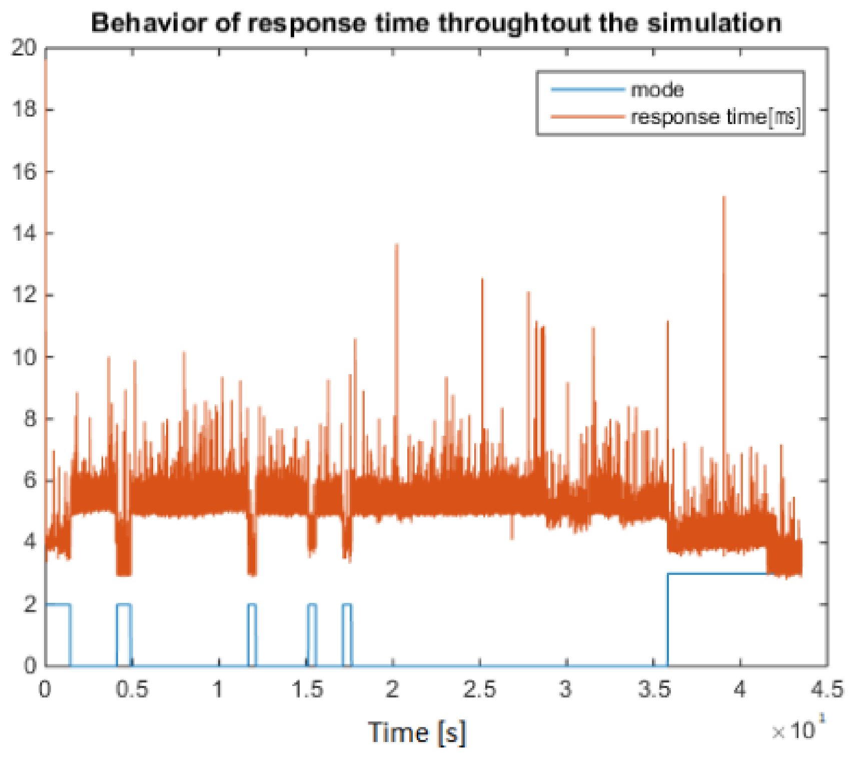


Source: The Author

The system proved to be robust to the injected faults during the simulation, demonstrating the success of the techniques adopted, increasing the system's dependability. However, nothing can be said about the meeting of the Requirement 6. The first reason is that we could not have the access of reliability data of all platforms used in this work. Another reason is that, as the target failure rate is about 10−7 chance of hour failure/flight hour, it will take several hours of operation to make any conclusions given the small probability required. Lastly, a prototype more resistant to bad contact failures is needed to be built.

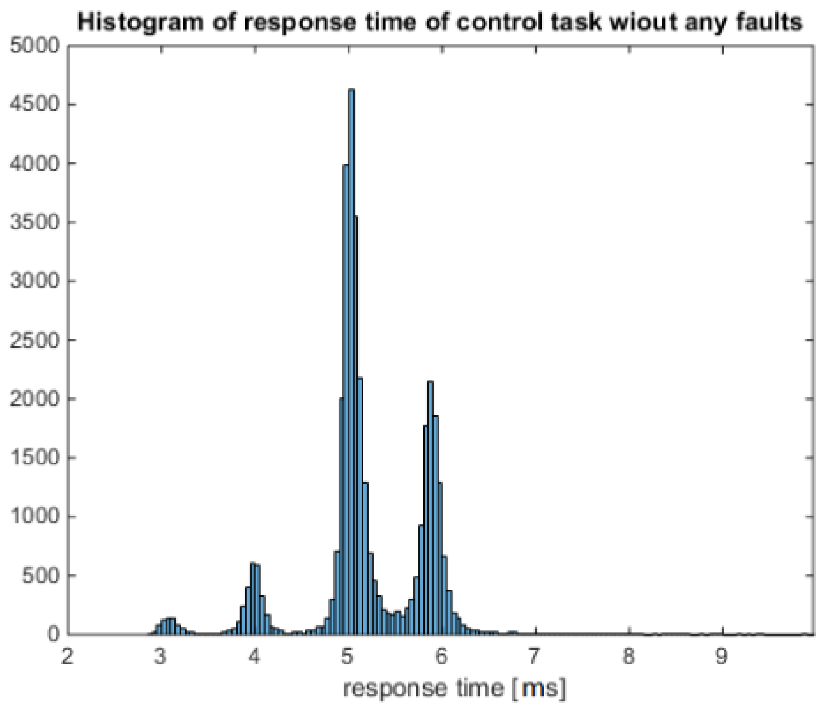
To analyze and validate the proposed solution related to the real-time requirement, the response time of the Controller thread should be obtained and, after, an offline validation must be done. Figure 44 shows the response time behavior throughout the simulation, and Figure 45 presents an histogram of the samples collected when the embedded system is in Mode 0, since it is the Mode when the longest system response time is achieved.

Figure 44: Behavior of the task's response time.



Source: The Author

Figure 45: Histogram of the task’s response time.



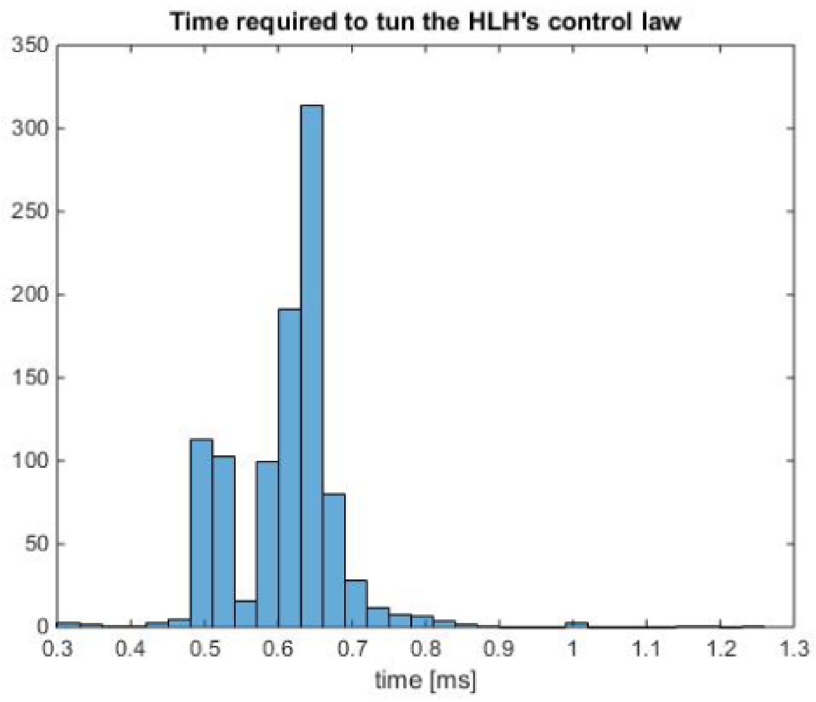
Source: The Author

As can been seen in the histogram, there is random behavior on the samples during the simulation. This behavior occurs due to we are collecting data on the side of the simulator that is running on a non-real time operating system in a multi-core processor. Despite of this behavior, the worst case response time value was 6,4 ms, which is demanding 64% of the sampling time.

From part of the data collected in the HLH, the controller took about 0.65 ms to run each instance of the control law, as shown in Figure 46. Therefore, we can conclude that the solution is poorly optimized regarding the communication latency between its components, demonstrating a demand for reducing this bottleneck.

Despite this bottleneck, 36% of the capacity is still available for Radio and Groundstation Threads and, as they have much longer deadline compared to the Controller thread, this solution in the HIL has potential to be validated. In addition, it still must be evaluated the response time of other threads in order to ensure the meeting of Requirement 7.

Figure 46: Histogram of the time required to run the HLH’s control law.



Source: The Author

5 Conclusions

This work proposed a safety-critical embedded system architecture to be applied in the flight control system of a VTOL-UAV. It uses a Jetson TX2 development kit, in view of growing demand for high computational cost algorithms. Different from current work related to the autopilot design, this thesis deals with the autopilot as a safety-critical system.

During the design of the solution, some researchers involved into the project of the UAV was interviewed. From the interviews, some functional and nonfunctional requirements were collected. Besides, in order to guide the dependability analysis, some aviation safety standards were consulted and, from a research on market, the dependability required for a UAV operation was inferred.

Consequently, during design phase a new hardware and software architecture was proposed. As the Jetson TX2 development kit does not have enough peripheral devices to communicate with all instrumentation required, an additional hardware, a Nucleo-F767zi board, was selected to do this interface. Their operating systems and software architectures were also defined. The Jetson TX2 development kit uses Ubuntu 18.04 and ROS, and Nucleo-F767zi board uses FreeRTOS with version 10.

In addition, some changes have been made to increase the dependability of the solution. Hardware redundancy and information redundancy techniques were used. They are an improvement of the architecture suggested by Vivekanandan et al. (2016). In order to increase the safety of the proposed architecture, we used a hot standby technique. This improvement makes sense because LLH was a critical point of failure and the most important hardware of the architecture.

In order to test the solution, a Hardware-in-the-loop simulation environment was designed in order to inject faults during the simulation. Then, experiments showed that the UAV remained operational despite the occurrence of faults, and the fault-tolerant strategies worked as expected. Therefore, it can be conclude that the solution could meet most of its requirements.

5.1 Future Work

Some future works of this project are:

- Make a comparison with other architectures.
- Investigate the adoption of RedHawk Linux.
- Perform a study of the communication latency between all components and the instrumentation specified.
- Obtain the hardware reliability data for numerical pre-validation of the architecture proposed;
- Obtain instrumentation reliability data and use them in the dependability analysis of the whole system;

- Search for a high performance hardware with sufficient communication peripherals, or develop a custom embedded hardware;
- Test other fault tolerance strategies, for example, the Self-Purging Redundancy Architecture and GPU’s hardening techniques.
- After going through a robust manufacturing process, thoroughly validate the solution for several hours in order to evaluate if the required dependability level is reached.

Bibliography

Abbott, D. (2011). Linux for embedded and real-time applications. Elsevier.

Alemzadeh, H., Iyer, R. K., Kalbarczyk, Z., & Raman, J. (2013). Analysis of safety-critical computer failures in medical devices. IEEE Security Privacy, 11(4), 14–26.

Arm, J., Bradac, Z., & Kaczmarczyk, V. (2016). Real-time capabilities of linux rtai. IFAC-PapersOnLine, 49(25), 401–406.

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput.,1(1), 11–33.

Barr, M. (1998). Programming Embedded Systems in C and C++. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1st edition.

Burns, A., Hayes, N., & Richardson, M. (1995). Generating feasible cyclic schedules. Control Engineering Practice, 3(2), 151–162.

Cardoso, D. N., Esteban, S., & Raffo, G. V. (2019). A nonlinear w∞ controller of a tilt-rotor uav for trajectory tracking. In 2019 18th European Control Conference (ECC) (pp. 928–934).

Chao, H., Cao, Y., & Chen, Y. (2010). Autopilots for small unmanned aerial vehicles: A survey. International Journal of Control, Automation and Systems, 8(1), 36–44.

Cheon, S., Ha, S., & Moon, T. (2016). Hardware-the-loop simulation platform for image-based object tracking method using small UAV. Digital Avionics Systems Conference, IEEE/AIAA 35th.

