Universidade do Minho
Escola de Engenharia
Departamento de Eletrónica Industrial

Pedro Guimarães

System Architecture for the ASBGo* Smart Walker

Master dissertation
Master Degree in Industrial Eletronics and Computers

Dissertation supervised by
Professora Doutora Cristina P. Santos

December 2017

ACKNOWLEDGEMENTS

Gostaria de agradecer à minha orientadora por me ter proporcionado a oportunidade de trabalhar num tema tão envolvente e abrangente como este. Ao longo deste último ano desenvolvi as minhas capacidades intelectuais e de trabalho mas foi sobretudo através do relacionamento interpessoal que agora me considero preparado para entrar no mundo de trabalho.

Agradeço a todos os meus colegas e amigos do curso, ao Sr. Carlos e ao meus borrachões de Felgueiras.

Faço ainda um forte agradecimento ao Professor João Luiz Afonso e à Professora Rosa Vasconcelos cujos valores e profissionalismo fazem jus e elevam o nome da Universidade do Minho.

Mãe, Pai, Inês....Obrigado por tudo e continuem comigo.

ABSTRACT

Weakness, mobility and balance problems are some of the obstacles that most certainly will go along with the last period of life, the old age. Also, those difficulties can strike young lives due to gait abnormalities resulted from degenerative diseases or even accidents. To patients with high motor deficit, traditional methods, such as wheelchairs, are usually prescribed, but the use of an assistive device that do not promotes the patient's recovery will eventually lead him to a restrict daily life as well as a notable loss of motor skills.

With previous questions in mind, the *Adaptive System Behavior Group (ASBG)* decided to develop a motorized smart walker capable of adapting to the needs of its users. The *Adaptive System Behavior Group Project (ASBGo)* counts already with four versions that have proved its worth in clinical environment and was renowned, for two consecutive times, as one of the best technological and innovating Portuguese research projects in the rehabilitation field. However, the electromechanical and software solutions of each prototype commonly impair the global development of the project, making each version obsolete, outdated or unusable. Now, it is time to go further and render these proof-of-concept devices in a mature version, excluding previous academic solutions and engineering a robust and trustworthy device that will establish this new rehabilitation concept.

This master thesis, addressed to rehabilitation robotics, describes the design and implementation of a system architecture for the *Adaptive System Behavior Group Project Star (ASBGo*)* . The implementation of a unified modular system architecture embraces the development of software components, electronic hardware and electromechanical modifications required to its implementation. This new prototype is an upgrade of the all ASBGo previous versions, in which the sturdy and user-friendly solutions implemented provide robust tools for future development and usability.

Firstly, the contextualization in the project was performed, including a brief study of robotic software platforms, the familiarity with the several ASBGo prototypes and the research of the best solutions to design a system architecture.

Secondly, following a Top-Down strategy, the work plan was established bearing in mind the considerations to design and implement a global system architecture: definition of the main functionalities and behaviours of the prototype and, simultaneously, the strategies to be followed in the hardware, electromechanical and software development.

Hereinafter, the development stage was conducted following an Hardware-Software co-design methodology. That strategy ensured that the design and implementation of electronic and electric circuits were in agreement with all system requirements guaranteeing trade-offs, robustness and safety.

During all the development process, validations of the system were constantly performed and, in the end, intensive experimentations of the final device were executed in the laboratory with the intervention of colleagues of the ASBG group.

RESUMO

Debilidade e dificuldades de mobilidade e equilíbrio são alguns dos problemas que muito certamente irão acompanhar o período final da vida, a velhice. Para além disso, esses problemas podem atingir jovens vidas devido a anomalias na marcha resultantes de doenças degenerativas ou até mesmo acidentes. Aos pacientes que apresentam um alto défice motor, métodos tradicionais, como cadeiras de rodas, são normalmente receitados. No entanto, o uso de dispositivos de assistência que não promovem a recuperação do paciente irão eventualmente levá-lo a uma vida restrita assim como a uma notável perda de capacidades motoras. Com tal ideias em mente, o *Adaptive System Behavior Group (ASBG)* decidiu desenvolver um andarilho inteligente motorizado capaz de se adaptar às necessidades dos seus utilizadores. O *Adaptive System Behavior Group Project (ASBGo)* já conta com quatro versões que provaram o seu valor em ambiente clínico e já foi reconhecido, por duas vezes consecutivas, como um dos projetos de investigação mais tecnológico e inovador na área de reabilitação. Contudo, as soluções eletromecânicas e de *software* de cada protótipo comprometem o desenvolvimento contínuo do projeto, tornando cada versão obsoleta, desatualizada e inutilizável. Agora, está na hora de ir mais longe e tornar estes dispositivos de prova de conceito numa versão mais madura, excluindo as soluções académicas anteriormente implementadas e concebendo um dispositivo robusto e fiável que irá afirmar este novo conceito de reabilitação.

A presente dissertação de mestrado, no âmbito da robótica de reabilitação, descreve o design e implementação de uma arquitetura de sistema para o *Adaptive System Behavior Group Project Star (ASBGo*)* . A implementação de uma arquitetura de sistema unificada e modular envolve o desenvolvimento de componentes de *software*, *hardware* eletrónico e modificações eletromecânicas necessárias à sua implementação. Este novo protótipo consiste numa melhoria avançada de todas as versões anteriores do ASBGo no qual as soluções vigorosas e acessíveis implementadas providenciam meios para desenvolvimento futuro e usabilidade.

Inicialmente, foi realizada a contextualização no projeto que incluiu uma breve pesquisa de plataformas de *software* robótico, a familiarização com os diferentes

protótipos ASBGo e o estudo das melhores soluções para a conceção da arquitetura do sistema.

Seguindo uma estratégia *Top-Down*, o plano de trabalhos foi estabelecido tendo em conta considerações para o design e implementação de uma arquitetura de sistema unificada: definição das principais funcionalidades e comportamentos do protótipo e, concomitantemente, as estratégias a ser seguidas no desenvolvimento eletromecânico, de *hardware* e de *software*.

Doravante, a fase de desenvolvimento foi acompanhada por uma metodologia de *Hardware-Software co-design*. Esta estratégia assegurou a concordância entre o design e a implementação de circuitos eletrónicos e elétricos e todos os requisitos do sistema garantindo desta forma, *trade-offs*, robustez e segurança.

Durante todo o processo de desenvolvimento, validações do sistema foram constantemente realizadas, sendo que no final testes intensivos ao produto final foram executados em laboratório com a intervenção dos colegas do grupo ASBG .

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

Application Programming Interface. 87, 89

Adaptive System Behavior Group. iii–vi, 1, 2, 4, 28, 33

Adaptive System Behavior Group Project. iii, v, vi, 1, 3–7, 9, 18, 27–29, 31, 32, 35, 115, 116, 122, 134, 140, 141, 143, 144, 146

Adaptive System Behavior Group Project Prototype III. 1, 27, 31–34, 120, 141

Adaptive System Behavior Group Project Star. iii, v, 1, 3, 5–7, 33, 35, 37–39, 43–48, 50, 65, 69, 89, 94–96, 99, 101, 118, 120–125, 128, 134–136, 139–146

Adaptive System Behavior Group Project Plus Plus. 1–4, 27, 32–36, 107, 111, 141, 143, 144

Bayesian Filtering Library. 12

Departamento de Eletrónica Industrial. 1

Degrees of Freedom. 24

Dynamic Time Warping. 19

General-purpose input output. 71, 72, 79, 81, 83

Global Positioning System. 24

Graphical User Interface. 40, 101–104, 123–135, 146, 147

Human-Machine Interface. 24, 31, 32, 39, 41, 89, 111, 122, 142, 143

Inter-Integrated Circuit. 43, 46, 73

Inertial Measurement Unit. 24, 31, 43, 49, 56, 57, 62, 68, 73, 74, 76, 93, 116, 128, 132, 133, 141

Inter-Process Communication. 21

Infrared Distance Sensor. 30, 37, 42, 44, 53, 62, 70, 76, 92, 114, 141

Interrupt Service Routine. 71, 76, 78, 81, 82, 86

k-Nearest Neighbors. 19

Kinematics and Dynamics Library. 11

Orocos Components Library. 11

Open Source Computer Vision. 10

Open RObot COntrol Software. 11, 12

Operating System. 10, 11, 39, 104, 125, 135

Printed Circuit Board. 51–53, 55, 56, 59, 61, 63–65, 117, 118, 145

Printed Circuit Boards. 51, 52, 59, 109–111, 119, 121, 143, 145

Pulse-Frequency Modulation. 22

Proportional, Integral and Derivative. 98–100, 105, 137, 146

Pulse-Width Modulation. 46, 47, 62, 79–81, 101, 145

Red, Green and Blue - Depth. 20, 39, 44, 111, 143, 146

Robot Operating System. 7, 12–18, 20–25, 35, 37, 38, 66, 86, 87, 89, 91, 93–104, 136, 142, 144–146

Revolutions Per Minute. 46

Real-time operating system. 39, 69

Real-Time Toolkit. 11, 12

Single Board Computer. 20, 21

Simultaneous Localization and Mapping. 23

Serial Peripheral Interface. 46

Stanford Artificial Intelligence Robot. 13

Universal Asynchronous Receiver-Transmitter. 46, 78

Universidade do Minho. 1, 28

Ultrasonic Range Finder. 37, 43, 44, 50, 57–59

Ultrasonic Range Finders. 31, 37, 43, 44, 50, 57–60, 62, 72, 76, 77, 93, 108, 109, 117, 143

eXtensible Markup Language. 12, 15

Yet Another Robot Platform. 10, 11

1

INTRODUCTION

This dissertation, framed in the Master's Degree in Industrial Electronics and Computers Engineering, branches of Robotics and Embedded Systems, presents the work developed during the past year in *Adaptive System Behavior Group (ASBG)* of the *Departamento de Eletrónica Industrial (DEI)* , in *Universidade do Minho (UM)* .

In the past years, the *Adaptive System Behavior Group Project (ASBGo)* was renowned as one of the best technological and innovating portuguese research projects in the rehabilitation field (Mediacode, 2017). This research already counts with a first rudimentary prototype (Prototype I), a second prototype that was pioneer in clinical environment (Prototype II), a third device that stated the ASBGo 's value in the assisted gait training (*Adaptive System Behavior Group Project Prototype III (ASBGo III)*), and a more recent version that remains incomplete despite its improvements and clinical validations (*Adaptive System Behavior Group Project Plus Plus (ASBGo++)*). Although ASBGo worth has been proved undergo rehabilitation sessions of ataxic patients in Hospital de Braga, it is now time to go further and render these proof-of-concept devices in a final improved and mature version that excludes the academic solutions used for primary implementation and conceptual validation.

This project, addressed to rehabilitation robotics, describes the development of a new system architecture for the *Adaptive System Behavior Group Project Star (ASBGo*)* . This new prototype will be an upgrade of the four previous versions of the ASBGo smart walkers, coupling several sensors and functionalities of each version. Therefore, the main goals of this dissertation are the design and implementation of a unified modular system architecture by developing software components, electronic hardware and electromechanical modifications required to its implementation, ensuring robust and user-friendly solutions, in order to engineer a trustworthy device that will establish a new rehabilitation concept.

1.1 MOTIVATIONS, SCOPE AND PROBLEM STATEMENT

Weakness, mobility and balance problems are some of the obstacles that most certainly will go along with the last period of life, the old age. Also, those difficulties can strike young lives due to gait abnormalities resulted from degenerative diseases or even accidents. To patients with high motor deficit, traditional methods, such as wheelchairs, are usually prescribed. In this specific case, the use of an assistive device that do not promotes the patient's recovery will eventually lead him to a restrict daily life as well as a notable loss of motor skills. Conventional walkers are also a typical option that presents some rehabilitation potential in contrast with wheelchairs. Nonetheless, for a certain number of patients, this kind of devices are set aside due to handling difficulties, unsuitability and lack of security. This way, introducing a motorized assistive device in a patient's rehabilitation can promote gait training with constant assistance and without much effort. This leads the patient to focus more in the self correction of his motor function and also to a substantial increase of his confidence.

With previous questions in mind, the ASBG decided to develop a motorized smart walker capable of adapting to the needs of its users. From that initial intent, some prototypes were developed, and the last one, the ASBGo++ smart walker, has already proved its worth in clinical rehabilitation. This device allows monitoring user's gait and posture in real time, promoting self-correction through the biofeedback strategies implemented, and performs an objective measurement of some physical parameters which enables a reliable assessment of a patient's evolution. However, due to the varied academic solutions that it is made of and due to the incomplete development, the ASBGo++ prototype works only as prof-of-concept and requires an electromechanical update in order to became a robust and trustworthy device.

The integration of embedded sensors, even those that are not actually coupled in ASBGo++ but are important in the gait rehabilitation, the development of a modular software architecture capable of merge all the different algorithms, which will assure the device's autonomy and enhance the user's monitoring and assessment, and the implementation of user-friendly and robust solutions capable of working without failure are next steps to reach.

1.2 OVERVIEW OF THE RESEARCH

In the last years, the functionalities of the ASBGo smart walkers were validated in medical environment at Hospital de Braga. The assortment of study cases in which the ASBGo has been involved proved the large spectrum of applications of this device and the promising results achieved from this close partnership have keep the both parts interested in innovating and improving this research.

In order to render the ASBGo++ smart walker into a multi functional rehabilitation assistive device, it is necessary to re-establish the electronic solutions of the prototype, taking as main principles the high stability, the security, the easy usage and low cognitive effort for the patient. As referenced before, the previous prototypes developed by the group are prof-of-concepts and this master dissertation, focus on the development of a new architecture, will gather electronic hardware and software solutions to carry the ASBGo* prototype nearest of a final and marketable version.

The first step to be taken into account in this research is the contextualization in the project. A brief study of robotic software platforms, the familiarity with the several ASBGo prototypes and the research of the best solutions to design a system architecture are included in this task.

Secondly, the work plan should be established bearing in mind the design of a global system. This step embraces the definition of the main functionalities of the prototype and, simultaneously, the strategies to be followed in the hardware, electromechanical and software development.

Hereinafter, the development stage should be conducted. In this phase it is extremely important to respect a Hardware-Software co-design methodology. Moreover, the design and implementation of electronic and electric circuits should be in agreement with all system requirements guaranteeing trade-offs, robustness and safety.

Finally, the validation of the system is performed alongside with its gradual development. It is expected the execution of exhaustive tests, in the laboratory, trying to cover every possible configuration of the device. The experimentations will be conducted by the developer as well as by other members of the ASBGo group.

1.2.1 *Methodological considerations*

Over the past years, the ASBG team integrated miscellaneous specialized students who contributed to the emergence of the several functionalities of the ASBGo prototypes. This foremost work was extremely important because resulted in the definition of features that a smart walker should integrate in order to became an exclusive assistance device capable of promote individualized gait trainings, raising the concept of physical rehabilitation. In fact, the features that have been classified by the group as important to motor function reestablishment, have been developed, implemented, tested and validated in clinical environment. Simultaneously, those that received medical approval, were chosen to be integrated in a final version of the device. On the other hand, the fact that they were developed by a different person with a proof-of-concept intention, created, in some cases, software incompatibilities. Thus, the first methodological consideration to be take into account is to follow a Top-Down strategy where a global and unify system is developed from root that, besides allowing to join various algorithms previously developed, can be expanded in the future with new features. This forces the re-implementation of all functional requirements, specifying the behaviours and/or functions of the system.

The second strategy is linked to the first consideration. If, at first sight, the software compatibility is the hurdle that disables gather all the features, also the compatibility between physical hardware and electronics is a challenge. In order to build a highly technological device that assures safety in its use, it is necessary to reinvent all the electronics, fostering strategies and solutions that allow it to work without failure. Thus, it is crucial to discard all the academic solutions that compose the current version (ASBGo++ prototype) and design a new robust system.

Due to the countless sensors that will be embedded in the system and to its inherent complexity, it should be followed a Hardware-Software Co-design methodology. In contrast to ordinary approaches, on whose the specification and design of hardware/software occurs separately, this methodology defines the system-level objectives by the analysis of trade-offs between hardware and software simultaneously.

Finally, the involvement of mechanical experts will enable to carry out the mechanical modifications required to assemble the new prototype and the close partnership with Hospital de Braga will ensure future validations in clinical environment.

1.2.2 *Goals and research questions*

This dissertation is framed in the development of a soft real-time system architecture for the ASBGo* smart walker. To accomplish this main objective the following goals have to be achieved:

Goal 1: Conduct a review of previous versions of the ASBGo smart walkers. An extensive survey about functionalities, mechanical structures, hardware, software and electronic solutions will enable the contextualization in this project.

Goal 2: After the understanding of the problem, the expected behaviour of the system and its functionalities will be settled up. Here, the features and requirements defined in *Goal 1* should be included.

Goal 3: Study the most common robotic software platforms and choose a framework suitable to the development of the system with the characteristics presented in *Goal 2*, defining also the methodological strategy that should be followed in the design and development process.

Goal 4: Design the new system architecture and planing the development phase. This task covers the conceptual development of all the system, which includes sensors data acquisition, communication protocols between processing units, security and safety methodologies, locomotion of the robot and human-machine interface.

Goal 5: Specification of the appropriated material to build the new architecture (including hardware selection since basic sensors to processing units, power supplies, cables and electronic components to create protection circuits), ensuring completely compatibility between them. Since the majority of the material is not available and it will be purchased, the compatibility is a crucial demand.

Goal 6: Implementation of real-time sensors data acquisition, locomotion of the robot and human-machine interfaces. The data acquisition will dispose all the sensors' data embedded in the system, enabling a low level abstraction easing future algorithm expansion.

Goal 7: Perform unit tests to each implemented functionality and global validation of the system in the lab station. The global validation will also be performed by other members of the group, in order to improve details in the human-machine interface.

Goal 8: Documentation of all the work developed. It will detail all the research made and the tools used, the analysis of the problem, the design and implementation of the final system, the final results mentioning the achievements, failures and conclusions and a configuration/user manual of the ASBGo* smart walker.

The following research questions (**RQ**) are expected to be answered:

RQ1: How many ASBGo prototypes exist, what functionalities they have and how their hardware and software solutions are implemented ?

RQ2: What are the main goals and the requirements that the new prototype should fulfill, in terms of behaviour and features ?

RQ3: What is the most proper robotic software platform to develop the new system architecture and what strategy should be followed to engineering it ?

RQ4: In how many conceptual blocks the system will be organized and how can they be specified ?

RQ5: What material had to be acquired and/or purchased to develop the new system architecture ?

RQ6: Does the mechanical structure requires physical modifications ?

RQ7: The implemented solution was successfully validated and responds to all the demands that were initially proposed ?

RQ8: What are the next steps that should be taken and what future work possibilities are available in order to improve the developed system ?

1.3 CONTRIBUTIONS TO KNOWLEDGE

Linked to the development of a new system architecture for the ASBGo* smart walker, this project covers distinct layers of work. The following statements list the main contributions of this Thesis:

- Design a suited system architecture for the ASBGo* smart walker, reinventing the several individual solutions of previous prototypes through the development of a modular and robust device, discarding the academic approaches that the previous systems are made off.
- Through the accurate hardware development and the implementation of a modular software architecture, the failure risk of the device is reduced, increasing the security of use by patients in clinical environment and its usability in general terms.
- The new modular architecture, which includes gathering and embedding all sensors of the previous versions and dispose the data collected in a high level layer, eases the access to those sensors' information and so, allows to explore new algorithms in order to expand and/or optimize the functionalities of the device.
- The development of a dedicate human-machine interface for the ASBGo* smart walker, through which all the interaction between the users and the system is accomplished, forces the inclusion of future application-level features in the dedicated interface.

1.4 THESIS OUTLINE

This document is organized in six chapters: Chapter 1 clarifies the main theme of this thesis revealing also the methods to strike the proposed problem, the fundamental goals, research questions and contributions to knowledge. Chapter 2 focus on the presentation of some theory related to robotic software platforms highlighting the *Robot Operating System (ROS)* and devices that use that platform. Chapter 3 introduces the history of the ASBGo by presenting the mission and ideas of the project. Also, all the previous versions are detailed. In Chapter 4 is described all the development process that led to the creation of the ASBGo* . The final result is revealed in the Chapter 5 as well as different tests performed in the new device. Finally, the conclusions and potential future work for the new device are covered in Chapter 6.

2 LITERATURE REVIEW

The aim of this dissertation is the software and hardware development thus creating a new version of the ASBGo . Hence, this chapter explores literature related to robots development, more specifically, assistive robots and algorithms and tools that ease the robots' software development.

First, it will be analyzed current platforms for robotics software development in order to know its characteristics and in which way they can help developers. Despite not being extremely necessary the use of such tools, it will be seen that they can offer a number of considerable advantages, like develop robots in standard patterns, which contributes to software reusability. Also, it will be seen that each presented framework has its own main use-case and so, this analysis will enable a sure decision about which one will fit in this project and why not use the others.

Secondly, exploring real assistive devices based on robotic software platforms, allows to define some conclusions and consider ideas and solutions based on those type of systems.

2.1 ROBOTICS SOFTWARE PLATFORMS

It is noticeable a continuous growth in the number of complex applications that rely on the processing of huge amount of data perceived from different sensors. Some applications like domotics, medicals and entertainment applications use multi-modal sensed data to extract environmental information and act accordingly. From the software engineering point of view, systems with a high level of complexity arouse many challenges (Ando and Kuffner, 2012). This way, the use of frameworks allows to simplify such complex scenarios by increasing the efficiency of creating new software improving develop productivity, quality, reliability, modularity and robustness of new software. Despite the differences, robots also share some functionalities at some point and so, the use of software frameworks for robots development is "a must".

Some tools cannot only be considered frameworks, which offer simulation and debugging tools, compilers, core libraries and *API*'s, but they can also act as a middleware layer focusing on some complex functionalities like a messaging passing infrastructure between processes. Platforms can be designed for a particular purpose: either to respond to perceived weaknesses of other frameworks or to emphasize aspects that are considered essential during the design process of a certain system (Serrano).

Figure 1: YARP logo (YARP, 2016a).

Mainly used in humanoid robotics, *Yet Another Robot Platform (YARP)* is an open-source framework that wraps a set of libraries, protocols and tools to keep modules and devices properly decoupled. It minimizes the effort devoted to infrastructure-level software development by facilitating code reuse, modularity and so, maximize research-level development and collaboration. YARP attempts to make robot software more stable and long-lasting, without compromising the ability to constantly change sensors, actuators, processors, and networks. It helps to organize communication between sensors, processors, and actuators, by encouraging the promotion of loose coupling, easing the gradual system evolution. YARP is not an operating system and it has no desire to be in control of the developer's system, being then, a reluctant middleware layer (YARP, 2016b; Tsardoulias et al., 2013; Metta et al., 2006).

Its components are (YARP, 2016b):

- *libYARP_OS* - is the core component of YARP . Interfaces with the operating system(s) to support easy streaming of data across many threads in different machines. Being written almost entirely in C++, YARP is also written to be *Operating System (OS)* neutral, and it is used on Linux, Microsoft Windows, Apple macOS and iOS, Solaris, and Android.

- *libYARP_sig* - performs signal processing tasks in an open manner and it is easily interfaced with other commonly used libraries, like *Open Source Computer Vision (OpenCV)* .

- *libYARP_dev* - interfaces with common devices used in robotics such as motors controllers, boards, cameras and more.

Concluding, the main features of YARP include support for inter-process communication, image processing and a class hierarchy to ease code reuse across different hardware platforms (Metta et al., 2006).

Another appealing project, known as *Open RObot COntrol Software (OROCOS)* (see figure 2), aims the development of a modular, general-purpose, open-source framework for robot's control. It is supported by different OS 's such as Linux, Windows and macOS (OROCOS, 2016b).

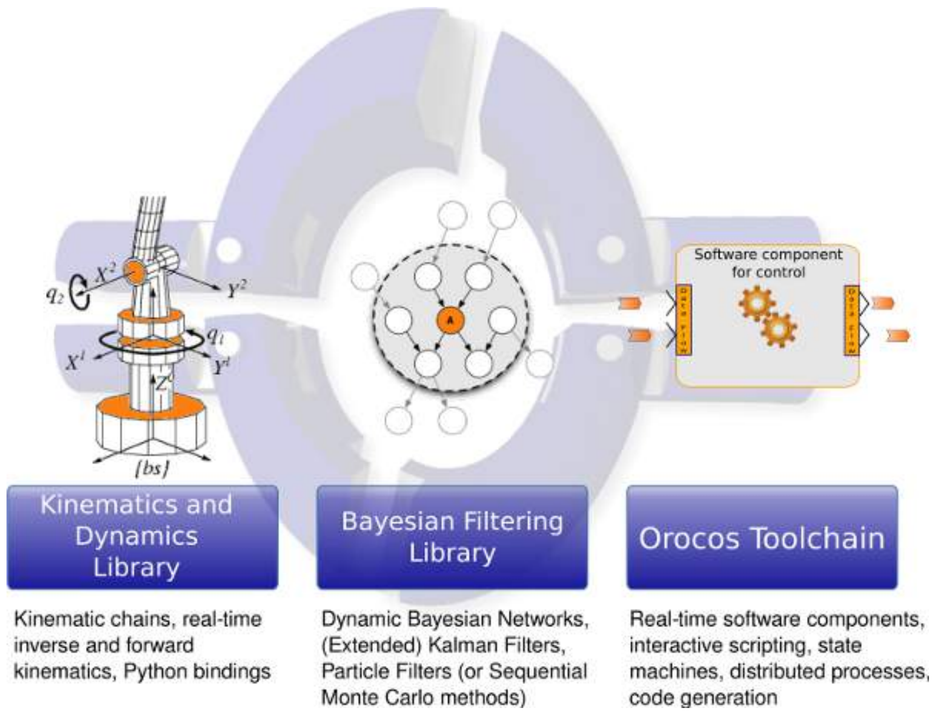

q_2
X^2
Y^2
q_1
X^1
Y^1
Z
{bs}
A
Software component for control
Data Flow
Data Flow
Kinematics and Dynamics Library
Bayesian Filtering Library
Orocos Toolchain
Kinematic chains, real-time inverse and forward kinematics, Python bindings
Dynamic Bayesian Networks, (Extended) Kalman Filters, Particle Filters (or Sequential Monte Carlo methods)
Real-time software components, interactive scripting, state machines, distributed processes, code generation

Figure 2: OROCOS project and logo (OROCOS, 2016a).

Consists in four C++ libraries (OROCOS, 2016b; Rutgeerts, 2007):

- The *Real-Time Toolkit (RTT)* provides the infrastructure and the functionalities to build real-time robotic applications in C++.
- The *Orocos Components Library (OCL)* provides some ready-to-use control components built using the RTT .
- The *Kinematics and Dynamics Library (KDL)* is a C++ library which provides support for robot (or general systems) kinematics and dynamics.

- The *Bayesian Filtering Library (BFL)* is an application independent framework for inference in recursive information processing and estimation algorithms based on Bayes' rule. Includes support for different Kalman variants, particle filters and it is easily extensible to include other Bayesian methods.

OROCOS applications are composed of software components, which form an application specific network. When using OROCOS , it is possible to use predefined components, contributed by the community, or build components. A single component may be capable of controlling a whole machine, or may be focused on a small part in the whole network of components, for example a component focused on signal filtering.

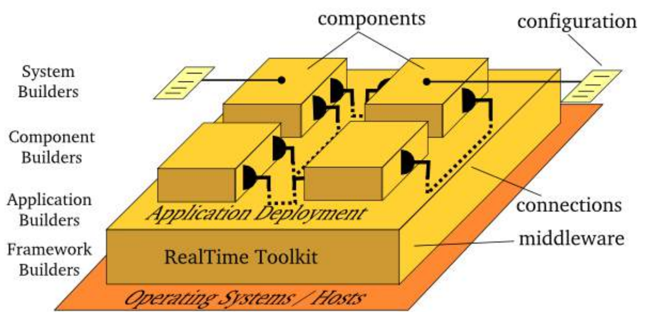


Figure 3: OROCOS RTT overview.

The components are built with the RTT (see figure 3) and can optionally make use of any other library (like a vision or kinematics toolkit). Most users interface components through their *eXtensible Markup Language (XML)* properties or command/method interface in order to configure their applications. Even so, there are five distinct ways in which an OROCOS component can be interfaced: through its (1) properties - run-time modifiable parameters, stored in XML files; (2) events - functions that are executed when a change in the system occurs; (3) methods - which are callable by other components to output a result immediately, just like a C language function; (4) commands - sent by other components to instruct the receiver to 'reach a goal'; (5) data flow ports - thread-safe data transport mechanism to exchange data between components. OROCOS allows to write hierarchical state machines using these primitives. This is the way of defining the application specific logic. State machines can be loaded/unloaded at runtime in any component (OROCOS, 2016b).

The frameworks presented are some of the major "competitors" of a third one that is about to be presented. Known as ROS , this robotic platform stands out in some positive aspects like its inter-platform operability, its inference to a modular software development and more reasons that will be detailed next. Also, by the fact that ROS is being increasingly adopted by research groups, commercial companies and governmental organizations, encourages the will to learn and to use such platform.

2.2 ROS - ROBOT OPERATING SYSTEM

ROS is an open-source framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior (ROS, 2016a). Provides various features such as message passing, distributed computing, code reusing, and so on. The project started in 2007 by Morgan Quigley as part of the *Stanford Artificial Intelligence Robot (STAIR)* project but the main development happened at Willow Garage (see `www.willowgarage.com/`) (Joseph, 2015).

Figure 4: ROS logo (ROS, 2016b).

The ROS community is growing very fast, having many users and developers worldwide (see `https://vimeo.com/146183080`). Most of high-end robotics companies are now porting their software to ROS . This way, the increase of ROS -based applications can generate a lot of job opportunities in industrial robotics field. Also, the impact of this platform in the field shows that the knowledge in ROS will be an essential requirement for a robotic engineer (Joseph, 2015).

Contrarily to what the name suggest, ROS is not an operating system. It runs on Unix-based platforms, primarily tested in Ubuntu, and so, it is considered a meta-operating system meaning that it provides some of the services of a full operating system (Joseph, 2015; Thomas, 2016).

Why ROS ? The pros and cons

The ROS platform is preferable because (see figure 5) (Joseph, 2015):

- It comes with ready-to-use functionalities that are totally configurable, have clear interfaces and can easily be integrated in a robot software. Besides being well documented, those capabilities are also very well implemented and validated. Autonomous navigation, which contains motion planning, obstacle avoidance and many other algorithms, is an example of a high-end capability to be added to mobile robots.

- Contains many simulation, debugging and visualization tools.
- It wraps device drivers for many sensors and actuators, easing the task to interface such components.
- It is composed by a message-passing middleware that allows the communication between different ROS processes called nodes. This processes can be programmed in any language supported by ROS like C, C++, Python or Java.
- It respects the development of software in blocks i.e. in different modules. The software becomes more comprehensible, more logic and most of all, since the processes (modules) are separated with clear interfaces, if one fails, the system can still run and not crash. This way, robustness is introduce in the system.
- Incorporates a concurrent resource handling mechanism allowing any number of ROS processes to consume a certain resource, like a sensor data, when it is available and perform a particular functionality. Thus, the complexity is reduced and it gets easier to debug the application.
- It has an active community to support any possible doubt. Through the web portal (see `http://answers.ros.org`), ROS users exchange ideas, questions and answers. Also, the ROS wiki website (see `http://wiki.ros.org/`) collects a vast documentation about practically everything that involves ROS and collects a considerable number of tutorials.

Figure 5: ROS equation (ROS, 2016a).

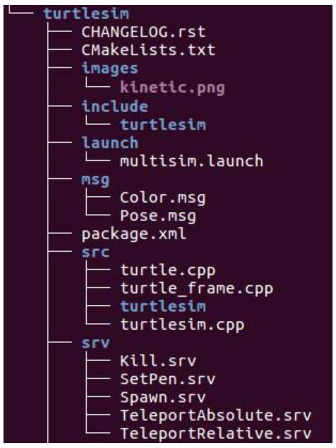
The ROS platform is sometimes avoided because (Joseph, 2015):

1. It needs a operating system to run.
2. It do not ensure real-time constraints. For hard real-time systems, where it is imperative to meet strict deadlines, ROS is not advised because all the overhead and latency that exits in it can compromise the system's time constraints. However, ROS tries its best to meet the deadlines of the system.

ROS file system level

Like in an operating system, the ROS file system is organized in a particular way. This characteristic confers to it a standard style of software development. The ROS files and folders are hierarchically organized as follows (Joseph, 2015; Martinez, 2016a):

1. *Meta packages* - is a term used to refer to a group of packages with a specific purpose. One example is the *navigation* meta package which contains a set of packages that contributes for that purpose. It contains the *amcl* package (a probabilistic localization system for the robot), the *path_planner* package (plans a path for the robot) and so on.
   a) *Packages* - are considered the basic unit of the ROS software. It gathers the ROS runtime processes, libraries, configuration files, build file and many more elements. Not all the following components are mandatory in a package. So, a typical structure of a ROS package is (follow *turtlesim* package's example in figure 6):
      i. *Includes* (*include* folder) - gathers the header files and libraries necessary in the package.
      ii. *Sources* (*src* folder) - stores the source codes written in programming languages compatible with ROS . The source codes can be future ROS processes (nodes) or just auxiliary code.
      iii. *Scripts* (*scripts* folder) - executable scripts that contain relevant commands to automate certain tasks for the package.
      iv. *Launch files* (*launch* folder) - keeps the launch files that are used, for example, to launch one or more ROS processes of the package or to define important parameters for those processes.
      v. *Messages* (*msg* folder) - contains the ROS messages files (more detail in 2.2 - *ROS implementation concepts*).
      vi. *Services* (*srv* folder) - contains the ROS services files (more detail in 2.2 - *ROS implementation concepts*).
      vii. *Package.xml* - XML file that defines properties about the package such as the package name, authors, maintainers, and dependencies on other packages.
      viii. *CMakeLists.txt* - CMake file that manages the build process for the package.



```
└── turtlesim
    ├── CHANGELOG.rst
    ├── CMakeLists.txt
    ├── images
    │   └── kinetic.png
    ├── include
    │   └── turtlesim
    ├── launch
    │   └── multisim.launch
    ├── msg
    │   ├── Color.msg
    │   └── Pose.msg
    ├── package.xml
    ├── src
    │   ├── turtle.cpp
    │   ├── turtle_frame.cpp
    │   ├── turtlesim
    │   └── turtlesim.cpp
    ├── srv
    │   ├── Kill.srv
    │   ├── SetPen.srv
    │   ├── Spawn.srv
    │   ├── TeleportAbsolute.srv
    │   └── TeleportRelative.srv
```

Figure 6: Typical ROS package (Martinez, 2016b).

There are bash commands to create, modify and work with these packages. Also, ROS provides a package with bash-like commands, called *rosbash*, used to navigate and manipulate ROS packages.

ROS implementation concepts

A system built using ROS consists in a network of processes, in one or more hosts, connected at runtime. That being said, to understand this level of operation of ROS , the following fundamental concepts must be taken into account (Joseph, 2015; Martinez, 2016a):

- *Nodes* - are the processes that perform computation. Since ROS was designed to be modular, a typical ROS node can be considered a software module that performs a certain task, for example, acquire the data from a sensor. This way, the aim of the nodes is to build simple processes, with communication interfaces to exchange data with each other, rather than a large process with all the functionality. Also, a system granulated in nodes eases the process of unitary debug.
- *Messages* - are typed data structure used by nodes to communicate with each other. It supports primitive types like integers, booleans, floating points and many more as well as arrays of those types. Also, messages can contain other messages and arrays of them.
- *Topics* - are containers, with a unique name, that transports messages. In one hand, a node sends messages (data) by publishing it to a topic and in

the other hand, a node interested in a certain kind of data must subscribe to the appropriate topic. There may be multiple concomitant publishers and subscribers for a single topic as well as a single node may publish and/or subscribe to multiple topics. The production and consumption of information are decoupled meaning that publishers and subscribers are not aware of each others' existence. When created, a topic must have a message type associated and the nodes that access to it, must respect that type.

- *Services* - are used for a request/response interaction between nodes. For some ROS nodes, synchronous transactions are more appropriated and so, the publish/subscribe model is not enough. This way, there will be a client node that sends request messages to a server node that provides a certain service and responds back with a result. While the server performs the service, the client waits for the response.

- *Master* - The ROS master is the crucial element of the system. The master provides a name registration and a lookup mechanism that allows the nodes to exchange messages or invoke services. So, in a certain robot (or a general system) using ROS , the master must be running in the computer system.

Working with ROS

An example of a simple application using ROS will be presented. Some concepts addressed in 2.2 - *ROS implementation concepts* will be used. So, the simple application follows the idea presented in figure 7.

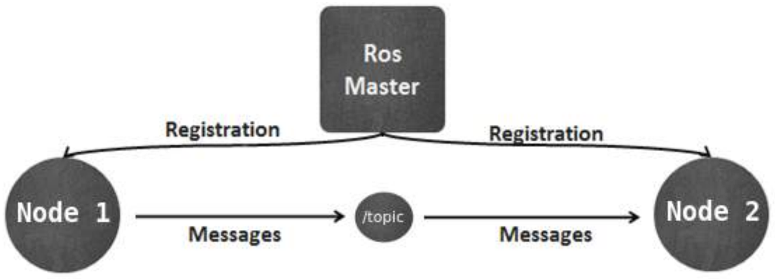


Figure 7: Simple application blocks.

The *node1* in figure 7 is our publisher node, called *camera_acquisition_node*. It will acquire the data from a camera and it will publish the data in a topic called *camera_frames*. In the other side, the *node2* will subscribe to that topic and so, when a message is available in the topic, the *processing_node* (our *node2*) will consume the information and for example, process the current camera frame with a computer vision algorithm. The type of the message of the topic must be of an image type of ROS .

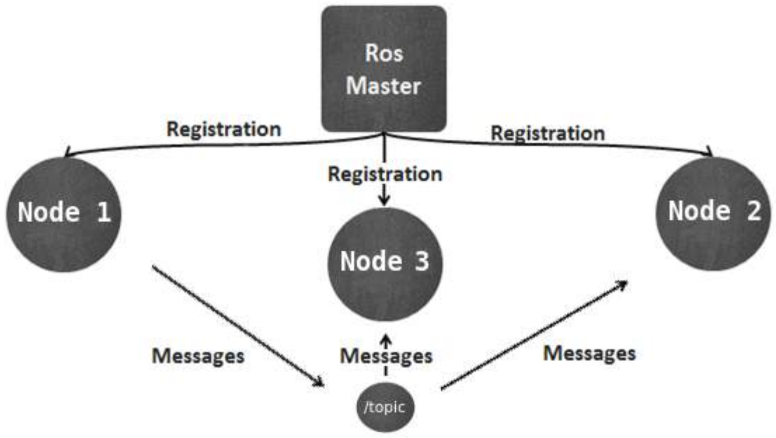


Figure 8: Simple application blocks - extended.

Now, as it can be seen in figure 8, if we want to add a new process (a new ROS node) to perform another task, it is possible and quite trivial. We added the *node3*, called *display_image*, which also subscribes from the *camera_frames* topic and just displays the camera images on a screen. So, we can have as many processes as we want that consume the data from a camera and execute independent tasks.

2.3 ros-based assistive devices

Given the assistive nature of the ASBGo and taking in advantage the study of the ROS platform in 2.2, some examples of assistive robots built with the support of ROS are detailed next. These assistive systems can sense, process sensory information and then perform actions that benefit people with disabilities and people of the older age group. Some are designed to provide a certain kind of diagnostic (measurement and assessment), others for therapeutic benefit (improvement of a certain body function) and some uses both purposes.

Studying ROS -based assistive devices allows to acquire some architectural, design and implementation ideas and/or solutions that could be helpful for the development journey. The following presentations cover a simple overview of the device, giving a highlight over the ROS concepts and the architectural aspects of the system.

SmartWalker: an intelligent robotic walker (Shin et al., 2016)

SmartWalker (figure 9) is a robotic walker, consisting in a high-tech extension of a regular walker, equipped with sensors and actuators. It as two modes of operation: the autonomous and assistive mode.

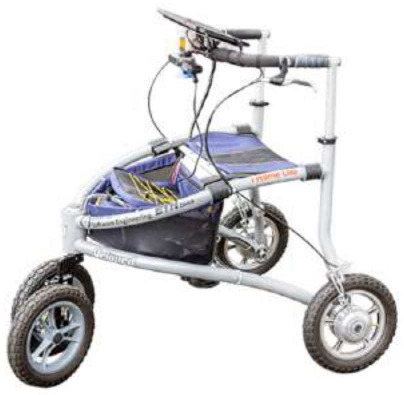


Figure 9: *SmartWalker*.

In the autonomous mode, the *SmartWalker* can receive user commands through its real-time gesture-based user interface and navigate around the environment accordingly. The underlying algorithm uses a *k-Nearest Neighbors (k-NN)* classifier with *Dynamic Time Warping (DTW)* to classify gestures. As people can unintentionally make gestures similar to the commands, the interface has also the ability to ignore unintentional gestures by detecting the users' attention along with the gesture recognition.

In the assistive mode, it is not required to push the walker. The device supports the user by detecting the legs and by controlling the speed according to the walking speed and the ground inclination. The walking speed is computed by detecting the leg movements using a laser range scanner and by combining this information with the ground inclination and the state of brakes in the controller, resulting in the appropriate speed for the walker.

The *SmartWalker* is composed of a walker frame enhanced with sensors, actuators, and appropriate software:

- As to the **hardware**:

  The walker consists of a normal walking frame, two hub engines, a laser range scanner, an inclinometer, and a rotatable camera. The front wheel has no motor, conferring stability and maneuverability to the device. The rear wheels are powered by electrical bike motors. Each hub engine, located at the rear wheels, contains a Hall effect sensor for measuring the rotational speed of the each wheel. The laser range scanner, at the bottom center of the walker, is used for obstacle avoidance; the inclinometer, on top of the scanner, measures the pitch of the ground; and the *Red, Green and Blue - Depth (RGB-D)* camera, which is attached to a motor for 360°rotation and placed below the handlebar, is for gesture recognition. The camera has an operation range of 0.8 to 3.5 meters and works indoors only.

The *SmartWalker* has two processing units connected via a local network. The first one, a tablet-PC, provides a touch user graphical application (see figure 10) and is where most of the computation is running. The second device is a BeagleBone Black 5 *Single Board Computer (SBC)* used for message passing between the tablet and the sensors/actuators.

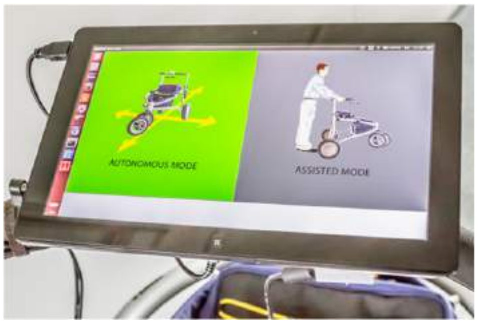


Figure 10: *SmartWalker* graphical user interface.

- As to the **software**:

  The software is distributed between the tablet-PC and the SBC . In the tablet runs the main control application and the gesture recognition module, whereas the BeagleBone acquires raw data from the sensors and also receives commands from the tablet to enable the actuators.

  As it can be seen in the figure 11, in the tablet-PC runs the main control application developed in another robotics programming framework, *Roboscoop* (Rusakov et al., 2014). In the main processing unit also runs a ROS node, written in C++, responsible for the gesture recognition functionality that uses image processing libraries from ROS . The communication between these two

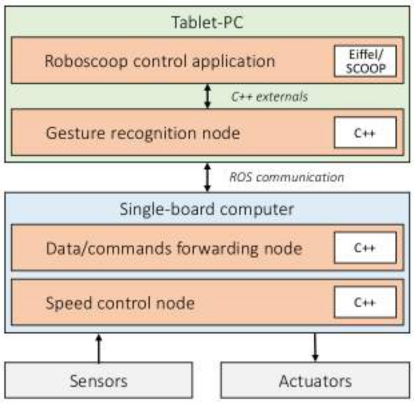


Figure 11: *SmartWalker* software distribution.

processes (ROS node and main control application) is made through *Inter-Process Communication (IPC)* .

In the SBC , it run ROS nodes that: manage data/commands between the Tablet-PC and the SBC , acquire the sensors' data and control the speed of the motors. The communication between the two processing units is accomplished through the use of a network communication provided by ROS .

Smart walker with navigation features for blind people

Another smart walker (figure 12), work of Wachaja et al. (2014), provides different navigation capabilities to assist blind people with or without walking disabilities.

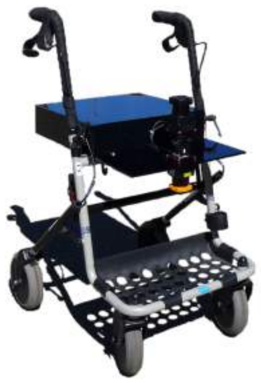


Figure 12: The walker.

The system consists in an off-the-shell walker, adapted with sensors and data-processing capabilities. The walker was developed in a modular fashion allowing the easy integration of the sensing and processing unit in other walker brands. ROS was used to simplify, by the use of existing and available modules, and also to increase the software flexibility.

Following figure 13, the walker includes two planar laser range finders, one to calculate the egomotion by laser scan matching and the other, continuously tilted by a servo motor, senses the three-dimensional environment. By fusing the egomotion estimation with the measurements of the second scanner it is possible to obtain a dense three-dimensional point cloud *i.e.* a surface of the environment. An approach that enhanced the terrain classifiers from robotics was developed allowing to detect hazardous obstacles from the point cloud.

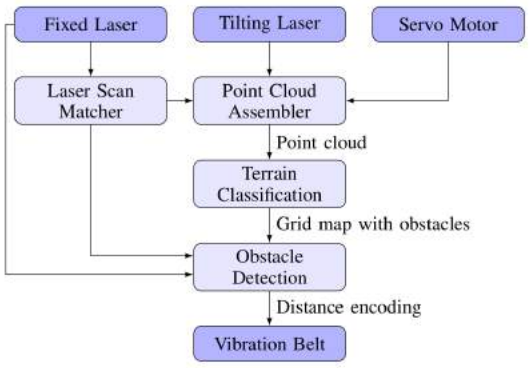


Figure 13: Simple system architecture of the smart walker.

The distances to nearby obstacles are computed by fusing traversability information from the classifier and the data from the fixed laser. The information is then relayed to a vibration belt via bluetooth. The belt comprises five vibration motors evenly distributed and provides an haptic feedback to the user. Obstacle distances are encoded with *Pulse-Frequency Modulation (PFM)* meaning that closer objects result in the respective motor vibrating with a higher repetition rate.

Continuing the software to be based on the ROS framework, another functionality was developed for this same smart walker (Wachaja et al., 2015). The device uses the same sensors, the laser range finders, and vibration motors in the walker's handles for vibrotactile feedback are added. The processes of terrain classification and obstacle detection presented in the previous functionality remain, but now, it is used publicly available ROS modules for path planning based on the Dijkstra algorithm - an algorithm which detects the shortest path between nodes in a graph. The path planner considers a map as well as the eventual obstacles. The map is created beforehand with

Simultaneous Localization and Mapping (SLAM) algorithm also being able to perform this process and the path planning in parallel.

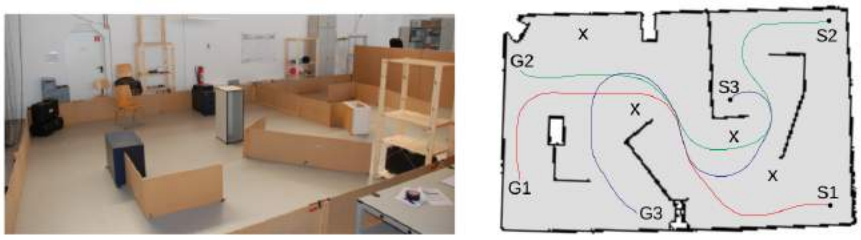


Figure 14: Test environment (left) and the corresponding map with planned paths (right).

The controller module on the system guides the user through the environment with for vibration signals generated by the vibration motors on the handles: go straight, turn left, turn right and goal reached. Each signal is repeated continuously until it is overwritten by another one. This functionality eases the life of individuals with visual impairment allowing their safe navigation in an environment.

Essex Wheelchair - a ROS-Based Multi-sensor Navigation of Intelligent Wheelchair

This project presents a ROS -based multi-sensor navigation for an intelligent wheelchair (see figure 15) that can help the elderly and disabled people (Li et al., 2013).

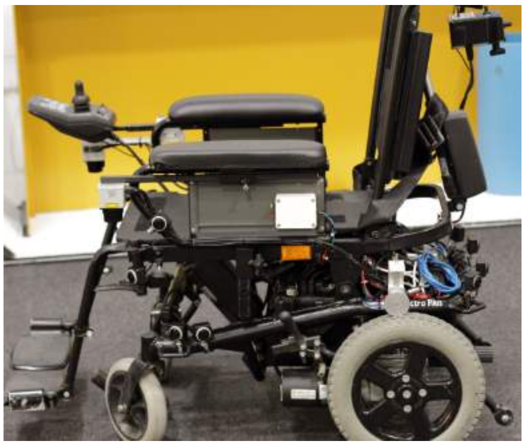


Figure 15: *Essex* wheelchair.

As to the system architecture (see figure 16):

- A commercial powered wheelchair is equipped with embedded computers, control electronics, wireless networks and multiple sensors such as: optical odometry encoders, two laser scanners, 12 sonar sensors, a 9*Degrees of Freedom (DOF) Inertial Measurement Unit (IMU)* , a camera, a microphone and a *Global Positioning System (GPS)* .

- A low level controller, installed on the embedded computer running Linux, provides interfaces to sensors and motor actuators. A server process running at the same embedded computer publishes sensor data and subscribes to commands for the motors' actuation. Unified data type, data structure of sensors information and actuator commands are defined inside. Also, serial boot-loaders have been installed in the modules with the embedded computer to facilitate future development.
- On the high level controller, a ROS based system is developed, consisting in the client. It provides autonomous navigation and an interaction interface. The *Human-Machine Interface (HMI)* bridge allows the users to actuate the navigation by the interface according to their preference.

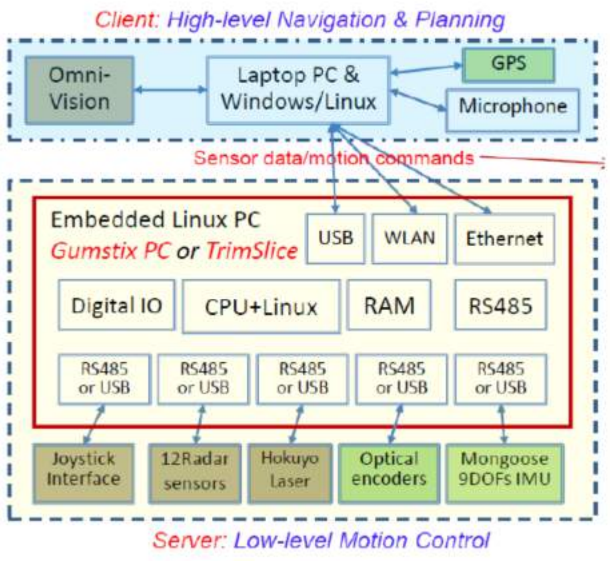


Figure 16: System architecture of *Essex*.

As to the wheelchair navigation system:

- As previously referred, the infrastructure of the wheelchair control system is based on ROS . Also, the navigation system is based on the ROS navigation stack. So, through a communication node, sensor data is sent from the low level controller to the high level controller. The sensor data is represented as a sensor message of ROS .

- Once the sensor message and transformation message, which represents the transform between two coordinate frames in free space (Saito, 2016), are published, another node generates the map for the navigation.
- Having the map, a planner and action node use specific messages to compute a trajectory, and then, motion control can be applied.
- A motor drive node subscribes to velocity messages, broken into linear and angular parts. Motors commands are generated, published through the communication node and consumed in the low level control for the motors' actuation.

3 ASBG PROJECT

The ASBGo started in 2011 when the first smart walker of the group born (Martins, 2011). Since then, several prototypes were developed in order to achieve a motorized assistive device capable of improve the physical condition of real patients (see figure 17) (Martins, 2016; Alves, 2016; Caetano, 2017).

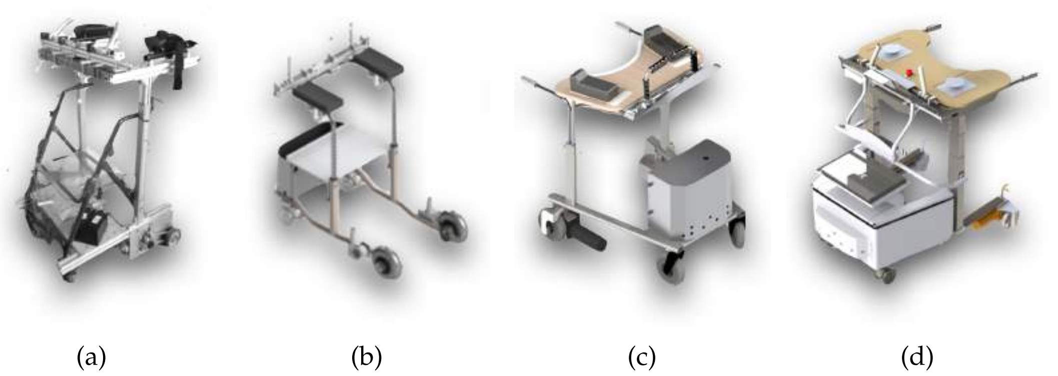


Figure 17: Evolution of ASBGo prototypes: (a) Prototype I; (b) Prototype II; (c) ASBGo III ; (d) ASBGo++ .

This chapter introduces the ASBGo , highlighting the main goals of the group project and the prototypes that resulted from the continuous research. Note that each prototype has a distinguish nomenclature and so, it is advised to memorize their names in order to avoid getting lost during the explanation of the project.

3.1 WHAT? TO WHAT? AND HOW?

The easiest way to understand a project is responding to three simple questions: *"What? To what? and How?"*. The *"What?"* is the main goal of a project and can be defined as its mission. *"To what?"* defines the future application context, which means, the vision of the project. The resources used to accomplish the project answer to the question *"How?"* and are defined as its values.

Following the previous logic, it can be affirmed that the ASBGo was born with the aim of create a four-wheeled motorized smart walker to define a new rehabilitation concept. Through a multidisciplinary team that involves partnerships between UM , Orthos XXI and Hospital de Braga, the group intents to improve the quality of life of patients with motor and balance disorders.

As previous referred to in section 1.1 - *Motivations, scope and problem statement*, the ASBG decided to develop a motorized smart walker capable of adapting to the needs of its users. The desire to engineer an assisted device that can promote physical rehabilitation and, on the other hand, retard the loss of motor function, is related to the increasing number of people with motor disorders caused by age and/or specific pathologies, such as ataxia. In order to accomplish it, the group kept the focus in the main characteristics presented in a row, whom distinguish the ASBG smart walkers from the other walkers commonly used as assistive devices:

1. Present a locomotion with constant linear velocity, which allows the users to conduct their efforts only in the cyclic gait training and not in the movement of the walkers (Martins et al., 2015);
2. Reflex the user's intentions in their behavior since they can be driven through a handlebar with sensors, promoting multitasking gait training and user's independence (Martins et al., 2013b, 2014d);
3. Present high physical support provided by a table which can sustain the major part of the user's weight, increasing the secure sensation and reducing the fall risk (Rodrigues, 2014);
4. Integrate a set of sensors that allow to monitor the users' gait performance and assess the surrounding environment, giving biofeedback to the patient and enabling autonomous behaviour in risk situations (Martins, 2016).

3.2 asbgo prototypes

In order to completely understand the ASBGo and, in specific, each one of its prototypes, it was revised chronologically (see figure 18) the improvements made in each version, summarized the relevant characteristics that distinguish them from each other and research topics that the group has covered in the past years.

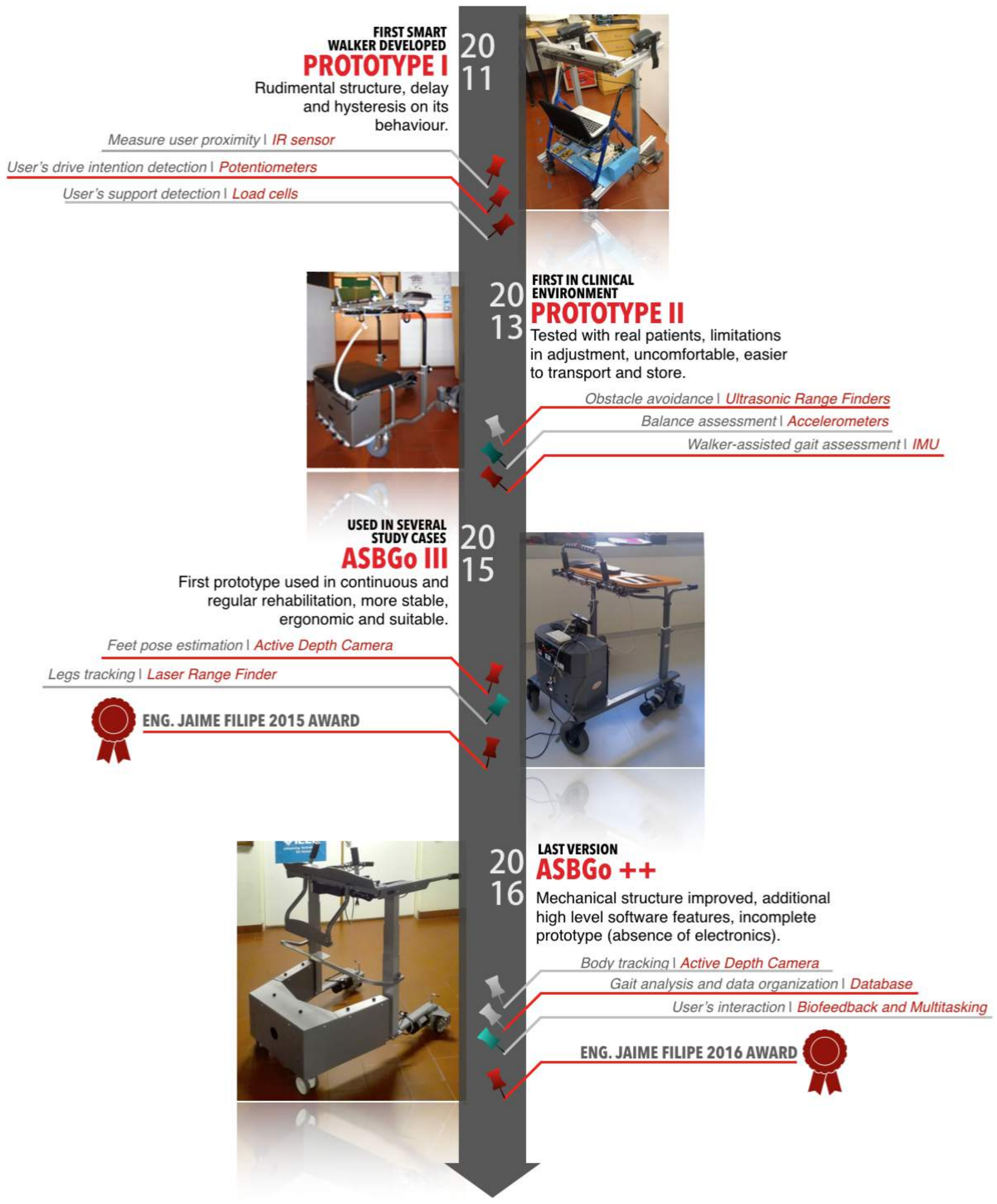

FIRST SMART WALKER DEVELOPED
PROTOTYPE I
20 11
Rudimental structure, delay and hysteresis on its behaviour.
Measure user proximity | IR sensor
User's drive intention detection | Potentiometers
User's support detection | Load cells
20 13
FIRST IN CLINICAL ENVIRONMENT
PROTOTYPE II
Tested with real patients, limitations in adjustment, uncomfortable, easier to transport and store.
Obstacle avoidance | Ultrasonic Range Finders
Balance assessment | Accelerometers
Walker-assisted gait assessment | IMU
USED IN SEVERAL STUDY CASES
ASBGo III
20 15
First prototype used in continuous and regular rehabilitation, more stable, ergonomic and suitable.
Feet pose estimation | Active Depth Camera
Legs tracking | Laser Range Finder
ENG. JAIME FILIPE 2015 AWARD
20 16
LAST VERSION
ASBGo ++
Mechanical structure improved, additional high level software features, incomplete prototype (absence of electronics).
Body tracking | Active Depth Camera
Gait analysis and data organization | Database
User's interaction | Biofeedback and Multitasking
ENG. JAIME FILIPE 2016 AWARD

Figure 18: Chronological representation of the ASBGo research work (Rodrigues, 2014; Martins, 2016; Alves, 2016; Caetano, 2017).

3.2.1 *Prototype I*

In 2011, a conventional four-wheeled walker, not motorized, supplied by Orthos XXI was adapted and transformed in the first prototype of the group. The *Prototype I* was developed only as a proof-of-concept with the intention to verify the requirements and functionalities that a smart walker should embrace. Thus, its heavy and rudimental structure, made of iron materials, is not detailed in this summary (Martins, 2016).

The main functionalities of this prototype were: the detection of user's drive intentions through an angular potentiometer and a joystick settled in the handlebar; the locomotion of the robot, through the use of two motors and a system of pulleys and belts for each wheel, according to user's commands; the measurement of the user's proximity through an *Infrared Distance Sensor (IR)* sensor placed bellow the handlebar pointed to the user; and the measurement of the user's support in the walker obtained by two load cells placed in each one of the forearm-support (Martins, 2016, 2011; Martins et al., 2014c).

It should be noticed that the strategy of driven control used did not satisfied the group, since the behaviour of the robot presented hysteresis and delay. Moreover, the information collected from the IR sensor and from the load cells was not electronically linked to the robot system or between each other. Although it could be acquired, it required post signal processing, and there was no way to cross-linking data, restriction that is not relevant in an only proof-of-concept context.

3.2.2 *Prototype II*

The second prototype was designed mainly to rectify the mechanical, structural and ergonomic problems of the *Prototype I*. Some requirements, as height adjustment, the inclusion of a harness support, more comfortable forearm-supports and an improved design, allowed the inclusion of the device in clinical environment, where it was tested by some patients (Martins, 2016).

In clinical environment, the biggest constraint pointed to the device was the lack sense of confidence and safety by the users. Moreover, the handicap in the individual size adjustment decrease the usability of the walker. Thus, the improved mechanical structure of *Prototype II* was not enough to give the patients the reliance and the confort required to undergo gait training with a motorized smart walker (Martins, 2016). However, the potentiality shown by this prototype as an assisted recover device prompt the continuity research work in the ASBGo .

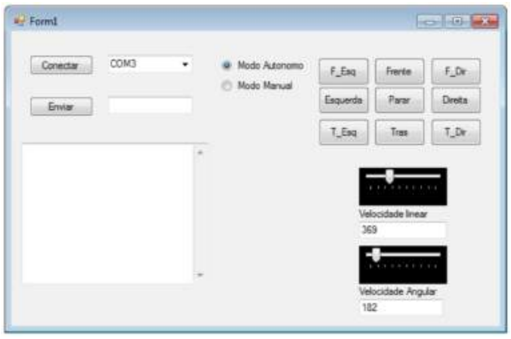
In terms of electronics, the motorized locomotion system remained the same but it was directly coupled to the wheels of the walker and so, using a direct connection it was possible to increase the torque of the wheels and save space. The joystick was replaced by a linear potentiometer, improving the driven control strategy and the behaviour of the device (Martins et al., 2014d, 2013a). The insertion of *Ultrasonic Range Finders (URFs)* allowed to conduct the first proof-of-concept studies in obstacle avoidance (Faria et al., 2014). Moreover, the user's balance assessment was initialized through the placement of accelerometers in the patient's body, and a more complete walker-assisted gait assessment method arisen from the use of IMU s in the patient undergo rehabilitation (Tereso et al., 2014; Martins et al., 2014a).

3.2.3 *ASBGo III*

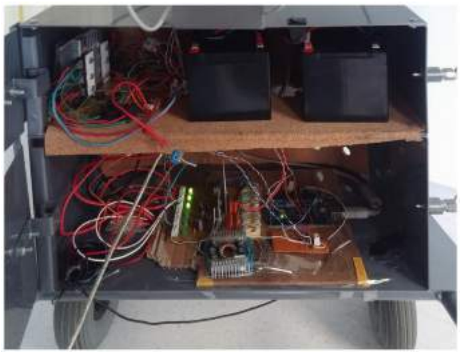
The ASBGo III may be considered the most important prototype of the group since it gathered the main characteristics to be used in continuous clinical rehabilitation, as happened over the past years (Martins et al., 2015).

The ASBGo III is recognized by a solid mechanical structure, suitable to majority of the patient's size, which includes a wooden table to provide support and safety to the patient. It presents also a more attractive design and despite its poor and academic electronic solutions (see figure 19b (Caetano, 2017)), its behaviour follows the intentions of the user that are captured by a new handlebar. This new handlebar also uses two potentiometers (an angular and a linear as well as *Prototype II*) but it is characterized by an improved design which eases driving of the device. The locomotion system is also the same of *Prototype II* but, the use of an *Arduino* to interface the potentiometers and the motors' drivers allowed the development of an HMI to control the device remotely. However, it should be noticed the simplicity of this HMI once it consisted in an *executable file* whose functionalities were only define the velocities and send the directions for the robot's locomotion. Also note that the velocities are not defined in appropriated units (see figure 19a) (Caetano, 2017). Another usability restriction is related to the charge of the device. As it can be visualized in figure 19c, the charging process makes use of an external setup that can jeopardize the safety of the individual that is manipulating the system.

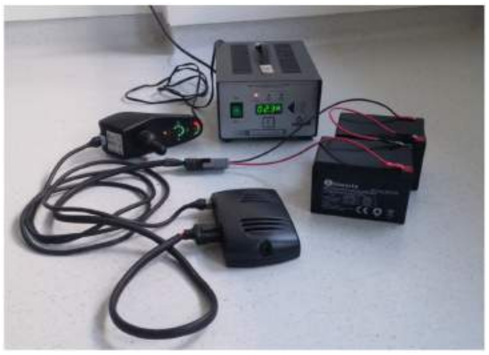
Aside from its use in clinical environment, the ASBGo III enabled the expansion of research work in feet pose estimation and legs tracking making use of an active depth camera and a laser range finder, respectively (Page et al., 2015; Martins et al., 2014b). Although very interesting algorithms have been developed, their employment is conditioned by the post processing data required, which restricts its use. Moreover, as previously happened with the sensors integrated in *Prototype II*, due to the lack of a software architecture, there is no way to cross-linking data and use the information of the several sensors together (Caetano, 2017).



(a)



(b)



(c)

Figure 19: Details of ASBGo III : (a) HMI ; (b) Electronics; (c) Power supply charging.

3.2.4 *ASBGo++*

The ASBGo++ is the last version of the group and, although not finished, worked as the basis for this master thesis. In its most recent research, the ASBGo worked in the main goals presented in a row (Caetano, 2017; Alves, 2016):

1. Review of all the functionalities of *Prototype I*, *Prototype II* and ASBGo III , summarising the sensors or devices required by each feature, respectively;

2. Placement in clinical environment and follow up several cases studies in rehabilitation with ASBGo III , promoting the communication between ASBG members, clinical team and patients;

3. Identification of electromechanical and software constraints of the ASBGo III , settling the main requirements to produce a new improved prototype: the ASBGo++ ;

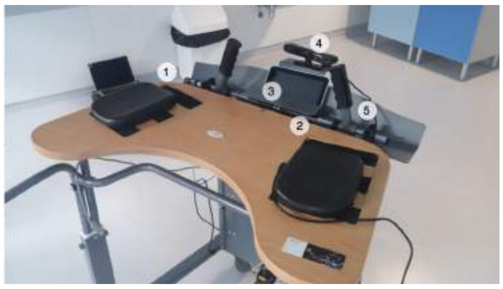
4. Design of the mechanical structure of ASBGo++ and manufacture in enterprise environment;

5. Development of high level applications that can be used by the medical team in order to potentiate the use of the ASBGo++ in gait rehabilitation;

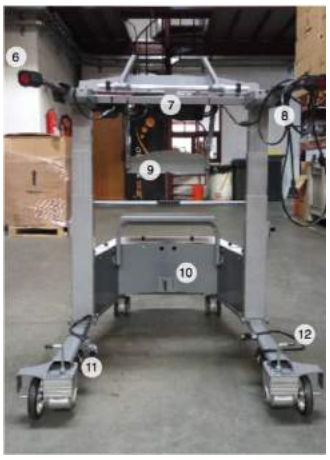
6. Validation, in clinical environment, of the new ASBGo++ handlebar and of some applications developed, such as: lower limbs monitoring and biofeedback; database of spatiotemporal gait parameters; and multitasking and reaction time measurement.

The above list shows that the ASBGo++ is defined by an improved mechanical structure and additional high level software features (Alves, 2016; Caetano, 2017). However, the ASBGo++ is an unfinished prototype due to the absence of electronics as well as lack of a defined system architecture (note that the development of electronics was not enumerated above and neither any of the previous prototypes have a system architecture).

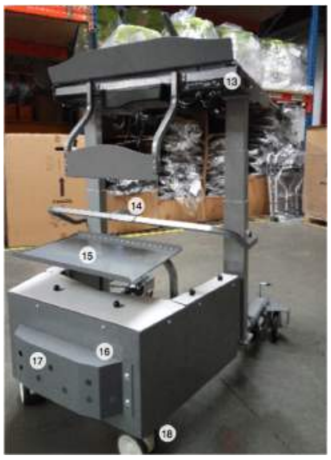
The electrical and software requirements of ASBGo++ included the development of a system architecture, embedding all the sensors used in previous prototypes and the development of a modular software architecture in order to engineer a unified device that correlate three main fields: hardware, low-level and hight-level software (Caetano, 2017). Thus, this master thesis arises to continue the work of this unfinished version, fulfilling the electronic and electrical requirements previously identified and making use of the mechanical structure of ASBGo++ (figure 20) to assemble a new prototype, which will be called ASBGo* .



(a)



(b)



(c)

Figure 20: ASBGo++ : (a) new handlebar mounted in ASBGo III in order to validate its use undergo gait training; (b) Posterior view; (c) Lateral view; (number description available in Table 1) (Caetano, 2017).

Table 1: Mechanical supports and positions of the ASBGo++ 's hardware components.

N°	Description	N°	Description
1	Angular potentiometer	10	Laser
2	Linear potentiometer	11	*DC* Motor
3	Display	12	Safety brake
4	Camera support number 1	13	Height adjustment system
5	Handlebar (support and driving)	14	Camera support number 3
6	Handle grip (only for rear support)	15	External PC support
7	Infrared sensor	16	Enclosure (electronics)
8	Load cells	17	Ultrasonic range finders
9	Camera support number 2	18	Caster Wheels

4 ASBGo* SMART WALKER: THE NEW PROTOTYPE

To tackle unreliability, proof-of-concept implementations and architecturally poor systems of all the previous prototypes, the ASBGo* arises. This new assistive device takes the ASBGo to a new level in a way that future development follows now a bright path. The good architectural design of the system and the solid software base developed eases, to future developers, the task of implement, add, test and include new features to enrich the new device.

Besides its robustness, the new prototype was developed to be used in clinical environment and so, the easy usage and the accessibility were always a priority. That being said, the ASBGo* is ready to be integrated in medical environment to help patients to improve from their motor impairment and so, helping patients to recover the joy and the confidence in their locomotion. The value of the device is also on the fact that it supports the medical team during the rehabilitation process.

To achieve this new prototype, all the previous versions were studied and the considerations addressed in the last version (the ASBGo++) were also taken into account. From here, the new architecture started to be outlined. Gathering the functional requirements for the new device, a conceptual model in blocks was delineated to, in the end, obtain an idea and an overview of the global system. Being, at first sight, the behavior and the functionalities analyzed and defined, the necessary material was explored, specified and purchased. This is considered a crucial phase where the most simple cable until the powerful processing unit must be reviewed in detail in order to ensure compatibilities and performance so that, in the long run, everything fits perfectly. Following a co-design method, hardware and software for the ASBGo* was developed. Respecting the requirements defined, the proper electronic and electric circuits were created and various software components, from a more low-level to a more high-level layer, were also implemented. The software architecture is built using ROS , a robotic software platform that given its generic nature, its tools, its inference on modular development, its documentation, its message-passing infrastructure and its positive evolution, revealed to be an asset for the new system. Also, during the development of the system's architecture, mechanical modifications, that were not predicted, were performed (presented in Chapter 5, subsection 5.1.1).

4.1 FUNCTIONAL REQUIREMENTS OF THE SYSTEM

From the study of the first three prototypes and given the considerations defined in the last version, the ASBGo++ , it resulted the following functional requirements. So, the system must:

1 Acquire data from a set of sensors in real-time. Updated information of the sensors must be disposed, in specific units, in a high-level layer and must be easily accessed in order to be developed/explored algorithms and software features;

2. Monitor the battery's voltage;

3. Freeze in case of critical battery;

4. Stop in case of emergency;

5. Move multi-directionally, more specifically, in eight directions;

6. Allow its locomotion through the use of the handlebar or the remote controller;

7. Control the velocity of its locomotion using a control algorithm;

8. Detect some possible errors (e.g. processing unit failure, motors errors);

9. Provide a local graphical user interface allowing the configuration and the use of the device (e.g. perform rehabilitation sessions);

10. Present an embedded database containing the personal information of patients as well as information about their sessions;

11. Store the information of the database in a personal cloud, enabling the access of patients' data through the personal desktops of the medical team;

12. Present an interactive game that allows multitasking gait training and detects the reaction time of the patients undergo rehabilitation;

13. Assess patient's gait and posture, providing biofeedback;

14. Detect the distance between the device and the patient;

15. Monitor the support of the patients' forearms in the smart walker's support table;

16. Compute and show the patient's balance;

17. Present a safety mode for the locomotion.

18. Record videos of the patients' gait and posture during a session;

19. Record several spatiotemporal parameters of a patient during a session;

20. Calculate its traveled distance;

21. Navigate autonomously to a point localized in a beforehand created map avoiding possible obstacles;

22. Present a shared control navigation, where the drive intentions of the users and the autonomous intentions of the system are taken into account to compute a final locomotion command.

Besides the definition of several functionalities for the system, this thesis have kept its focus in the complete implementation of the first nine requirements (colored), which can be defined as the main ones, granting to the ASBGo* the ability to be used in rehabilitation sessions and to be further improved. However, it should be noticed that the sensors' data acquisition component, from point 1, includes all the sensors, even those that are not currently in use (IR sensor, URFs , load cells). All the sensors' information is disposed, in its specific units, in a high level layer and so, ready to be used in high level algorithms. For example, the distance between the walker and the user is acquired by an IR sensor and its value, converted in centimeters, is dispose, at a certain frequency, in the *sensors_values* and *infrared_values* topics of ROS . This information can be used to detect the presence of a patient and then be included in a safety control strategy. Likewise, the distance measured by each *Ultrasonic Range Finder (URF)* is converted in centimeters and it is published in the *sensors_values* and *urf_values* ROS topics created, which simplifies the implementation of a future autonomous navigation algorithm. So, due to the methodology used, which covers the complete development of electronics for all the sensors' data acquisition, the good architectural design of the system and the solid software base, created the necessary resources to implement all the other requirements listed above and even to expand the list.

4.2 SYSTEM OVERVIEW

Being the functional requirements specified, a conceptual model of the new system was created. Figure 21 shows the resulted system's architecture overview, revealing the different components of the ASBGo* (ElekTRICKS, 2016; Alves, 2016; SuportLaser, 2017; M, 2017; Beetronics, 2017; RobotShop, 2017; BestApplePrice, 2017; Millsaps, 2015).