Czerniejewski, A., Dantu, K., & Ziarek, L. (2018). juav: A real-time java uav autopilot. In 2018 Second IEEE International Conference on Robotic Computing (IRC) (pp. 258–261).

d. Santos, F. F., Draghetti, L., Weigel, L., Carro, L., Navaux, P., & Rech, P. (2017). Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W) (pp. 169–176).

da Silva, G. M. (2017). Sistema de gerenciamento de energia fotovoltaica aplicado a um veículo aéreo não-tripulado.

del Pino, J. C. (2016). Diseño y Desarrollo Preliminar de una Plataforma UAV Tilt-Rotor para Misionesde Búsqueda y Rescate.(Proyecto EMERGENTIA).

Dhurjaty, S. & Atre, A. (2016). A hand-cranked, affordable defibrillator for resource-poor settings. In 2016 IEEE Global Humanitarian Technology Conference (GHTC) (pp. 542–546).

Donadel, R. et al. (2015). Modeling and control of a tiltrotor unmanned aerial vehicle for path tracking.

Dubrova, E. (2013). Fault-Tolerant Design. Springer Publishing Company, Incorporated.

Ellingson, G. & McLain, T. (2017). Rosplane: Fixed-wing autopilot for education and research. In 2017 International Conference on Unmanned Aircraft Systems (ICUAS) (pp. 1503–1507).

Fletcher, J. (1982). An arithmetic checksum for serial transmissions. IEEE Transactions on Communications, 30(1), 247–252.

Frenzel, L. E. (2015). Handbook of serial communications interfaces: a comprehensive compendium of serial digital input/output (I/O) standards. Newnes.

Fulton, R. & Vandermolen, R. (2017). Airborne Electronic Hardware Design Assurance: A Practitioner’s Guide to RTCA/DO-254. CRC Press.

Gans, N., Dixon, W., Lind, R., & Kurdila, A. (2005). A hardware-in the-loop simulation platform for vision-based control of unmanned air vehicles. Mechatronics.

Gaska, T., Watkin, C., & Chen, Y. (2015). Integrated modular avionics - past, present, and future. IEEE Aerospace and Electronic Systems Magazine, 30(9), 12–23.

Gmytrasiewicz, P. J. & Durfee, E. H. (1992). Decision-theoretic recursive modeling and the coordinated attack problem. In Proceedings of the First International Conference on Artificial Intelligence Planning Systems (pp. 88–95). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Grigulo, J. & Becker, L. B. (2018). Experimenting sensor nodes localization in wsn with uav acting as mobile agent. In 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), volume 1 (pp. 808–815).

Hitt, E. F. (2006). Avionics Development and Implementation, Chapter 5: Fault Tolerant Avionics. CRC Press, 1st edition.

Janson, K., Treudler, C. J., Hollstein, T., Raik, J., Jenihhin, M., & Fey, G. (2018). Software-level tmr approach for on-board data processing in space applications. In 2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS) (pp. 147–152).

Kahe, G. (2018). Triple-triple redundant reliable onboard computer based on multicore microcontrollers. International Journal of Reliability, Risk and Safety: Theory and Application, 1(1), 7–15.

Kato, S. & Yamasaki, N. (2009). Semi-partitioned fixed-priority scheduling on multi-processors. In 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium (pp. 23–32).

Knight, J. C. (2002). Safety critical systems: challenges and directions. In Proceedings of the 24th International Conference on Software Engineering. ICSE 2002 (pp. 547–550).

Lakos, J. (1996). Large-scale C++ Software Design. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Lala, J. H. & Harper, R. E. (1994). Architectural principles for safety-critical real-time applications. Proceedings of the IEEE, 82(1), 25–40.

LaMeres, B. J., Harkness, S., Handley, M., Moholt, P., Julien, C., Kaiser, T., Klumpar, D., Mashburn, K., Springer, L., & Crum, G. A. (2015). Radsat-radiation tolerant smallsat computer system.

Lamport, L., Shostak, R., & Pease, M. (1982). The byzantine generals problem. ACM Trans. Program. Lang. Syst., 4(3), 382–401.

Lara, A. V., Nascimento, I. B., Arias-Garcia, J., Becker, L. B., & Raffo, G. V. (2018). Hardware-in-the-loop simulation environment for testing of tilt-rotor uav's control strategies. XXII Congresso Brasileiro de Automática.

Lara, A. V., Rego, B. S., Raffo, G. V., & Arias-Garcia, J. (2017). Desenvolvimento de um ambiente de simulação de vants tilt-rotor para testes de estratégias de controle. Proc. of the XII Simpósio Brasileiro de Automação Inteligente, (pp. 2135–2141).

Liu, C. L. & Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM, 20(1), 46–61.

Liu, J. W. S. W. (2000). Real-Time Systems. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st edition.

Louali, R., Gacem, H., Elouardi, A., & Bouaziz, S. (2017). Implementation of an uav guidance, navigation and control system based on the can data bus: Validation using a hardware in the loop simulation. In 2017 IEEE International Conference on Advanced Intelligent Mechatronics (AIM) (pp. 1418–1423).

Meier, L., Honegger, D., & Pollefeys, M. (2015). Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In 2015 IEEE International Conference on Robotics and Automation (ICRA) (pp. 6235–6240).

Milluzzi, A., George, A., & George, A. (2017). Exploration of tmr fault masking with persistent threads on tegra gpu socs. In 2017 IEEE Aerospace Conference (pp. 1–7).

Miranda, G. M. T. (2017). Multi-core model predictive control strategy for a tilt-rotor uav in system-in-the-loop simulation.

Mozafari, S. H. & Meyer, B. H. (2015). Hot spare components for performance-cost improvement in multi-core simt. In 2015 IEEE International Symposium on Defect and

Fault Tolerance in VLSI and Nanotechnology Systems (DFTS) (pp. 53–59).

Murphy, N. & Barr, M. (2001). Watchdog timers. Embedded Systems Programming, 14(11),

79–80.

Nemati, F. (2010). Partitioned scheduling of real-time tasks on multi-core platforms.

Oliveira, D. A. G., Rech, P., Quinn, H. M., Fairbanks, T. D., Monroe, L., Michalak, S. E., Anderson-Cook, C., Navaux, P. O. A., & Carro, L. (2014). Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison. IEEE Transactions on Nuclear Science, 61(6), 3115–3122.

Oyetoke, O. (2015). Embedded systems engineering, the future of our technology world; a look into the design of optimized energy metering devices. International Journal of Recent Engineering Science (IJRES).

Pilla, L. L., Rech, P., Silvestri, F., Frost, C., Navaux, P. O. A., Reorda, M. S., & Carro, L. (2014). Software-based hardening strategies for neutron sensitive fft algorithms on gpus. IEEE Transactions on Nuclear Science, 61(4), 1874–1880.

Rahme, J. & Xu, H. (2017). Dependable and reliable cloud-based systems using multiple software spare components. In 2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (Smart- World/SCALCOM/UIC/ATC/CBDCom/IOP/SCI) (pp. 1–8).

Rego, B. S. & Raffo, G. V. (2016). Path tracking control based on guaranteed state estimation for a Tilt-rotor UAV. XXI Congresso Brasileiro de Automática.

Saidi, S., Ernst, R., Uhrig, S., Theiling, H., & de Dinechin, B. D. (2015). The shift to multicores in real-time and safety-critical systems. In Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis, CODES '15 (pp. 220–229). Piscataway, NJ, USA: IEEE Press.

Salehi, M., Ejlali, A., & Al-Hashimi, B. M. (2016). Two-phase low-energy n-modular redundancy for hard real-time multi-core systems. IEEE Transactions on Parallel and Distributed Systems, 27(5), 1497–1510.

Sari, B. & Reuss, H.-C. (2018). Fail-Operational Safety Architecture for ADAS Systems Considering Domain ECUs. Technical report, SAE Technical Paper.

Stankovic, J. A. (1988). Misconceptions about real-time computing: a serious problem for next-generation systems. Computer, 21(10), 10–19.

Trilaksono, B. R., Triadhitama, R., Adiprawita, W., & Wibowo, A. (2011). Hardware-in-the-loop simulation for visual target tracking of octorotor UAV. Aircraft Engineering and Aerospace Technology: An International Journal.

Vivekanandan, P., Garcia, G., Yun, H., & Keshmiri, S. (2016). A simplex architecture for intelligent and safe unmanned aerial vehicles. In 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (pp. 69–75).

Walls, C. (2012). Embedded software: the works. Elsevier.

Wang Sujing, Wang Lide, Shen Ping, Liu Biao, & Nie Xiaobo (2008). Research on electronically controlled fuel injection system. In 2008 IEEE Vehicle Power and Propulsion Conference (pp. 1–5).

Watkins, C. B. & Walter, R. (2007). Transitioning from federated avionics architectures to integrated modular avionics. In 2007 IEEE/AIAA 26th Digital Avionics Systems Conference (pp. 2.A.1–1–2.A.1–10).

Zhaolin Yang, Feng Lin, & Chen, B. M. (2016). Survey of autopilot for multi-rotor unmanned aerial vehicles. In IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society (pp. 6122–6127).

Appendix A: Interviews

This appendix describes all the information obtained in the first round of a series of interviews conducted with members of the PROVANT project in order to ascertain the key characteristics desired for the ProVANT 4.0. This text is part of the development of the master's thesis of the present author and aims to organize the hardware and software requirements necessary to meet the expectations of the interviewees. In all, we interviewed 6 Ph.D. or Master's students, each responsible for a different research topic involving ProVANT 4.0. In addition, 12 general questions were asked. The following sections describe the answers to each question.

A.1 Questions and Answers

A.1.1 Give a brief description of your work and how it will depend on the hardware and software architecture of UAV 4.0.?

Interviewee 1: "I work with robust nonlinear fault-tolerant control and fault detection and isolation strategies based on set-theoretic methods, with application to the UAV case. Such strategies are computationally costly, involving the online solution of nonlinear optimization problems with several constraints. Hardware/software supporting parallelization is very important. Reliable memory management and constant sampling time (input and output) are also required. Time delays (in communication and other things) should be minimal."

Interviewee 2: " My work is to develop linear and nonlinear robust controllers to provide path tracking and cope with the tilt-rotor UAV 4.0 entire flight envelope. The controllers must be embedded on the aircraft. Therefore, the hardware must provide the best performance as possible, in sense of execution speed. Besides ensure that commands sent to the actuators will be executed and data from the experiment will be saved for analysis. ”

Interviewee 3: “ My work is about some formulations of MPC, such as economic MPC, robust MPC, and stochastic MPC, as well as, the integration between them. ”

Interviewee 4: “ Applying verification techniques in the UAV design process. For this purpose, we are using Model Checking, and static and automatic method, complemented with Runtime Verification (RV), a dynamic method. In order to apply RV, we depend on the software system, because this technique will check the current execution against the desired properties that generates a monitor which is instrumented into the code. The hardware will be used in a HIL simulation to test and validate the work. ”

Interviewee 5: “ I work with Model Predictive Controllers, which are a optimal control approach to control systems. Solving the optimal control problem to control a tilt-rotor is very demanding computationally, so the hardware should allow parallelization throw a GPU, a multi-core processor, or a FPGA. As far as the software architecture is concerned, the control algorithm is quite complex, so I will need to program with object oriented programming in C++ (more computationally efficient than java and alternatives) and a way to control the timing to send the input in the correct sample times. ”

Interviewee 6: “ My work focuses in experiment and validation of localization methods for wireless sensor networks, the main idea is to localize the sensors using only beacon messages and RSSI from WiFi ESP8266 radios. The mobile node, which is the one provided with a low cost GNSS RTK system, broadcast its position while traversing a WSN, the beacons transmitted are stored in the nodes on the field and, with a post processed algorithm, it is able to estimate its own position without the need of a GNSS embedded in the static sensor itself. For a more specific application, such as in agriculture or monitoring of remote areas (islands, volcanoes and harsh environments), it is required an UAV capable of covering big areas, so the UAV 4.0 would be great for the application of a future work in this topic. ”

A.1.2 What are the basic behaviors and required functionalities of the ProVANT 4.0?