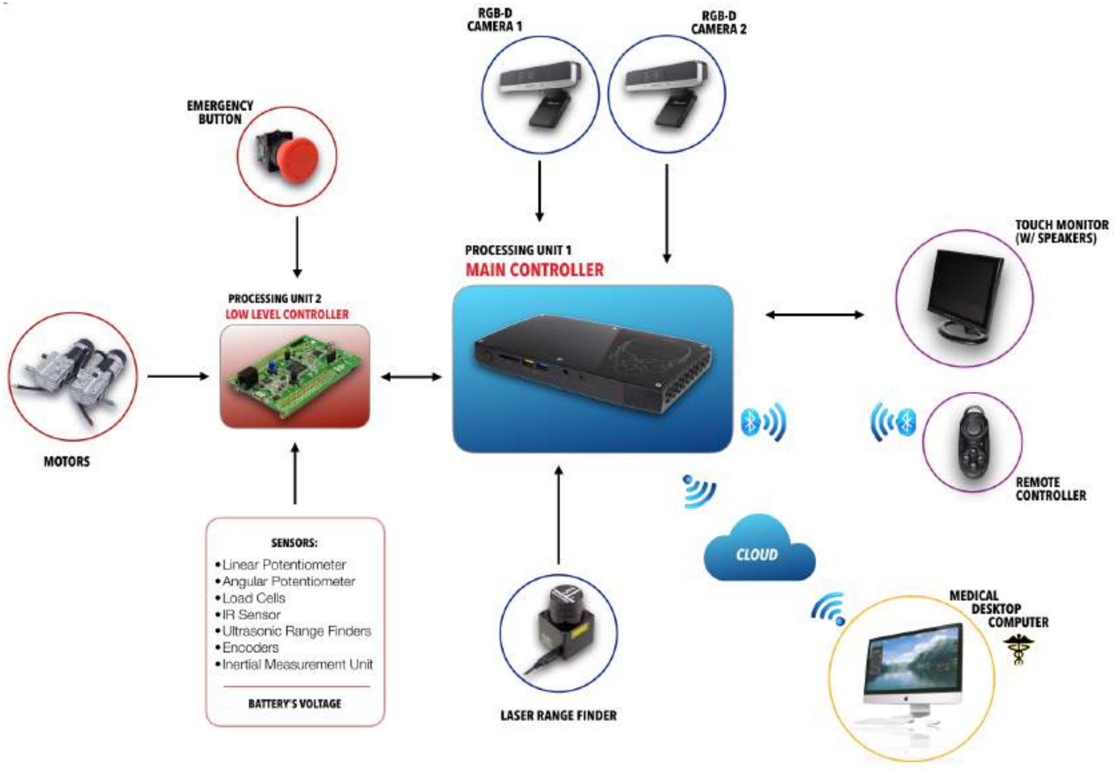


Figure 21: ASBGo* system overview.

The system is composed by two processing units. The first one, *Processing Unit 1* in figure 21, is considered the *Main Controller* of the system. Empowered with a great processing capacity, this unit will perform heavy computation and in it, is where the high-level software is developed and runs. That software is built using ROS and so, the OS installed in the *Main Controller* is the *Ubuntu 14.04* distribution of the Linux OS . The choice of a powerful on-board computer also allows to develop software, run simulations and use computationally intensive programs and tools easing the work to future developers. Since that there is the idea of creating a personal *cloud* for the ASBGo* , the *Main Controller* must be able to connect to the network through *WiFi*.

The second processing unit, *Processing Unit 2*, is considered the *Low-Level Controller*. It consist in a development board and is where the more low-level software is developed. This software enables the access to the sensors, motors and other components.

The two RGB-D cameras are required for the functionalities that will perform computer vision algorithms and the laser range finder, which scans the environment, will be required for the autonomous navigation.

Both the touch monitor and the remote controller are points of interface between the users and the system. With a touch monitor, the interaction is more practical and interactive and by presenting embedded speakers, it allows to develop features that can make use of sound. Also, being the monitor coupled to the walker, it is possible for the developer to edit, run and test new applications directly on the device. The idea for the remote controller, besides presenting buttons, is to contain a small joystick allowing to drive the ASBGo* .

Bearing in mind the functional requirements proposed to be fully implemented in this thesis, it results two main blocks: *Real-time sensors data acquisition and locomotion* and *Human-Machine interface (HMI)*.

4.2.1 *Real-time sensors data acquisition and locomotion*

The smart walker collects and processes information from a series of embedded sensors allowing it to understand the surrounding environment, present an autonomous behavior and act accordingly. As it can be seen in figure 22, the development board (*Low-Level Controller*) is used for the low-level control, i.e., sensors data acquisition, data's management and motors actuation. Besides that, it handles the emergency events and monitors the battery's voltage also checking if it is in critical state (discharged). It can be advanced the use of a *Real-time operating system (RTOS)* , the *FreeRTOS*, that leads to a modular and more manageable low-level application running in multitasking, with synchronism and protection against data corruption.

Through a serial communication, the *Main Controller* and the *Low-Level Controller* exchange data that can consists in, for example, sensors' data, commands and so on. A principle defined for this block is that the data acquired from the sensors is sent to the *Main Controller* in order to be handled in the high-level. With this modular approach, the *Low-Level Controller* gets abstracted and is only known, for future developers, as a component with certain inputs and outputs. As to the smart walker's locomotion, a control strategy to get the desire velocity for the walker must be included.

Apart from the software, all the electronic circuits for the acquisitions are a must in this conceptual block.

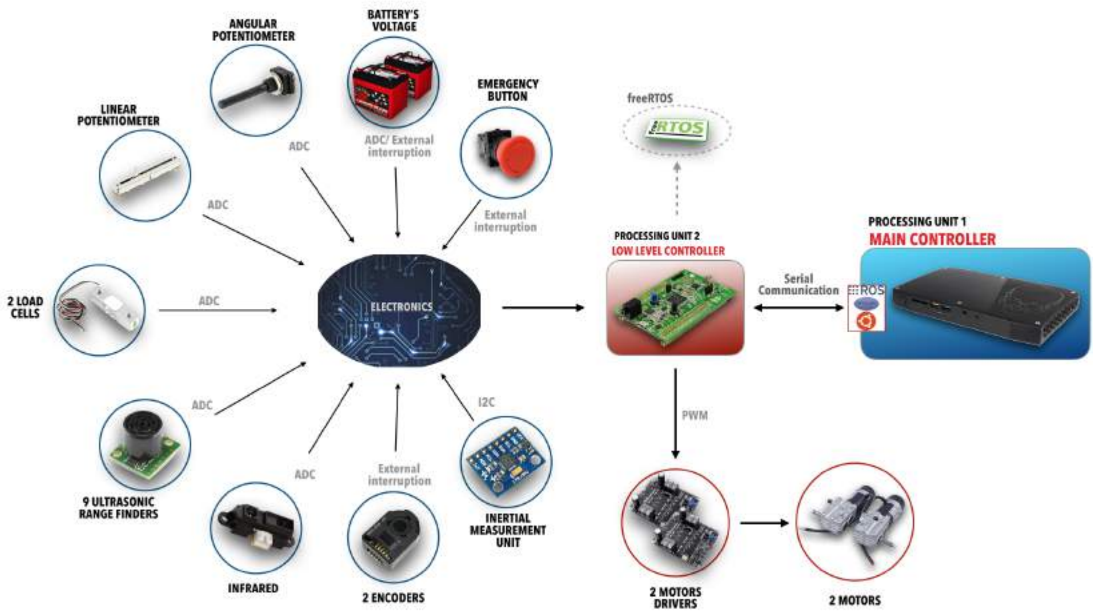


Figure 22: Real-time sensors data acquisition and locomotion block (LXbattery, 2017; ColdFireElectronica, 2017; Electronics, 2017; Technologies, 2017; Robotics, 2017; Elektronik, 2017; Solarbotics, 2017; FindIC, 2017a; Inc, 2016).

4.2.2 *Human machine interface*

In order to be easily used in the medical environment by medical specialists, the interaction with the system must exist and must abstract all the technical concepts (software and electronic concepts). Besides that, it is intended the creation of intuitive and user-friendly interfaces to avoid confusions and disruptions but rather offer a pleasant, simple and attractive way of use the smart walker (see figure 23).

The local *Graphical User Interface (GUI)* must provide means to configure parameters related to the locomotion, perform rehabilitation sessions, check the state of the device and configure and run application-level features like the biofeedback or the interactive game.

As to the remote controller, if the patient is not able to drive the device using the handlebar, the medical specialist can drive it remotely. Also, the buttons may be convenient to start or stop the system.

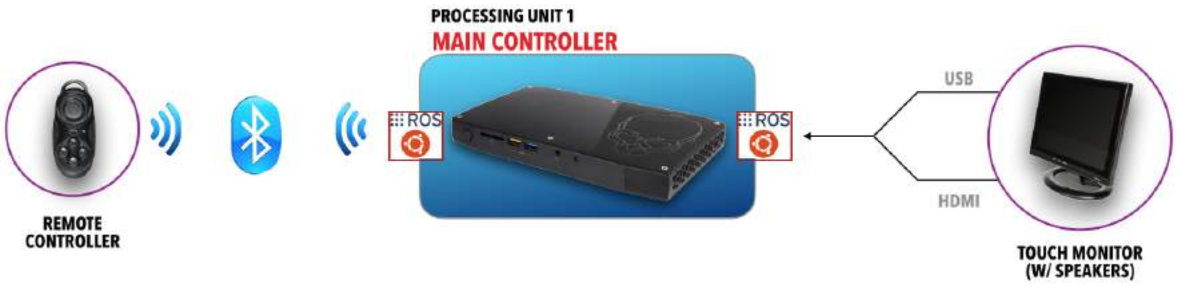


Figure 23: Human machine interface block.

4.3 HARDWARE AND ELECTROMECHANICAL DEVELOPMENT

In this section it is detailed all the process of physical development *i.e* in terms of material specification and electric and electronic circuits development.

4.3.1 *Material specification*

In order to build the new system, many material has to be used and so, some had to be re-used and some purchased. This way, it will now be presented, in categories, the material gathered and some details about it, its purpose in the system and relevant commentaries will be mentioned. The categories are: *sensors*, *processing units*, *actuators*, *HMI* , *power supply and safety systems* and *cables*. Note that not all the material will be presented in here making sense to present, for example, the electronic components in the electronic circuits development and some structural components in the mechanical modifications.

4.3.1.1 *Sensors*

LINEAR POTENTIOMETER

A simple 100kΩ slide potentiometer was acquired. It presents a sliding movement and it is coupled in the handlebar of the smart walker allowing to change the direction of the walker's locomotion. The output, in voltage, is proportional to the position of the slider's handle.

Figure 24: Linear potentiometer.

ANGULAR POTENTIOMETER

The angular potentiometer acquired presents an ohmic value of 470kΩ. In this case, the output value is proportional to the angle between the wiper and its original position. It is coupled in the handlebar and allows to detect if it is to drive forward or not.

Figure 25: Angular potentiometer.

INFRARED SENSOR

To detect the distance between the patient and the smart walker, a *Sharp GP2Y0A21* was collected. IR sensors detect possible objects by the emission of IR radiation and by receiving that light which is reflected by the object. The output depends on the angle measured. The detection range of the *Sharp GP2Y0A21* is approximately 10 to 80 centimeters and the output of the sensor consists in a simple analog voltage that depends on the distance measured. It is advised by the manufacturer to put a 10 μF capacitor (or larger) across power and ground close to the sensor to stabilize the power supply.

Figure 26: IR sensor.

LOAD CELLS

Two micro load cells, measuring forces from 0 up to 50 kilograms, were gathered. The load cell converts the load on it into very small electrical signals. It contains a Wheatstone bridge configuration constituted by strain gauges where,

Figure 27: Load cell.

by applying a force in the load cell, the strain gauge is deformed, its electrical resistance changes proportionally and the output voltage changes. This single-point load cells only get stressed if a force is applied to one of the ends. The two outputs allows to get the voltage between the strain gauge terminals but, due to its small value (dozens of mV), it must be amplified. The loads cells acquired are the *CZL635* from *Phidgets*.

INERTIAL MEASUREMENT UNIT

The new ASBGo* counts with one *GY-521 MPU6050* from *InvenSense*. This IMU contains both a 3-axis gyroscope and a 3-axis accelerometer allowing measurements of both independently. It uses the *Inter-Integrated Circuit (I2C)* protocol for communication and it is powered with a voltage ranging from 3.3V to 5V. Its purpose on the new device is to acquire inertial data from the patients to assess their balance or their fall risk.

Figure 28: IMU .

ENCODERS

Figure 29: Encoder.

Two encoders are now coupled in the device, one in each rear wheel. The sensors are the *HEDS-5500* from *Avago* and consist in a three channel optical incremental encoders. The sensor utilizes a film codewheel allowing resolutions of 500 counts per revolution. However, due to the two relevant outputs, which are two square waves in quadrature, the resolution quadruples because it is possible to detect a rising edge transition and a falling edge transition for each wave. The encoders are used to calculate the traveled distance and the velocity of the wheels.

ULTRASONIC RANGE FINDERS

The *LV-MaxSonar-EZ* URFs acquired, use high frequency sound to detect objects in a range of 0 to approximately 6 meters. The URF provides different ways of output like: analog output, by serial communication, or by a pulse width. It also reserves a strategy that allows to start an acquisition whenever

Figure 30: URF .

it is wanted using a pulse and, it outputs a pulse when an acquisition is over. This methodology is an asset because the nine URF present in the new ASBGo* are disposed in such a way that if the acquisition of each started at the same time there would be many interferences. So, using the start pulses, it is possible to acquire the information of the URFs in a looping chain, i.e., one acquires and when it finishes, it commands others to start and so on.

RGB-D CAMERA

Figure 31: RGB-D camera.

The new *Intel RealSense F200 Camera* wraps several components like: color camera, depth camera with IR light source and a microphone array. It allows to perform face analysis, detection and tracking, hand and finger tracking, gesture recognition and speech recognition. With this kind of technology, numerous features for the ASBGo* can be explored and be easily integrated in the system. One example is the use of one camera for the assessment of the patients' gait and another for the posture's assessment by the use of computer vision algorithms.

LASER RANGE FINDER

For future implementations of autonomous navigation features, the *Hokuyo URG-04LX-UG01 Scanning Laser Rangefinder* is included. Consists in a small and accurate laser scanner with a detectable range of 200 millimeters to 5.6 meters. Also, it scans in a 240°area with a 0.36°angular resolution and a scanning time of 100ms/scan.

Figure 32: Laser range finder.

4.3.1.2 *Processing units*

MAIN CONTROLLER

The on-board computer, also known as *Main Controller*, presented on the ASBGo* can be considered the most valuable acquisition for the new assistive device. All the development, simulations and research can be made in this computer and, in it, is where this developed applications run. Parameters such as the size, the consumption, the performance and the peripherals, were all taken into account and resulted in the selection of the powerful *Intel NUC6i7KYK* also known as *Skull Canyon*.

Figure 33: *Main Controller*.

The mini-PC, with a *6th generation Intel Quad-Core i7-6770HQ* 2.6 to 3.5 GHz processor and a *Intel Iris Pro Graphics*, unleashes maximized performance for intense computation and intensive workloads. It can be experienced cutting-edge performance from the start with fast boot, and multitasking that is effortless whether moving between application, files, or web sites. It supports up to 32GB of RAM and contains two M.2 slots with support for a 42 or 80 millimeters SATA or PCIe SSD. It includes four USB 3.0 ports, SD card slot, one headphone/microphone jack, one mini DisplayPort and an HDMI 2.0 display port. Also, *NUC6i7KYK* comes with *Intel 802.11ac wireless*, the fastest wireless of *Intel* available today along with built-in *Bluetooth 4.2*. As to the eletrical carateristics, this computer is powered with a voltage of 19V and presents a consumption of 120W.

Since the mini-PC comes with no RAM and no storage unit, a *Crucial 8GB DDR4 2133 MT/s* RAM and a *250GB Samsung 960 EVO Series M.2* SSD were selected. Also, to expand the USB ports of the *Intel NUC6i7KYK* the *Manhattan 7-Port USB 2.0 HUB* was acquired and, to ease the work of the developer, the *Logitech MK270*, a wireless keyboard and mouse, was purchased.

Figure 34: *Low-Level Controller*.

LOW LEVEL CONTROLLER

The *Low Level Controller* of the ASBGo* system is developed in the *STM32F4-Discovery* development board, a low-cost and low-power development board based on a high-performace microcontroller, the *STM32F407VGT*. The board was elected due to the significant performance capabilities and due to the multiple channels that peripherals offer. It includes a *ST-LINK/V2* embedded debug tool, one *ST-MEMS* digital accelerometer, a digital microphone, one audio digital-to-analog converter with integrated speaker driver, LEDs, two push-buttons, one for reset and one for the user, and a USB On-The-Go micro-AB connector.

The powerful *STM32F407VGT* microcontroller features a 32-bit *ARM Cortex-M4* RISC (Reduced Instruction Set Computer) core operating at a frequency of up to 168 MHz. It features a floating point unit (FPU) single precision, a full set of DSP (Digital Signal Processor) instructions, memory protection unit (MPU) which enhances application security and incorporates high-speed embedded memories. Among that, the device also offers a three 12-bit analog-to-digital with multiple channels, two digital-to-analog converters, twelve general-purpose 16-bit timers including two *Pulse-Width Modulation (PWM)* timers for motor control and two general-purpose 32-bit timers. It includes up to fifteen communication interfaces like I2C , *Universal Asynchronous Receiver-Transmitter (UART)* , *Serial Peripheral Interface (SPI)* and so on.

usb-to uart serial converter

The communication between the two controllers consists in a serial communication. This way, the FT232 USB UART Board (Type A) was acquired. It is an accessory board that features a USB-to-UART serial converter FT232 and a USB Type A connector. It is possible to select the output voltage level as 5V or 3.3V.

Figure 35: USB-to-UART serial converter (WAVESHARE, 2017).

4.3.1.3 *Actuators*

motors

The actuators in the ASBGo* are two 24V, 100W brushed motors with angular velocities of 4600 *Revolutions Per Minute (RPM)* for each rear wheel. However, each motor contains a gearbox with a ratio of 19:1 and so, the actual angular velocity is approximately 242 RPM . Also, the motors come with an electromechanical brake system. Besides its 1.8A of nominal current, each motor, depending on the load and the velocity, can drive currents of about 10A.

Figure 36: DC brushed motors.

Figure 37: Motors driver.

MOTORS DRIVERS

To drive each motor, two *Cytron 30A 5-30V Single Brushed DC Motor Driver* were purchased. It allows bi-directional control, has a maximum peak current of 80A (1 second) and supports 30A continuously. As to the input voltage, it ranges from 5V to 30V. The drivers also have an input for the PWM with a logic level input of 3.3V and 5V.

4.3.1.4 *Human machine interface*

TOUCHSCREEN MONITOR

To allow the interface between the user and the smart walker, a 12 inches touchscreen monitor from *Beetronics* was purchased. It presents a strong casing, a LED panel with an aspect ratio of 4:3 and presents HDMI, VGA and RGA connections and USB for the touch functionality. Besides that, it ensures compatibility with *Linux* and the drivers are easy to get. The monitor presents a configurable menu with many options and, in terms of electric specification, it is powered with 12V, it has a consumption of less than 15W and so, a maximum current of approximately 2A.

Figure 38: Touchscreen monitor.

Figure 39: Remote controller.

REMOTE CONTROLLER

To control the ASBGo* remotely, a remote gamepad from *Elegiant* was acquired. It is small and lightweight, convenient to carry and the buttons and the joystick are responsive and not sticky. It is compatible with *Linux* and works with *Bluetooth* wireless technology. Being a low power device, the battery lasts many hours and it takes about 30 minutes to charge by using a Micro USB cable. The *Bluetooth* connection seems stable up until a distance of 10 meters.

4.3.1.5 *Power supply and safety systems*

batteries

The power supply source for the new device was a target of many study. Since the higher voltage required for the system is 24V for the motors, two 12V rechargeable *AGM DEEP-CYCLE sealed silicon* batteries from *Zenith* were gathered and are connected in series. Due to the consumptions of the system (the 6A from the on-board computer, the 2A of the touchscreen monitor and the possible 10A of each motor) the batteries acquired present a capacity of 36Ah meaning that they ensure, approximately, 36A during an hour. This way, the battery of ASBGo* can last, in the medical environment, about 1 or 2 days before the need to be charged.

Figure 40: 12V battery.

step-down modules

Since the battery's voltage is about 24V, step-down modules were purchased in order to get a lower and stable voltage to power other components of the system. Five modules of *DROK* were acquired to get voltages of 19V, 12V, 5V, 3.3V and a stable 24V value.

Figure 41: Step-down (Amazon, 2017).

relays

Two types of relays were collected: a thermal relay and normally open relays with electromagnetic operation.

For the first type, it was acquired one *W54-XB1A4A10-30*. Its purpose is to protect the entire system and so, it breaks the circuit when there is current superior to 30A in the main net of the electric system. This value was chosen accordingly to the consumption of the entire system. Also, this relay presents a button for visual indication.

Figure 42: Thermal relay (FindIC, 2017b).

For the second type of relays, it was acquired one *RLAC/4-24* from *Nagares*, to power ON or shut down the entire system when a switch is pressed, and two *RLP/5-24*, also from *Nagares*, used to open the power circuit of the motors when the emergency button is pressed. The *RLAC/4-24* handles 40A and the *RLP/5-24* handles 22A.

(a)

(b)

Figure 43: Relays (Nagares, 2017a,b).

emergency button

A simple normally closed emergency button was selected. When pressed, it opens the circuit.

Figure 44: Emergency button.

4.3.1.6 *Cables*

blue and brown wire

For the connections of the electric system, it was set the use of blue wire where there is positive voltage and brown wire for the ground. The wire section was taken into account due to the possible values of current.

three-wire and four-wire cables

Cable with three and four wires were collected to allow connections to the different components (e.g. sensors) that have three and four pins, respectively. This avoids the existence of many loose wires that can lead to a messy and confuse electric system. Also, with all the specific wires of a sensor gathered in a cable, the accessibility to specific wires gets easier.

four-wire coiled cable

A specific coiled cable was acquired to be used with the IMU . Since this sensor will be on the patient's body, the connection between the sensor and the system must be stretchable in order to keep the patient comfortable. The cable has four wires corresponding to the number of used pins in the IMU .

ethernet cables

For the connection of each one of the nine URFs , it was used ethernet cables to take advantage of the number of wires in them. Six pins of each URF are utilized and so, using ethernet cables avoids take to much space in the URFs ' acquisition board also easing the connections to the board. Note that the cables are only used to take advantage of the number of wires and so, ethernet protocols are not being used.

hdmi

For the transmission of audio and video to the touchscreen monitor, an HDMI cable was purchased.

4.3.2 *Electric system*

Figure 45 represents the global eletric system of the ASBGo* . Some relevant blocks are highlighted and will be detailed.

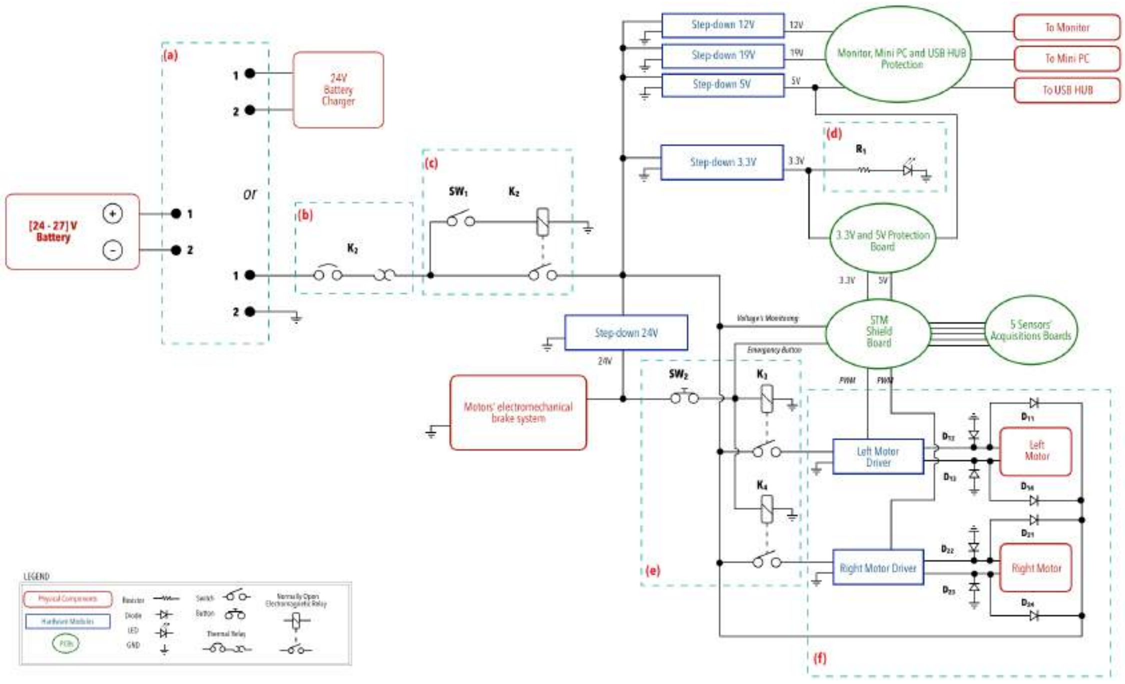


Figure 45: ASBGo* electric system.

Block *(a)* intends to show that the 24V power source of the new device can be connected (manual connection using a plug) to a 24V battery charger or to the ASBGo* system. It was set this approach because this way, only one of the options can be chosen (connect or the charger or the system to the battery) and so, it is impossible to make the mistake of charge and use the battery to power the system simultaneously.

Block *(b)* wraps the thermal relay (*K1*) that protects the entire system in case of currents superior to 30A. Block *(c)* presents the power switch (*SW1*) that in case of switched ON, it excites the coil of the relay *K2*, the contact gets closed, and the system gets ON. If switched OFF, electrical current stops to flow to the coil, the *K2* contact opens and the system gets OFF.

Block *(d)* consist in a visual feedback for the user, using a red LED, indicating that the system is ON.

Despite the battery's voltage being indicated with a 24V value, in reality, this value is a little bit higher (approximately 26V). So, to get a stable 24V is used a step-down module. The output voltage, besides power the electromechanical brake system of each motor, feeds the block *(e)* where it is represented the emergency button *SW2*. When the button is not pressed, the coil of relay *K3*, for the left motor, and the coil of relay *K4*, for the right motor, are excited and so, the contacts are closed and electrical current can flow to the motors. When *SW2* is pressed, it opens the circuit that powers the coils, the contacts that allow to power the motors get open and the motors stop.

The purpose of block *(f)* is to highlighted the combination of diodes existing to each motor. When *SW2* is pressed, the contacts of relays *K2* and *K3* open instantly and so, due to the inductive properties of the motors and being the supply current interrupted, voltage spikes can occur. This way, it is used a combination of freewheeling diodes (to each motor and for both clockwise and counter-clockwise rotational direction), allowing the motors to draw current until the energy gets dissipated. Without this method, voltage spikes are generated and other circuits of the system get affected.

4.3.3 *Electronic system: PCBs*

It will be now presented particular parts of the electric system, more specifically, the *Printed Circuit Boards (PCBs)* developed. The electronic circuits are divided in three categories: *sensors acquisition* boards, *shield* board and *protection* boards. It was used the *Printed Circuit Board (PCB)* design and schematic software called *EAGLE*.

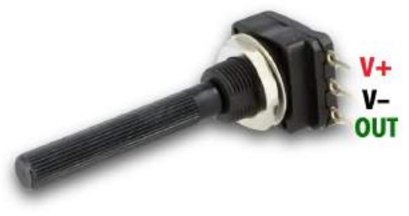
4.3.3.1 *Sensors acquisitions*

For the creation of the sensors acquisitions PCBs , a modular approach was followed. So, for the sensors acquisition boards it was developed a PCB to each sensor or for a special group of sensors. These unit PCBs are then connected to the "motherboard" of the system, which is the *Shield* board, that will be discussed further. With this modular method, it gets easier to debug and test the boards and also, if one of them is affecting the rest of the circuits it is only necessary to remove it from the *shield*. To each board, four holes were made in their corners: two to screw the board to the *Shield* when coupled and other two to screw to the structure of the walker so that the board gets stable. Next, it is presented the circuits and the respective PCB to each unit board.

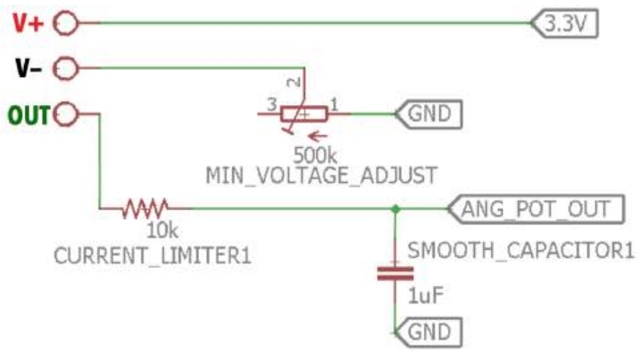
potentiometers and infrared sensor

Due to the simplicity of the circuits and since all the sensors from this board are coupled in the superior part of the walker (next to the handlebar), they were gathered in the same PCB .

Figure 46b presents the circuit for the acquisition of the angular potentiometer's data. It is powered by a voltage of 3.3V and, being its ohmic value of 470 kΩ, a trimmer potentiometer (*MIN_VOLTAGE_ADJUST* in figure 46b) of 500 kΩ is used to allow the possibility to define a minimum output voltage. In the output of the potentiometer (*OUT*), there is, as good practice, a resistor to limit the current that flows to the *STM32F4* pin and also, using a capacitor to ground, the output of the potentiometer is smoothed.



(a)



(b)

Figure 46: Angular Potentiometer: (a) Pin assignment; (b) Circuit.

As to the circuit for the acquisition of the linear potentiometer's data (see figure 47) and the IR sensor's data (see figure 48), they follow the same principle of the angular potentiometer's circuit. However, there is not the possibility to define a minimum output value.

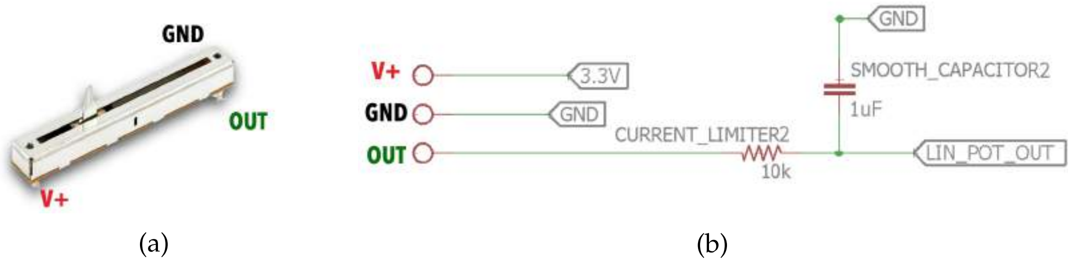


Figure 47: Linear Potentiometer: (a) Pin assignment; (b) Circuit.

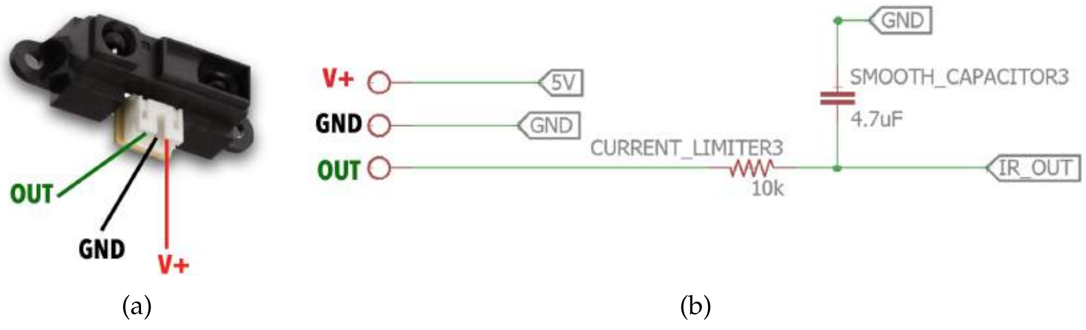


Figure 48: Infrared sensor: (a) Pin assignment; (b) Circuit.

As it can be seen in figure 49, the outputs of the sensors are forwarded to pin headers which allow the interface with the *Shield* board. Also, the power voltages and ground are also connected to the pin headers to receive 3.3V, 5V and ground from the *Shield*.

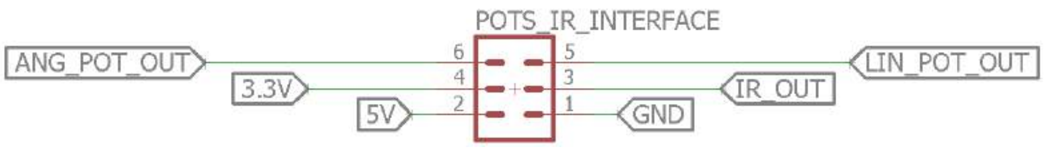


Figure 49: Pin headers, from the potentiometers and infrared sensor's board, to interface with the *Shield* board.

It resulted the PCB presented in 50, where in *(a)* it is represented the circuit for the IR sensor, in *(b)* the circuit for the linear potentiometer, in *(c)* the circuit for the angular potentiometer and in *(d)* the interface to the *Shield* board.

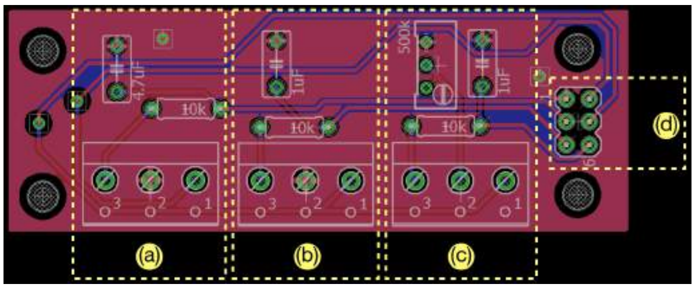


Figure 50: Potentiometers and infrared sensor's board.

load cells

For each load cell (left and right), it is used an instrumentation amplifier, an *INA126P*, to amplify the difference between the outputs of the load cell (*V1+* and *V1-* in figure 51). With a 1 kΩ trimmer (*GAIN_POT1*), it is possible to adjust the amplification gain. Also, an 1 kΩ resistor between the output of the *INA126P* and ground is included to pull-down the output to zero when there is no force applied to the load cell. It is used, in parallel with the resistor, a 10 nF capacitor to smooth the output and a 3.3V zener diode to limit the output voltage. Figure 51b shows the circuit for just one load cell. However, the circuits are equal for each sensor.

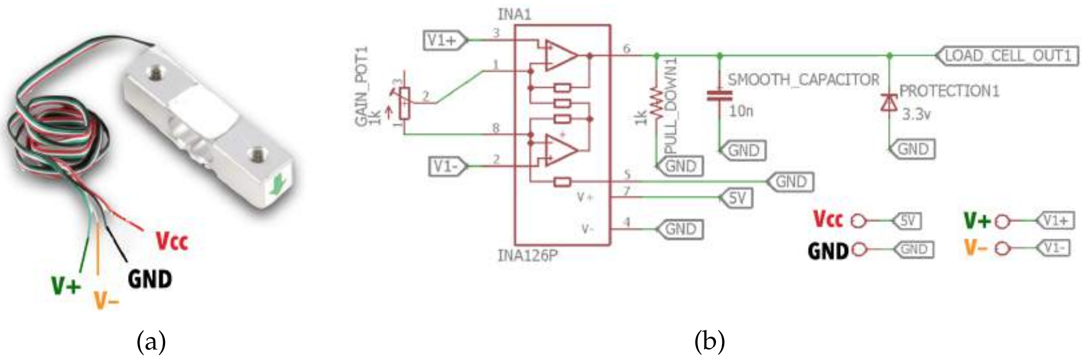


Figure 51: Load cell 1: (a) Pin assignment; (b) Circuit.

As it can be seen in figure 52, the outputs of the load cells are forwarded to pin headers, which allow to redirect the same outputs to the *STM32F4* pins in the *Shield* board. Also, 5V and ground are connected to the pin headers to allow receive those potentials from the motherboard.

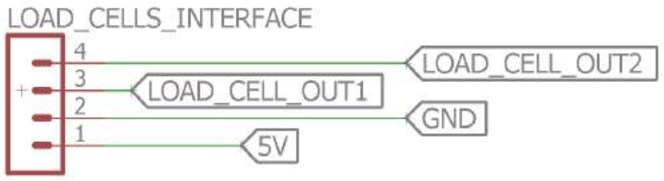


Figure 52: Pin headers, from the load cells' board, to interface with the *Shield* board.

Figure 53 reveals the resulted PCB . It is represented in *(a)* the interface to the *Shield*, in *(b)* the circuit for the left load cell and in *(c)* the circuit for the right load cell.

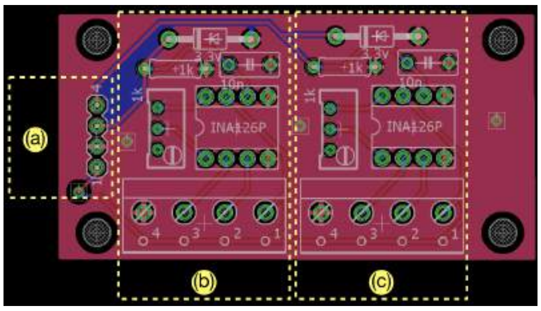


Figure 53: Load cells' board.

encoders

The circuits for each encoder are equal. It just consists in pull-up resistors between the power voltage, 3.3V, and each output channel of the encoder (*CHA* and *CHB*). Figure 54b represents a circuit for one of the encoders.