Interviewee 1: “The ProVANT 4.0 should be able to perform teleoperated flights, and autonomous hover and forward flights (tracking a desired trajectory). Load transportation tasks must be also taken into account (load suspended by a cable in hover flight, but retracted in forward flights). ”

Interviewee 2: “Besides the common features, it must provide an environment to make easy implementation of controllers (Similar to the ProVANT simulator) and a way to "debug" and test the controllers before performing real tests. Furthermore, the wire connections must ensure that "bad contacts" will not happen during flight.”

Interviewee 3: “Its basics abilities are performing hover and forward flights with or without suspended load. Moreover, it should be able to perform tasks within kilometers range using global positioning systems and vision systems to deal with environment peculiarities.”

Interviewee 4: “The UAV will automatically measure wind speed and display the speed on the ground station. The ground station will automatically estimate the time to return, taking into account the wind and display this information to the operator. The ground station will automatically warn the operator if the fuel/battery is not sufficient to return.”

Interviewee 5: “It is need a way to read sensors, send input to actuators and send/receive data via radio for telemetry.”

Interviewee 6: “RTK GNSS for waypoint or autonomous navigation, communication with sensor nodes and telemetry. VTOL capability.”

A.1.3 What is the hardware performance needed to your work be carried out?

Interviewee 1: “The control and state estimation strategies involved are very computationally demanding. They require the online solution of nonlinear optimization problems with several constraints, and the execution of several nonlinear state estimators in parallel.”

Interviewee 2: “The sensors must report correct measurements, with minimum noise as possible. The actuators must perform the commands correctly. The controller’s loop time must be smaller than 10 milliseconds.”

Interviewee 3: “It needs to have high computational power since most of the controllers of my work are costly. It would be important to have the possibility of doing GPU coding.”

Interviewee 4: “We need a high performance of the hardware because we intend to use a HIL Simulation to test and validate our work to raise the level of reliability in the system.”

Interviewee 5: ‘I don’t have a way to specify exact required performance yet, but as I answered in question 2, the algorithm is very computationally expensive. The communication should also being able to send all estimated states and controls in a sample time, either via radio, or via serial communication.”

Interviewee 6: “For the RTK algorithm: any system capable of running LINUX OS with at least 500Mhz processor and 100MB of free memory will be OK. For the sensor nodes: ESP8266 radio + cpu modules.”

A.1.4 How many and which algorithms need to be executed and stored simultaneously in the main memory during the UAV 4.0 operation?

Interviewee 1: “A control algorithm based on solving a constrained nonlinear optimization problem (a nonlinear MPC, for instance), a fault diagnostic algorithm (probably also based on optimization), and a bank of nonlinear set-based state estimators running in parallel (the number of state estimators varies from one case to another).”

Interviewee 2: “It is not possible to define the exact number of algorithms. But, it is necessary to execute the controller, the filtering algorithm (To improve measurements), the algorithm that detects error and instability (to execute some action during instability and system's fault). Besides save data for analysis."

Interviewee 3: "Basically, two threads need to be made, one for the controller itself and the other for obstacle detection. However, through the course of my work, maybe it will be necessary to run some parts of the controllers in parallel or even in an embedded system."

Interviewee 4: "Control algorithms and algorithms that interface with the UAV subsystems."

Interviewee 5: "The algorithm involves a data processor for LIDAR raw output, which includes 3 computational geometry algorithms(small and not demanding) and a processing part to convert this data to structured constraints. It also involves a optimization part, which includes a NLP solver(IPOPT like) a HP-adaptive pseudospectral transcription algorithm."

Interviewee 6: "Algorithms: RTK algorithm, waypoint navigation algorithm, wireless sensor communication"

A.1.5 How would be an ideal interface of Hardware/Software for your work to be done?

Interviewee 1: "The hardware/software interface should be as reliable as possible, regarding memory management, task scheduling, multi-core processing, and sensor/actuator data management (send and receive data to/from sensors and actuators)"

Interviewee 2: "An interface similar to the ProVANT simulator."

Interviewee 3: "Simple, with a clear and well-described framework. Moreover, it should be modular in a sense that all of the users' efforts can be directed to their own work without having to worry about how the whole Hardware/Software works."

Interviewee 4: "I have not yet identified an ideal interface for my work."

Interviewee 5: “One that reads my mind and program the embedded hardware by itself”

Interviewee 6: “Any Linux OS Single Board Computer with, at least, three UARTs. WiFi ESP8266 radios as sensor nodes and Autopilot for waypoint navigation (GNSS)”

A.1.6 What would be a good way to test and debug your work?

Interviewee 1: “The best way do debug these strategies is to be able to simulate sensor and actuator failures online. By this way, the fault-tolerant control and fault detection and isolation strategies can be effectively tested, anticipating all the faulty situations for which these strategies are specially designed, prior to the occurrence of real component malfunctions.”

Interviewee 2: “Using "Hardware in the loop" with a possibility to print partial results in some interface.”

Interviewee 3: “It would be interesting to have some interface allowing to see some selected signals in a graphic fashion. Also, to have some methods to export those signals.”

Interviewee 4: “A good way to test my work is using HIL Simulation.”

Interviewee 5: “Any way that from my PC and program embedded hardware through a USB or other convenient way.”

Interviewee 6: “Test: practical tests in laboratory and on the field. Debug: log files and debug messages in real time of operation.”

A.1.7 How would you like to obtain flight data?

Interviewee 1: “Flight data should be stored in the embedded system (I don?t know exactly how, maybe a flash memory), or sent to the ground station. The desired trajectory, performed trajectory (all the system states), control signals, sensor measurement (and frequency), input and output data of state estimators, and also the output from the fault diagnoser should be stored.”

Interviewee 2: “During a flight, some data should be sent from the UAV to a ground station. Moreover, after the flight, it must provide the possibility to download the whole data from the UAV’s memory.”

Interviewee 3: “Graphically, in a ground station for example, and in output files that can be imported easily to others software.”

Interviewee 4: “In a web page, where we can obtain flight data wherever we are.”

Interviewee 5: “Getting Telemetry data, using a compiler debugger, memory analyzer software, a runtime analyzer and using a profiler can be very helpful.”

Interviewee 6: “Long Range telemetry link between UAV and PC/tablet.”

A.1.8 What input and output data are required to your application run properly?

Interviewee 1: “The fault-tolerant control strategies require information on the control signals, sensor measurement (and frequency), and desired trajectory (all desired states).”

Interviewee 2: “Input: attitude (In Euler angles), position (x,y, and z, w.r.t. an inertial ground station), servomotors’ angle and angle of deflection of aerodynamic surfaces. Output: Servomotors’ torque, thrusters’ voltage and aerodynamic surfaces angle of deflection.”

Interviewee 3: “All states (orientation, position, servomotors’ angles, load’s angles), control input (thrusts, torques, control surfaces angles), vision sensing (camera information), and battery-related variables (current, tension, etc).”

Interviewee 4: “A input is all of data that control algorithm will need to run. The output is the event trace that this algorithm generates.”

Interviewee 5: “No preference, though, telemetry is usually sent through some radio device, as xBee for example.”

Interviewee 6: “input: GNSS rover (UAV) raw data, GNSS BASE station raw data (NTRIP or local), IMU measurements. Output: RTK GNSS solution to UAV autopilot, position from RTK to sensor nodes, telemetry messages.”

A.1.9 What about the frequency of these data?

Interviewee 1: “Output data should be sent in the lowest actuator frequency (12 ms), and input data should be received in multiples of it (in the case of sensor measurement with lower frequency, i.e., it can be 12 ms, 24 ms, 36 ms, etc).”

Interviewee 2: “The only constraint of frequency in the sample time. The smallest sample time is probably the necessary to control the actuators, probably around 10-20 ms.”

Interviewee 3: “Around 85 Hz or bigger (12 ms or less).”

Interviewee 4: “It depends on the period of the threads.”

Interviewee 5: “No preference, though, telemetry is usually sent through some radio device, as xBee for example.”

Interviewee 6:

“GNSS rover (UAV) raw data: 5Hz; GNSS BASE station raw data (NTRIP or local): 1Hz; RTK GNSS solution to UAV autopilot:5Hz; position from RTK to sensor nodes: 5Hz; telemetry messages: 57600bps”

A.1.10 For a proper documentation, what should be included in the software and hardware manual?

Interviewee 1: “Every technical detail regarding the project of the software and hardware architecture should be documented. A hard requirement is that the next developer should be able to continue the work only by reading the manual (despite basic technical and theoretical knowledge required, of course. However, these should be briefly explained, or referenced to another material for the ProVANT reader).”

Interviewee 2: “Everything must be detailed, using the most number of figures as possible. Each step of the code must be commented.”

Interviewee 3: “The whole hardware and software framework description and a brief explanation about the methods (or functions in case of procedural languages) as well as its interfaces. Moreover, a simple getting start tutorial for users that don’t need to get detailed knowledge of the application.”

Interviewee 4: “Functional and non-functional requirements. high and low level features of each components.”

Interviewee 5: “Hardware and software architecture description, API descriptions, how the software and hardware is intended to be used, a high level description on how to program the hardware.”

Interviewee 6: “Software: Commented code, manual for developers, manual for user guide; Hardware: Manual for assemble parts (with mechanical parts catalog)”

A.1.11 If a UAV get into unrecoverable failure mode, what kind of countermeasures should be done?

Interviewee 1: “In the worst case, the system should be able to switch to a control strategy capable of perform safe vertical landing. An alert signal should be sent to the ground station.”

Interviewee 2: “The system must switch the implemented controller to a nonlinear one that drives the UAV to land somewhere. The "backup" controller must control just the UAV’s altitude and attitude, being as simple as possible.”

Interviewee 3: “It should have a redundancy controller running during all flight and able to safely put the vehicle on hover and then land in a safe place. If the vehicle gets in failure mode, this controller must be activated.”

Interviewee 4: “The UAV should enter safe mode and use its resources to identify the area where it is flying to take the possible measures, from emergency landing to turning off the engines if there are no risks in its environment.”

Interviewee 5: “Some sort of safe landing, if possible. "Unrecoverable failure mode" is very general, it is hard to say what this really means.”

Interviewee 6: “Turn on failsafe mode in order to taking it back to home position and take control of the drone; if not possible land where it is; Instantly power off if reachable; if not reachable try to find it through the telemetry GNSS messages (on the map).”

A.1.12 Which ones of the UAV 4.0 failures do you consider tolerable?

Interviewee 1: “Sensor failures (wrong measurement, lack of measurement), unusual communication delays, actuator failures (loss of potency), and maybe the complete loss of a propeller (in this case, I think it is reasonable to not require trajectory tracking anymore, but only stabilization. Maybe still trajectory tracking if in forward flight mode, due to the aerodynamic surfaces).”

Interviewee 2: “fault from data sent from the UAV to the ground station.”

Interviewee 3: “Lost of the control surfaces (it could still perform hover) and lost of some sensor information (as long as it still possible to estimate them).”

Interviewee 4: “Data latency and loss (not much time) Delay receiving commands.”