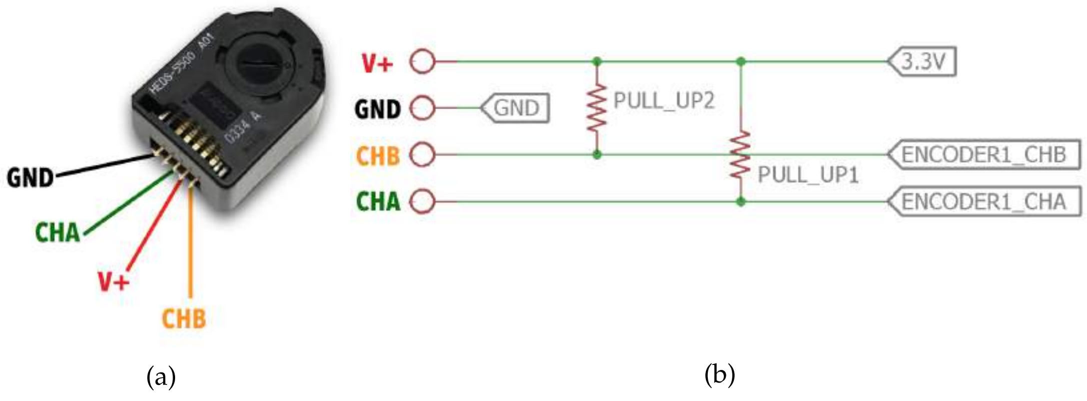


Figure 54: Encoder 1: (a) Pin assignment; (b) Circuit.

As to the interface to the *Shield* (see figure 55), the outputs of the two output channels of each encoder (*ENCODER1_CHA* and *ENCODER_CHB* in figure 54b) are forwarded to pin headers. Also, 3.3V and ground are connected to the pin headers.

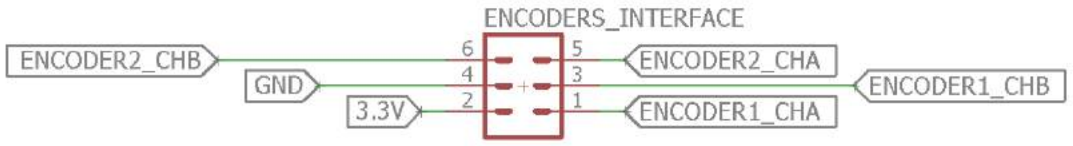


Figure 55: Pin headers, from the encoders' board, to interface with the *Shield* board.

As it can be seen in figure 56, which represents the final encoders' PCB , the block *(a)* consists in the pin headers to interface with the *Shield*, the block *(b)* the circuit for the left encoder and the block *(c)* the circuit for the right encoder.

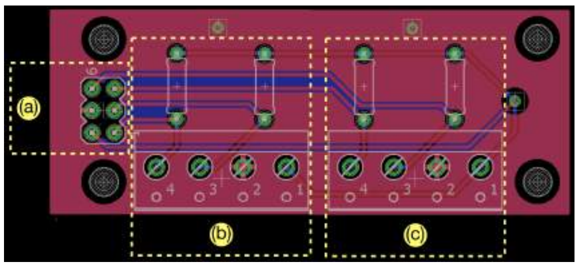


Figure 56: Encoders' board.

inertial measurement unit

The PCB for the *MPU6050*'s data acquisition, is the only board that is not screwed up to the *Shield* board. Since the purpose of the IMU is to provide inertial data from the patient, the board must be on the patient's body. So, it is used female USB connectors to connect the IMU 's board to the motherboard (see figure 57).

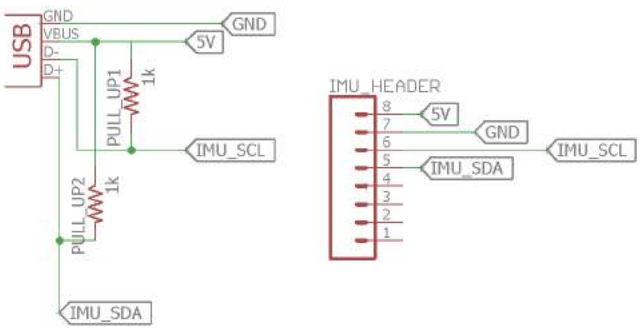


Figure 57: IMU 's circuit.

From the female USB connector comes the 5V, to power the IMU , and ground. The *IMU_HEADER* in figure 57 is where it is connected the sensor. The communication pins, *IMU_SCL* and *IMU_SDA*, are forwarded to the USB connector. Also, due to the long length of the cable that connects the two boards, it was added pull-up resistors to the communication pins.

Figure 58 presents the final result. It is represented in *(a)* the female USB connector, to interface with the *Shield*, and in *(b)* the circuit for the sensor's acquisition.

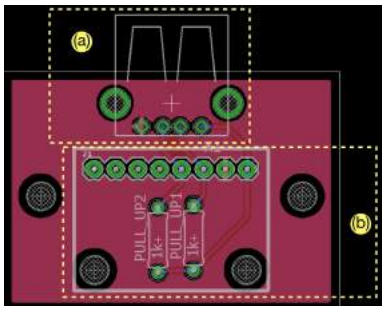


Figure 58: IMU 's board.

ULTRASONIC RANGE FINDERS

As it was referred in 4.3.1.1 - *Sensors*, ethernet cables are used for the connection between each URF sensor and the acquisition board. So, it was necessary to solder some wires of the ethernet cables to each sensor. Since the color sequence of the ethernet wires can be different, it was established a sequential order taking into account the position of the wire in the ethernet connector but not its color.

Figure 59 presents the pins used for the acquisition (note that the red 'X' represents a pin or a wire that is not used/connect).

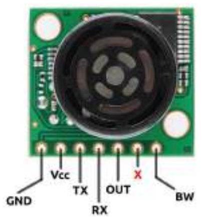


Figure 59: URF 's pins assignment.

Following figure 60, the connections between the URF sensor and the ethernet cable should be made following the alignment sequence, taking into account the position of the pins in the male ethernet connector. Then, through this match it is possible to know the color of the wire that has to be soldered to the respective pin of the sensor.

Figure 61a shows the direction to where the URFs are pointing: Six of them are pointing forward (F); two are pointing up (U); and one is pointing down (D). As discussed in 4.3.1.1 - *Ultrasonic Range Finders* in *Sensors*, if the detecting areas of the several URFs intersect, it can occur interferences in the readings of the sensors. So,

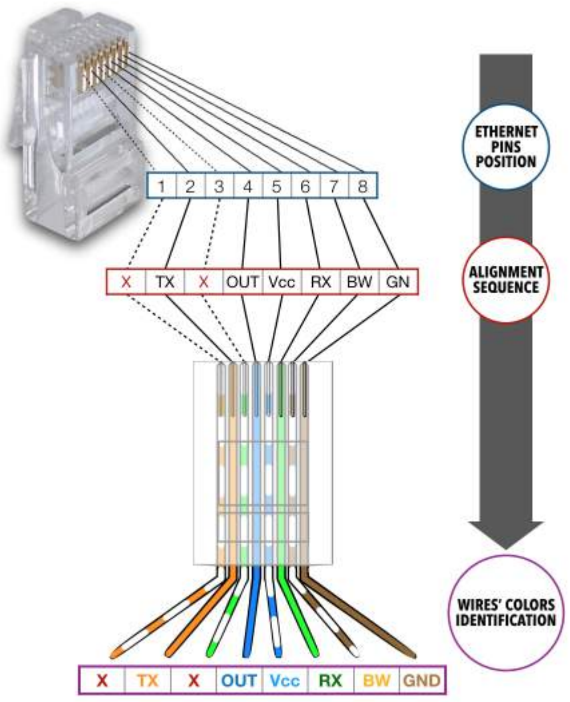
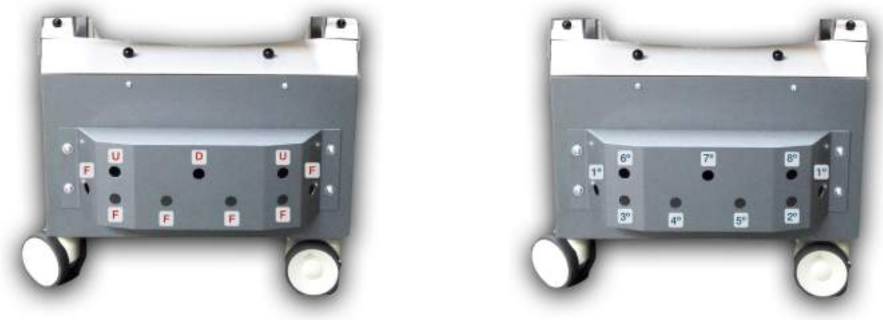


Figure 60: Process to identify the correspondent URF pin to the respective ethernet wire.

since the detecting areas of the lateral URFs do not cross, they receive the initial pulse from the *STM32F4* and start the acquisition at the same time. When the acquisitions finish, a pulse is sent to other sensor and so on. Figure 61b reveals the sequence of acquisitions, which is performed in loop. When the last URF finishes the acquisition, it sends a pulse to the initial URFs and so, the acquisition in loop starts over again.



(a) (b)

Figure 61: URFs : (a) Pointing direction; (b) Sequence of acquisitions.

Whenever it is necessary to stop the loop of acquisitions, the URFs must be reset (power OFF and the power ON). So, as it can be seen in figure 62, it is used a normally open relay that allows to open the circuit that powers the URFs for a fraction of time and then close, reseting the sensors. When a pulse is sent to the base of the 2N2222 transistor, the coil of the relay gets excited and the contacts of the relay close. Otherwise, if no pulse is sent to the base of the transistor, the contacts of the relay open.

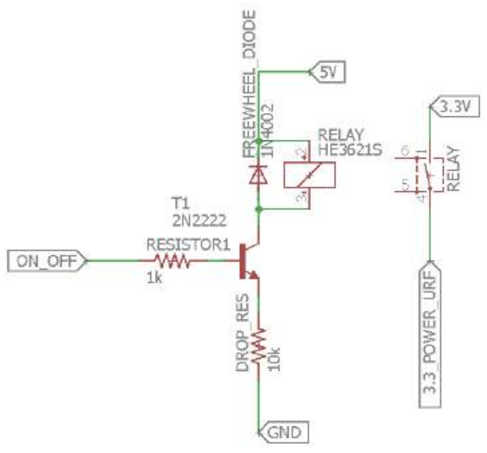


Figure 62: Circuit to reset the URFs .

In figure 63 is presented the pin headers that contain all the outputs of each URF (*SONAR*x), the power voltages and ground, the pulse to reset the URFs (*ON_OFF*) and the pulse that starts the loop of acquisitions. Since the URFs that acquire simultaneously can receive a pulse from the *STM32F4* pin and from the last sensor of the sequence, it is used a diode to prevent that the pulse from the last sensor of the sequence goes to the *STM32F4* pin when the acquisitions are in loop.

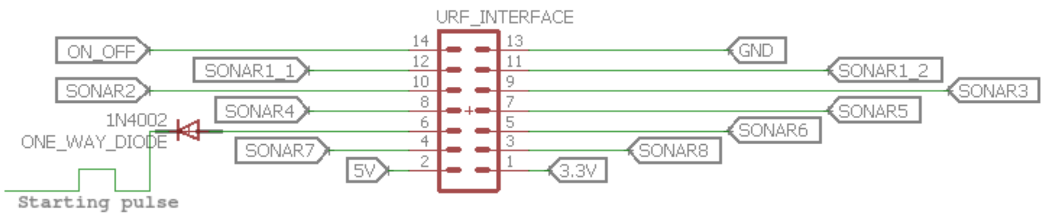


Figure 63: Pin headers, from the ultrasonic range finders' board, to interface with the *Shield* board.

Figure 64 reveals the PCB for the acquisition of the URFs ' data. Block *(a)* consists in the circuit to reset the URFs , block *(b)* presents the pin headers to interface with the *Shield* board and block *(c)* gathers the nine female ethernet connectors.

4.3.3.2 *Shield*

The *Shield* is considered the motherboard of the PCBs . In it, all the information of the other boards is gathered, it presents the circuit to detect an emergency and circuits to monitor the battery voltage and is where it reigns the *Low-Level Controller*.

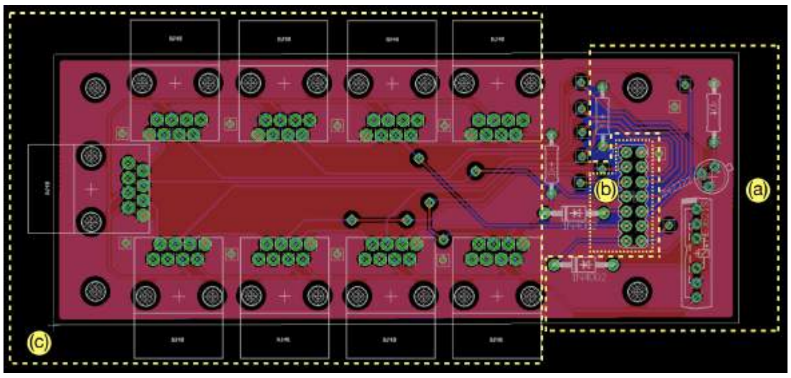


Figure 64: URFs ' board.

As to the circuits in the board, figure 65 shows the circuit that detects if the battery reached a critical voltage. Using a voltage divider it is possible to drop the voltage from the critical value to 0.7V in *Vbe* (base-emitter voltage). So, in one hand, if the voltage of the battery is above the critical voltage, *Vbe* is greater than 0.7V and so, the trasistor saturates and the *CRITICAL_BATTERY_PIN* gets the value of zero. In the other hand, if the battery voltage gets below the critical voltage, *Vbe* will be lower than 0.7V, the transistor gets in cut-off mode and so, a pulse is sent to the *STM32F4* meaning that the battery reached the critical value. To set the critical voltage, i.e, voltage which will trigger an interruption, it is used a trimmer potentiometer in the voltage divider circuit.

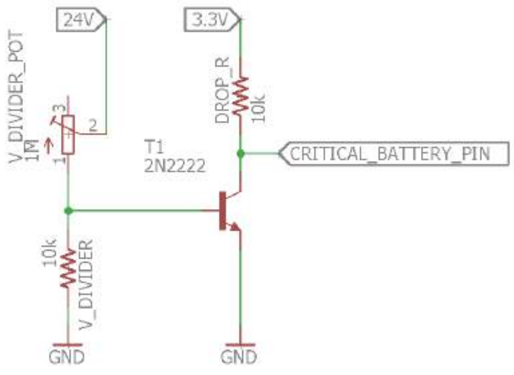


Figure 65: Critical battery detection circuit.

To monitor the battery's voltage, it is used a voltage divider (see figure 66) that scales the range voltage of [0 - 30V] to [0 - 3V]. This way, it is possible to get the voltage of the battery in a scale of [0 - 3V].

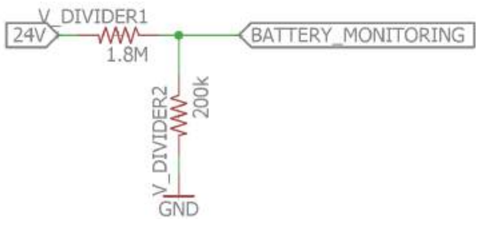


Figure 66: Battery's voltage monitoring circuit.

As to the emergency button's circuit of figure 67, if the button is not pressed, the input for the circuit is 24V and, through a voltage divider, the *EMERGENCY_PIN* gets the value of approximately 3V. Otherwise, if the emergency button gets pressed, the circuit opens and the *EMERGENCY_PIN* gets the value of 0V. Depending on this transitions, the *Low-Level Controller* knows if the button is pressed or not. The capacitor used helps to prevent the bouncing from the contacts of the emergency button.

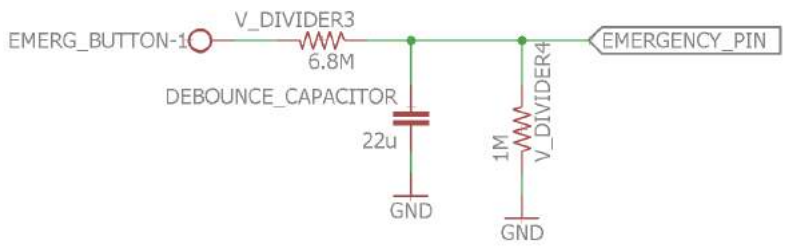


Figure 67: Circuit for emergency detection.

Figure 68 reveals the *Shield* board. The several blocks in the figure represent important characteristics of the PCB that will be explained next.

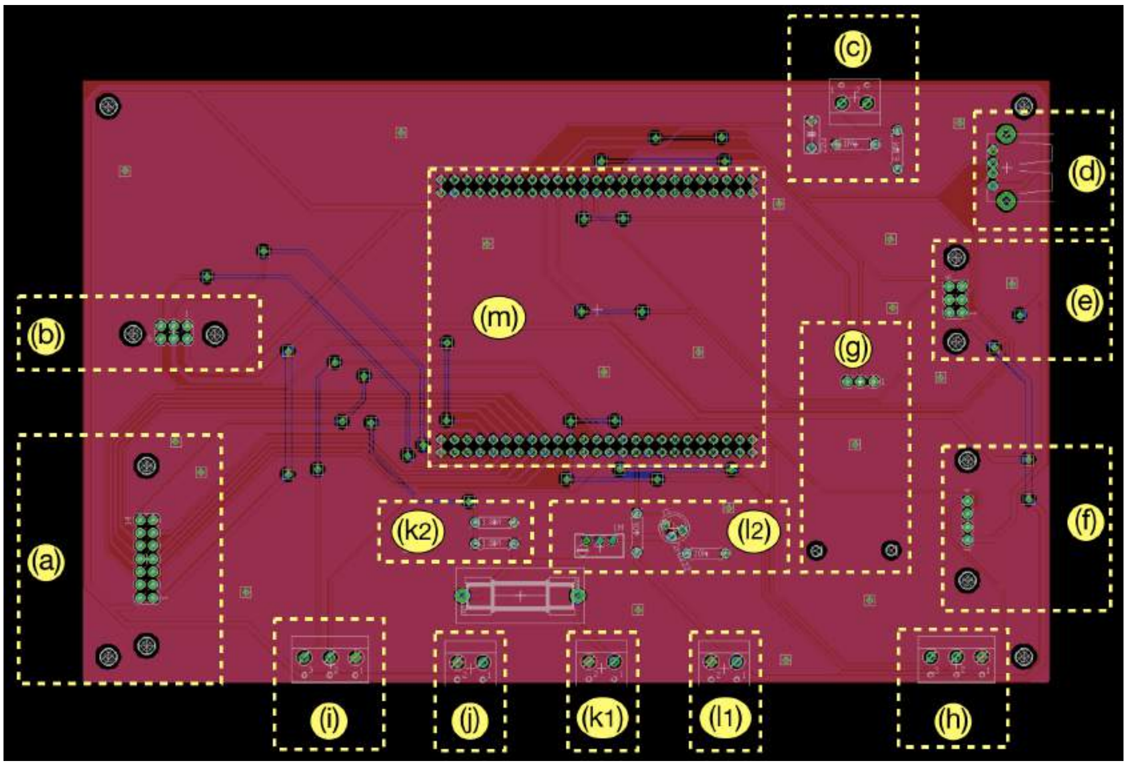


Figure 68: *Shield* board.

As to the blocks:

(a) Where it is coupled the URFs ' board;

(b) Where it is coupled the potentiometers an IR sensor's board.

(c) Circuit for the emergency detection;

(d) Female USB connector to allow the connection with the IMU 's board;

(e) Where it is coupled the encoders' board.

(f) Where it is coupled the load cells' board.

(g) Where it is coupled the USB-to-UART serial converter;

(h) Connections for the right motor's driver (the connections, from left to right, are: direction; PWM ; ground);

(i) Connections for the left motor's driver (the connections, from left to right, are: direction; PWM ; ground);

(j) Connector for the 5V;

(k) (1) Connector for the battery's voltage; (2) circuit that monitors the battery's voltage;

(l) (1) Connector for the 3.3V; (2) circuit that detects if the battery's voltage is critical;

(m) Where it is coupled the *Low-Level Controller* unit.

Figure 69 highlights the *Low-Level Controller*'s pins assigned to the outputs of the different circuits that have been explained. The outputs of the figure are grouped by colors according to their purpose.

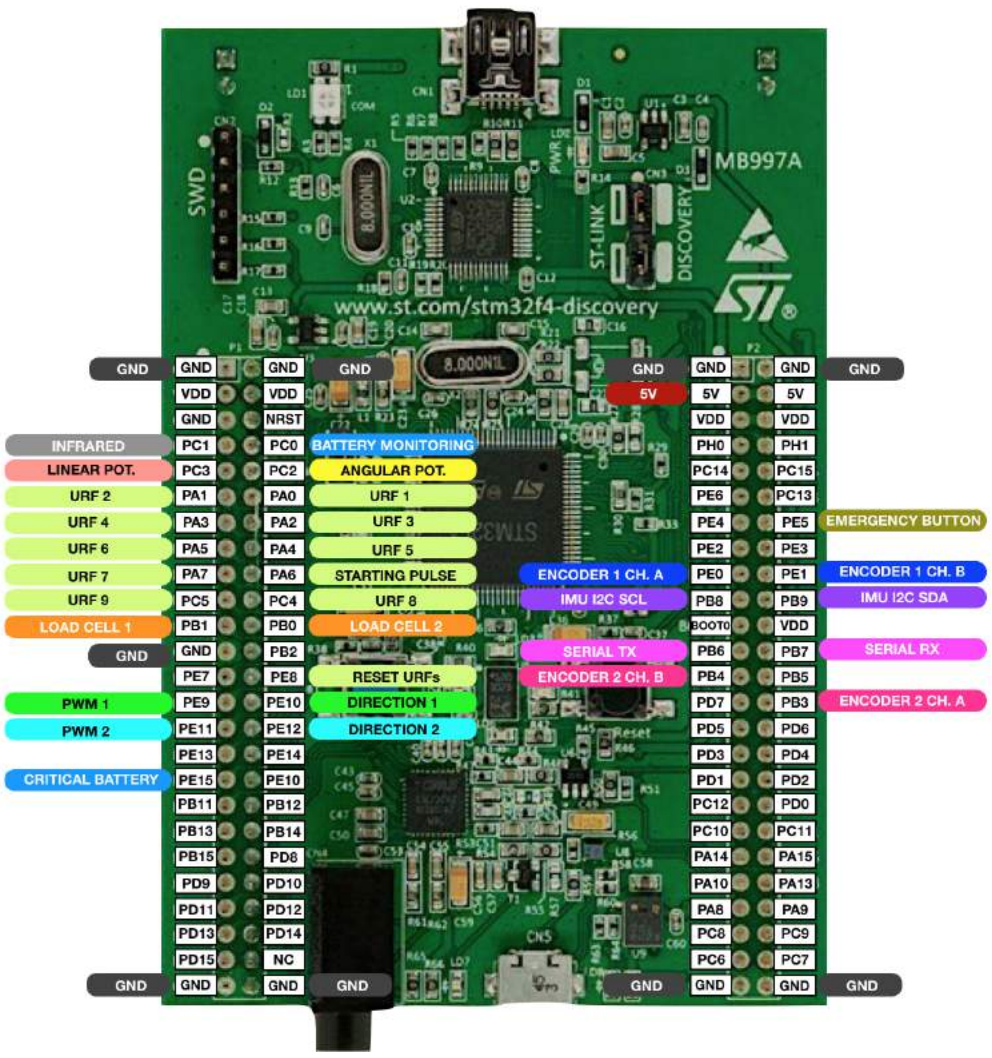


Figure 69: *Low-Level Controller* pin assignment.

4.3.3.3 *Protection*

3.3V AND 5V PROTECTION

The present PCB protects the components connected to 3.3V and 5V from over-voltage and short circuits.

As it can be seen in figure 70, the circuit is the same for both voltages. The only difference is the value of the zener diode which is rated with the voltage that is necessary to limit (i.e. to limit 3.3V it is used a zener with a breakdown voltage of 3.3V and to limit 5V it is used a zener with a breakdown voltage of 5.1V).

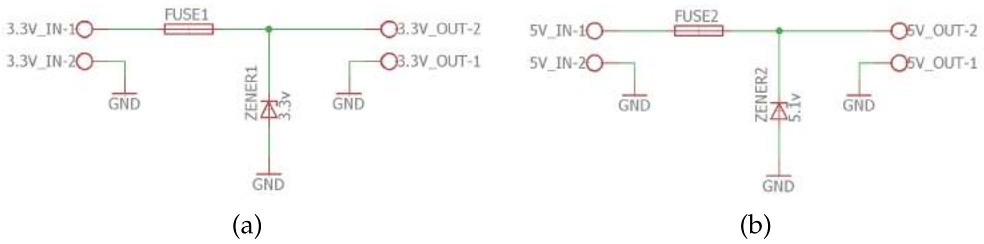


Figure 70: Protection board 1: (a) 3.3V protection; (b) 5V protection.

If a short circuit occurs, the fuse will brake open. As to the protection against over-voltage, the zener tries to limit the voltage to the load if that voltage exceeds the zener's threshold voltage. Also, if the voltage is exceeded and the zener breaks down and conducts current, the fuse will break open protecting the zener and the rest of the circuit from over-voltage.

The resulted PCB is shown in figure 71 where in *(a)* is highlighted the circuit for the 3.3V protection and in *(b)* the 5V protection.

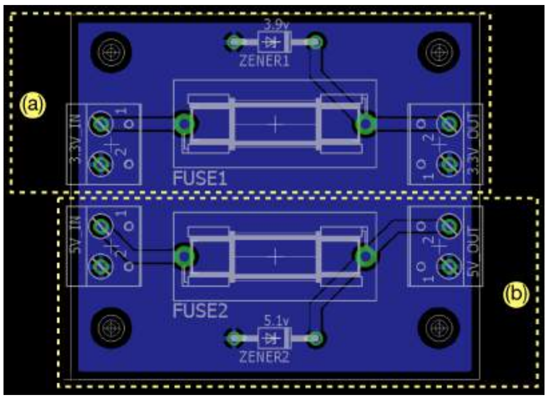


Figure 71: 3.3V and 5V protection board.

monitor, usb hub and on-board computer protection

To protect the touchscreen monitor, the USB HUB and the on-board computer (*Main Controller*) against possible short circuits, it was implemented a protection board for them. Remember, the power voltage for the computer is 19V, for the monitor is 12V and for the USB HUB is 5V.

Figure 72 shows the circuit for the protection of the *Main Controller*. Since the rated current of the on-board computer is approximately 6A, the fuse must brake open at this current value. It is also used a voltage divider to get a voltage of 3.3V from 19V (in this case), to power a LED that allows to know if the fuse opened. To protect the other two elements, the circuit is quite the same. The differences are the value of the fuse and the combination of resistors of the voltage divider that powers the LED.

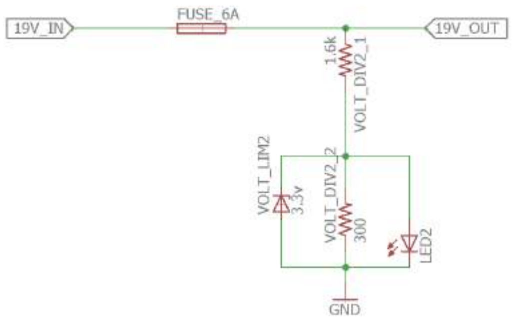


Figure 72: *Main Controller*'s protection circuit.

Figure 73 reveals the final PCB . As it can be seen, the block *(a)* represents the circuit to protect the touch screen monitor, the block *(b)* the circuit to protect the *Main Controller* and the block *(c)* the circuit to protect the USB HUB.

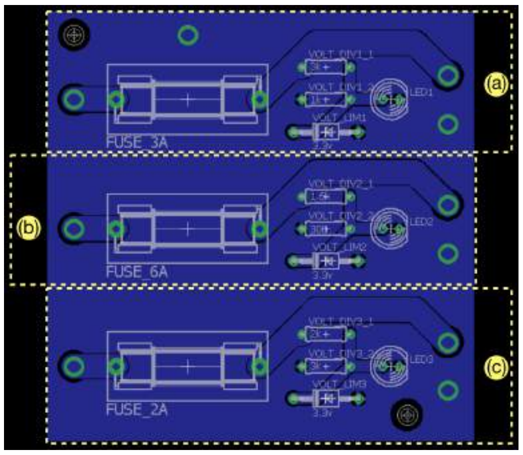


Figure 73: Monitor, USB HUB and on-board computer protection board.

4.4 SOFTWARE DEVELOPMENT

This section addresses the development of the ASBGo* 's software from a more low-level layer to a more high-level layer.

4.4.1 *Overview*

To accomplish the blocks specified in 4.2 - *System Overview* and to confer behaviors to the new device, software was designed and implemented. Figure 74 recalls the two controllers of the system: the *Low-Level Controller* and the *Main Controller*.

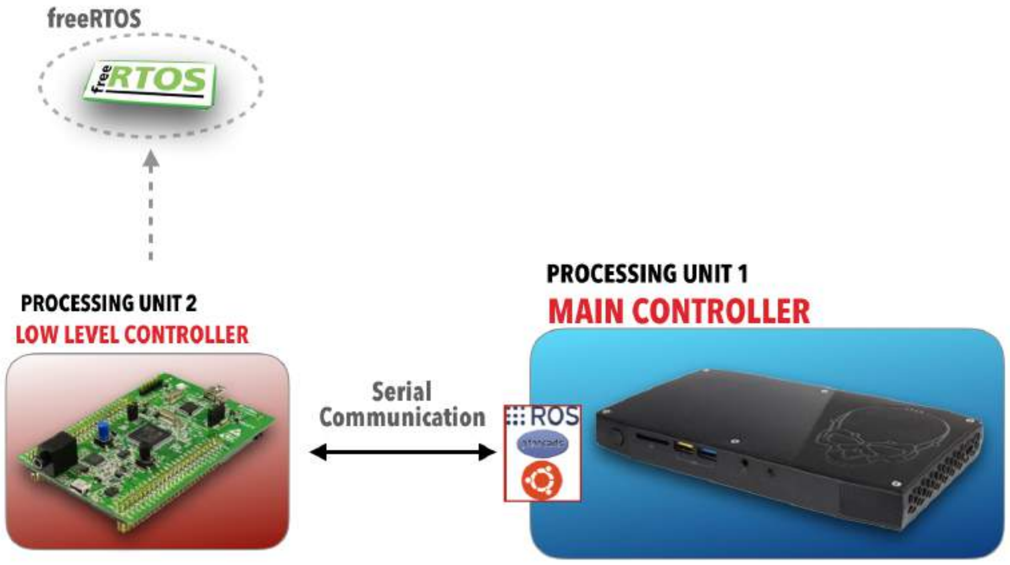


Figure 74: Controllers of the system.

The role of the *Low-Level Controller* is to interface with hardware such as sensors, motors, emergency button and battery. Also, this controller manages all the input and output data, acts according to commands sent by the *Main Controller* and also reacts to certain events and warns the *Main Controller*.

The more high-level software, developed in the *Main Controller*, is built using ROS allowing to defined a solid architecture easing the process of future development. In this main processing unit, the sensors' data coming from the *Low-Level Controller* are disposed and updated at a certain frequency and different algorithms based on these data and interfaces for the user are implemented.

4.4.2 *Communication system*

For the communication between the *Low-Level Controller* and the *Main Controller*, a protocol for communication was designed. Based on a serial communication protocol, the specific protocol of communication for the new system establishes a particular framing for the messages and a set of commands and acknowledgments.

4.4.2.1 *Message framing*

Figure 75 shows the different frames defined for the communication between the two controllers. Each one respects the presented framing during the creation of the messages.

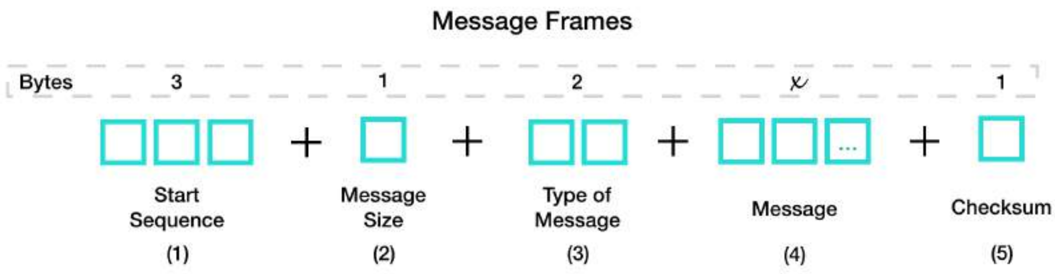


Figure 75: Message's frames.

As to the frames:

1. Represents the start sequence of the message and is constituted by three bytes. Whenever one of the parties of the communication receives the start sequence it means that a new message will be exchanged. In case of the communication gets lost in the exchange of messages, the start sequence of new messages will allow the communication to get back in track;

2. One byte reserved for the message size;

3. Represents the type of message to be exchanged. It can be of the type *information*, e.g. a start command, or it can be of the type *data*. Two bytes are reserved for this frame;

4. The payload frame has undefined size (x) because it depends on the message that is going to be send.

5. The checksum is used to detect if errors occurred during the transmission of the message. It has one byte reserved;

4.4.2.2 *Messages*

It will be know presented messages that are exchanged between the two controllers in each direction. So, the *Main Controller* can send to the *Low-Level Controller*:

- Wake up command;
- Greeting command;
- Life signal command;
- Start commands;
- Stop command;
- Restart command;
- Freeze command;
- Motors' data;
- Frequency for receiving the sensors' data.

In the other hand, the *Low-Level Controller* sends to the *Main Controller*:

- Acknowledgments according to the commands of the *Main Controller*;
- Low battery warning;
- ON/OFF emergency warning;
- Serial communication error warning;
- IMU disconnected warning;
- Sensors' data;
- Battery's voltage data;

4.4.3 *Low-level controller*

The *Low-Level Controller* takes care of the low-level part of the ASBGo* . The application is developed in C language using the *Keil µVision IDE*. Also, the low-level application is built upon an RTOS which consists in a program that schedules execution in a timely manner, manages system resources, and provides a solid base for developing application code in a multitasking environment. The RTOS used is the *FreeRTOS*. It has a very small memory footprint, low overhead, and very fast execution providing the core real time scheduling functionality, inter-task communication, timing and synchronization primitives only. It will be now presented the structure of the application as well as the different modules and their relations.

4.4.3.1 *Structure*

Figure 76 shows the different modules of the low-level application hierarchically divided. There are a relation of direct association from the top of the hierarchy to the bottom. This means that modules from a level above use modules from the level bellow. With this approach it is gets easier to design and understand the conceptuality of the application.

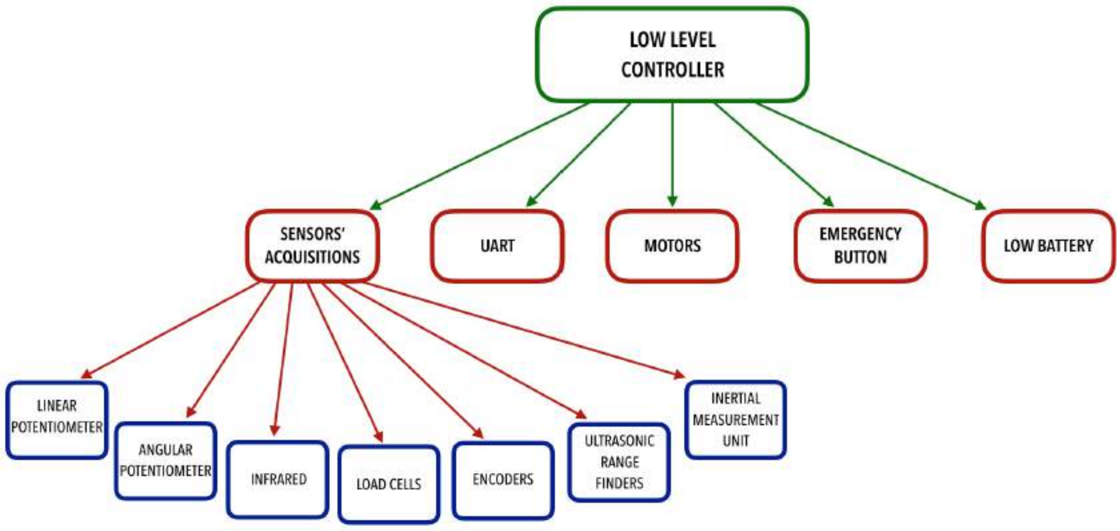


Figure 76: *Low-Level Controller* modules.

In the top of the hierarchy is the *low level controller* module. This main module includes all the others present in the second level of the hierarchy: the *UART* for the serial communication, the module that takes care of the acquisition of the sensors' data, the module that manages the motors, the *emergency button* that detects and handles an emergency event and finally, the module that monitors the battery's voltage. In turn, the *sensors' acquisition* module gathers all the specific modules of each sensor managing all their data and their operation.

4.4.3.2 *Modules*

It will be know detailed each module of the low-level application. Figure 77 reveals the legend for the diagrams that come next.

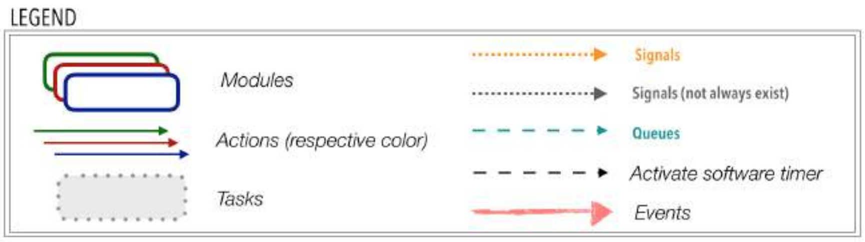


Figure 77: Legend of the modules' diagrams.

potentiometers, infrared and load cells

The acquisition of the linear and angular potentiometers, the IR sensor and the load cells is quite the same. It is done using the ADC1 of the *Low-Level Controller* with a resolution of 10 bits. So, in this modules are defined functions to: configure the ADC; configure the DMA (Direct memory access); perform an acquisition; and a function that handles possible errors that can occur during the configurations. Also, there is a initialization function that wraps the configuration functions. It is used the DMA so that when an acquisition is performed the data acquired is directly stored in a pre-defined position of the memory and, at each acquisition, the data is overrode and so, updated. Figure 78 illustrates the relations of the module.