Interviewee 5: “Occasional faults on sensors can be tolerated, if not frequent.”

Interviewee 6: “Loss of signal with RC; Low battery; Magnet sensor error; GPS error; Engine failure.”

Appendix B: Nucleo boards's firmware

B.1 IDE

Several IDE (Integrated Development Environment) can be used to design code for a STM microcontroller. For this work it was used the System Workbench for STM32 and free IDE available in https://www.openstm32.org/. This is an IDE based on Eclipse and to download it, a register is required. For a complete manual of these application access https://www.eclipse.org/documentation/ and choose the manual according to the eclipses version of the IDE downloaded.

B.2 StmCubeMX

According to STMicroelectronics[13], "STMCubeMX is a graphical tool that allows an easy configuration of STM32 microcontrollers and microprocessors through a step-by-step process.". One can download it, using the link https://www.st.com/en/development-tools/stm32cubemx.html. A complete manual can be found in the same link of the download.This software set the project up even with code required for use of FreeRTOS in the application

B.3 Project Setup

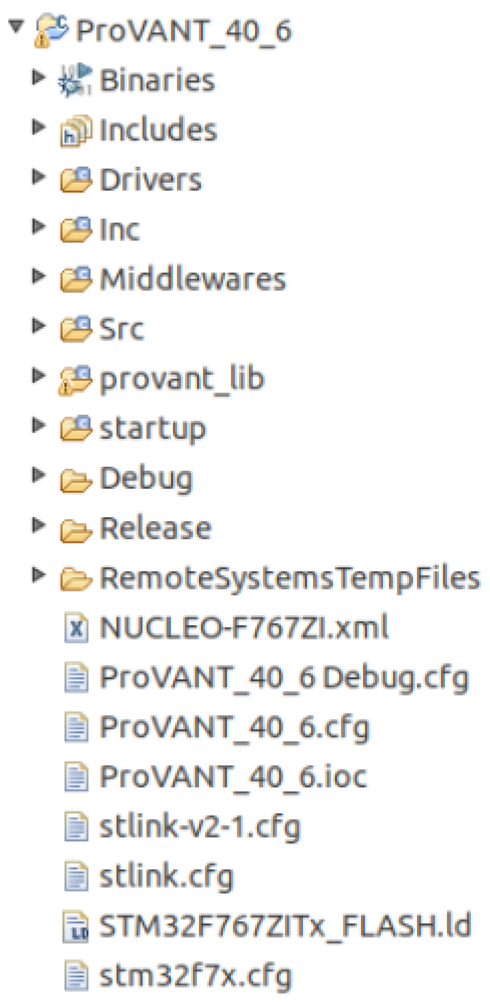
The source code of the LHL project can be found in the github repository https://github.com/Guiraffo/provant-software. The Code available is set up just for HIL

[13] www.st.com

simulation. In order to implement codes for communication with instrumentation, the respective driver settings required must be configured with the StmCubeMX.

The project has the following organization:

Figure 47: Project organization.



```
▼ ProVANT_40_6
  ▶ Binaries
  ▶ Includes
  ▶ Drivers
  ▶ Inc
  ▶ Middlewares
  ▶ Src
  ▶ provant_lib
  ▶ startup
  ▶ Debug
  ▶ Release
  ▶ RemoteSystemsTempFiles
    NUCLEO-F767ZI.xml
    ProVANT_40_6 Debug.cfg
    ProVANT_40_6.cfg
    ProVANT_40_6.ioc
    stlink-v2-1.cfg
    stlink.cfg
    STM32F767ZITx_FLASH.ld
    stm32f7x.cfg
```

Source: The Author

With the exception of the provant_lib folder, all organization is defined with default by StmCubeMX. The content of each folder is:

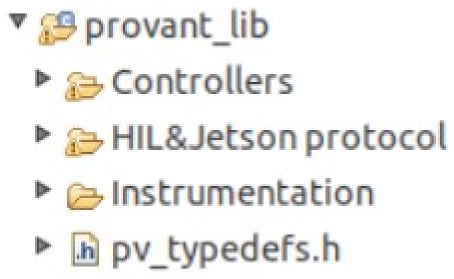
• **Driver:** Hardware Abstraction Layer and CMSIS code;

• **Inc:** Some headers related to the main function and some features related to interruption, timers and system definition;

- **Middlewares:** FreRTOS source code;
- **Src:** Implementation of the main function and some features related to interruption, timers and system definition;
- **provant_lib:** folder with custom code designed by the application, for example the implementation of the simple control laws used to provide fault tolerance for HLH failures;
- **Debug:** Files generated by the compilation in debug mode;
- **Release:** Files generated by the compilation in release mode.

This provant_lib folder has the following organization:

Figure 48: provant_lib folder's organization



Source: The Author

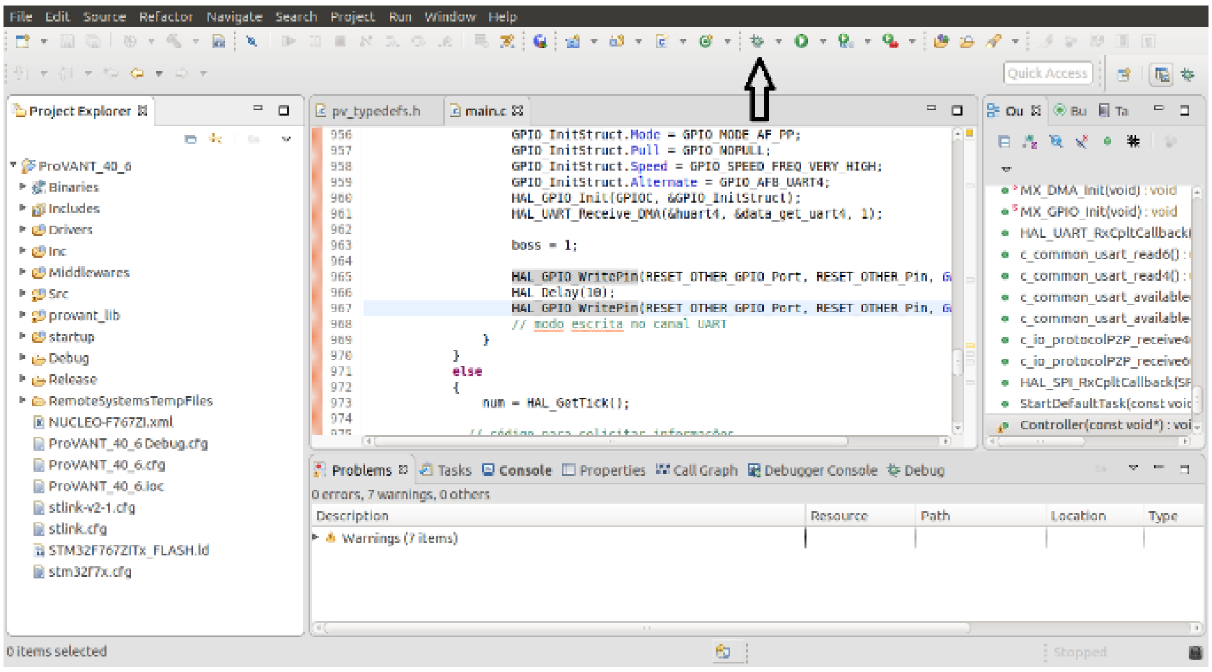
- **Controllers** folder with implementation of control laws;
- **HIL&Jetson protocol** is the place where the source code of communication protocol used with the HIL and the Jetson;
- **Instrumentation** is the place where the custom code to be design in order to communicate with the sensors and actuators.
- **pv_typedefs.h** is a file with some definition of structures used to the input and output data. It is reused from the previous work on ProVANT 1.0 and ProVANT 2.0 development.

B.4 Compilation and upload

For the first compilation and upload of a project, after it is created, click with the of right button on the project name, go to “Debug As” and then, click with the left button on “Ac6 STM32 C/C++ Application”.

From now on the compilation and upload are performed. Click with left button on icon demonstrated in Figure 49.

Figure 49: How to compile the project



ProVANT_40_6
Binaries
Includes
Drivers
Inc
Middlewares
Src
provant_lib
startup
Debug
Release
RemoteSystemsTempFiles
NUCLEO-F767ZI.xml
ProVANT_40_6 Debug.cfg
ProVANT_40_6.cfg
ProVANT_40_6.ioc
stlink-v2-1.cfg
stlink.cfg
STM32F767ZITx_FLASH.ld
stm32f7x.cfg

```
                GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
                GPIO_InitStruct.Pull = GPIO_NOPULL;
                GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
                GPIO_InitStruct.Alternate = GPIO_AF8_UART4;
                HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
                HAL_UART_Receive_DMA(&huart4, &data_get_uart4, 1);

                boss = 1;

                HAL_GPIO_WritePin(RESET_OTHER_GPIO_Port, RESET_OTHER_Pin, G
                HAL_Delay(10);
                HAL_GPIO_WritePin(RESET_OTHER_GPIO_Port, RESET_OTHER_Pin, G
                // modo escrita no canal UART
            }
        }
        else
        {
            num = HAL_GetTick();
```

MX_DMA_Init(void) : void
MX_GPIO_Init(void) : void
HAL_UART_RxCpltCallback
c_common_usart_read6() :
c_common_usart_read4() :
c_common_usart_available
c_common_usart_available
c_io_protocolP2P_receive4
c_io_protocolP2P_receive6
HAL_SPI_RxCpltCallback(SP
StartDefaultTask(const voi
Controller(const void*) : voi

0 errors, 7 warnings, 0 others

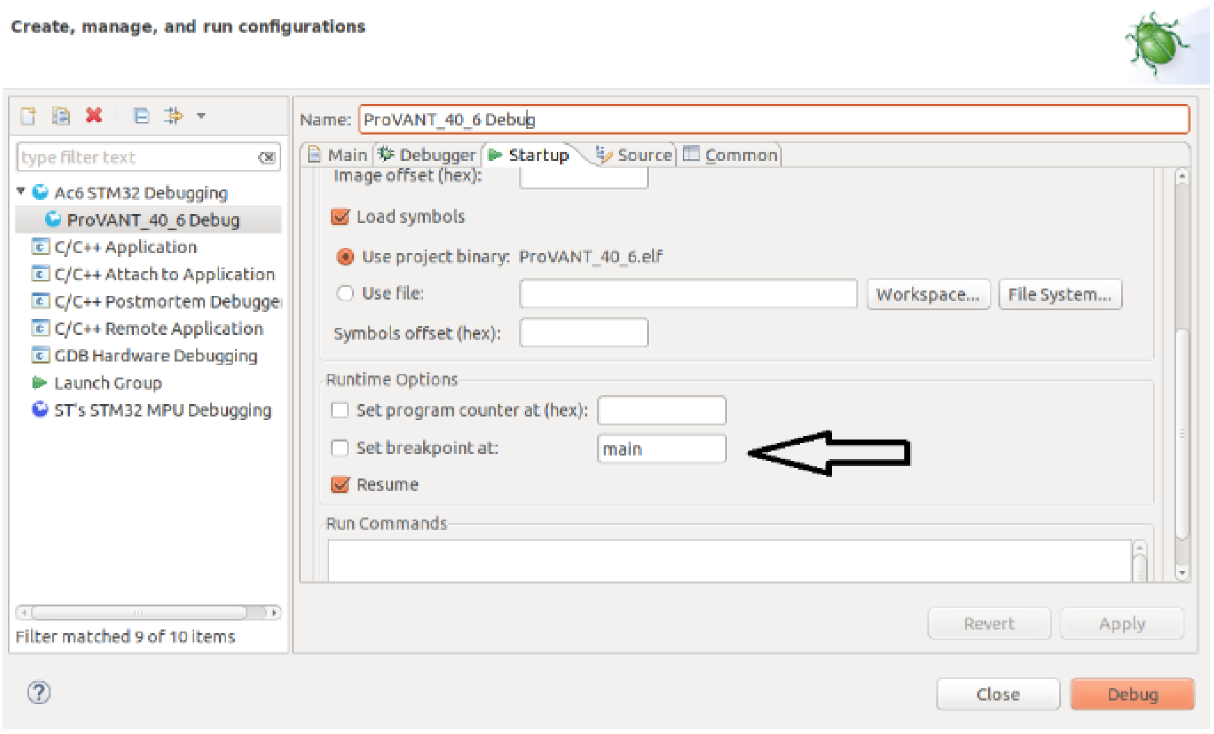
Description	Resource	Path	Location	Type
Warnings (7 items)				

0 items selected

Source: The Author.

Before uploading the code, verify if the field “set the breakpoint at” in the Startup of the Debug Configuration Window is unchecked, as shown in Figure 50. If it is checked, uncheck it.

Figure 50: Startup Settings


Create, manage, and run configurations

Name: ProVANT_40_6 Debug

Main | Debugger | Startup | Source | Common

Image offset (hex):

[✓] Load symbols

● Use project binary: ProVANT_40_6.elf

○ Use file:

Symbols offset (hex):

Runtime Options

[ ] Set program counter at (hex):

[ ] Set breakpoint at: main

[✓] Resume

Run Commands

Source: The Author

The upload of the code for both Nucleo Boards must be one at a time. After the process upload the code for both boards, reset them at the same time with the RESET button in the board. The first board that you release the button will be the main board and the last, the backup board.

Appendix C: Jetson's software

C.1 Installation of Ubuntu 18.04 and ROS 2

Jetson TX2 is a platform that has already an ubuntu 16.04 installed by default. But, in order to install the last version (Crystal) of ROS 2 until the date of this thesis, it is needed to install Ubuntu 18.04. For this, the designer should download the Nvidia SDK manager and run it.

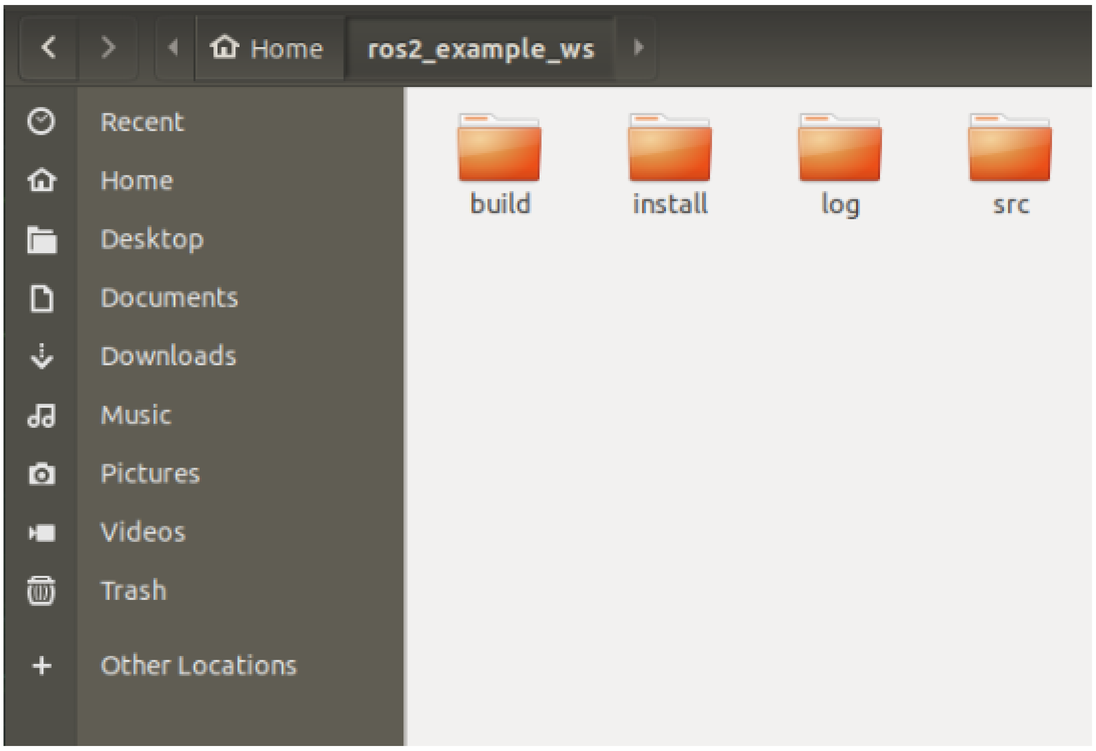
Official updated information on how to download and use the Nvidia SDK manager can be found in the following site https://docs.nvidia.com/sdk-manager/index.html and a video about how to install Ubuntu 18.04 is at the following link https://www.youtube.com/watch?time_continue=134&v=s1QDsa6SzuQ.

Besides, in order to install ROS 2, do the step by step of https://index.ros.org/doc/ros2/Installation/Crystal/Linux-Install-Binary/ in the Jetson TX2 after Ubuntu 18.04 be installed.

Lastly, create the workspace of ROS 2 in order to design new ROS packages. The step by step is in https://index.ros.org/doc/ros2/Tutorials/Colcon-Tutorial/

We make for these work a workspace named ros2_example_ws, but any name can be chosen. After it is well created the workspace will appear as the Figure 51.

Figure 51: ROS workspace organization

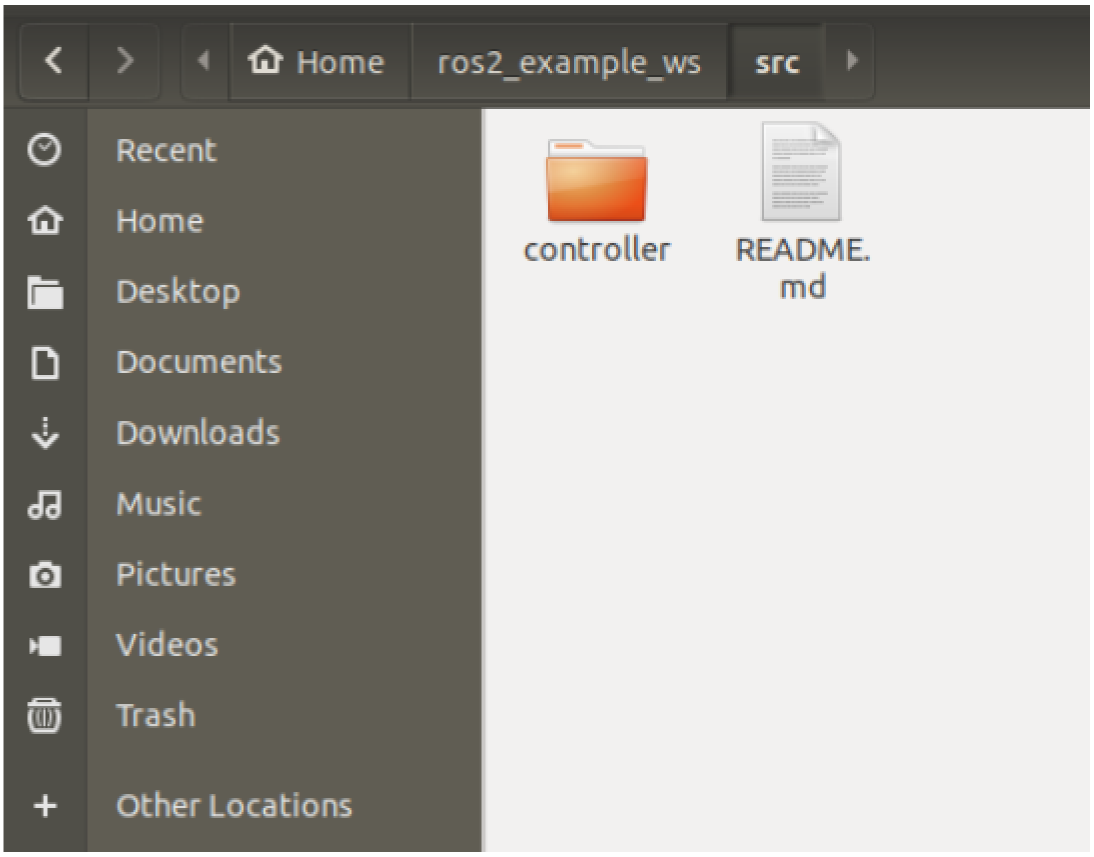


Source: The Author

C.2 Project Setup

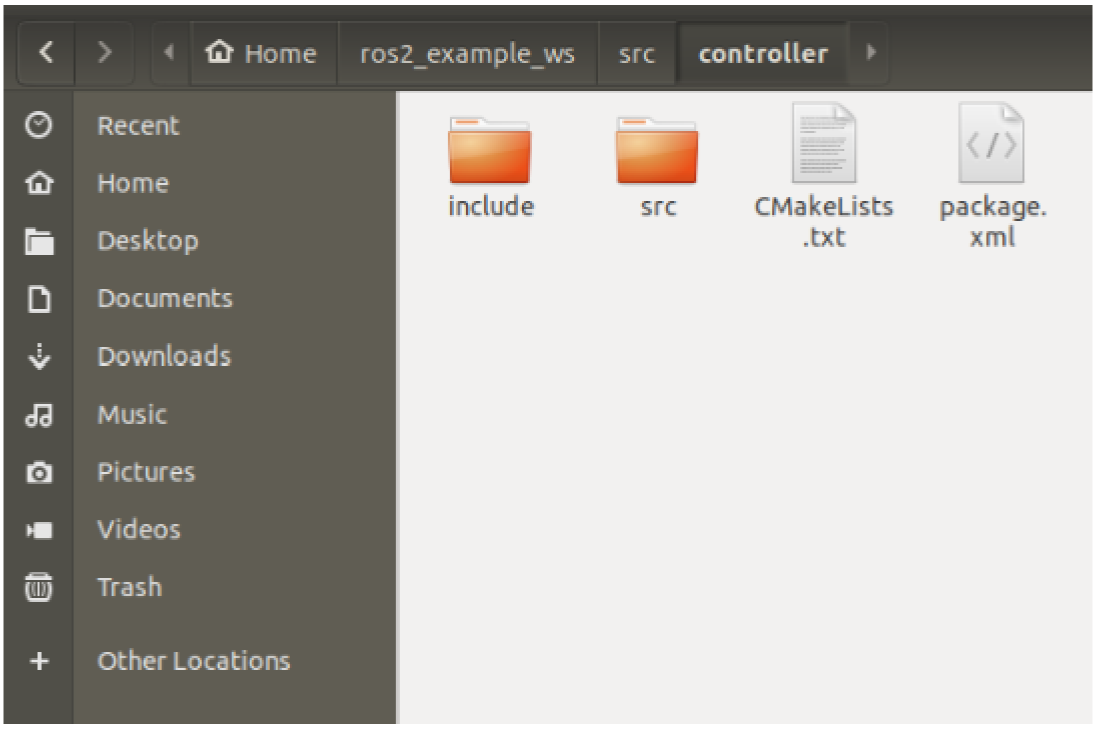
Clone the source code of the project in the github's repository https://github.com/Guiraffo/provant-software. The result will be as the Figure 52.