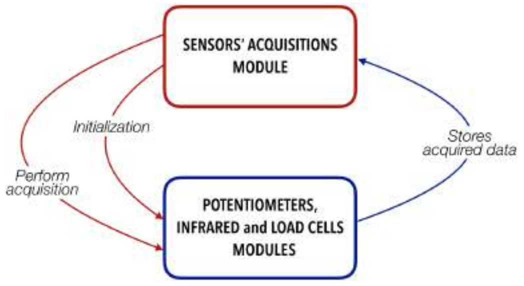

SENSORS' ACQUISITIONS MODULE
Initialization
Perform acquisition
Stores acquired data
POTENTIOMETERS, INFRARED and LOAD CELLS MODULES

Figure 78: Diagram of the potentiometers, infrared and load cells' modules

encoders

The outputs of the encoders consist in phase-shifted pulses (from channel A and B). When channel A leads channel B, it means that the motor is rotating in one direction. Else, if channel B leads channel A, then the motor is rotating in the opposite direction. To detect this pulses (ticks), it is used a configuration of external interruptions meaning that when a transition is detected, positive or negative, an *Interrupt Service Routine (ISR)* is called. This way, the present modules has functions to: configure the *General-purpose input output (GPIO)* pins for interruptions at positive and negative transitions; enable and disable the external interrupts; and a function that handles possible errors that can occur during the configurations. Figure 79 illustrates the relations of the module.

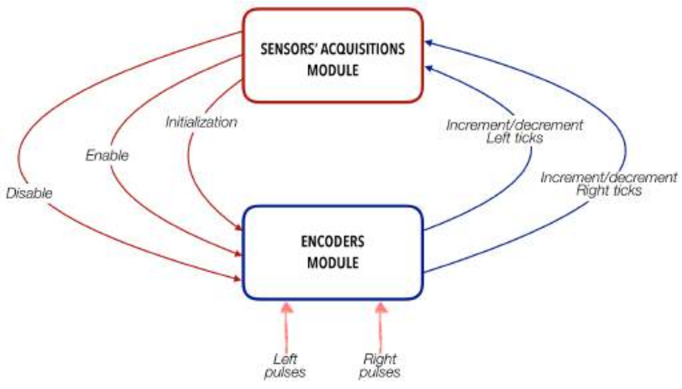

SENSORS' ACQUISITIONS MODULE
ENCODERS MODULE
Initialization
Enable
Disable
Increment/decrement Left ticks
Increment/decrement Right ticks
Left pulses
Right pulses

Figure 79: Diagram of the encoders module.

The flowchart from figure 80 shows one of the four ISR s. The four ISR s correspond to the two channels of each encoder. In this case, it is presented the ISR for the channel A of the left encoder. When a transition in channel A occurs, it is evaluated if it was a rising or falling transition. If it was a rising one, it is then evaluated the state of the channel B in which, depending on its state, a variable that stores left encoder ticks is incremented or decremented. In the other hand, if it was a falling transition, the channel B is equally evaluated to determine in which direction the motor is rotating in order to the left encoder ticks' variable be incremented or decremented. The other ISR s follow the same principle.

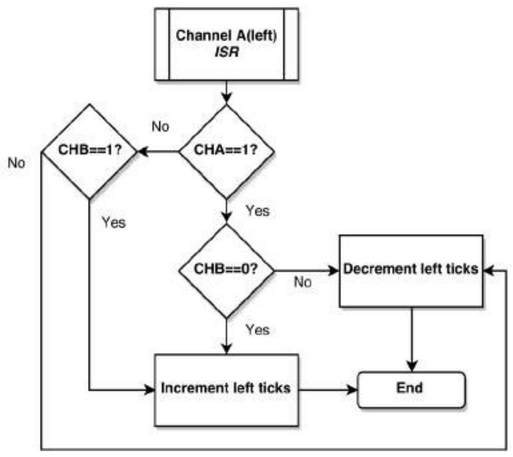


Figure 80: Flowchart of the ISR for the channel A of the left encoder.

ULTRASONIC RANGE FINDERS

The acquisitions of the URFs ′ data are done using nine channels of the ADC2 of the *STM32F4* with a resolution of 10 bits. Figure 81 shows the relations of the module.

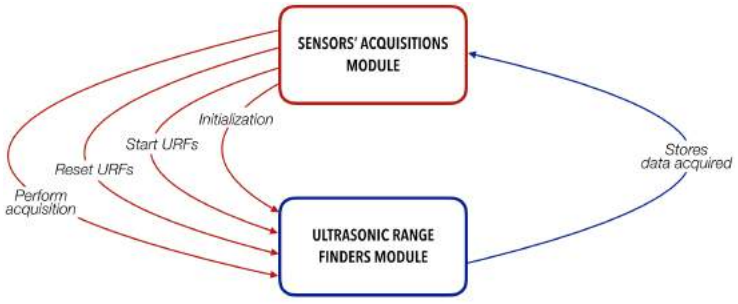


Figure 81: Diagram of the ultrasonic range finders module.

This module contains functions to:

- Configure the ADC2. It is configured to perform nine conversions in nine specific channels. When a conversion is requested for the ADC2, nine conversions are performed sequentially;
- Configure the DMA. When the acquisitions are done, the data acquired is directly stored in pre-defined positions of the memory without being necessary the intervention of the processor;
- Configure two GPIO pins. Both pins are configured as digital outputs where one is to send a pulse to reset the URFs and the other is to send an initial pulse in order to start the acquisitions;
- Start the URFs . In this function, the starting pulse is sent and the URFs acquire distance values in a looping chain;
- Reset the URFs by sending a pulse which powers OFF and then powers ON the URFs ;
- Start an ADC conversion. When this function is called, the analog outputs values of the URFs are read and converted by the ADC;
- Handle possible errors that can occur during the configurations.

INERTIAL MEASUREMENT UNIT

To acquire data from the *MPU6050*'s accelerometer and gyroscope, it is used the I2C communication protocol. Figure 82 pictures the relations of the module. In this module several macros are defined storing special bytes used for the I2C communication with the *MPU6050*.

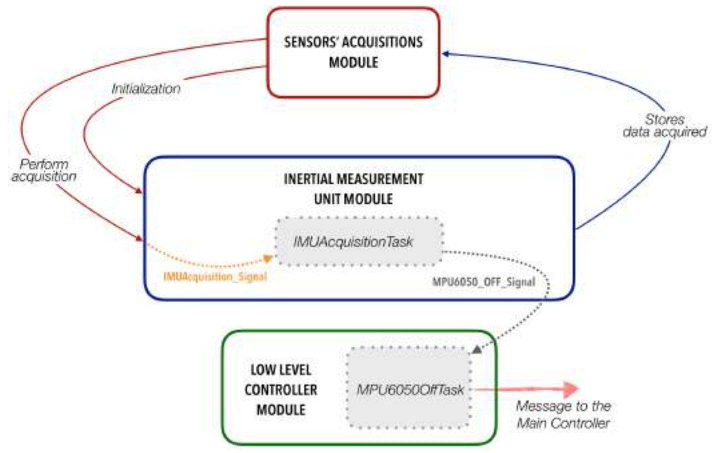


Figure 82: Diagram of the inertial measurement unit module.

So, it presents functions to:

- Configure the I2C1 of the *Low-Level Controller* for the communication with the IMU ;
- Initialize the *FreeRTOS* objects of this module such as tasks and semaphores;
- Check if the IMU is "alive". To make such verification, it is necessary to send a special byte and if the IMU is "alive", it responds with its address. This way it is possible to know, during the execution of the application, if the sensor is connected to the system or if it was disconnected;
- Configure the IMU such as configure the data sample rate, the accelerometer, the gyroscope and their resolutions. Before executing this configurations, it is tested if the IMU is "alive";
- Start an acquisition of the IMU 's data;
- Request accelerometer's data;
- Returns the data read from the accelerometer;

- Request gyroscope's data;
- Returns the data read from the gyroscope;
- Handle possible errors that can occur during the configurations.

As to the *FreeRTOS* objects there is:

- A semaphore (*IMUAcquisition_Signal*) which is released when it is called the function to start an acquisition;
- A task (*IMUAcquisitionTask*) that blocks until it is signaled by the *IMUAcquisition_Signal*. When this semaphore is released, this tasks checks if the IMU is "alive", requests and gets accelerometer's data, requests and gets gyroscope's data and stores this information in specific positions of the memory which are overrode at each acquisition;
- A semaphore (*MPU6050_OFF_Signal*) that is released when it is detected that the IMU is not "alive";
- A task (*MPU6050OffTask*) that blocks until it is signaled by the *MPU6050_OFF_Signal*. When this semaphore is released, this task warns the *low level controller* module so that a warning is sent to the *Main Controller*.

sensors' acquisitions

This module can be considered like an interface between the *low level controller* module and the sensors' modules. This means that the present module is "commanded" by the main module to start or stop acquisitions, it handles every sensor, based on the functions provided by each module of the sensors, and it outputs the data of all the sensors.

In this module is defined a *struct*, called *Sensors*, that gathers all the variables corresponding to the sensors' data (see listing 4.1). Also, a global object of the *Sensors* struct type is instantiated. With this struct, it is possible to encapsulate all the sensors' data in one unified data structure in an organized way easing the accessibility and manipulation of the data.

```
struct Potentiometers{
  uint16_t angular;
  uint16_t linear;
};
```

```
struct LoadCells{
  uint16_t left;
  uint16_t right;
};

struct UltrasonicRangeFinders{
  uint16_t one;
  uint16_t two;
  uint16_t three;
  uint16_t four;
  uint16_t five;
  uint16_t six;
  uint16_t seven;
  uint16_t eight;
  uint16_t nine;
};

struct Encoders{
  long leftTicks;
  long rightTicks;
};

struct IMU{
  int16_t accelerometer_X;
  int16_t accelerometer_Y;
  int16_t accelerometer_Z;
  int16_t gyroscope_X;
  int16_t gyroscope_Y;
  int16_t gyroscope_Z;
};

struct Sensors
{
  struct Potentiometers potentiometer;
  uint16_t infrared;
  struct LoadCells load_cells;
  struct UltrasonicRangeFinders ultrasonic_range_finder;
  struct Encoders encoder;
  struct IMU imu;
};
```

Listing 4.1: *Sensors* struct

Figure 83 shows the relations of the module.

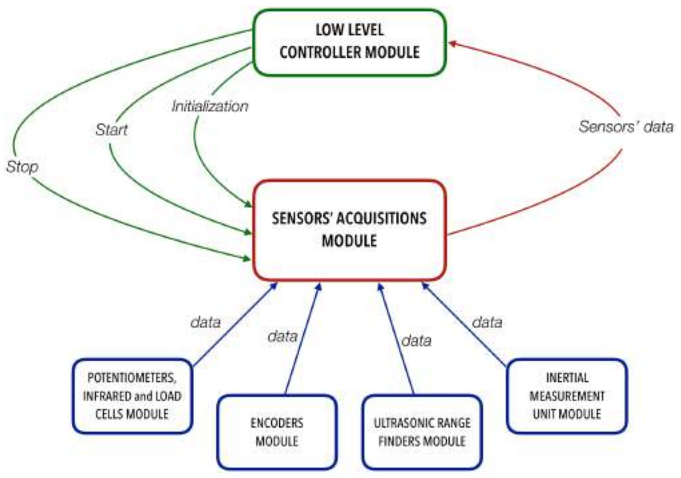

LOW LEVEL CONTROLLER MODULE
Stop
Start
Initialization
Sensors' data
SENSORS' ACQUISITIONS MODULE
data
data
data
data
POTENTIOMETERS, INFRARED and LOAD CELLS MODULE
ENCODERS MODULE
ULTRASONIC RANGE FINDERS MODULE
INERTIAL MEASUREMENT UNIT MODULE

Figure 83: Diagram of the sensors' acquisitions module.

The *sensors' acquisitions* module contain functions to:

- Configure the acquisition timers. An ISR is called every time the timers reach the configured time (expire). Inside the ISR it is requested to the sensors' modules to perform acquisitions. So, there are two timers:
  - TIMER3: configured with a frequency of 1kHz. At each 1 millisecond, the ISR of TIMER3 is executed and it is requested the acquisition of the data of the potentiometers, IR , load cells and IMU ;
  - TIMER4: configured with a frequency of 100Hz. At each 10 milliseconds, the ISR of TIMER4 is executed and it is requested the acquisition of the URFs ' data. There is a specific timer for this sensors due to the latency that is introduced by the acquisitions in a loop chain. So, it is pointless to acquire at a higher frequency rate because the outputs of the nine URFs take a few milliseconds to get updated.
- Clean the *Sensors* struct;
- Initialize the timers and the sensors by calling the initialization functions of each sensor's module;
- Start the acquisition timers and enable the interruptions for the encoders' ticks detection;

- Stop the acquisition timers, reset the URFs and disable the interruptions for the encoders' ticks detection;
- Handle possible errors that can occur during the configurations.

UART

The *UART* module takes care of the serial communication between the *Main Controller* and the *Low-Level Controller*.

In this module is defined a *struct*, called *Data*, that wraps a pointer to bytes, which is used to store the message, and a byte, which is used to store the size of the message (see listing 4.2). This way, it is introduced in the application a specific data structure to handle with the messages to receive or to send.

```
struct Data
{
  uint8_t *message;
  uint8_t size;
};
```

Listing 4.2: *Data* struct

Figure 84 gives the relations of the module.

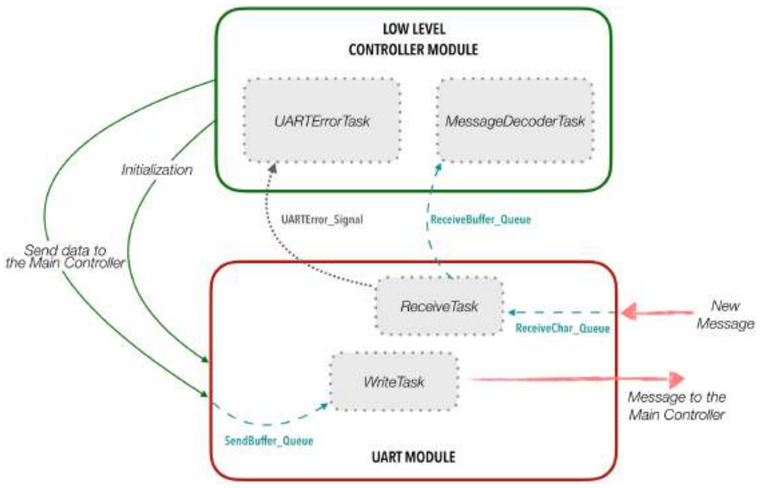


Figure 84: Diagram of the UART module.

The *UART* module presents functions to:

- Configure the USART1 of the *Low-Level Controller* using a baudrate of 115200;

- Initialize the *FreeRTOS* objects for this module such as semaphores, message queues and tasks;

- Prepare a message to be sent. This function performs the framing process discussed in 4.4.2.1 - *Message Frame*, producing a valid message that is compatible with the communication protocol existing between the two controllers. This way, auxiliary functions, such as functions to perform the checksum, to get the size of the messages and to add the type of the message to the final message, are defined, complementing the process of creating a new message that is about to be sent. After, it is declared a pointer to an object of the *Data* struct type, memory is allocated to it and to its fields, the information of the new created message is stored in the fields of the object and the pointer is queued in a specific message queue called *SendBuffer_Queue*;

- Handle the received bytes from the *Main Controller*. This functions analyzes each received byte in order to know in which frame of the message, which is being received, it is on. For example, if nothing has been received and a new message arrives, this function will start to analyze the sequence of the first three bytes to check if it corresponds to the start sequence. Then it receives the message size and so on. In the end, a checksum of the received message is performed and if it equals to the received checksum, the message is valid and it was successfully received;

- Queue the valid message received. When a message is successfully received, this function declares a pointer to an object of the *Data* struct type, allocates memory to it and to its fields, according to the received message, and stores the message information in the new object. Then, the object that contains the message and its size is queued in a specific message queue (*ReceiveBuffer_Queue*) which is accessed by the *low level controller* module in order to be decoded;

- Handle possible errors that can occur during the configurations.

As to the *FreeRTOS* objects there is:

- A message queue (*ReceiveChar_Queue*) that queues single bytes received from the message coming from the *Main Controller*. So, when a byte is received, an ISR from UART is triggered, the byte is read and queued in this specific queue;

- A task (*ReceiveTask*) that blocks until there is data in *ReceiveChar_Queue*. When bytes are available, this task consumes the byte in the queue and the function that analyzes bytes is called in order to analyze the byte received;
- A message queue (*ReceiveBuffer_Queue*) that stores successfully received messages to be consumed and decoded by the *low level controller* module. The type of data of this queue consists in pointers to objects of the *Data* struct type. When the information is consumed and used, the memory of the allocated pointer is deleted as well as the fields of the object.
- A message queue (*SendBuffer_Queue*) that, after the framing process, stores new created messages to be sent to the *Main Controller*. The type of data of this queue consists in pointers to objects of the *Data* struct type;
- A task (*WriteTask*) that blocks until there is data in *SendBuffer_Queue*. When data is available, its fields (message and message size) are accessed and a new message is sent to the *Main Controller*. After used, the pointer to the object of a *Data* struct type that was consumed is deleted as well as its fields;
- A semaphore (*UARTErrors_Signal*) that is released when the communication is lost. When, at some point, the communication fails and can not get back in track, this signal is released;
- A task (*UartErrorTask*) that blocks until it is signaled by the *UARTErrors_Signal*. When this semaphore is released, this task, declared in this module but defined in the *low level controller* module, sends a warning to the *Main Controller*.

MOTORS

The *motors* module is responsible to generate the PWM signals in order to actuate the motors. In this module a struct, called *Motors*, is defined and its fields correspond to the duty cycles for the right and left motors' PWM s. An object of this struct is instantiated (*motors*) and in it is where the duty cycle values to each motor will be stored. Figure 85 pictures the relations of the module. The *motors* module collects functions to:

- Configure the TIMER1 of the *Low-Level Controller*. It is configured in PWM mode with two output channels, one to each motor;
- Configure two GPIO pins as digital outputs. Each pin is used to set the rotation direction of each motor;

- Initialize the *FreeRTOS* objects for this module such as message queues, tasks and software timers;
- Starts the timers for each PWM output. When this function is called, the duty cycles are set to 0;
- Update the fields of the *motors* object and so, new duty cycles for the PWM s of each motor are set. Also, depending on the signal (positive or negative) of the new duty cycle, it is set the rotational direction of each motor;
- Stop the timer for each output channel. Also, the duty cycle values of the *motors* object are set to zero;
- Smoothly stop the motors when the *Low-Level Controller* receives a stop command. With this approach, the *Main Controller* does not have to worry about control the stop process because the *Low-Level Controller* takes care of imperatively stop the motors. When this function is called, a software timer of the *FreeRTOS* (*StopMotorsSmoothly_Timer*) is activated;
- Handle possible errors that can occur during the configurations.

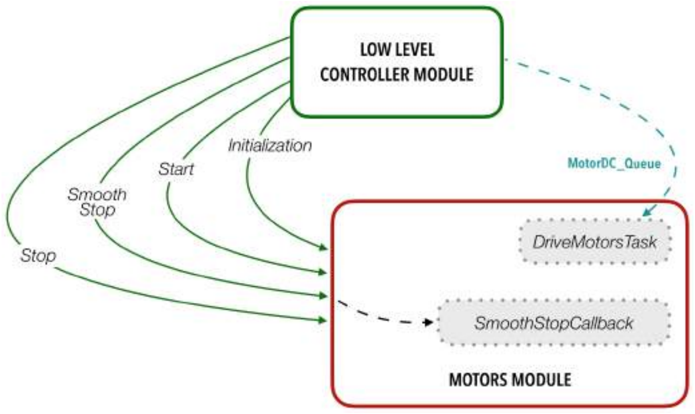


Figure 85: Diagram of the motors module.

As to the *FreeRTOS* objects there is:

- A message queue (*MotorsDC_Queue*) that stores two bytes corresponding each byte to the value of the duty cycle to each motor. The *low level controller* module receives those values from the *Main Controller* and queues it in the *MotorsDC_Queue*;

- A task (*DriveMotorsTask*) that blocks until there is data on the *MotorsDC_Queue*. When the *low level controller* queues new duty cycle values, this task wakes, consumes the data on the queue and calls a function that updates the duty cycle values and the PWM s for the motors;

- A software timer that is activated when it is necessary to smoothly stop the motors. At each 0.3 seconds, the software timer's callback (*StopMotorsSmoothly_Timer*) is executed and it gradually reduces the duty cycle of the PWM of each motor until it reaches the value 0. When the motors stop, the software timer is disabled;

EMERGENCY BUTTON

The *emergency button* module handles an emergency event. Using an external interrupt, it is possible to know if the button is or was pressed or release. Note that when the button is pressed it is necessary to release it manually, otherwise, it stays pressed. So, when a transition is detected, an ISR is executed and it is evaluated and defined a new state of the button. Figure 86 shows the relations of the module.

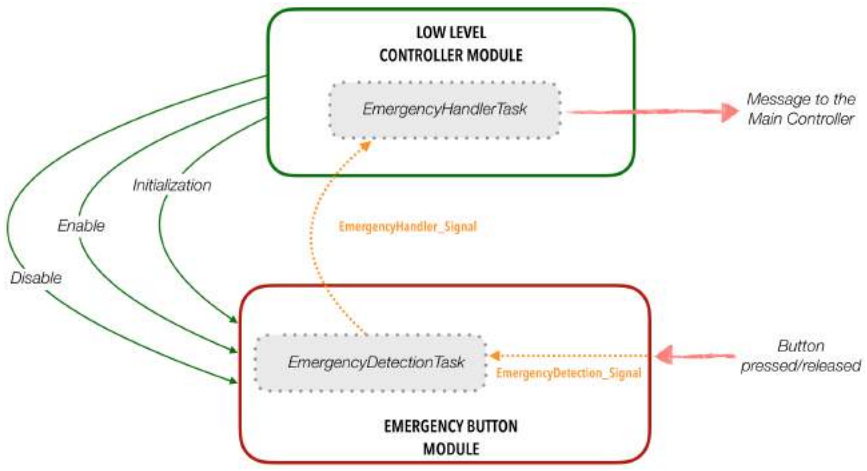


Figure 86: Diagram of the emergency button module.

So, the module presents functions to:

- Configure a GPIO pin in external interrupt mode detecting rising and falling transitions;

- Initialize the *FreeRTOS* objects for this module such as semaphores and tasks;

- Enable the external interrupt;
- Disable the external interrupt;
- Read the digital value that is on the pin;
- An initialization function that wraps all the configuration functions and performs an initial checks of the button's state;
- Handle possible errors that can occur during the configurations.

As to the *FreeRTOS* object there is:

- A semaphore (*EmergencyDetection_Signal*) that is released when the ISR is executed (a detection occurred);
- A task (*EmergencyDetectedTask*) that blocks until it is signaled by the *EmergencyDetection_Signal*. When this signal is released, it is verified which transition occurred and, depending on that, the *EmergencyHandler_Signal* is released;
- A semaphore (*EmergencyHandler_Signal*) is released when the button gets in a new state;
- A task (*EmergencyHandlerTask*) that blocks until it is signaled by the *EmergencyHandler_Signal*. When this signal occurs, the current state of the button is sent to the *Main Controller* by the *low level controller* module (this task is declared in the *emergency button* module but it is defined in the *low level controller* module).

low battery

The *low battery* module monitors the battery's voltage and detects when the voltage reaches a critical state. For the first case it is used the ADC3 of the *Low-Level Controller* with a resolution of 12 bits, to get a digital value of the battery's voltage (which is scaled in a range of [0 - 3V]). For the second case it is used a pin in external interrupt mode. Note that when the critical voltage is detected, an ISR is executed. Figure 87 illustrates the relations of the module. It contains functions to:

- Configure the ADC3;

- Configure the DMA. When a conversion of the ADC is performed, the data acquired is diretcly stored (without the intervention of the processor) in a specific position of the memory. At each acquisition, the value is overrode and so, updated;
- Configure a GPIO pin in external interrupt mode to detect falling transitions;
- Initialize the *FreeRTOS* objects for this module such as semaphores, tasks and software timers;
- Enable the external interrupt, to detect the battery's critical voltage, as well as enable the battery's voltage monitoring;
- Disable the external interrupt and the battery's voltage monitoring;
- Perform an initial verification to see if the battery's voltage is already in its critical state;
- Handle possible errors that can occur during the configurations.

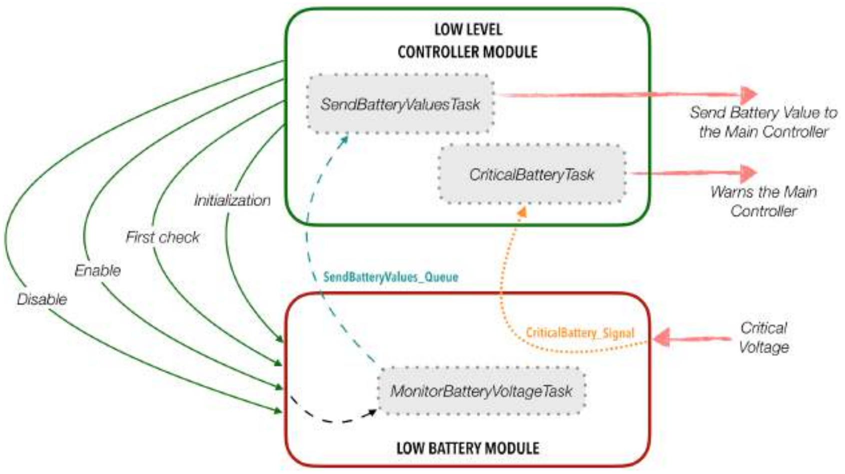


Figure 87: Diagram of the low battery module.

As to the *FreeRTOS* objects there is:

- A semaphore (*CriticalBattery_Signal*) that is released when an external interrupt occurred (battery's voltage in its critical state);

- A task (*CriticalBatteryTask*) that blocks until it is signaled by the *CriticalBattery_Signal*. When this signal is released, this task, declared in the *low battery* module but defined in the *low level controller* module, sends a warning to the *Main Controller* and "freezes" all the *Low-Level Controller*.
- A software timer which is activated when the battery's voltage monitoring is enabled. Its callback (*MonitoringBatteryVoltageCallback*) is executed at each 5 second. During the execution, an ADC3 conversion is performed and the acquired value is queued into the *SendBatteryValue_Queue*;
- A message queue (*SendBatteryValue_Queue*) that stores integers corresponding to the value resulted from the ADC3 conversion of the battery's voltage;
- A task (*SendBatteryValuesTask*) that blocks until there is data in the *SendBatteryValue_Queue*. When a value of the battery's voltage is available, this task, which is declared in the *low battery* module but defined in the *low level controller*, consumes the information and sends this data to the *Main Controller*

low level controller

The *low level controller* module is the main block of the low-level application. It is the one that make the decisions and acts accordingly to the information of all the other modules. This module is the only one that decides which message needs to be sent to the *Main Controller* and also is the one that decodes the received messages and commands the low-level application. Note that by the explanation of the others modules, it is possible to understand the job of the *low level controller* module and how it interacts with the others. So,

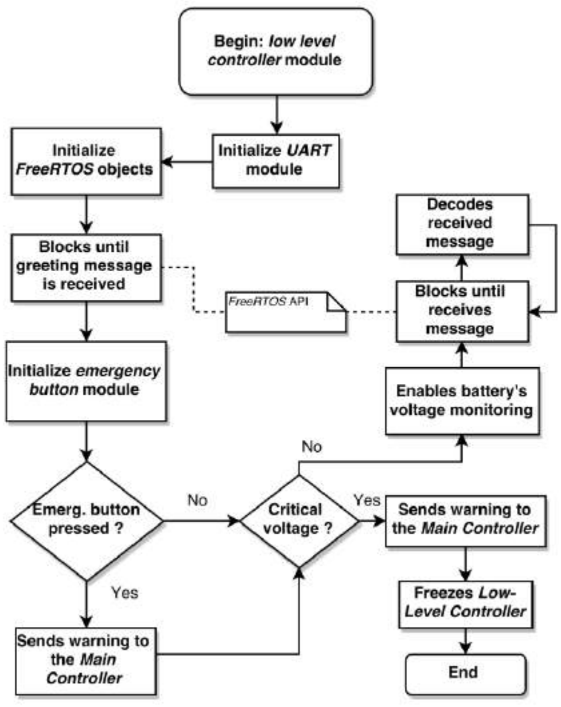

Begin: low level controller module
Initialize UART module
Initialize FreeRTOS objects
Blocks until greeting message is received
FreeRTOS API
Initialize emergency button module
Emerg. button pressed ?
No
Yes
Sends warning to the Main Controller
Critical voltage ?
No
Yes
Enables battery's voltage monitoring
Blocks until receives message
Decodes received message
Sends warning to the Main Controller
Freezes Low-Level Controller
End

Figure 88: Flowchart of the *low level controller* module.

to clarify this module, it will be
only explained exclusive details and the main flow of the program. Figure 88 shows a flowchart of the *low level controller* module's startup.

As to the flowchart:

- At the beginning, relevant initializations are performed such as the initialization of the *UART* module and the initialization of the *FreeRTOS* objects for the *low level controller* module;

- Then, this module blocks in a task (*MessageDecoderTask*) waiting the existence of messages (coming from the *Main Controller*) in the *ReceiveBuffer_Queue* of the *UART* module in order to decode them;

- At the beginning of the program, this main module only waits for a greeting message. It will do nothing until it receives a greeting message from the *Main Controller*;

- Receiving a greeting message, it means that the *Main Controller* is ready and so, the *low level controller* module performs second initializations like initialize the *emergency button* module and the *low battery* module. During the initialization of this modules two things happen: it is verified the state of the emergency button and it is verified if the battery's voltage is in critical state. If it is, the *Main Controller* is warned to be turned off and the *low level controller* freezes itself i.e. it does absolutely nothing more. Contrarily, if the battery's voltage is still acceptable, it is enabled its monitoring and the *low level controller* module goes again to an idle state waiting for messages (commands) from the *Main Controller*;

- Since it was already received a greeting message, the incoming messages from the *Main Controller* are commands to perform some kind of action.

While blocked in the *MessageDecoderTask*, the *low level controller* module can receive messages such as:

- A 'life signal' message. In order to verify if the *Low-Level Controller* is still "alive", the *Main Controller* sends a periodic message in which the *low level controller* module has to acknowledge and answer;

- A 'start' message. The *low level controller* module can receive starting messages to start the sensors' acquisitions, start the motors or both:

  - When it is to start the acquisitions of the sensors' data, the *sensors' acquisitions* module is initialized and started. From there, the specific field of the *Sensors* struct will start to be filled by the specific sensor's module. Also, it is initialized the TIMER2 of the *Low-Level Controller* with a frequency of 10Hz. This timer is used to send all the sensors' data to the *Main Controller*. So, at each 100 milliseconds, an ISR of the TIMER2 is executed and a semaphore (*SendSensorsData_Signal*) is released. The *SendSensorsDataTask* task waits the signal of *SendSensorsData_Signal* and when it is signaled, it reads the values of the *Sensors* struct (from the *sensors' acquisitions* module) and sends the data to the *Main Controller*;
  - When the motors are enabled, the *motors* module waits duty cycle values.

- A 'stop' message. When the *low level controller* module receives a 'stop' command, it stops the *sensors' acquisition* module, stops the timer used to send the sensors' data to the *Main Controller* and stops and disables the motors (if enabled);
- A 'restart' message. When a restart message is received, the *Low-Level Controller* performs a software restart and starts from the beginning of the application;
- A 'freeze' message. The *Main Controller* has also the ability to detect if the battery's voltage is critical based on the battery's value that it receives (see the *low battery* module). This way, the *Main Controller* can also send a 'freeze' command in order to the *Low-Level Controller* "freezes" itself;

Besides receiving message, it can be received data such as:

- Motors' data. The *low level controller* module receives, from the *Main Controller*, duty cycle values for the motors. Those values are queued in the *MotorsDC_Queue* which are then consumed and used in the *motors* module;
- New send frequency. By default, the frequency to send the sensors' data is 10Hz. However, the *Main Controller* has the power to define a new frequency. The range is [1 - 800] Hz;

4.4.4 *Main controller*

The *Main Controller* gathers the more high-level software. Built using ROS , the software developed covers the requirements that the blocks defined in 4.2 - *System Overview* demand.

Since the use of the *Indigo* distribution of ROS , the operating system that is installed in the *Main Controller* is the *Ubuntu 14.04*. For the code implementation, it was installed the *QtCreator* IDE with a ROS plug-in to ease the process of built software with ROS . The current developed software is written in C++ and QML. However, for future expansion, new code can be written in the same language or in another language compatible with ROS . For the communication subsystem, that will be discussed next, it was used the POSIX threads (pthreads) libraries which consist in a standard based thread *Application Programming Interface (API)* for C/C++ and allows to introduce concurrence in the execution of a program. For the rest of the software, it was used the *roscpp* API of ROS .

It will be now detailed the communication subsystem developed as well as all the ROS processes.

4.4.4.1 *Communication subsystem*

To accomplish the communication between the two controllers, it was created a class for the serial communication called *Serial*. More specifically, the *Serial* class consists in a concrete class that derives from a class that defines an interface for communications, called *ICommunication*. This abstract class gathers methods and attributes common of a communication process and so, envisioning a possible software expansion, future communication protocols developed should derive from the interface *ICommunication*. Figure 89 presents the class diagram of the communication subsystem from the *Main Controller*.

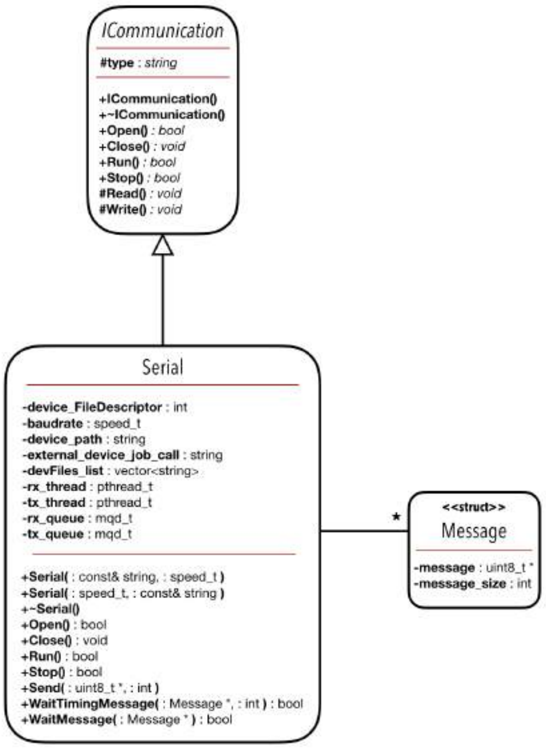


Figure 89: Communication subsystem class diagram.

The *ICommunication* class contains pure virtual methods that must be implemented by the derived classes. There are methods to: open and close the communication; run and stop the communication; and methods to read and write data. As to the attributes, it contains a string attribute which will serve to store the type of the communication. Also, the communication subsystem contains a struct called *Message* that can be use as a generic data structure for the communication. The *Message* struct wraps a pointer to bytes, which will store a message, and an integer, which will store the message's size.

The serial communication is the type of communication used between the two controllers and so, the *Serial* class, derived from the *ICommunication*, was implemented. So, the *Serial* class contains:

- Two constructors that have a parameter which consists in the baudrate of the serial communication. The difference between them is that one accepts a string that will serve to search and find the device that responds to the that string in order to get the device's file path for pairing. The other constructor accepts a string with the device's path;
- A method to open the communication. If, in the constructor, it was passed as argument a string with a call message, this *'Open'* method will search for all the serial devices, open the communication and send the call message. If the device does not responds, the communication is closed and it is attempted to communicate with another device. Otherwise, if a device responds the same call message, the communication was successfully opened and remains opened. When the communication succeeds, it is stored the file descriptor of the device because it is needed to perform the communication operations;
- A *Close()* method which closes the communication with the device;
- A *Run()* method that configures and starts the messages queues and the threads:
  - The *Read_Thread* is in an infinite loop calling the pure virtual method *Read()* that was implemented. This method blocks the threat until it receives a message. When it is received, it is analyzed the sequence of the first three bytes to verify if it equals the start sequence. If it does not, the rest of the message is discarded. If it does, the rest of the message is received and if the calculated checksum equals the received checksum, the message is queued into the *rx_queue*;
  - The *rx_queue* is a message queue that stores pointers to objects of the *Message* struct type. When it is necessary to queue a message, it is declared a pointer

to a *Message* struct and memory is allocated to it and to the field that will store the message. Then, the information is copied to the object and the pointer is queued. After used, the allocated memory must be freed.

  - The *Write_Thread* is in an infinite loop calling the pure virtual method *Write()* that was implemented. This method blocks the thread until there is data in the *tx_queue*. When data is available, the thread wakes and since the data consists in a pointer to an object of the *Message* struct type, the message's size is taken into account and the message itself is fetched in order to be sent to the serial device;
  - The *tx_queue* is a message queue that stores pointers to objects of the *Message* struct type and it is used to signal the *Write_Thread* to send the queued message to the serial device.

- A *Stop()* method that cancels the threads and closes the message queues;
- A public method that is used to send messages. As parameters it receives an array of bytes, consisting in the message, and an integer that consist in the message size. This method performs the framing process discussed in 4.4.2.1 - *Message Frame*, in order to obtain a valid and compatible message. Then, a pointer to a *Message* struct is declared and memory is allocated to it and to the field that will store the message. The information is copied to the object and its pointer is queued on the *tx_queue*;
- Two more methods that complement this serial API . Both methods are used to block and wait until there is data in the *rx_queue* (wait until a message is received). The methods return a pointer to a *Message* struct which allows the access to the information of the received message and use it. The difference between this two methods is that one blocks infinitely until there is a message and the other waits a message during a specific time;

4.4.4.2 *ROS nodes*

Different software components were designed and implemented respecting the two main blocks targeted: *Real-time sensors data acquisition and locomotion* and *Human-Machine interface (HMI)*. In this part, it will be presented different ROS nodes that confer functionalities to the ASBGo* and also allowed to define a software base for future development. So, the nodes are divided as follows (note that the nodes are in *italic*):

- Real-time sensors data acquisition and locomotion:
  - *LowLevelController_node* (1)
  - Low Level Controller's sensors data algorithms:
    - *CriticalBatteryDetector_node* (2)
    - Locomotion:
      - *DriveHandlebar_node* (3)
      - *SetpointsInterface_node* (4)
      - *VelocityCalculator_node* (5)
      - *PIDLocomotionEnabler_node* (6)
      - */left_motor/pid_controller* (7)
      - */right_motor/pid_controller* (7)
      - *MotorsPWM_node* (8)
      - *MotorsDetections_node* (9)
- Human-machine interface:
  - *LocalGUI_node* (10)
  - *RemoteController_node* (11)

It will be now discussed the purpose of each node and some considerations will be given. Also, it will be revealed, to each node, the topics to which it subscribes and the topics to which it publishes. This way, it is possible to understand the inputs and the outputs of the software processes allowing the comprehension of how the system works.

(1) LOWLEVELCONTROLLER_NODE

The *LowLevelController_node* is considered, from the *Main Controller* point of view, the bridge between the *Main Controller* and the *Low-Level Controller* and so, it is the entity that sends and receives information to and from the *Low-Level Controller*. Due to the complexity of the node, it was developed a class, called *_LowLevelController*, that is instantiated in the node and dictates its execution flow . This way, the software of this node gets encapsulated and more organized. Figure 90 gives a representation of the *LowLevelController_node* and the related topics.

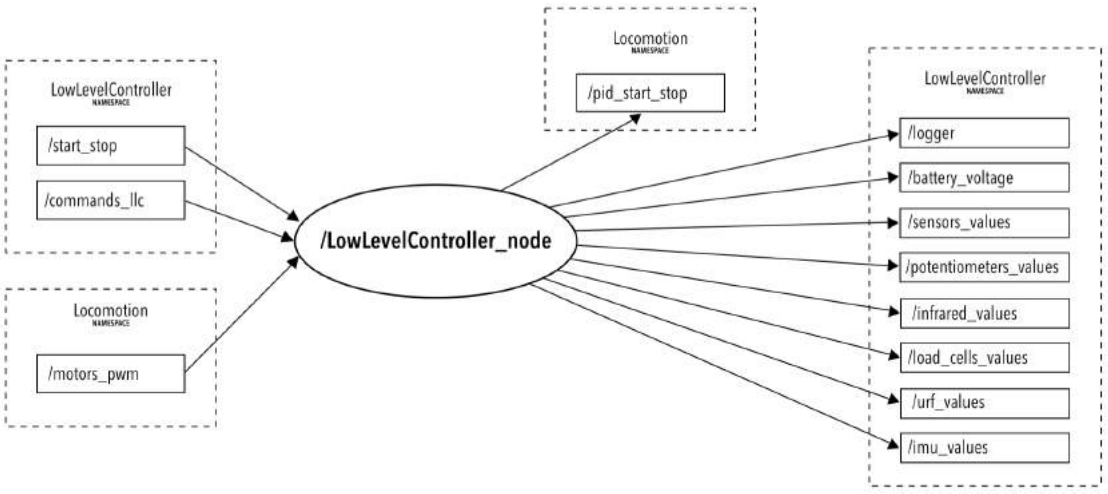


Figure 90: *LowLevelController_node* ROS graph.

As to the *_LowLevelController* class:

- It subscribes to the *LowLevelController/start_stop* topic that is created in the beginning of the node. The message's type of the topic is *bool* and so, when a *true* message is received the node starts its main execution. Otherwise, if a *false* message is received the *Low-Level Controller* unit is restarted and the node shutdowns itself;

- When the node starts its main execution, a serial communication is created (an object of the *Serial* class is instantiated) and is passed as argument a specific call message to be sent to the *Low-Level Controller*. When it responds, it is detected the specific device file in order to validate the communication (see 4.4.4.1 - *Communication subsystem*). If the communication was successfully opened, the communication threads start and also it starts a thread (*WaitingMessages_Thread*) from the *_LowLevelController* class;

- The *WaitingMessages_Thread* pthread is in an infinite loop calling a method from the *Serial* class that blocks until a message is received from the serial device (in this case, the *Low-Level Controller*). When a message is received it is decoded and actions are performed accordingly.

- It is relevant to say that at this point, to every command that is sent to the *Low-Level Controller* a acknowledgment is expected and so, if it is not received it is assumed that something is wrong and a message is reported to the *LowLevelController/logger* topic (it will be discussed later);

- Being everything configured and ready, a greeting message is sent to the *Low-Level Controller*. If the *Low-Level Controller* greets back, everything is ok and the execution continues;
- Battery's values are periodically received from the *Low-Level Controller*, converted into a voltage value and published in the *LowLevelController/battery_voltage* topic. Also, a life signal message is periodically sent to the *Low-Level Controller* in order to know if it is still "alive" and nothing went wrong. Again, if it not responds, a message is published to the *LowLevelController/logger* topic;
- The *LowLevelController_node* publishes to the *LowLevelController/logger* topic relevant log messages. It can be messages to warn that something was wrong or to warn that some event was triggered like the low battery event, the emergency button pressed/released event and so on;
- From this point, the node is kind of in an idle state. It is just waiting for commands from other nodes that are received through the *LowLevelController/commands_llc* topic in which the *LowLevelController_node* subscribes to. The commands received are interpreted and specific messages are sent to the *Low-Level Controller* unit. It can be commands to enable/disable specific sensors (choose the ones that will acquire data), to enable/disable the motors, to start/stop the action in the *Low-Level Controller* and to "freeze" or restart the *Low-Level Controller*;
- If a 'start' command is received (from the *LowLevelController/commands_llc* topic) and if all the sensors were enabled, the *LowLevelController_node* will receive from the *Low-Level Controller* unit the sensors' data. This data is stored in a struct of sensors' data equal to the one represented in 4.1. The sensors' data is received as raw information (a set of bytes) and then it is casted to the sensors' struct type so that the information gets on the specific fields automatically. The data of each field of the sensors' struct is converted to specific units, e.g. the data of the IR is converted to centimeters, the data from the load cells is converted to kilograms and so on, and then this data is published to specific topics. The potentiometers' values are published to the *LowLevelController/potentiometers_values* topic, the IR 's value to the *LowLevelController/infrared_values* topic, the load cells'

values to the *LowLevelController/load_cells_values* topic, the URFs ' values to the *LowLevelController/urf_values* topic and the values of the IMU to the *LowLevelController/imu_values* topic. Also, all the sensors' values converted are published in the *LowLevelController/sensors_values* topic that gathers the information of all the sensors. It is relevant to say that this topics do not accumulate the values published but instead, it updates them;

- If the motors are enabled and a 'start' command is received, the *LowLevelController_node* publishes to the *Locomotion/pid_start_stop* topic and subscribes to information from the *Locomotion/motors_pwm* topic which consist in duty cycles values to be sent to the *Low-Level Controller*;
- There is a method in the *_LowLevelController* class that allows to send to the *Low-Level Controller* unit a new frequency value corresponding to the frequency in which the sensors' data will be received.

(2) CRITICALBATTERYDETECTOR_NODE

The *CriticalBatteryDetector_node* is responsible to detect if the battery's voltage has reached a critical state. Figure 91 reveals the inputs and outputs of this software component.

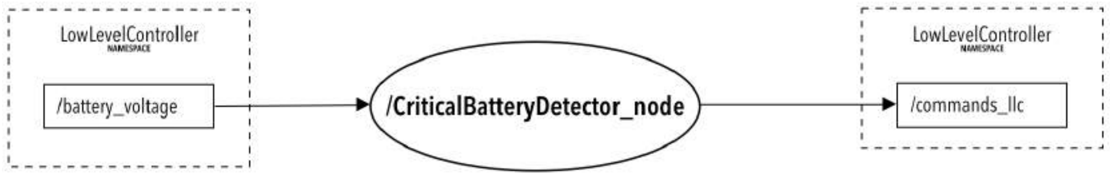


Figure 91: *CriticalBatteryDetector_node* ROS graph.

As to the node:

- When launched, it takes two parameters. The first one, *battery_critical_voltage*, corresponds to the voltage that is considered a critical voltage for the battery and so, for the system. The second parameter, the *n_detections*, corresponds to the number of times that is detected the critical voltage until a warning message is sent;
- It subscribes to the *LowLevelController/battery_voltage* topic in which, when a battery value is available, the node consumes it and checks if the received

voltage is greater than the *battery_critical_voltage*. If it is not a counter variable is incremented, otherwise, the counter variable is decrement (if greater than zero). When the counter variable reaches the *n_detections*, a warning is reported;

- It publishes to the *LowLevelController/commands_llc* topic a message (command) to warn the *LowLevelController_node* that the battery's voltage reached a critical value.

(3) drivehandlebar_node

The *DriveHandlebar_node* is responsible to monitor the positions of the handlebar of the ASBGo* in order to generate a locomotion command. Figure 92 shows the inputs and outputs of the *DriveHandlebar_node*.

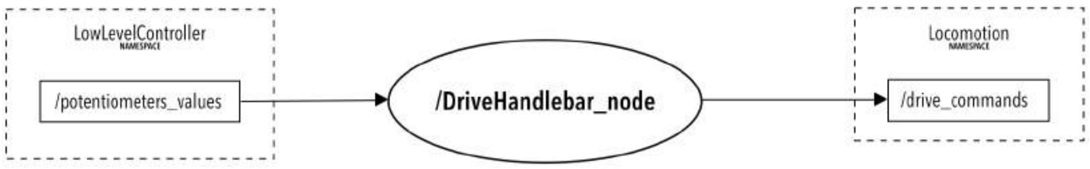


Figure 92: *DriveHandlebar_node* ROS graph.

So, as to the node:

- When launched, it takes five parameters consisting in five threshold values: one is to detect if it is to go forward, other two to detect if it is to go in front-left or front-right direction, and the other two to detect if it is to turn completely to the left or to the right;

- It subscribes to the *LowLevelController/potentiometers_values* topic in which, when potentiometers' values are available, they are compared with the threshold values in order to be generated a drive command;

- When there is a drive command, it is published to the *Locomotion/drive_commands* a string corresponding to the drive command. Such commands can be to: move front, move to the front left, move to the front right, turn completely to the left, turn completely to the right and stop.

(4) setpointsinterface_node

The *SetpointsInterface_node* consists in a interface between the processes that generate drive commands and the processes that control the motors. According to the drive commands, it is outputted relevant information for the nodes responsible for the motors' control. Figure 93 illustrates the inputs and outputs of the *SetpointsInterface_node*.

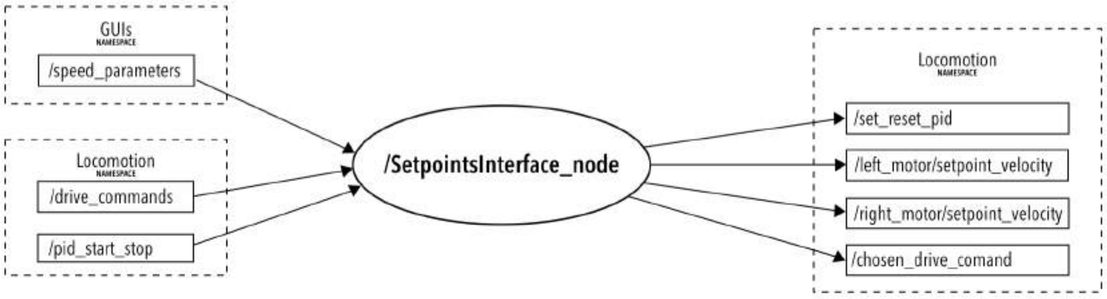


Figure 93: *SetpointsInterface_node* ROS graph.

So, as to the node:

- It always stores the state of the locomotion, e.g. if the locomotion is disabled the state is in *NONE_STATE*, if the ASBGo* is moving forward the state is in *FRONT_STATE* and so on;
- When the node is launched, it is in a *NONE_STATE* and also, when a *false* value comes from the *Locomotion/pid_start_stop*, a topic in which the node subscribes to, it means that the locomotion was disabled and so, the state variable of this node gets the value of *NONE_STATE*;
- When the locomotion is enabled, drive commands will be received through the *Locomotion/drive_commands* topic. When received, those commands are decode and setpoint velocities to each motor are defined and published to the *Locomotion/left_motor/setpoint_velocity* and *Locomotion/right_motor/setpoint_velocity* topics;
- It will be in another node that when there is a 'stop' drive command and the velocity of the walker reaches zero, the controller of the motors is disabled temporarily. Eventually, if it is received a different command (in this node) the motors' controller must be enabled again. For this, the *SetpointsInterface_node* publishes a *true* value to the *Locomotion/set_reset_pid* topic;
- Since the *SetpointsInterface_node* gives a final locomotion command (setpoint velocities), it also publishes, to the *Locomotion/chosen_drive_command* topic, the drive command received to be used by others nodes for important algorithms;

- Since this node outputs setpoint velocities, it must know which velocities to publish. The definition of the velocity consists in two parameters: the velocity between [0.1 - 1]m/s^2 and the curvature percentage ([1 - 100]%) which sets the angular velocity for the curves. When the node is launched, there are a default velocity and curvature. However, the *SetpointsInterface_node* subscribes to the *GUIs/speed_parameters* topic which allows to define new values for the parameters that define the locomotion of the walker (velocity and curvature);

- For example, when a 'front right' command is received and the velocity was defined with a value of 1m/s^2 and the curvature 50%, the setpoint velocity for the left motor will be 1m/s^2 and for the right motor will be (*defined_velocity* - ((*defined_curvature* / *100*) x *defined_velocity*)) which translates in (*1* - ((*50* / *100*) x *1*)) that equals to a velocity of 0.5m/s^2 and so the walker curves to the right.

(5) velocitycalculator_node

The *VelocityCalculator_node* is responsible to calculate the current velocity of each motor of the ASBGo* . Figure 94 reveals the inputs and outputs of the node.

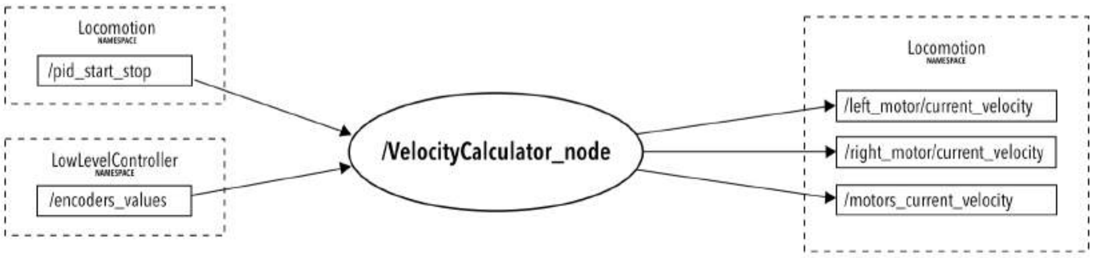


Figure 94: *VelocityCalculator_node* ROS graph.

So, as to the *VelocityCalculator_node*:

- When launched, it takes three parameters: the ticks per revolution of the left and right encoder and the radius of the walker's wheels;

- It subscribe to the *Locomotion/pid_start_stop* topic so that when the locomotion is disabled the node stops calculating the velocity of each motor and when the locomotion is enabled, it starts to calculate the velocities;

- It subscribes to the *LowLevelController/encoders_values* topic to receive the ticks of each encoder. When received, the ticks are summed to a respective variable that accumulates the ticks of each encoder;

- There is a ROS timer that at each 200 milliseconds calls a callback that calculates the distance traveled and the current velocity of each motor (in this 200 milliseconds period) based on the ticks accumulated, the ticks per revolution of the encoder and the wheels' circumference. After the calculation, the accumulation variables are clean;
- The velocities calculated are respectively published to the *Locomotion/left_motor/current_velocity* and *Locomotion/right_motor/current_velocity* topics. Also, both velocities are published together to the *Locomotion/motors_current_velocity*.

(6) PIDLOCOMOTIONENABLER_NODE

The *PIDLocomotionEnabler_node* manages the start/stop of the motors' controller process and also can request the reset of the duty cycle values of the motors. Figure 95 illustrates the inputs and outputs of the node.

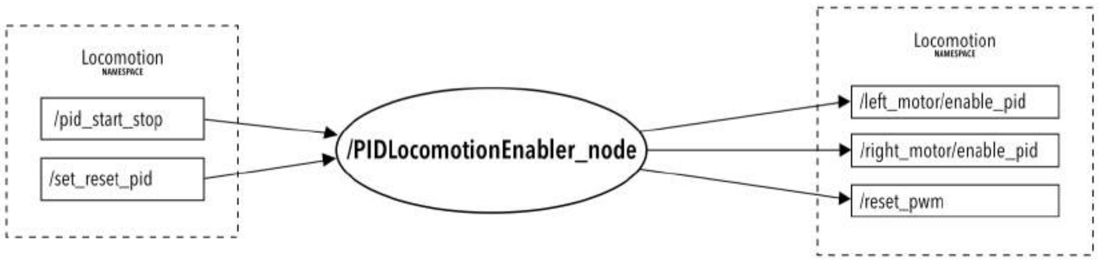


Figure 95: *PIDLocomotionEnabler_node* ROS graph.

So, as to the *PIDLocomotionEnabler_node*:

- It subscribes to the *Locomotion/pid_start_stop* topic meaning that when a *false* value is received, the locomotion will be disabled and when a *true* value is received the locomotion is enabled;
- It subscribes to the *Locomotion/set_reset_pid* topic meaning that when a *false* value is received, the locomotion will not be disabled but instead will be "asleep" and when a *true* value is received the locomotion is "waked";
- It publishes boolean values to the *Locomotion/left_motor/enable_pid* and *Locomotion/right_motor/enable_pid* topics to enable/disable the motors' controller processes and also publishes boolean values to the *Locomotion/reset_pwm* in order to reset (or not) the duty cycles of each motor.

(7) PID CONTROLLERS

There are two *pid_controller* nodes: the */right_motor/pid_controller* for the right motor and the */left_motor/pid_controller* for the left motor. The *Proportional, Integral and Derivative (PID)* controller consists in a control loop feedback strategy in which is calculated the error through the difference between a desired value for a certain variable and the measured value of that certain variable. Then, it is applied a proportional, integral and derivative correction to the error calculated and an output is generated.

So, to apply the PID strategy in the system, a package from ROS , called *pid*, was download and installed which allowed to avoid "re-eventing the wheel" in the implementation of a PID controller. This already implemented package is well validated and also well documented in the wiki of ROS . This way, two *pid* nodes are launched, one for the right motor and one for the left motor. The inputs and outputs of the nodes have the same meaning but are duplicated (respective to each motor) as it can be seen in figure 96.

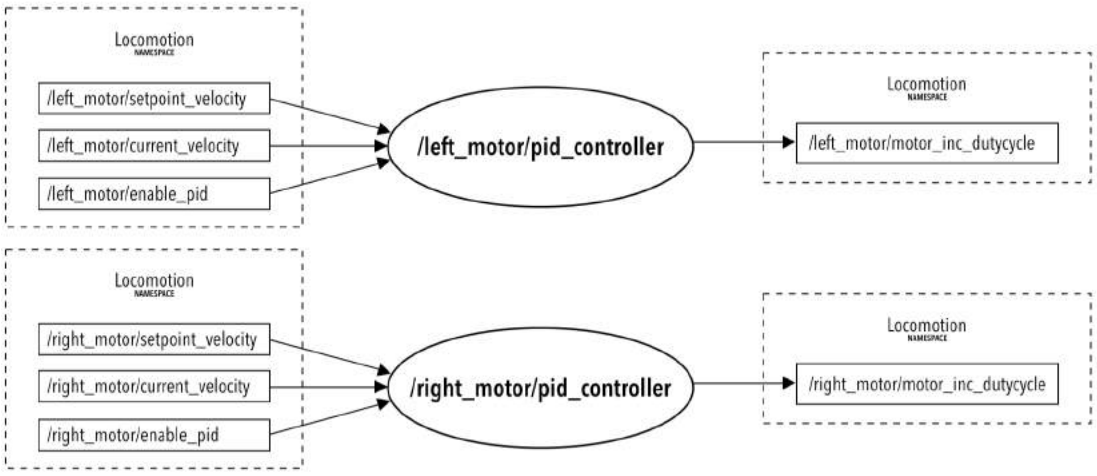

Locomotion
NAMESPACE
/left_motor/setpoint_velocity
/left_motor/current_velocity
/left_motor/enable_pid
/left_motor/pid_controller
Locomotion
NAMESPACE
/left_motor/motor_inc_dutycycle
Locomotion
NAMESPACE
/right_motor/setpoint_velocity
/right_motor/current_velocity
/right_motor/enable_pid
/right_motor/pid_controller
Locomotion
NAMESPACE
/right_motor/motor_inc_dutycycle

Figure 96: ROS graph of the *pid_controller* nodes.

So, as to the *pid* type nodes:

- They subscribe to *.../setpoint_velocity* topics that will contain the setpoint variables (in this case velocities) for the PID controller algorithms;
- They subscribe to *.../current_velocity* topics that will contain the measured variables (in this case measured velocities) for the PID controller algorithms. The *pid* controller only acts when a measured value is received. This means

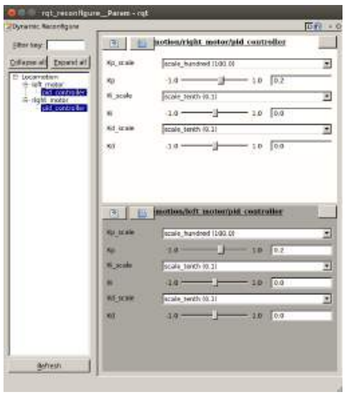
that the frequency of operation of the PID controller algorithm depends on the frequency in which measured values are published to the *.../current_velocity* topics;

- They subscribe to *.../enable_pid* topics that enable or disable the PID controller algorithm;

- They publish to *.../motor_inc_dutycycle* topics the output of the PID controller.

For the ASBGo* system, it was defined that the output of the PID controllers consist in values that are summed or subtracted to the current duty cycle values of each motor. Instead of applying a direct duty cycle value, it is used increments or decrements of the duty cycle. This allows to get a gradual and slow response for the velocity, which consists in a requirement of the ASBGo* due to its purpose (physical rehabilitation of patients with gait disorders).

When the nodes are launched, several parameters can and must be set. The important ones are the gains of the PID controller. For this system it is only used a proportional control. By testing different approaches with all the types of control (proportional, derivative and integral), it was verified that none presented satisfied results like the approach that uses only the proportional and the incremental and decremental values for the duty cycle.



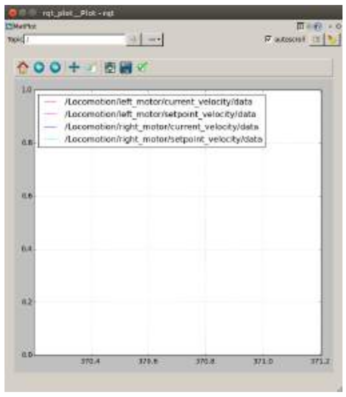
/motion/right_motor/pid_controller

Kp_scale: scale_hundred (100.0)
Kp: -1.0 … 1.0 — 0.2
Ki_scale: scale_tenth (0.1)
Ki: -1.0 … 1.0 — 0.0
Kd_scale: scale_tenth (0.1)
Kd: -1.0 … 1.0 — 0.0

/motion/left_motor/pid_controller

Kp_scale: scale_hundred (100.0)
Kp: -1.0 … 1.0 — 0.2
Ki_scale: scale_tenth (0.1)
Ki: -1.0 … 1.0 — 0.0
Kd_scale: scale_tenth (0.1)
Kd: -1.0 … 1.0 — 0.0

(a)


rqt_plot__Plot - rqt
MatPlot
Topic
autoscroll
/Locomotion/left_motor/current_velocity/data
/Locomotion/left_motor/setpoint_velocity/data
/Locomotion/right_motor/current_velocity/data
/Locomotion/right_motor/setpoint_velocity/data
1.0
0.8
0.6
0.4
0.2
0.0
370.4
370.6
370.8
371.0
371.2

(b)

Figure 97: ROS *pid* features: (a) Dynamically reconfigurations of gains; (b) Plotting PID variables.

ROS provides a way of configure the gains dynamically in order to test different configurations, in real-time, to in the end check the best combination and so, set static values for the gains. Also, it provides a way of monitoring, in a real-time plot, the different variables of the PID controller (see figure 97).

(8) motorspwm_node

The *MotorsPWM_node* is responsible to manage the output of the PID controllers in order to be outputted duty cycles for the motors' actuation. Figure 98 shows the inputs and outputs of the node.

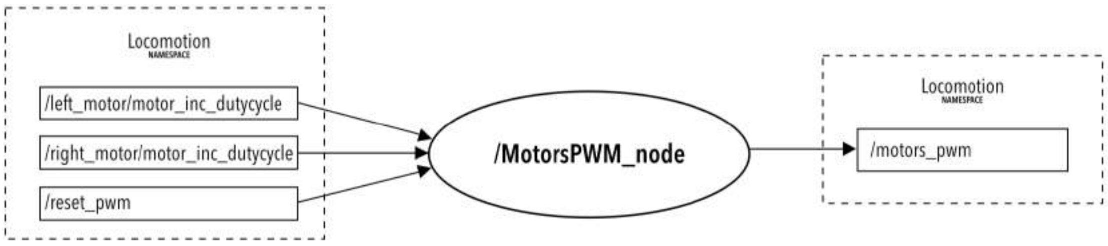


Figure 98: *MotorsPWM_node* ROS graph.

So, as to the *MotorsPWM_node*:

- It subscribes to the *Locomotion/left_motor/motor_inc_dutycycle* and *Locomotion/right_motor/motor_inc_dutycycle* topics which will contain the outputs (increment or decrement fragments for the duty cycle) of the respective *pid_controllers* nodes. When the node is launched, the variables that continuously store the actual value of the duty cycle to each motor, *left_dc* and *right_dc*, present a zero value;

- When values are received from the *Locomotion/left_motor/motor_inc_dutycycle* and *Locomotion/right_motor/motor_inc_dutycycle* topics, they are respectively summed (or subtracted) to the *left_dc* and *right_dc* variables. There are also conditions that limit the duty cycles to a maximum value of 100 and a minimum value of -100;

- When new duty cycles values are generated, they are published to the *Locomotion/motors_pwm* topic in order to be consumed by the *LowLevelController_node* and so, sent to the *Low-Level Controller* unit;

- It subscribes to the *Locomotion/reset_pwm* in which when a *true* value is received, the duty cycle variables are reset and published to the *Locomotion/motors_pwm* topic.

(9) MOTORSDETECTIONS_NODE

The *MotorsDetections_node* is responsible to, given a 'stop' drive command, detect when the motors stop in order to put the locomotion in a sleep state (disable *pid_controllers* nodes and reset duty cycles). Since the motors present an electromechanical brake system, it is possible to rotate a handle, in each motor, which decouples the motor from the motor's shaft. This way it is possible to push the walker. Also, if PWM s are applied to the motors, they will rotate but the shaft will not. This constitutes a problem in a way that if drive commands are generated, the motors will rotate and since the motors' shaft will not and the encoders are coupled to the shafts, the system will detect null velocities and will try to compensate more and more until maximum duty cycles are reached. To avoid this, it is taking into account the current drive command and the motors' velocities. When it is given a drive command to move the walker, it is started a ROS timer. After three seconds, it is executed a callback and it is checked if the velocities are still zero. If they are, it means that the handles of the electromechanical brake system of the motors were rotated and so, a command to reset the duty cycles is given. Figure 99 reveals the inputs an outputs of the node.

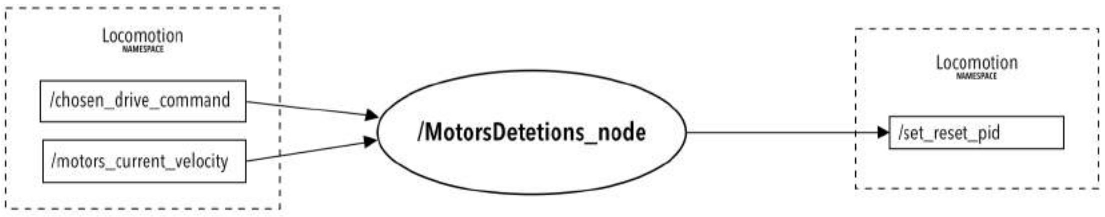


Figure 99: *MotorsDetections_node* ROS graph.

So, as to the *MotorsDetections_node*:

- It subscribes to the *Locomotion/chosen_drive_command* topic to be aware of the current drive command;
- It subscribes to the *Locomotion/motors_current_velocity* topic to be aware of the current velocities of the motors;
- It publishes boolean values to the *Locomotion/set_reset_pid* topic in order to put the locomotion in sleep or wake mode.

(10) LOCALGUI_NODE

The *LocalGUI_node* consists in the local GUI of the ASBGo* . It was used QML, a

JSON-like declarative language for the design of the interface, and it was written a C++ class, the *_LocalGUI*, that serves as an intermediary between the front-end (interface) and the back-end (the rest of the ROS nodes). The main execution of the node is in the class. So, it receives information from others nodes, processes the information and acts on the front-end. Also, when the users navigate through the interface, the front-end calls specific methods from the class in order to be sent information to the other nodes. Figure 100 shows the inputs and outputs of the node.

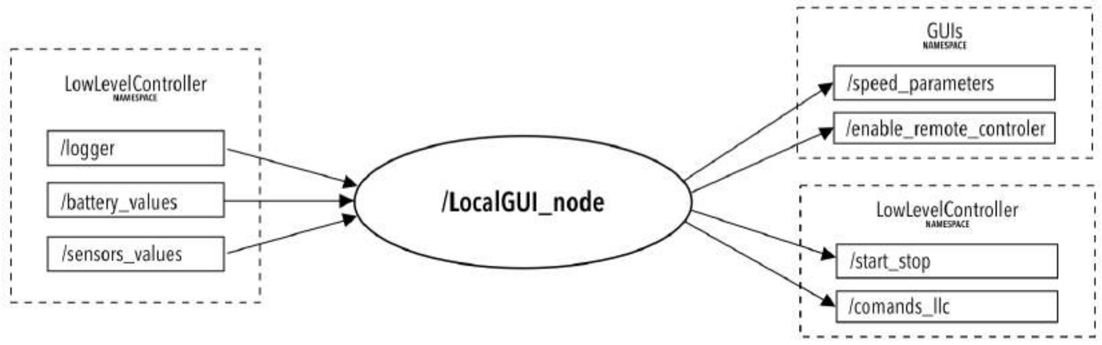


Figure 100: *LocalGUI_node* ROS graph.

So, as to the *LocalGUI_node*:

- It subscribes to the *LowLevelController/logger* topic in order to receive logs from the *LowLevelController_node* and act in the front-end according to the log received;
- It subscribes to the *LowLevelController/battery_values* topic in order to get the battery's voltage, calculate an estimation of the percentage of the battery and print that value in the front-end;
- It subscribes to the *LowLevelController/sensors_values* topic in order to get the sensors' data to be displayed in the interface;
- It publishes to the *GUIs/speed_parameters* topic the parameters that define the locomotion of the walker (velocity and curvature). Those values are chosen by the user in the graphical interface and published inside the class.
- It publishes to the *GUIs/enable_remote_controller* topic boolean values to enable/disable the remote controller. Note that the remote controller is only used to drive the walker and so, it is only enabled in specific pages of the local GUI ;

- It publishes to the *LowLevelController/start_stop* topic boolean values to start/stop the main execution of the *LowLevelController_node*. A *true* value is published to this topic in the startup of the GUI and a *false* value before shutdown the system;
- It publishes to the *LowLevelController/commands_llc* topic commands for the *LowLevelController_node* like commands to: choose the sensors that will operate, enable/disable the motors and 'start'/'stop' the action in the *Low-Level Controller* unit. This commands are sent, inside the class, according to the configurations of the user in the local GUI (front-end).

(11) REMOTECONTROLLER_NODE

The *RemoteController_node* is responsible to handle the information from the remote controller. Note that the remote controller is only used to drive the walker. Figure 101 illustrates the inputs and outputs of the nodes.

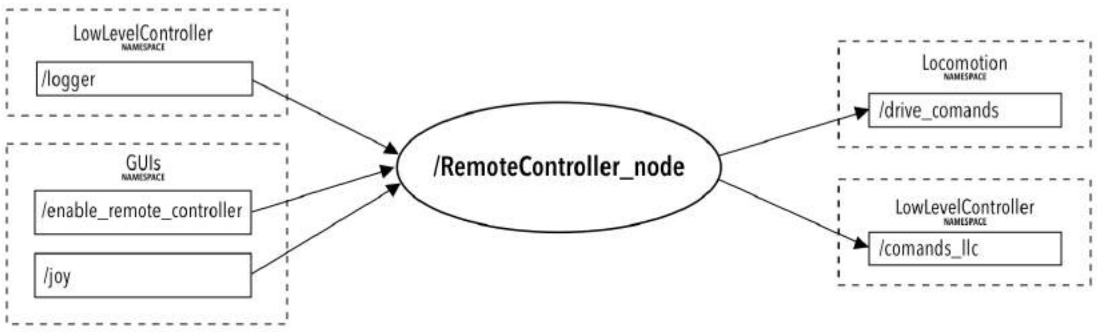


Figure 101: *RemoteController_node* ROS graph.

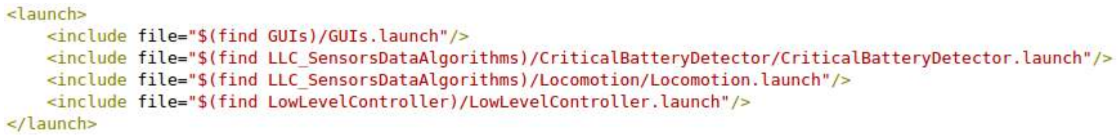
So, as to the *RemoteController_node*:

- It subscribes to the *GUIs/joy* topic that contains the current state of each one of the remote controller's buttons and axes. To have this type of topic, it was download and installed a ROS package called *joy*. This packages consists in ROS driver for generic Linux joysticks. The package contains the *joy_node*, a node that interfaces with the controller and publishes the information to the *GUIs/joy* topic. It is necessary to give the path to the device file of the remote controller;
- It subscribes to the *GUIs/enable_remote_controller* topic which is used to enable/disable handle the states received from the *GUIs/joy* topic. Since the remote controller is only used to drive the walker, it is only enabled in specific pages of the local GUI ;

- It subscribes to the *LowLevelController/logger* topic to take into account some logs of the *LowLevelController_node*;
- It publishes to the *LowLevelController/commands_llc* topic commands such as the 'start' and 'stop' commands in order to start/stop the action in the *Low-Level Controller* unit (sensors' acquisitions and enable motors);
- It publishes to the *Locomotion/drive_commands* topic drive commands according to the coordinates of the remote controller's joystick. Note that it is impossible to drive the walker backwards with the handlebar but it is possible with the remote controller.

4.4.4.3 *System startup*

When the *Main Controller* is turned ON, and after booting the *Ubuntu 14.04* OS and perform the login, a startup shell script is executed. That script sets up environment variables needed by ROS and then, a ROS launch file created, called *MainController.launch*, is launched.



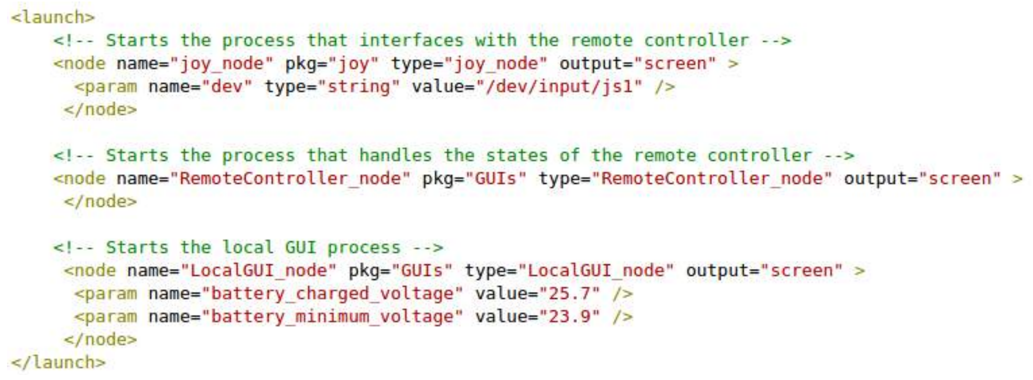
```
<launch>
    <include file="$(find GUIs)/GUIs.launch"/>
    <include file="$(find LLC_SensorsDataAlgorithms)/CriticalBatteryDetector/CriticalBatteryDetector.launch"/>
    <include file="$(find LLC_SensorsDataAlgorithms)/Locomotion/Locomotion.launch"/>
    <include file="$(find LowLevelController)/LowLevelController.launch"/>
</launch>
```

Figure 102: *MainController.launch*

As it can be seen in figure 102, the *MainController.launch* executes another four ROS launch files that are responsible to launch the different nodes discussed in 4.4.4.2 - *ROS nodes*. As also showed in 4.4.4.2 - *ROS nodes*, some nodes take some parameters when launched which allows to change the value of some important variables of the nodes without the need of re-compilation of the source code.

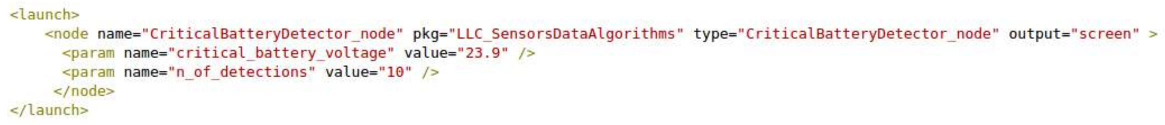
Future nodes can be launched inside the already existing launch files (if it makes sense conceptually) or a new launch file can be created and included in the *MainController.launch*.