Figure 52: src folder content



Source: The Author

Figure 53: controller folder content

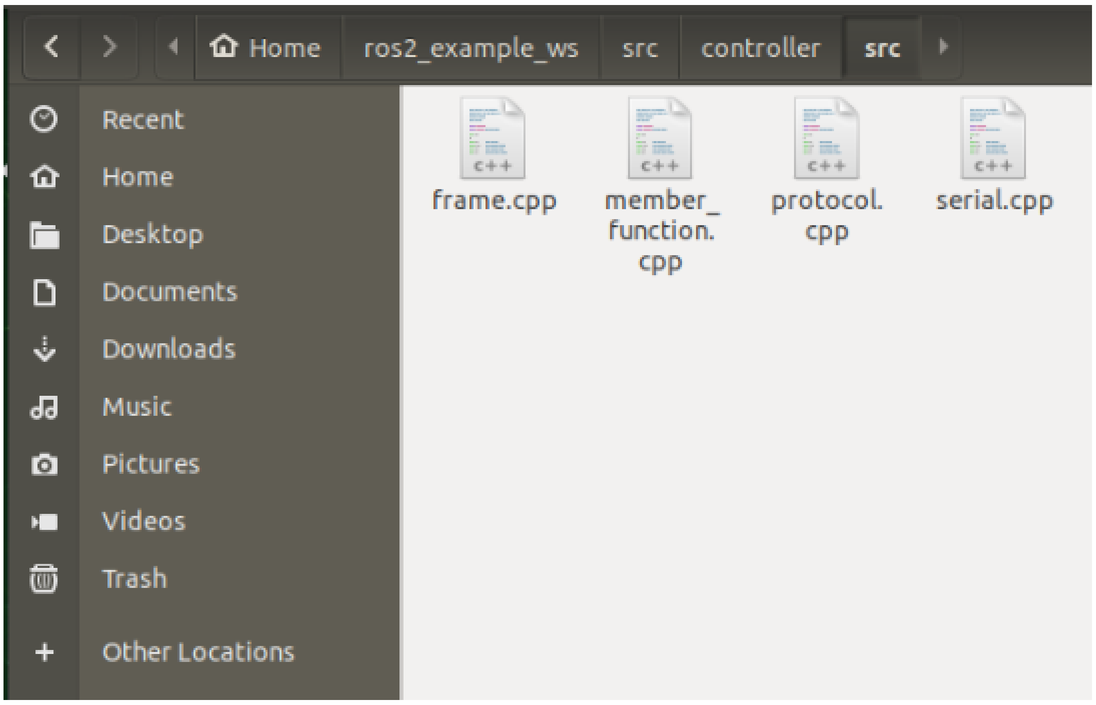


Source: The Author

The src folder is where cpp files can be found. They are the implementation of the structure of control that are shown in Figure 54. Frame.cpp consists the structure of the unit of data that is sent each time by the communication between LHL and HLH. protocol.cpp consists on the implementation of the communication protocol. serial.cpp consists on the abstraction of the drivers command by the protocol.cpp. Lastly, member_function.cpp is the implementation of the controller.

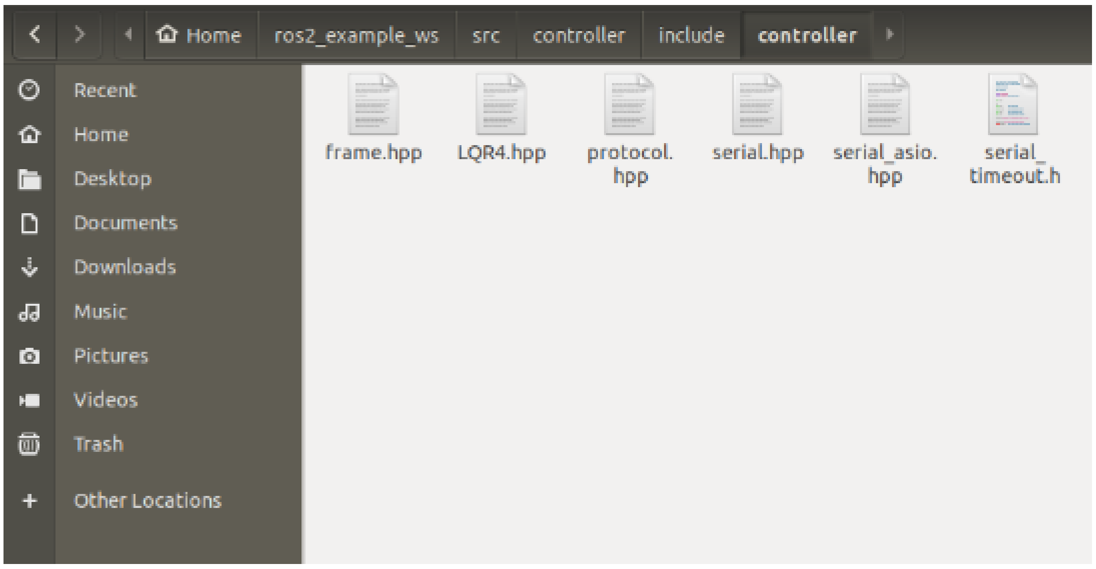
The include folder is where the headers files can be found. They are shown in Figure 55. They are the interface of the implementation that was described before. One important header file is the LQ4.hpp and it has an example control law. For each control law designed, another header that must be included in the project.

Figure 54: controller’s src folder content



Source: The Author

Figure 55: controller’s include folder content



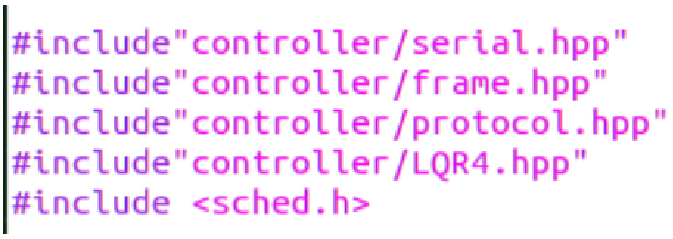
Source: The Author

C.3 Including new control law

As it was already said, for each control law a library must be included in the project. This header file consists in a class similar the one implemented in ProVANT Simulator. However, some configuration must be done in order to use this new control law. For more details of the ProVANT Simulator, read its user manual[14].

First of all, it must be included in the member_function.cpp the new control law as it is shown in Figure 56.

Figure 56: Headers of memeber_function.cpp

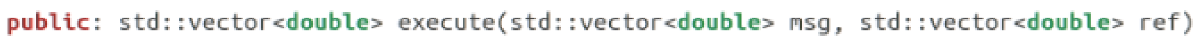


```
#include"controller/serial.hpp"
#include"controller/frame.hpp"
#include"controller/protocol.hpp"
#include"controller/LQR4.hpp"
#include <sched.h>
```

Source: The Author

Next, it must be changed the parameters of the execution method created to run in the ProVANT Simulator, as illustrated in Figure 57.

Figure 57: Declaration of execute() method



```
public: std::vector<double> execute(std::vector<double> msg, std::vector<double> ref)
```

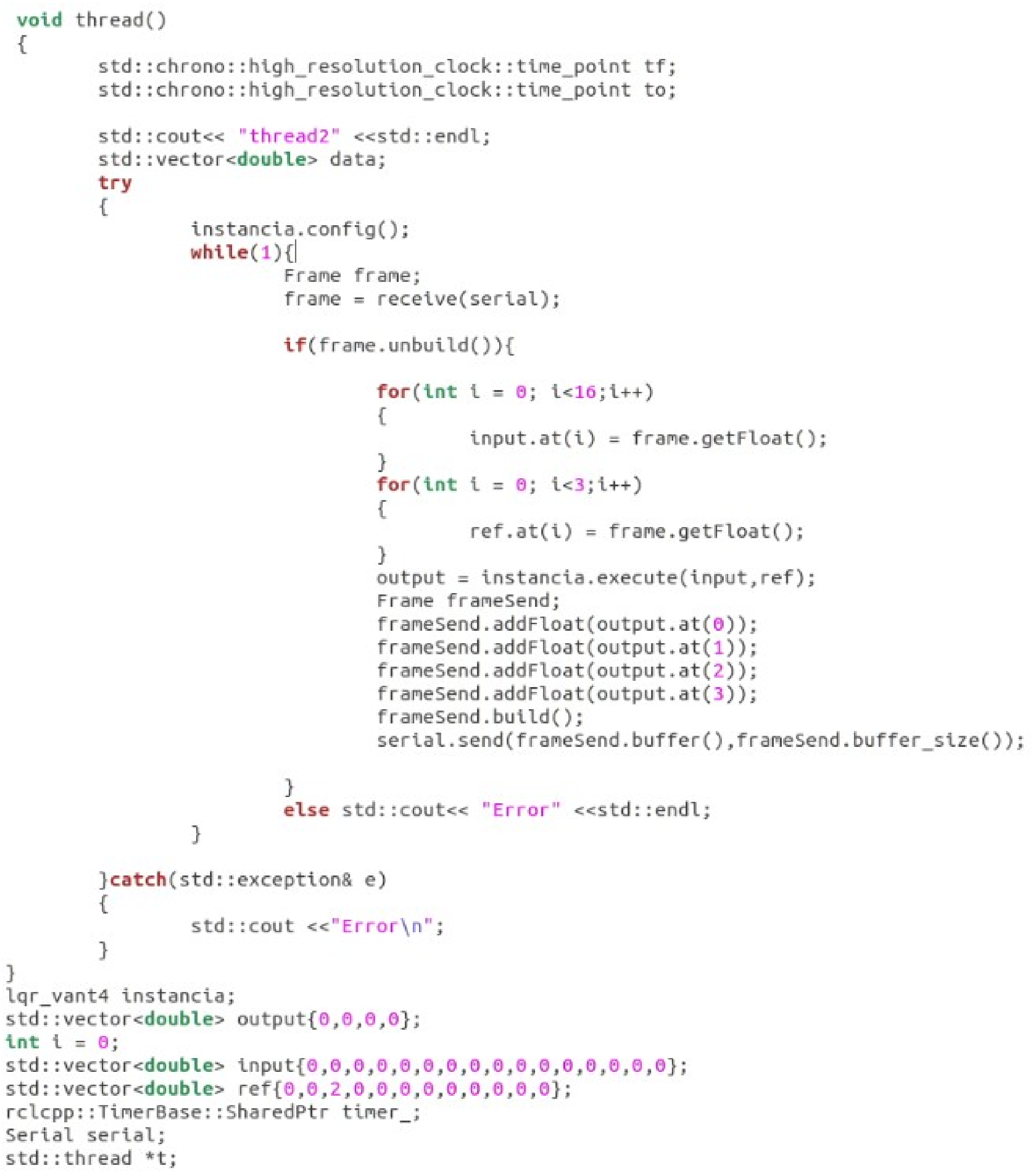
Source: The Author

Lastly, change the type of instance and the number and configuration of its parameters

[14] https://github.com/Guiraffo/ProVANT-Simulator/blob/master/doc/Manual.pdf

in the member_function.cpp, according with the data sent by the LHL according to shown in Figure 58.

Figure 58: Content of the code implemented in the execute method



```
void thread()
{
        std::chrono::high_resolution_clock::time_point tf;
        std::chrono::high_resolution_clock::time_point to;

        std::cout<< "thread2" <<std::endl;
        std::vector<double> data;
        try
        {
                instancia.config();
                while(1){
                        Frame frame;
                        frame = receive(serial);

                        if(frame.unbuild()){

                                for(int i = 0; i<16;i++)
                                {
                                        input.at(i) = frame.getFloat();
                                }
                                for(int i = 0; i<3;i++)
                                {
                                        ref.at(i) = frame.getFloat();
                                }
                                output = instancia.execute(input,ref);
                                Frame frameSend;
                                frameSend.addFloat(output.at(0));
                                frameSend.addFloat(output.at(1));
                                frameSend.addFloat(output.at(2));
                                frameSend.addFloat(output.at(3));
                                frameSend.build();
                                serial.send(frameSend.buffer(),frameSend.buffer_size());

                        }
                        else std::cout<< "Error" <<std::endl;
                }

        }catch(std::exception& e)
        {
                std::cout <<"Error\n";
        }
}
lqr_vant4 instancia;
std::vector<double> output{0,0,0,0};
int i = 0;
std::vector<double> input{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
std::vector<double> ref{0,0,2,0,0,0,0,0,0,0,0,0};
rclcpp::TimerBase::SharedPtr timer_;
Serial serial;
std::thread *t;
```

Source: The Author

C.4 Compilation and running

For computation of the project, execute in the terminal the following command wih the terminal in the workspace folder:

```
colcon build
```

For running the controller, execute the following commands:

```
sudo su
echo 1 > /sys/devices/system/cpu/cpu1/online
echo 1 > /sys/devices/system/cpu/cpu2/online
cd /usr/bin/
jetson_clocks
exit
setserial /dev/ttyTHS2 low_latency
ros2 run controller timer_member_function
```

Appendix D: Detailed electronic design

This appendix describe some details about the hardware architecture. First, it gives a table with all suggested hardware specifications. Next, it defines how the hardware will be connected in low level designed according to peripherals of LLH and HLH. Lastly, as there are two LLH in this architecture, it is discussed how to use Multiple master/client configuration on serial bus described before.

D.1 Hardware specification

Table 4: Hardware specification

Number	Device	Description
1	Jetson Tx2	HLH
2	Nucleo f767zi	LLH
1	Navio 2	2x IMU, 1x GPS and 1 Barometer
1	ADIS164890	IMU
1	3DR Pixhawk Airspeed Sensor Kit	Pitot tube
1	MB2530 IRXL-MaxSonar-CS3	Sonar
1	UBLOX NEO-M8T	GPS RTK module
2	Flat Maxon motor brushless EC 45 flat Ø42.8mm - 50 Watt	Brushless motor for fast servo-motor
2	Maxon controller ESCON 36/3	ESC for fast servo-motor
2	Maxon Sensor Encoder MILE, 512 CPT	Encoder for fast servo motor
4	Hitec D145SW Digital HV devices	Servo-motor
2	AXI 5345/14 HD 3D Extreme V2	brushless motor for propeller
2	Mezon 160 ESC	ESC for propeller
1	OrangeRx R1020X	Radio receiver
1	Device for communication with ground station	to be decided
1	Power system manager	custom device
1	Max3232	RS232/UART
2	KNACRO RS422 to TTL UART MCU Serial Port Signal Mutual Conversion Module with Over-Voltage Over-Current Protection-3.3V	RS422/UART

Source: The Author

D.2 Connection between hardwares

Table 5: Connection between hardwares

Device 1	peripheral	Device 2	Converter
Nucleo f767zi	PPM	OrangeRx R1020X	
Nucleo f767zi	USART2	Device for communication with ground station	
Nucleo f767zi	USART3	Maxon Sensor Encoder MILE, 512 CPT	RS422/UART
Nucleo f767zi	UART4	Maxon Sensor Encoder MILE, 512 CPT	RS422/UART
Nucleo f767zi	UART5	Jetson XT2	
Nucleo f767zi	PWM	Hitec D145SW Digital HV	
Nucleo f767zi	UART7	Power system manager	
Nucleo f767zi	UART8	MB2530 IRXL-MaxSonar-CS3	RS232/UART
Nucleo f767zi	PWM	Mezon 160 ESC	
Nucleo f767zi	PWM	Mezon 160 ESC	
Nucleo f767zi	I2C3	3DR Pixhawk Airspeed Sensor Kit	
Nucleo f767zi	I2C4	Navio2	
Nucleo f767zi	SPI1	Navio2	
Nucleo f767zi	PWM	Hitec D145SW Digital HV	
Nucleo f767zi	PWM	Hitec D145SW Digital HV	
Nucleo f767zi	PWM	Hitec D145SW Digital HV	
Nucleo f767zi	PWM	Maxon controller ESCON 36/3	
Nucleo f767zi	PWM	Maxon controller ESCON 36/3	
Jetson XT2	USB	UBLOX NEO-M8T	
Jetson XT2	SPI	ADIS164890	

Source: The Author

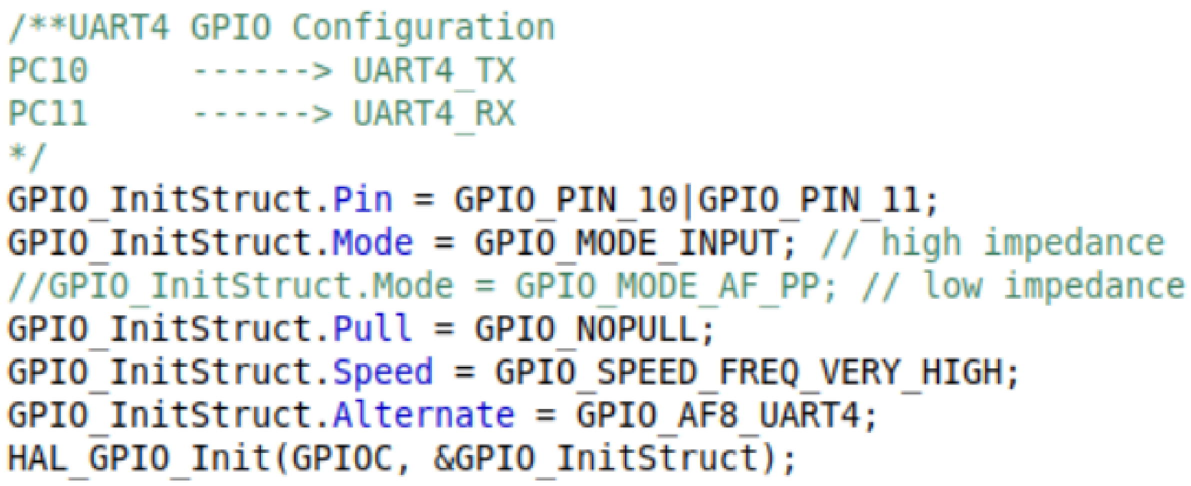
D.3 Multiple master/client configuration on a serial bus

UART, SPI and I2C are protocols that by default must have only one component that asks for services for the rest of components. Particularly, UART is a protocol of communication that allows communication of only two components. However, I2C and SPI have one component that can communicated with several others.

Nonetheless, the proposed solution demands two components for asking services instead of only one. As consequence of this demand, some different configurations should be done during the development of this work. Usually, the pins that write data are configured in low impedance and the pins that reads data are configured in high impedance, but we should avoid the connection of two pins in low impedance on the same bus. To solve this problem, we configure the main board as required by default and made a different configuration for the backup board, making all pins that can write data in high impedance. This configuration is exemplified in Figure 59, that present an example for UART and is located in the file named “stm32f7xx_hal_msp.c” inside the “src” folder.

Thus, after the detection of main LLH failure, the backup LLH resets the main board and change the configuration of the write data pins for low impedance as performed by default. This change of configuration is made in “main.cpp” file inside the “controller” thread.