Figure 103 shows the first launch file executed in the *MainController.launch*, the *GUIs.launch*. It is launched: (1) the *joy_node*, the one that interfaces with the remote controller. It takes a parameter consisting in the Linux device file path of the remote controller; (2) the *RemoteController_node*, responsible to handle the states of the remote controller; and (3) the *LocalGUI_node* where it runs the local GUI of the device. It takes two parameters consisting in the maximum and minimum voltages of the battery.



```
<launch>
    <!-- Starts the process that interfaces with the remote controller -->
    <node name="joy_node" pkg="joy" type="joy_node" output="screen" >
      <param name="dev" type="string" value="/dev/input/js1" />
     </node>

    <!-- Starts the process that handles the states of the remote controller -->
    <node name="RemoteController_node" pkg="GUIs" type="RemoteController_node" output="screen" >
     </node>

    <!-- Starts the local GUI process -->
     <node name="LocalGUI_node" pkg="GUIs" type="LocalGUI_node" output="screen" >
      <param name="battery_charged_voltage" value="25.7" />
      <param name="battery_minimum_voltage" value="23.9" />
     </node>
</launch>
```

Figure 103: *GUIs.launch*

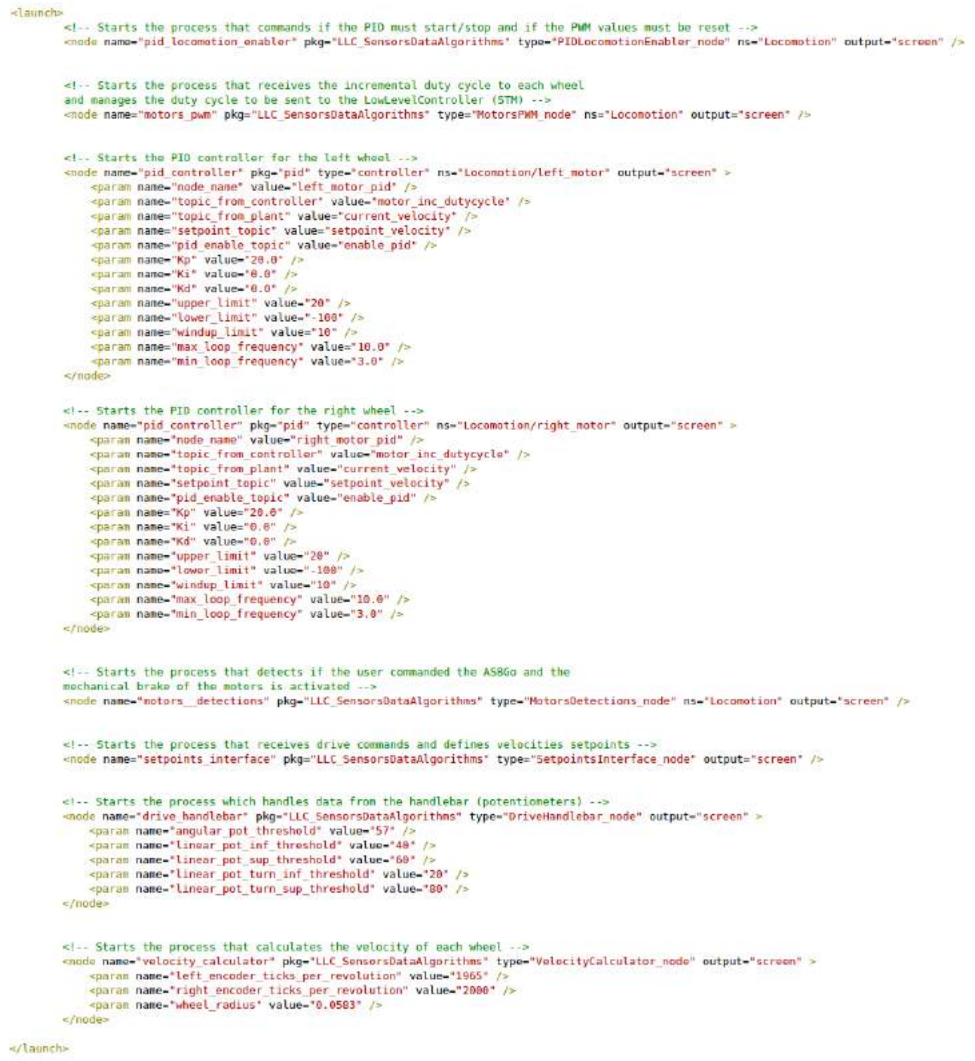


```
<launch>
    <node name="CriticalBatteryDetector_node" pkg="LLC_SensorsDataAlgorithms" type="CriticalBatteryDetector_node" output="screen" >
      <param name="critical_battery_voltage" value="23.9" />
      <param name="n_of_detections" value="10" />
     </node>
</launch>
```

Figure 104: *CriticalBatteryDetector.launch*

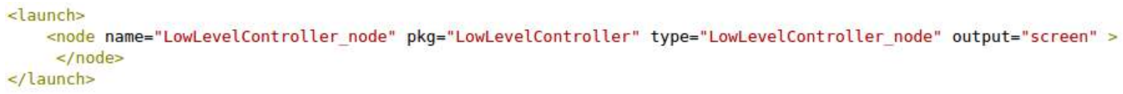
The second launch file executed in the *MainController.launch* is the *CriticalBatteryDetector.launch* (see figure 104). It launches the *CriticalBatteryDetector_node* with one parameter that consists in the minimum battery's voltage and another that consists in the number of detections that are counted before a warning is triggered.

Figure 105 shows the third launch file executed, the *Locomotion.launch*. In this file is launched: (1) the *PIDLocomotionEnabler_node*, responsible to manage the PID controllers (enable/disable the control algorithms and reset the motors' duty cycles); (2) the *MotorsPWM_node*, which manages the duty cycles of each motors, (3) the *pid_controller* nodes for each motor with the gains of the controller as parameters; (4) the *MotorsDetections_node*, responsible to detect a peculiar situation of the motors and also if they are stopped; (5) the *SetpointsInterface_node*, which decodes drive commands and generates velocity setpoint values; (6) the *DriveHandlebar_node* that manages the state of the walker's handlebar in order to output a drive command. It takes five parameters that correspond to threshold values used in the evaluation of the state of the handlebar; and (7) the *VelocityCalculator_node*, responsible to calculate the velocity of each wheel of the walker. It takes as parameters the ticks per revolution of each encoder and the radius of the wheels.



```
<launch>
        <!-- Starts the process that commands if the PID must start/stop and if the PWM values must be reset -->
        <node name="pid_locomotion_enabler" pkg="LLC_SensorsDataAlgorithms" type="PIDLocomotionEnabler_node" ns="Locomotion" output="screen" />


        <!-- Starts the process that receives the incremental duty cycle to each wheel
        and manages the duty cycle to be sent to the LowLevelController (STM) -->
        <node name="motors_pwm" pkg="LLC_SensorsDataAlgorithms" type="MotorsPWM_node" ns="Locomotion" output="screen" />


        <!-- Starts the PID controller for the left wheel -->
        <node name="pid_controller" pkg="pid" type="controller" ns="Locomotion/left_motor" output="screen" >
            <param name="node_name" value="left_motor_pid" />
            <param name="topic_from_controller" value="motor_inc_dutycycle" />
            <param name="topic_from_plant" value="current_velocity" />
            <param name="setpoint_topic" value="setpoint_velocity" />
            <param name="pid_enable_topic" value="enable_pid" />
            <param name="Kp" value="20.0" />
            <param name="Ki" value="0.0" />
            <param name="Kd" value="0.0" />
            <param name="upper_limit" value="20" />
            <param name="lower_limit" value="-100" />
            <param name="windup_limit" value="10" />
            <param name="max_loop_frequency" value="10.0" />
            <param name="min_loop_frequency" value="3.0" />
        </node>

        <!-- Starts the PID controller for the right wheel -->
        <node name="pid_controller" pkg="pid" type="controller" ns="Locomotion/right_motor" output="screen" >
            <param name="node_name" value="right_motor_pid" />
            <param name="topic_from_controller" value="motor_inc_dutycycle" />
            <param name="topic_from_plant" value="current_velocity" />
            <param name="setpoint_topic" value="setpoint_velocity" />
            <param name="pid_enable_topic" value="enable_pid" />
            <param name="Kp" value="20.0" />
            <param name="Ki" value="0.0" />
            <param name="Kd" value="0.0" />
            <param name="upper_limit" value="20" />
            <param name="lower_limit" value="-100" />
            <param name="windup_limit" value="10" />
            <param name="max_loop_frequency" value="10.0" />
            <param name="min_loop_frequency" value="3.0" />
        </node>


        <!-- Starts the process that detects if the user commanded the ASBGo and the
        mechanical brake of the motors is activated -->
        <node name="motors__detections" pkg="LLC_SensorsDataAlgorithms" type="MotorsDetections_node" ns="Locomotion" output="screen" />


        <!-- Starts the process that receives drive commands and defines velocities setpoints -->
        <node name="setpoints_interface" pkg="LLC_SensorsDataAlgorithms" type="SetpointsInterface_node" output="screen" />


        <!-- Starts the process which handles data from the handlebar (potentiometers) -->
        <node name="drive_handlebar" pkg="LLC_SensorsDataAlgorithms" type="DriveHandlebar_node" output="screen" >
            <param name="angular_pot_threshold" value="57" />
            <param name="linear_pot_inf_threshold" value="40" />
            <param name="linear_pot_sup_threshold" value="60" />
            <param name="linear_pot_turn_inf_threshold" value="20" />
            <param name="linear_pot_turn_sup_threshold" value="80" />
        </node>


        <!-- Starts the process that calculates the velocity of each wheel -->
        <node name="velocity_calculator" pkg="LLC_SensorsDataAlgorithms" type="VelocityCalculator_node" output="screen" >
            <param name="left_encoder_ticks_per_revolution" value="1965" />
            <param name="right_encoder_ticks_per_revolution" value="2000" />
            <param name="wheel_radius" value="0.0583" />
        </node>

</launch>
```

Figure 105: *Locomotion.launch*



```
<launch>
    <node name="LowLevelController_node" pkg="LowLevelController" type="LowLevelController_node" output="screen" >
     </node>
</launch>
```

Figure 106: *LowLevelController.launch*

The last executed launch file in the *MainController.launch* is the *LowLevelController.launch* (see figure 106). In this file is only launched the *LowLevelController_node*.

5

FINAL RESULT AND SYSTEM VALIDATION

It will be now presented the device resulted from all the effort and commitment dedicated to the project. Just to remember, figure 107 shows the starting point of this thesis, consisting in just the walker's mechanical frame. The frame was outlined in the ASBGo++ version and it was based on many relevant considerations that were also planned in that version. (*From now on, it is advised to open this document with the Adobe Acrobat Reader to allow the visualization of videos.*)

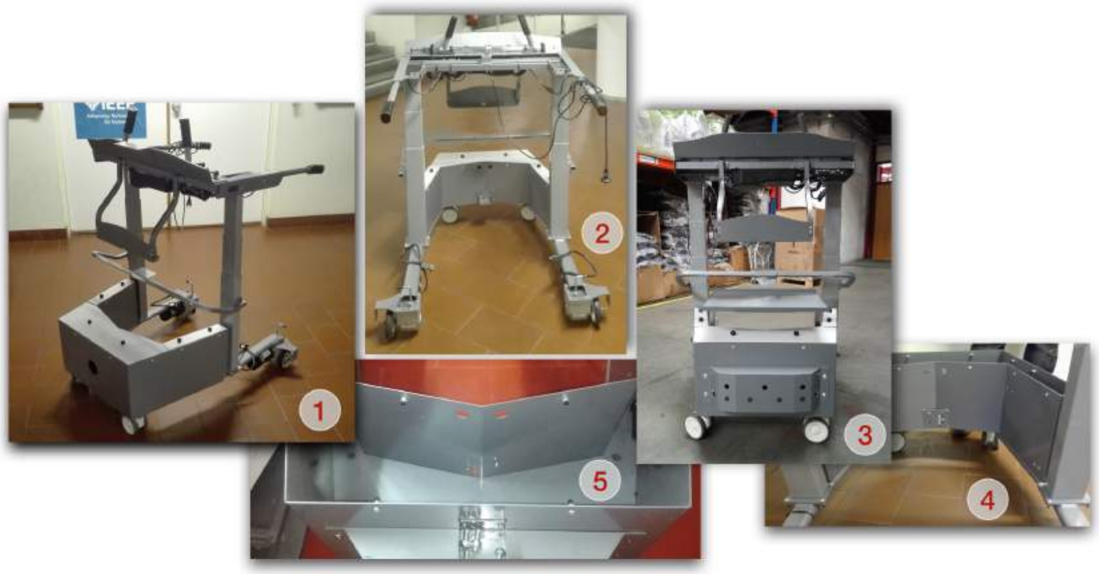


Figure 107: Walker's mechanical frame: (1)(2)(3) Walker's frame in different perspectives; (4) Interior section of the frame; (5) Walker's compartment that will contain all the electric system.

5.1 FINAL RESULT

5.1.1 *Mechanical modifications*

This subsection will present the mechanical modifications that arise as a requirement to integrate the electric system in the new device. All the modifications presented were performed by the author of this thesis with the support of a close familiar with knowledge and practice in mechanics.

Figure 108 shows the application of a new stopper on the handlebar in order to avoid the destruction of the plastic angular potentiometer.

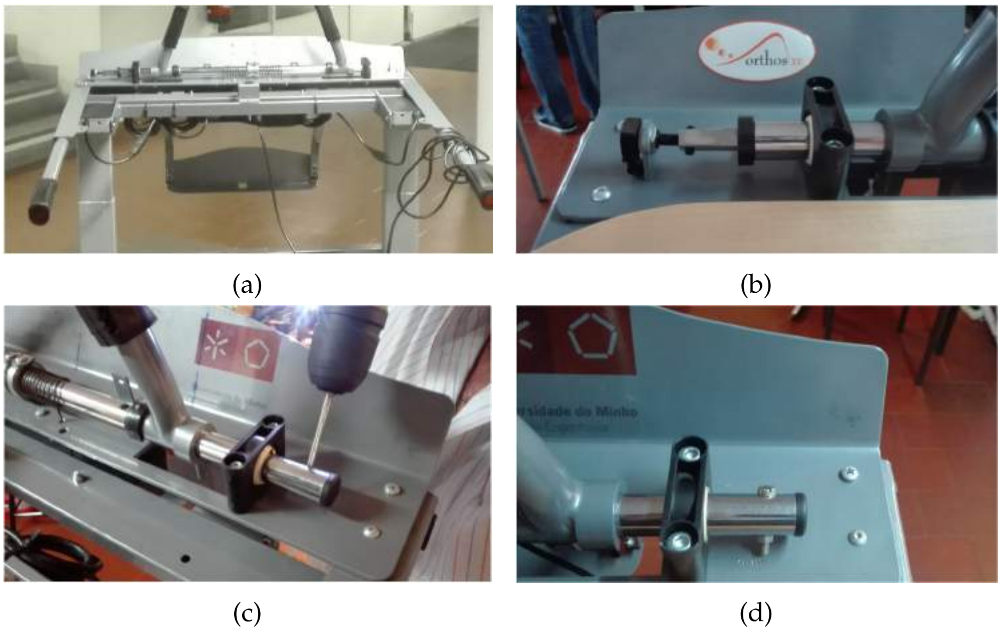


Figure 108: Application of a stopper on the handlebar: (a) Walker's handlebar; (b) Angular potentiometer; (c) Drilling the sliding pipe; (d) Stopper integrated.

Before the modification, when the handlebar was completely moved to the left, the sliding pipe was hitting the angular potentiometer degrading it (see figure 108b). So, by putting a stopper in the right side of the sliding pipe, the movement of the handlebar to the left is limited, not compromising the locomotion and preventing the destruction of the sensor (see figure 108d).

Figure 109 shows a mechanical modification related to the URFs in which it resulted a strategy that eases the application and the attachment of the sensors to the physical structure. In order to stably fix the sensors in the walker, two mechanical pieces were developed. This way, it is possible to put the sensors in the holes (which were also filled with rubber to add some friction) and by pressing the mechanical pieces to the sensors and by screwing them, the URFs get firmly attached (see figure 109b). Also, since two URFs are pointing up and one is pointing down (figure 109a), one of the mechanical pieces were cut in such a way that it forces this three sensors to point to their respective direction (figure 109b and 109c).

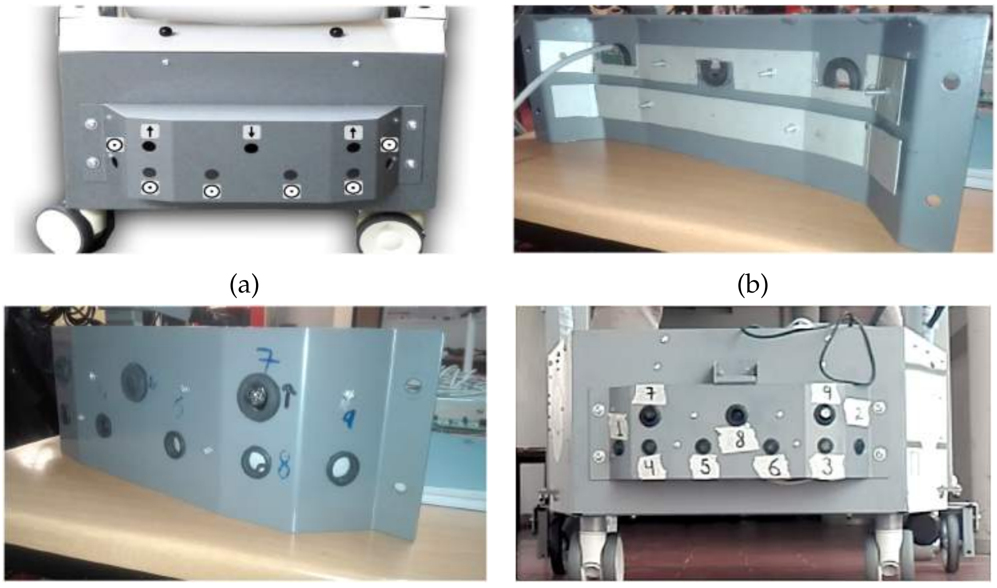


Figure 109: Strategy for the attachment of the URFs : (a) Pointing directions; (b) Structures developed that stably fix the sensors; (c) Sensors pointing to their respective direction; (d) All sensors coupled and pointing to the correct direction.

Since the space of the walker's compartment is very limited for all the electric system developed, the space had to be very well exploited in order to gather everything. Being the batteries the material with greater volume, they had to be mandatorily putted in the larger part of the compartment. On the other hand, being the PCBs and the motors' drivers thinner, it was decided that they would be attached to the sides of the compartment (see figure 110a). It was also planned to divide the larger part of the compartment in floors and so a mechanical piece, with the form of the compartment, was designed (see figure 110b). This allowed to allocate all the components in an organized way.

As referred, the PCBs and the motors' drivers were coupled to the sides of the walker's compartment. Being these parts very narrow, windows had to be made in order to increase the accessibility to the boards and so, to the sides of the compartment. Figure 111 presents the modification process.

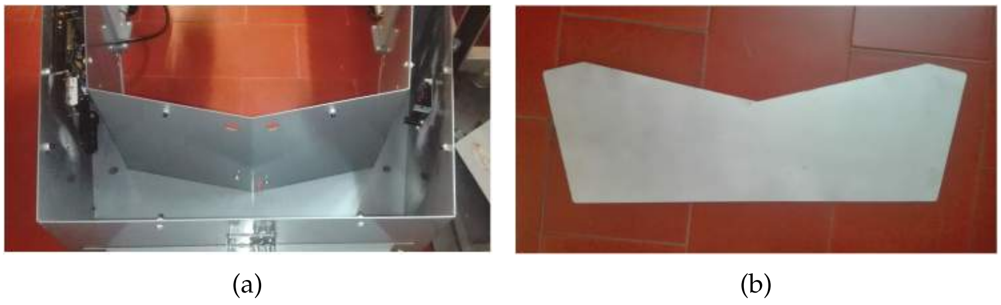


Figure 110: (a) Walker's compartment with the PCBs on the left side and the motors' drivers on the right side; (b) Mechanical piece that allows to divide the walker's compartment in floors.

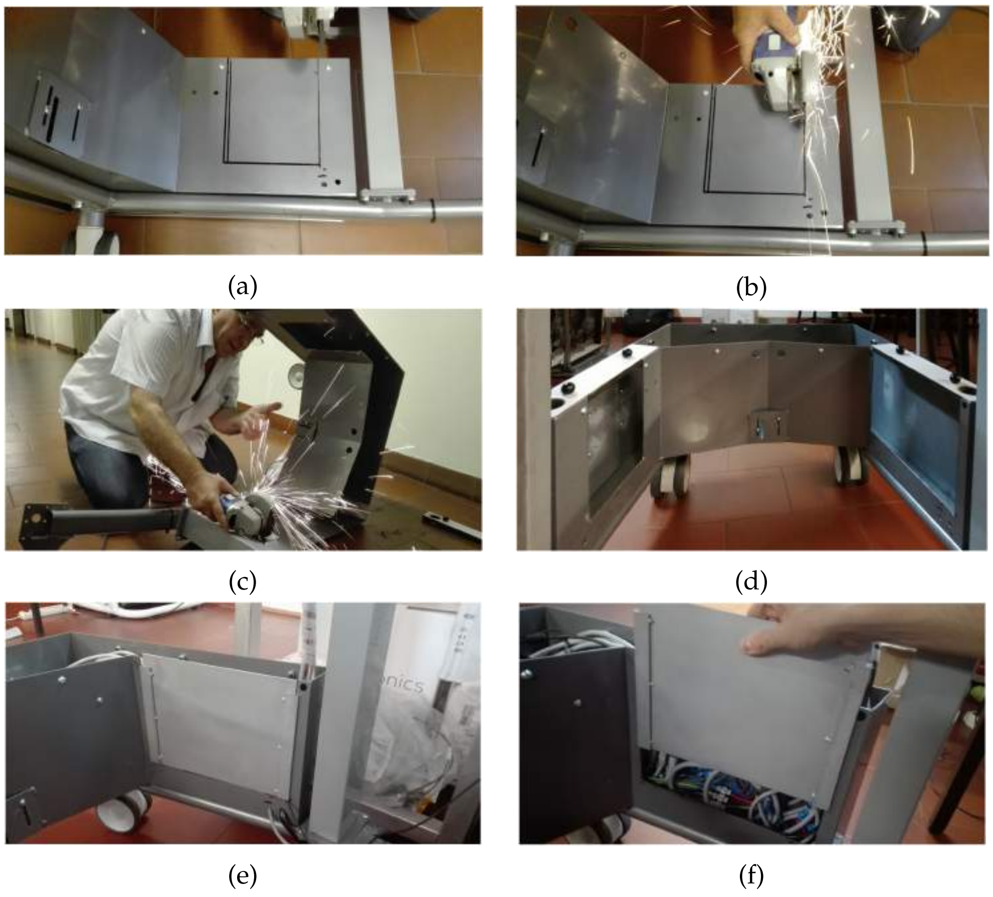


Figure 111: Making the access windows: (a) Planning the cuts; (b)(c) Cutting the frame; (d) Windows accomplished; (e) Right window closed; (f) Right window being opened.

To couple the PCBs and the motors' drivers to the compartment, many holes had to be drilled. Also, holes were made to allow passing the wires inside the compartment from one side to another (see figure 112b).

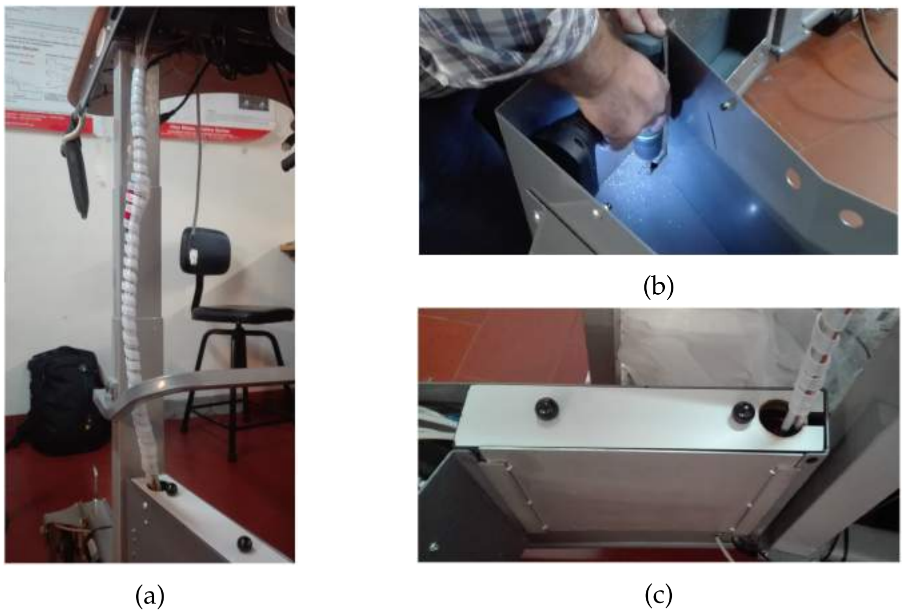
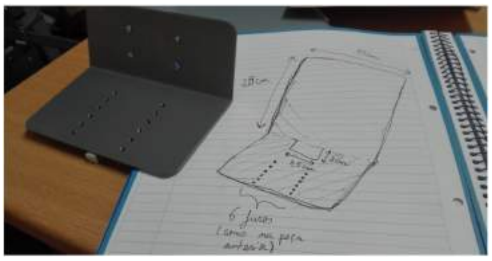


Figure 112: (a) Cables between the upper part of the walker and its compartment; (b) Holes drilled in the frame; (c) Holes and cuts in the wood lids.

Many wires must pass to the upper part of the walker and so, they were gathered in a plastic coiled cable clamp (figure 112a) and holes had to be made in the wood lids of the walker's compartment. Also, it were performed cuts in the same wood lids, consisting in a kind of paths where the wires can pass through, easing the extraction of the lids (figure 112c).

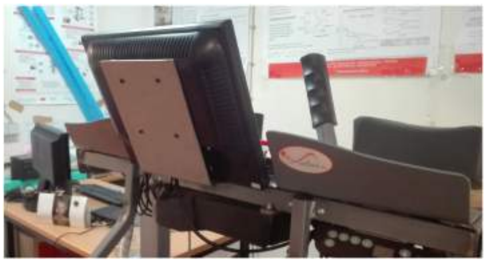
For the local HMI , a touchscreen monitor had to be added to new the device. It had to be coupled in a place visually accessible to the patient and also accessible to the medical specialist. Therefore, it was decided that the monitor had to be coupled behind the handlebar. However, in that position, it was planned (in the ASBGo++ version) that it would be a place that would support an RGB-D camera (figure 113a). That makes no difference because the camera can be coupled above the monitor. This way, it was taken in advantage the form of the mechanical piece that would support the camera (figure 113a) to design a new similar one, with bigger dimensions, to support the monitor (figure 113b). It resulted the piece that is supporting the monitor in figure 113d. Also, a considerable cut had to be made (figure 113c) in the upper part of the walker's frame to allow coupling the monitor. In the end, the monitor was successfully attached as it can be seen in figure 113d, 113e and 113f.

(a)

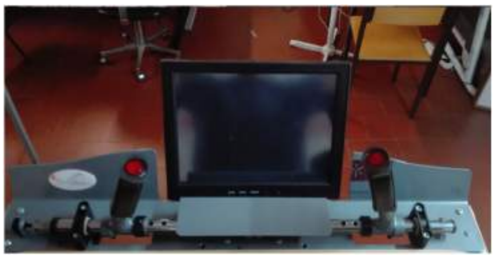


(b)

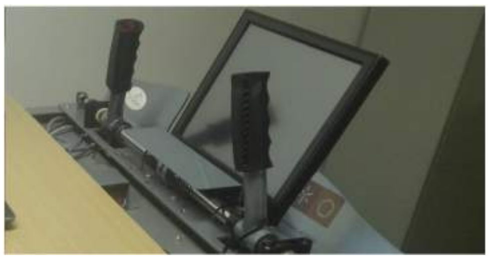
(c)



(d)



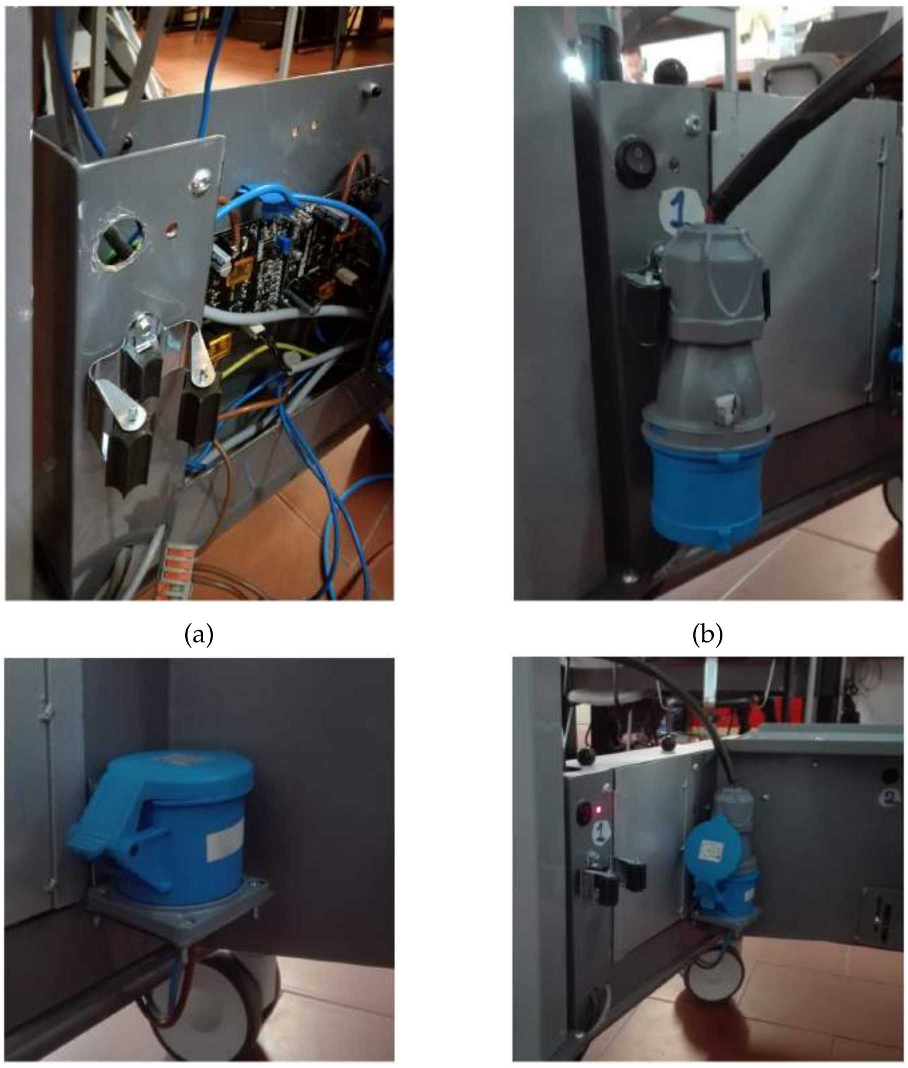
(e)



(f)

Figure 113: Mechanical modifications to couple the touchscreen monitor: (a) Location chosen and mechanical piece used to create a new similar one; (b) Old mechanical piece and a draw of the new one; (c) Cutting a portion of the upper part of the walker's frame; (d)(e)(f) Final result in different perspectives.

Figure 114a shows the holes that were made to couple the switch that allows to power the system ON and OFF and to couple the LED that indicates the state of the system i.e. if it is ON or OFF. It is also represented the holder that holds the male power plug connector of the system (see figure 114b). Figure 114c shows the female plug connector that is directly connected to the battery of the system. When the male power plug connector of the system is connected to the female plug connector, the system can be turned ON using the switch (figure 114d). Otherwise, if the male power plug connector of the battery charger (that will be explained later) is connected to the female plug connector, the battery of the system is charged.



(a)

(b)

(c)

(d)

Figure 114: (a) Holder for the male power plug connector and the holes for the switch that allows to power the system and for the LED that indicates the state of the system (ON/OFF); (b) Switch and LED coupled and the male power plug connector held; (c) Female plug connector that is connected to the battery of the system; (d) Male power plug connector connected to the female plug connector (battery) and system switched ON.

5.1.2 *Sensors and emergency button*

In this subsection it will be revealed the different locations of the sensors and the emergency button as well as the way that they are coupled.

Figure 115a shows the angular potentiometer and figure 115b the linear potentiometer that is bellow the mechanical piece pointed by the red arrow. Figure 116 reveals the position of the IR sensor in which the mechanical piece where it is coupled was also designed and created as well as holes had to be drilled.

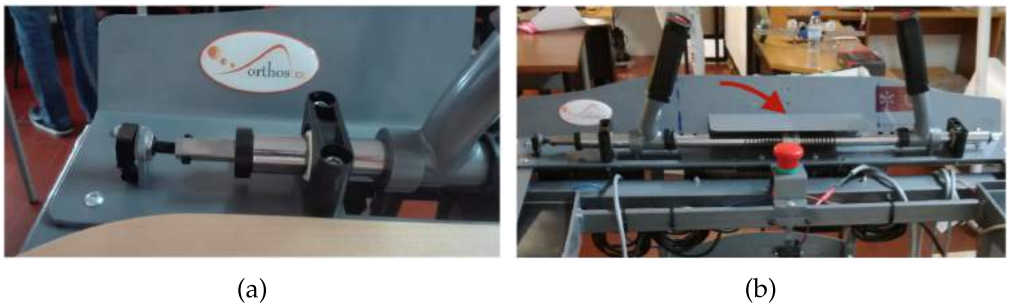


Figure 115: (a) Angular potentiometer coupled in the device; (b) Linear potentiometer bellow the mechanical piece pointed by the red arrow.

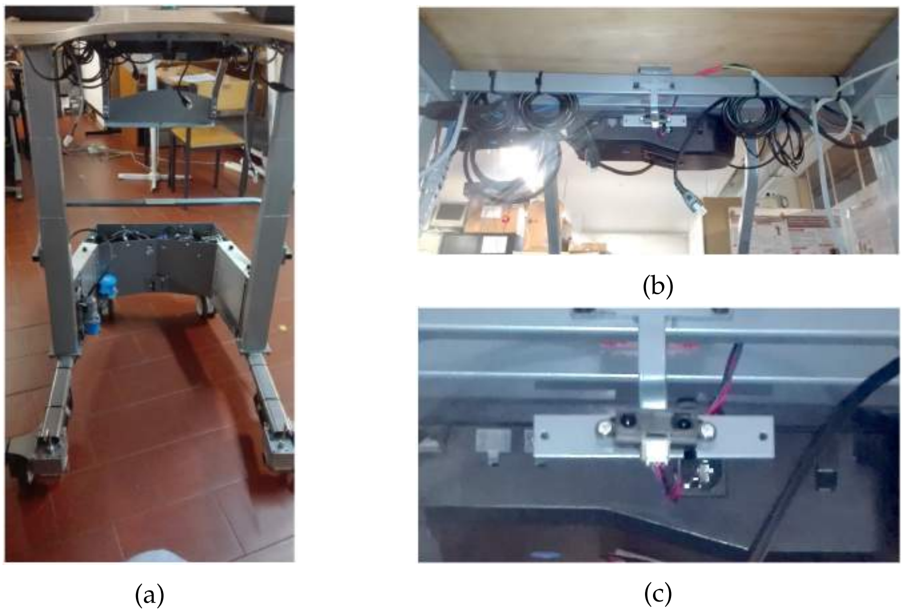
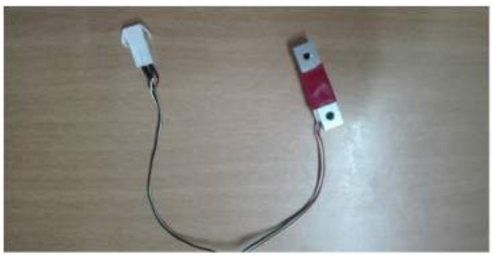
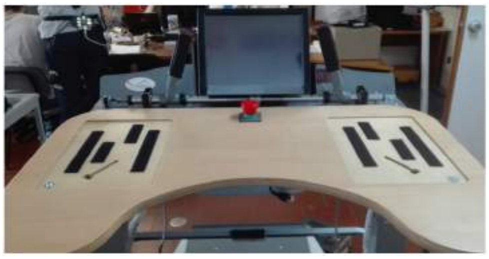


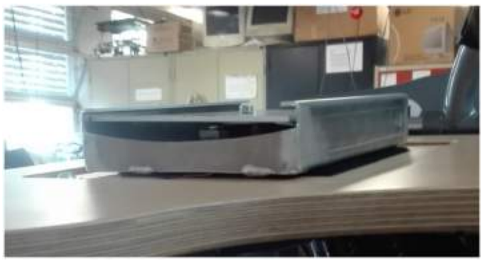
Figure 116: Position of the IR sensor (convergence to the sensor from (a) to (c)).

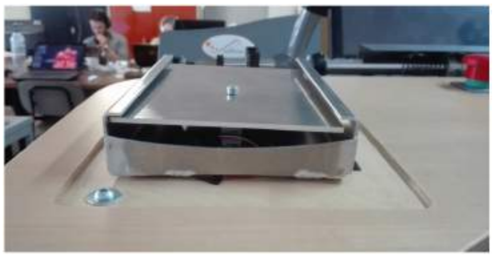
The load cells, as well as the rest of the sensors, can be easily detached. So, figure 117a presents one of the two load cells decoupled from the system. The sensor is then coupled to a kind of mechanical balance (figure 117d and 117e) which is then placed inside a foam padding. In turn, all the set is then placed on the wood platform as shown in figure 117f. From here, it is only necessary to connect the sensor to the system as demonstrated in figure 117c. The strategy to couple the load cells was done in collaboration with another member of the ASBGo as well as the table (figure 117b) was provided by the supporting company, the Orthos XXI.


(a)


(b)

(c)


(d)


(e)

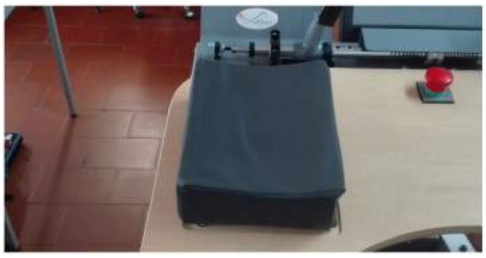

(f)

Figure 117: Load cells' application in the device: (a) Load cell; (b) Wood platform; (c) System's connector to the sensor; (d)(e) Mechanical balance containing the load cell; (f) Foam padding wrapping the mechanical balance and so, the sensor.

Figure 118 pictures the location of the left motor's encoder. The red arrow points to the encoder which is inside a mechanical piece. A customized shaft was designed and developed to link the sensor to the motor. For the right motor's encoder, the same principles are applied.