Figure 59: Example of peripheral pin as high impedance mode

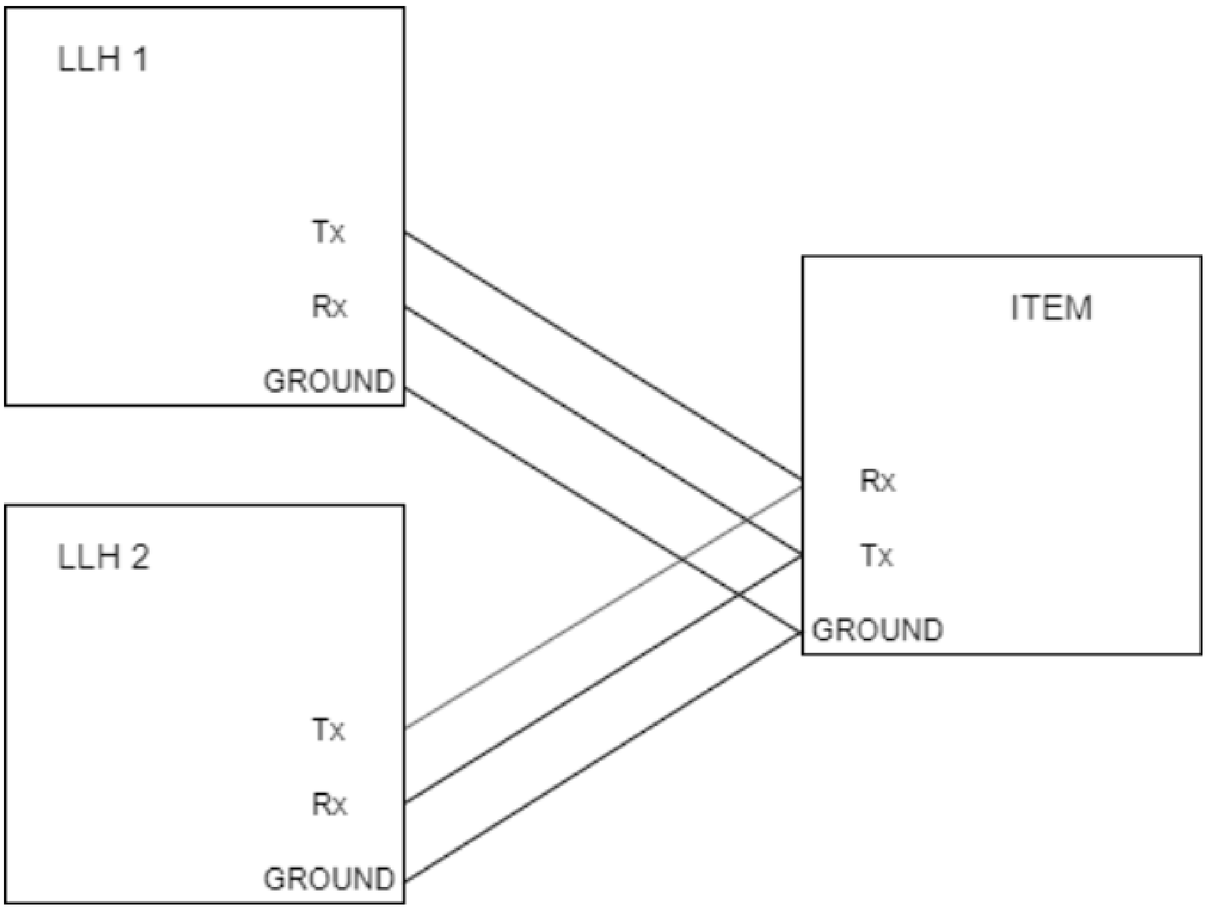


```
/**UART4 GPIO Configuration
PC10     ------> UART4_TX
PC11     ------> UART4_RX
*/
GPIO_InitStruct.Pin = GPIO_PIN_10|GPIO_PIN_11;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT; // high impedance
//GPIO_InitStruct.Mode = GPIO_MODE_AF_PP; // low impedance
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF8_UART4;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

Source: The Author

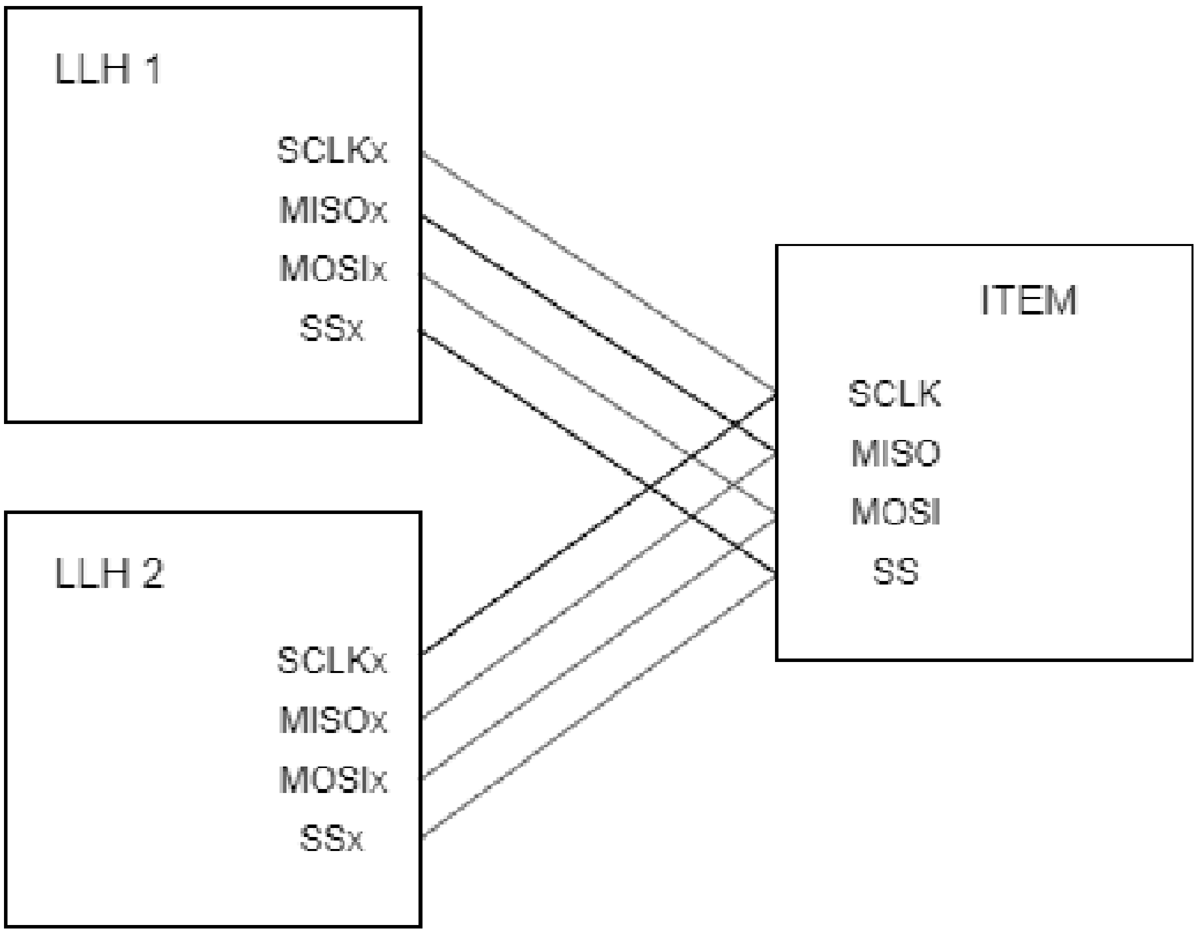
From now on, the connection between instrumentation and LLHs is done by default. It is shown in FiguresFigure 60, Figure 61 and Figure 62.

Figure 60: UART's connection



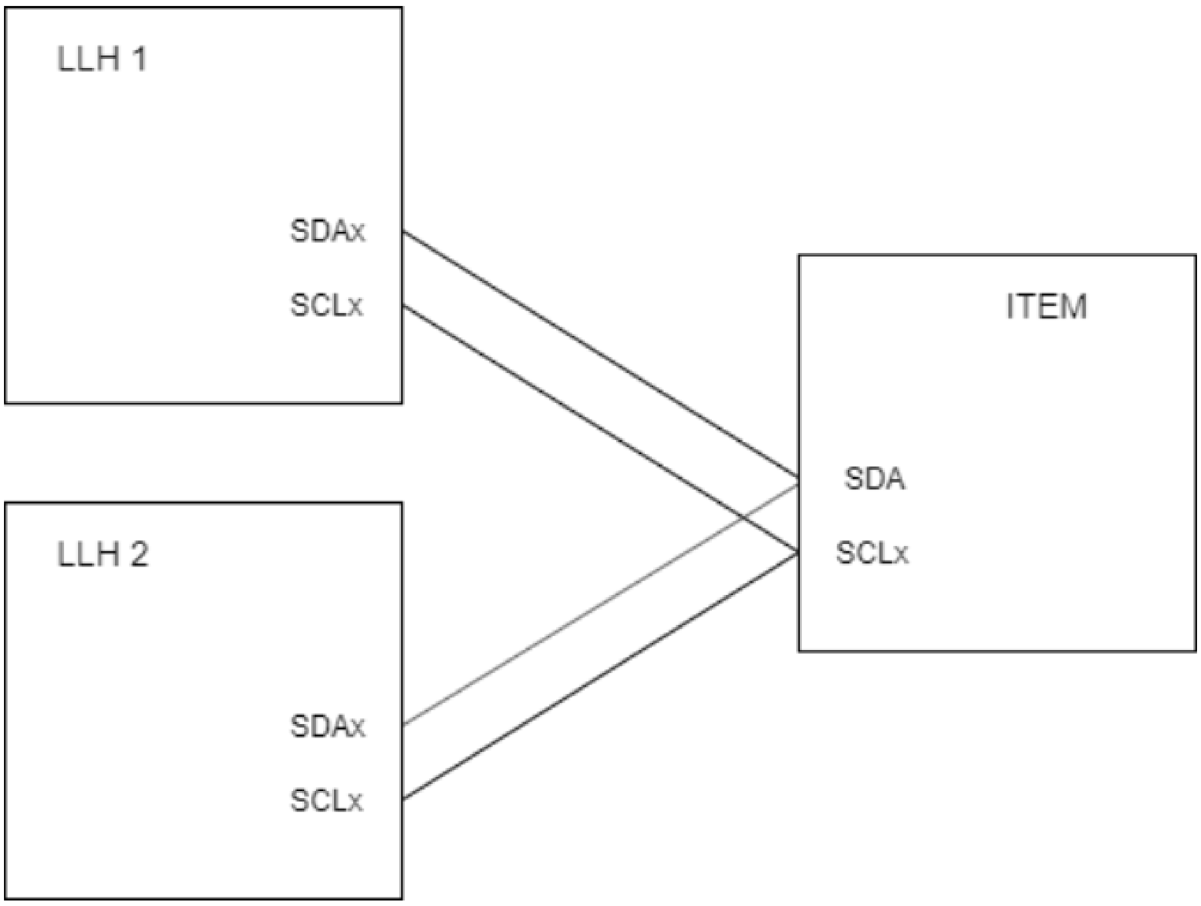
Source: The Author

Figure 61: Source: The Author



Source: The Author

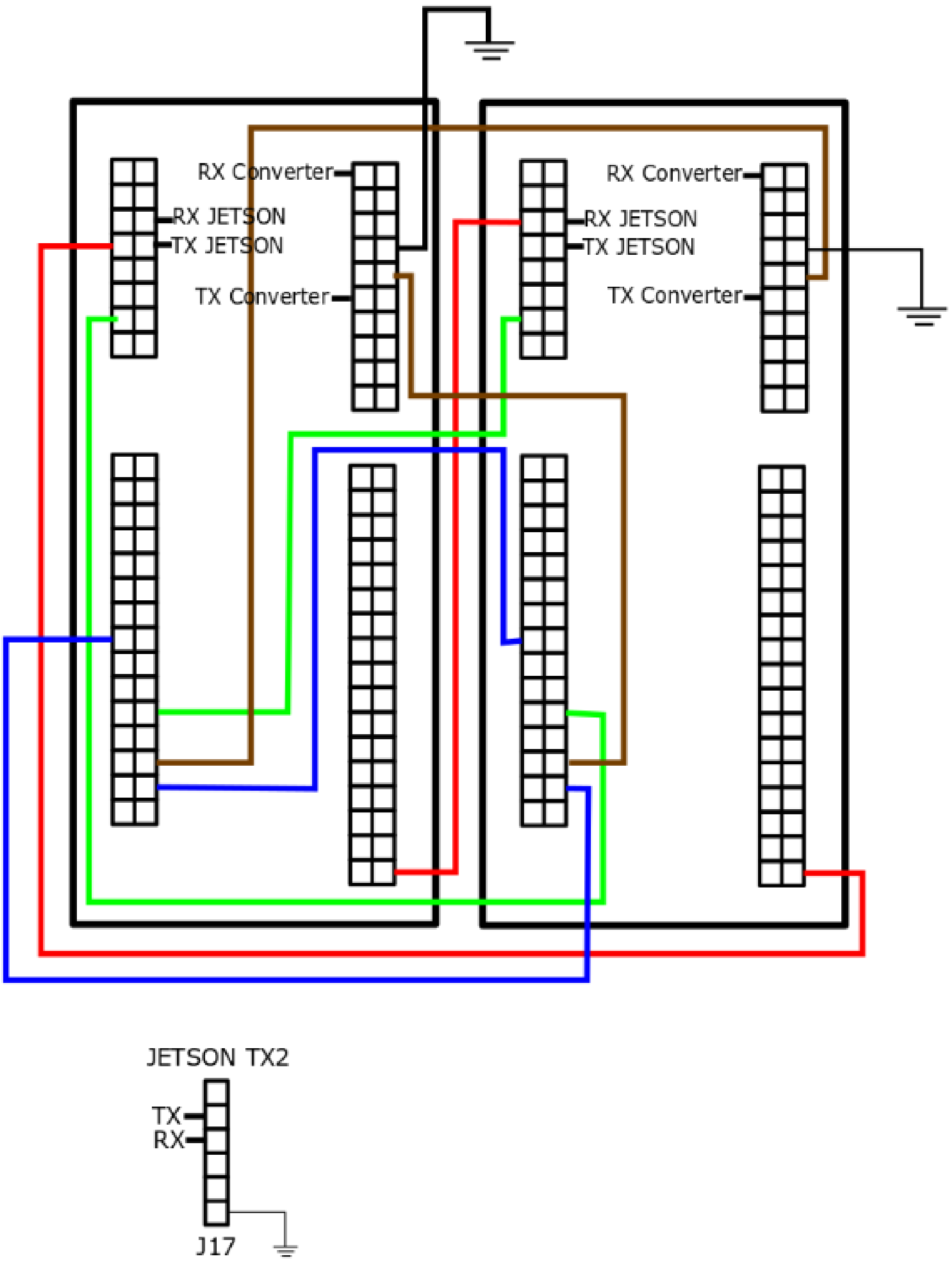
Figure 62: I2C’s connection.



Source: The Author

D.4 Schematic of the prototype built in this work

Figure 63: Schematic of the prototype.



Source: The Author