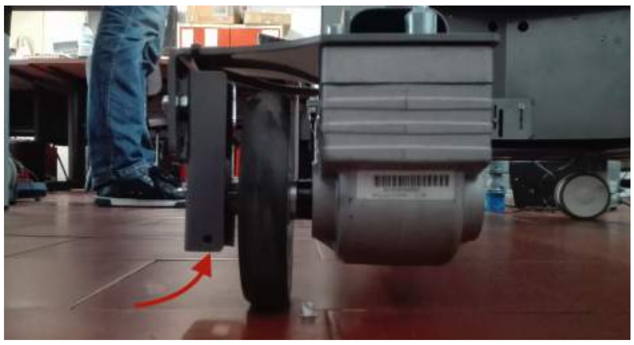
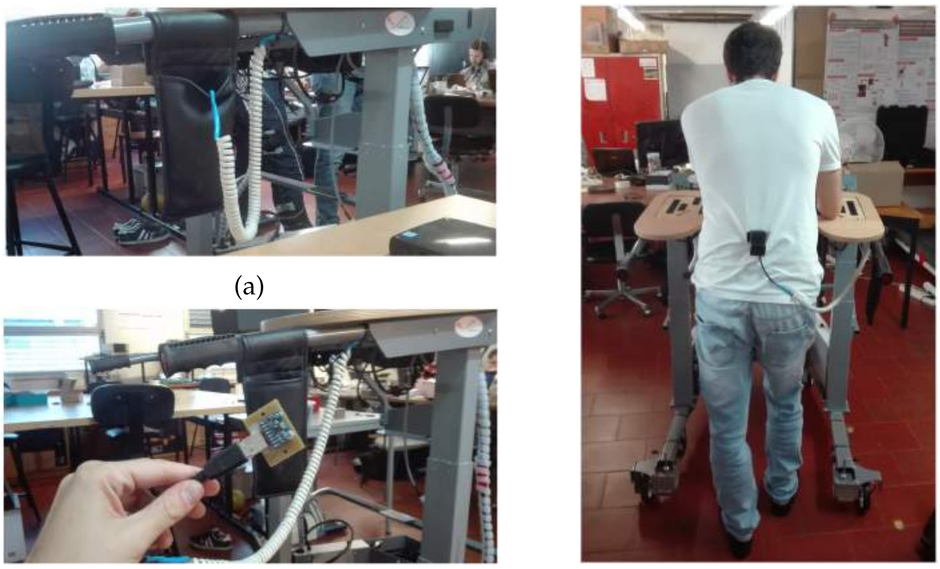


Figure 118: Encoder coupled to the left motor.

Figure 119b shows the IMU coupled to its acquisition board. The board is connected to the system through a coiled cable and it can be stored in a protection bag (figure 119a). It is presented in figure 119c the expected position of the sensor in the patient's body. The sensor is inside a small box that will be developed by a member of the ASBGo team.


(a)

(b)

(c)

Figure 119: (a) IMU stored inside a protection bag; (b) IMU and its acquisition board; (c) Simulation of the IMU in a person's body.

As to the URFs , figure 120 shows all those sensors well coupled and pointing to their respective direction.

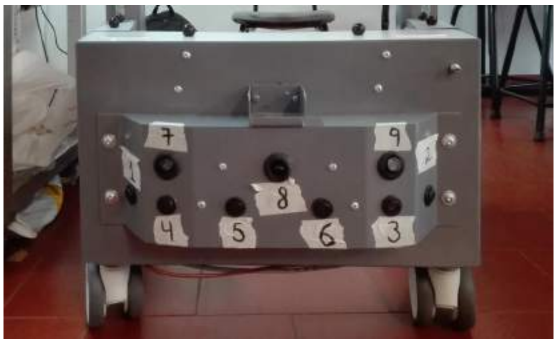


Figure 120: All the URFs coupled in the device.

Figure 121 reveals the emergency button coupled to the new device.

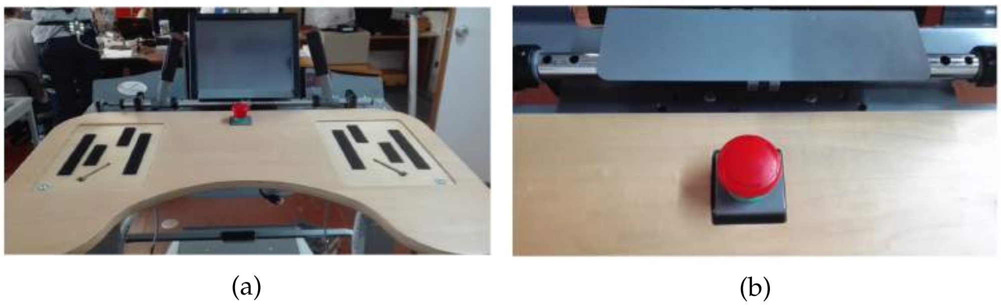


Figure 121: Emergency button coupled in the device.

5.1.3 *Material disposition*

As referred, it was noticed, during all the development, that the space of the walker's compartment was very limited. This way, all the space had to be very well exploited in order to all the components fit in an organized way.

Some of the thinner components were coupled to the sides of the compartment and its larger part was divided in floors.

Figure 122 and 123 illustrates the division in levels. It is relevant to say that in the third floor there is the material that have more permission to be accessed by future developers. Material such as the on-board computer, the USB HUB, the thermal relay (that indicates if a lot of current is flowing to the system) and the PCB that powers and protects the on-board computer, the USB HUB and the touchscreen monitor. In the other floors are present the more electrical components that consists now in an abstraction for future developers: in the second floor, the step-downs and the 3.3V and 5V protection PCB ; and in the first floor, the batteries.

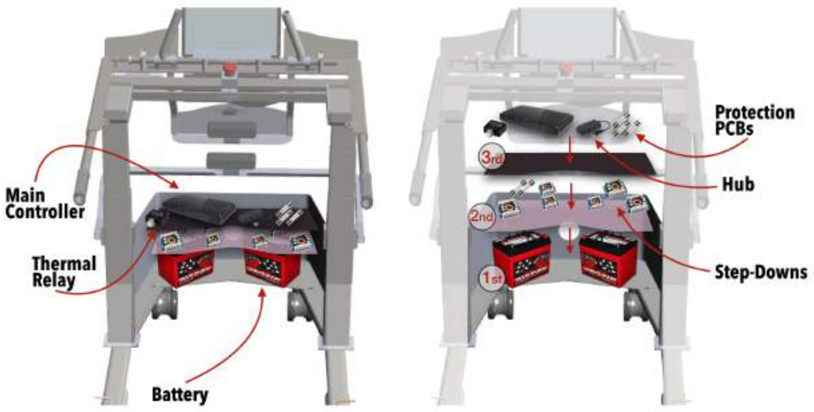
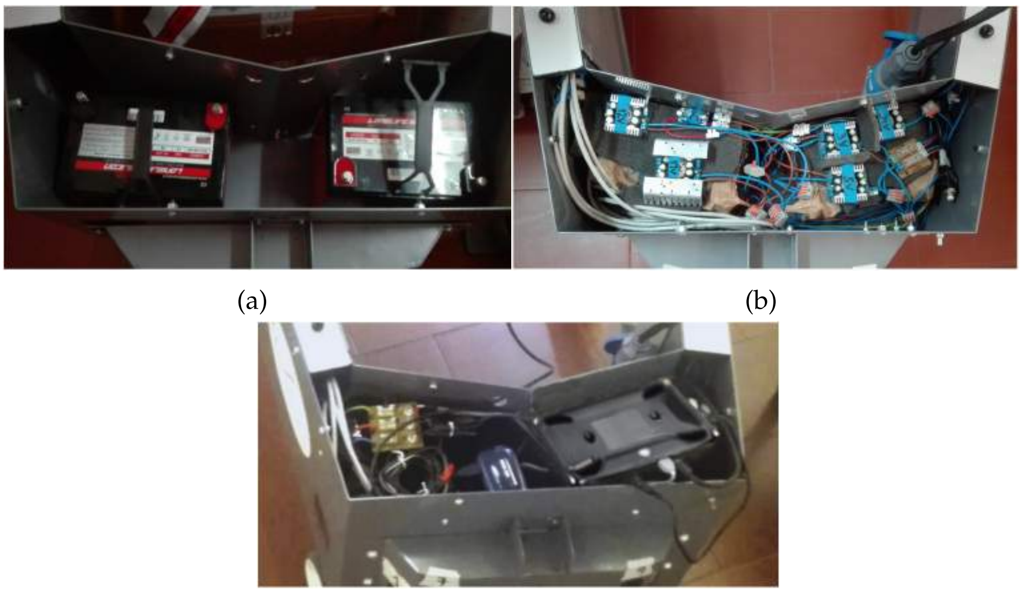


Figure 122: Larger part of the walker's compartment divided in levels.



(c)

Figure 123: The three floors of the larger part of the walker's compartment: (a) First floor; (b) Second floor; (c) Third floor.

As to the sides of the ASBGo* 's compartment (figure 124 and 125): in the left side there is the motors' drivers and the electromagnetic relays that enable/disable the motors in case of emergency and the relay that allows to power all the system ON/OFF; in the right side there are the acquisition PCBs as well as the *Shield* board.

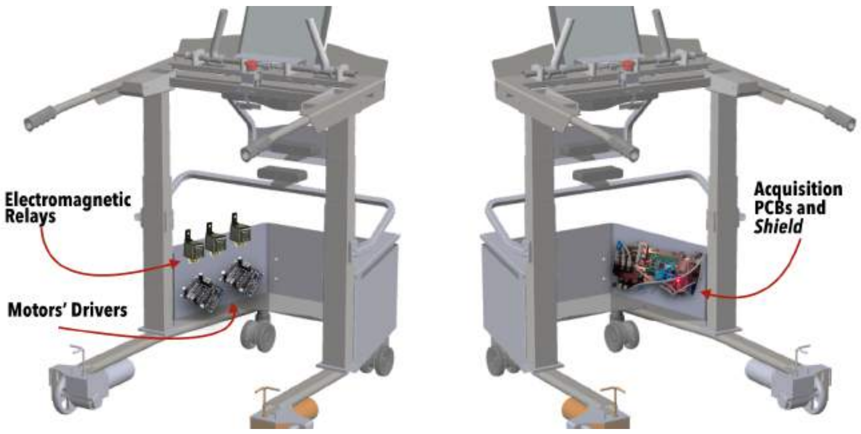
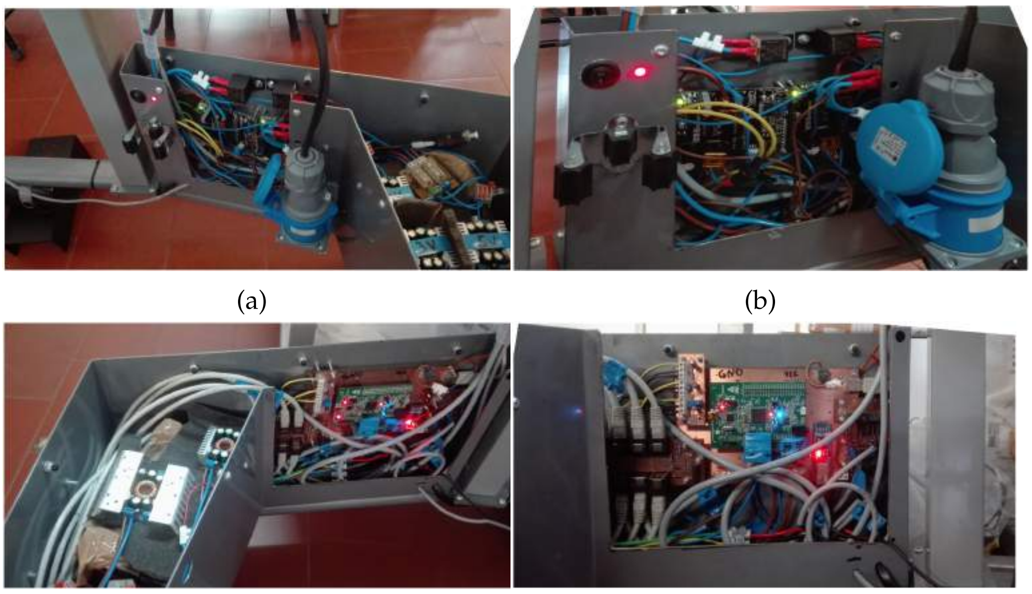


Figure 124: Sides of the walker's compartment.



(c)

(d)

Figure 125: The sides of the walker's compartment: (a)(b) Left side with the motors' drivers and relays; (c)(d) Right side with the acquisition PCBs coupled to the *Shield* board.

5.1.4 *Battery charging strategy*

A new and safe strategy to the charge the batteries of the ASBGo* was outlined and developed.

Figure 126a and 126b shows the ASBGo III prototype. To charge the batteries in this version it would be necessary to remove the batteries from the walker, connect the positive and negative pole to the positive and negative pole of the battery charger, respectively, and respect some other protocols. It consisted in a danger approach in which in case of non-compliance with the protocol, serious consequences could result.

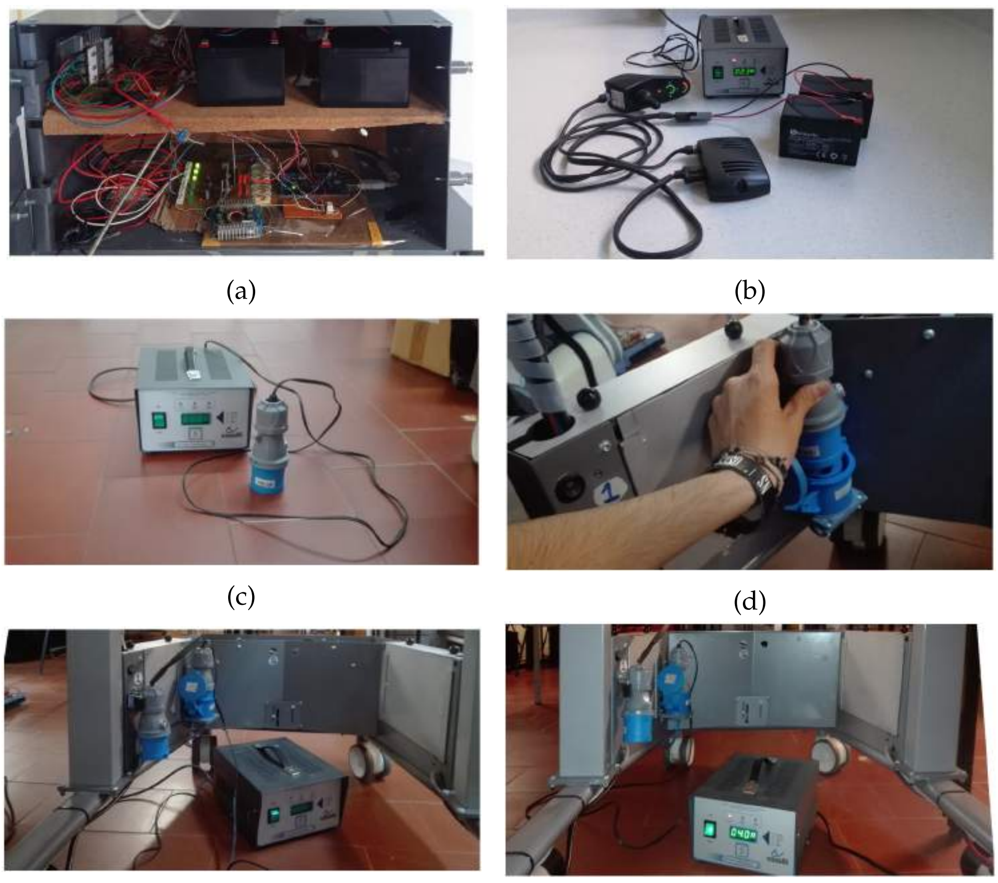
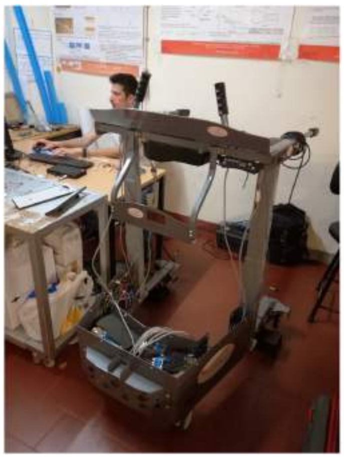

(a) (b) (c) (d) (e) (f)

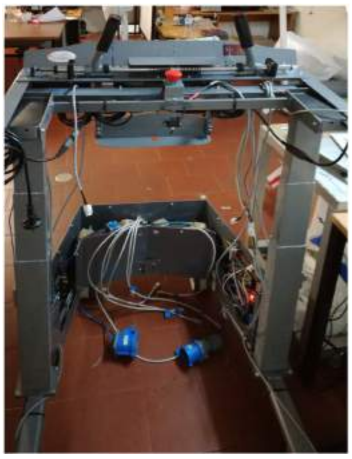
Figure 126: Battery charging strategy: (a) ASBGo III 's batteries; (b) ASBGo III 's batteries charging strategy; (c) ASBGo* 's battery charger; (d) Disconnecting the male power plug connector of the system from the female plug connector of the system (battery); (e) Male plug connector of the battery charger connected to the female plug connector of the system; (e) ASBGo* 's battery charging.

For the ASBGo* , a battery charger was adapted with a compatible male plug connector (figure 126c). This way, to charge the batteries it is necessary to power OFF the system and remove the male power plug connector of the system from the female plug connector (figure 126d). From here, it is only necessary to connect the male plug connector of the battery charger to the female plug connector (figure 126e), which is connected to the battery of the system, and turn ON the charger (figure 126f).

5.1.5 *Others*

The separation between the first and the second floor is accomplished with the use of foam (figure 127a). It is also used foam to accommodate and protect the PCBs .


(a)


(b)

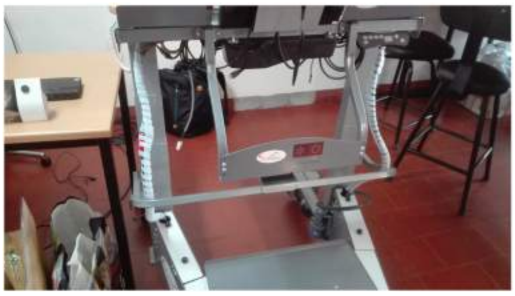
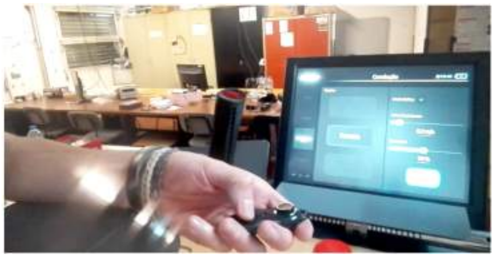

(c)

Figure 127: (a) Foam laid on the batteries consisting in the division between the first and the second floor; (b) Messy aspect; (c) Wires gathered and involved in a plastic coiled cable clamp.

As it can be seen in figure 127b, there is a great number of wires in the new device that transmit a visually messy aspect. To avoid that, the cables were involved in a plastic coiled cable clamp that firmly fix the cables, easing the maneuver of the these cables when e.g. the height of the walker is adjusted (figure 127c).

5.1.6 *HMI devices*

Figure 128a reveals the remote controller, which allows to drive the walker remotely, and the touchscreen monitor that allows the local interaction with the ASBGo* .


(a)

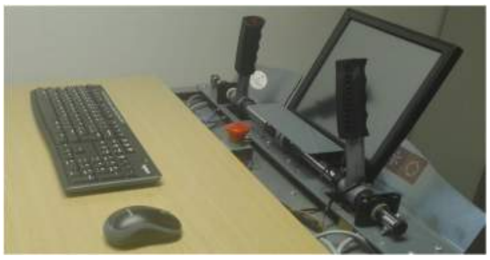

(b)

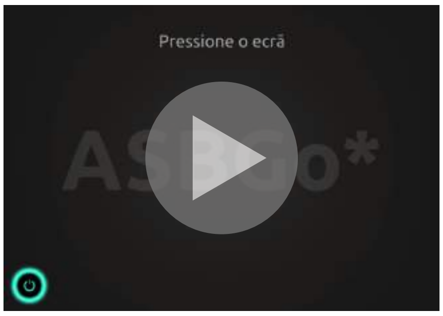
Figure 128: HMI devices: (a) Remote controller and touchscreen monitor; (b) Wireless mouse and keyboard for development.

Figure 128b shows another way of interaction with the system which is through the wireless mouse and keyboard of the walker. This way of interaction, which allows online development, simulation and test, is only used by ASBGo 's developers.

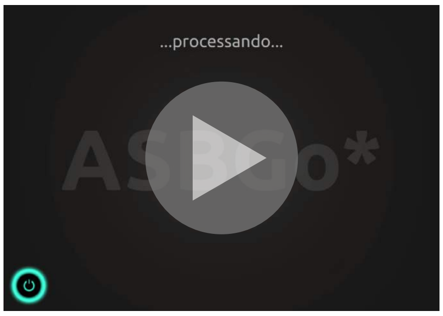
5.1.7 *Human-Machine interface: how to use*

It will be now demonstrated and explained the local GUI of the ASBGo* in order to highlight the user's perspective. As already referred, when the *Main Controller* is turned ON, all the nodes of the system, including the *LocalGUI_node*, are launched. It is in this specific node that runs the local GUI of the system. Since the new device will help patients in the Hospital de Braga in Portugal, the language used in the GUI is Portuguese. The ASBGo* 's GUI is exclusive and completely developed from root.

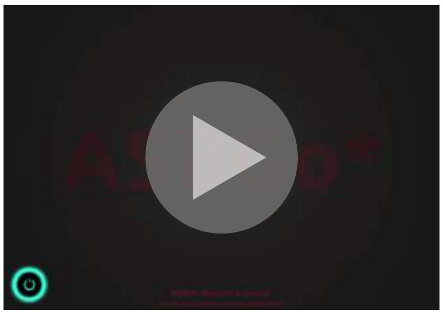
5.1.7.1 *Initial page*

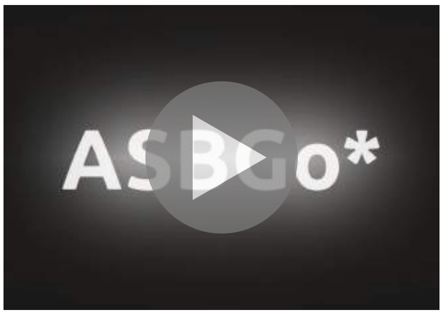


(a)



(b)


(c)



(d)

Figure 129: Local GUI initial page: (a) Waiting the indication of the user to start the system (*0:04*); (b) Connecting to the *Low-Level Controller* (*0:06*); (c) Connection failed (*0:07*); (d) Connection established successfully (*0:14*).

When the local GUI starts, an initial page appears (see video 129a) showing an animation with the name of the device and waiting infinitely for a user's touch in any part of the screen. Also, there is a shutdown button that turns the *Main Controller* OFF. Being the screen pressed, the *Main Controller* tries to connect with the *Low-Level Controller*, as it can be seen in video 129b, and two things can happen: if the connection fails for any reason, a warning appears as shown in video 129c and the system suggests to be turned OFF; if the connection was successfully established, the animation in video 129d is triggered and the main page is revealed.

5.1.7.2 *Main page*

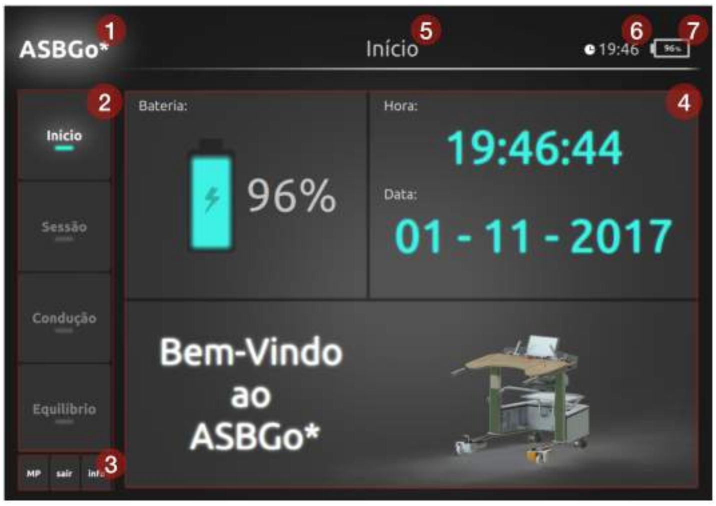


Figure 130: Local GUI main page.

As to the main page (figure 130):

(1) The main title of the application consisting in the name of the device;

(2) The main menu bar consisting in a scrollable menu that allows to select the different pages of the interface. Only one of the items can be selected and when it is, its name "gets ON" (white). By default (and at the beginning), the first item of the menu is selected;

(3) The little menu contains only three items that allows to go to the programmer mode, stop the system and shutdown the *Main Controller* and check some insight about the ASBGo* ;

(4) Represents the layout where the pages corresponding to the items of the main menu bar are loaded;

(5) The title of the current page of the main menu displayed;

(6) A simple digital clock displaying the hours and minutes;

(7) The percentage of battery. When the battery percentage gets to 0%, a red 'X' is displayed inside the little battery image (right superior corner).

5.1.7.3 *Little menu*

As referred, the little menu consists in a small menu with three buttons (three options) as it can be seen in figure 131.

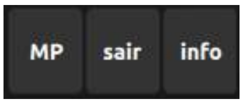
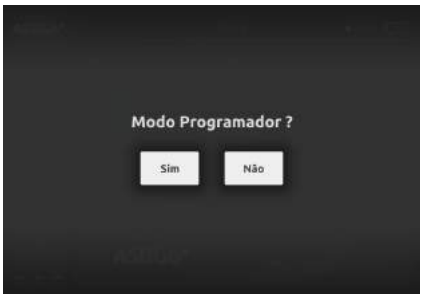


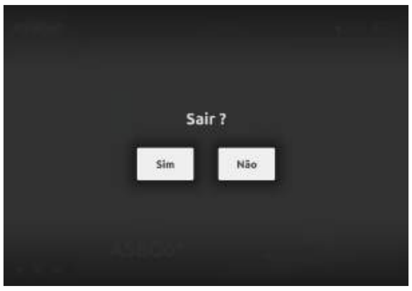
Figure 131: Local GUI little menu.

The first option, the *'MP'*, allows to go to the programmer mode. This means that when selected, the system stops running all the applications, low-level and high-level applications, and the *Main Controller* goes to the *Ubuntu 14.04* OS . From here, the on-board computer can be used as a general-purpose computer. The second option, the *'sair'*, turns OFF the *Main Controller*. Like in the previous option, all the applications of the system stop and the main processing unit shutdowns itself. The third option, the *'info'*, reveals some information and description of the new device, the ASBGo* .

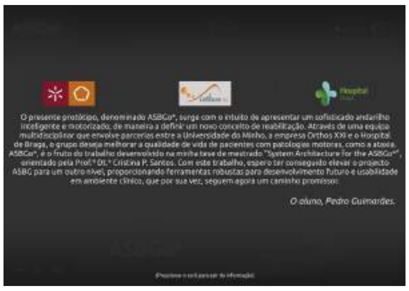
Figure 132 reveals what happens when each of the options are selected. For the *'MP'* (figure 132a) and *'sair'* (figure 132b) options, a confirmation is made. If the answer *'Não'* is selected, the little menu page vanishes and the previous state of the local GUI is re-established. As to the third option, the *'info'* (figure 132c), to close its little menu page it is necessary to touch any part of the screen.



(a)



(b)



O presente protótipo, denominado ASBGo*, surge com o intuito de apresentar um sofisticado andarilho inteligente e motorizado, de maneira a definir um novo conceito de reabilitação. Através de uma equipa multidisciplinar que envolve parcerias entre a Universidade do Minho, a empresa Orthos XXI e o Hospital de Braga, o grupo deseja melhorar a qualidade de vida de pacientes com patologias motoras, como a ataxia. ASBGo*, é o fruto do trabalho desenvolvido na minha tese de mestrado "System Architecture for the ASBGo*", orientado pela Prof.ª Dt.ª Cristina P. Santos. Com este trabalho, espero ter conseguido elevar o projecto ASBG para um outro nível, proporcionando ferramentas robustas para desenvolvimento futuro e usabilidade em ambiente clínico, que por sua vez, seguem agora um caminho promissor.

O aluno, Pedro Guimarães.

(Pressione o ecrã para sair da informação)

(c)

Figure 132: Local GUI little menu pages: (a) *'MP'*; (b) *'sair'*; (c) *'info'*.

5.1.7.4 *Main menu*

Based on the requirements of Chapter 4, section 4.1, a set of options was defined for the main menu. It is relevant to say that when one of the items is select, a respective page is loaded into the main menu pages layout (see *(4)* in figure 130). This way, in the future, if new requirements are outlined and if they have some influence in the local GUI , new options can be easily added to the main menu bar and also a respective page must be developed.

Although being established seven options, only four present a page. The others are currently unavailable and are for future development. The disabled options can be pressed but, since that they have no page, nothing happens. On the other side, if the available options are pressed, the respective page is loaded. Everytime that a new option is selected, a clean command is sent to the *LowLevelController_node* in order to the current configurations in it be reset. Then, depending on its purpose, the loaded page takes care of send configuration commands to the *LowLevelController_node*.

Figure 133 illustrates all the main menu options (unavailable options are in gray).

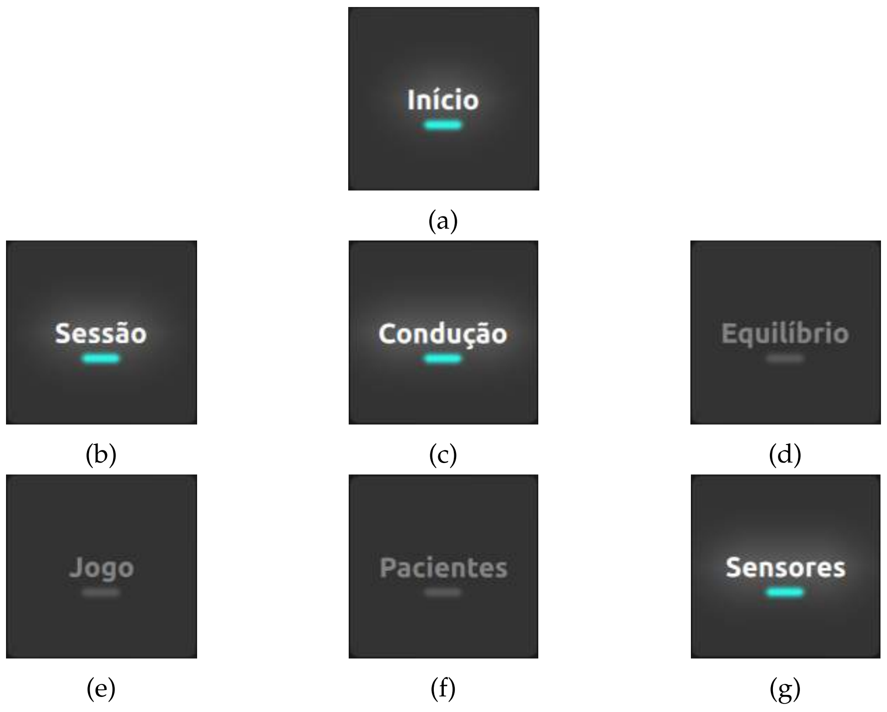


Figure 133: Local GUI main menu options: (a) '*Início*'; (b) '*Sessão*'; (c) '*Condução*'; (d) '*Equilíbrio*' (unavailable); (e) '*Jogo*' (unavailable); (f) '*Pacientes*' (unavailable); (g) '*Sensores*'.

It is important to refer that the four options developed are due to the nine requirements that were defined to be covered in the dissertation.

PAGES

***'Início'* page**

The *'Início'* page is considered the home page i.e. the first that appears (from the main menu options). Figure 134 reveals the *'Início'* page.

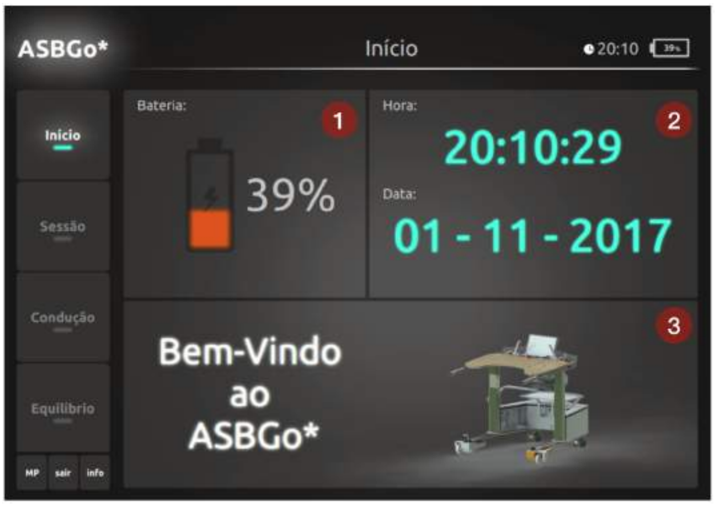


Figure 134: Local GUI *'Início'* page.

As to the *'Início'* page:

(1) A more complex representation of the battery percentage. It has three states (see video 135): (a) when the battery percentage is above 40%, the color inside the battery's image gets blue; (b) when the battery percentage is bellow 40% and above 20%, the color inside the battery's image gets orange. This means that the walker can perform one session, more or none (depends on the time of the session); and (c) if the battery percentage is bellow 20%, the color inside the battery's image gets red (session is not advised and the battery must be charged);

Figure 135: Local GUI battery percentage (*0:19*).

(2) A digital clock showing the hours, minutes and seconds as well as a presentation of the date in *dd - mm - yyyy;*

(3) A greeting message for the users and an image of the ASBGo* .

'Sessão' page

The *'Sessão'* page is where sessions are configured and performed. Figure 136 pictures the *'Sessão'* page.

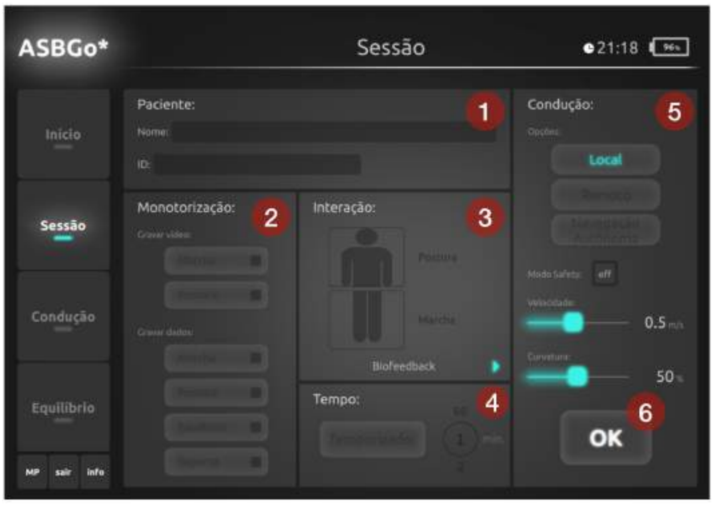


Figure 136: Local GUI *'Sessão'* page configurations.

As to the *'Sessão'* page:

(1) Identification of the patient (name and ID number) that is going to perform a rehabilitation session. This configuration has no purpose yet, but it is going to be important once created an embedded database;

(2) Configure the patient's assessment. In the future, it will allow to record video and store spatio-temporal parameters of the gait and posture of the patient. Also, it will be possible to store the data from the patient's balance, acquired by the IMU , and the data from the patient's support in the wood platform of the walker acquired by the load cells. Currently, when the *'Equilíbrio'* and *'Suporte'* options are selected, a configuration command is sent to the *LowLevelController_node* in order to be enabled the IMU and the load cells, respectively.

(3) The interaction configuration. This configuration has no purpose yet but will allow to choose one of two future functionalities. One is to select the gait and/or posture biofeedback and the other, the visual and/or auditive multitasking game.

(4) Choose a time mode for the session. If nothing is selected, the session will occur in 'chronometer' mode. In the other side, if the *'Temporizador'* button is selected, it can be chosen a number of minutes (maximum 60 minutes) and the session will occur in 'timer' mode;

(5) In this layout it is possible to configure the drive mode, local or remote mode (the autonomous navigation mode is for future development) and choose the desired velocity and curvature percentage for the walker's locomotion;

(6) When the user finishes the configuration of the system, it presses the 'OK' button to go to the session page.

Figure 137 shows an example of a session's configuration.

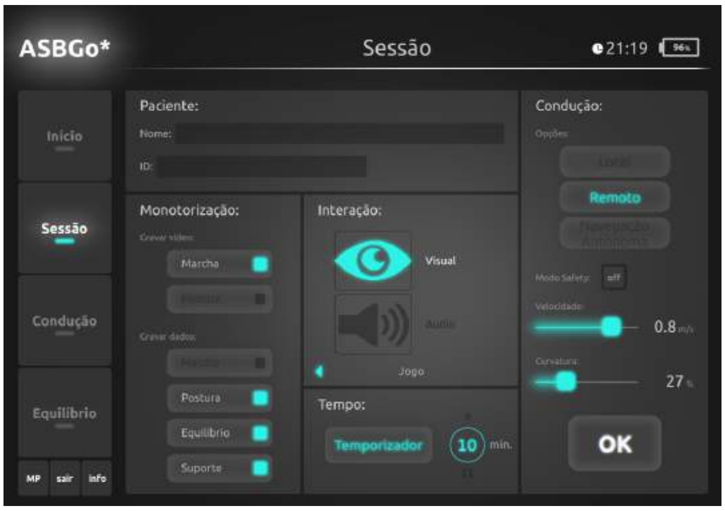
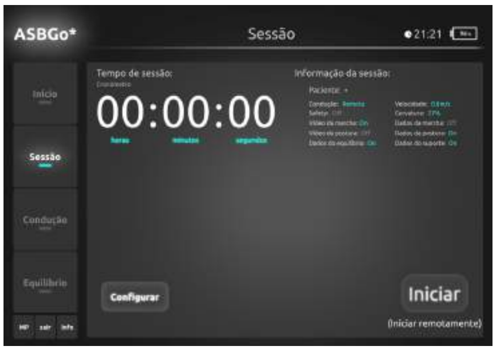
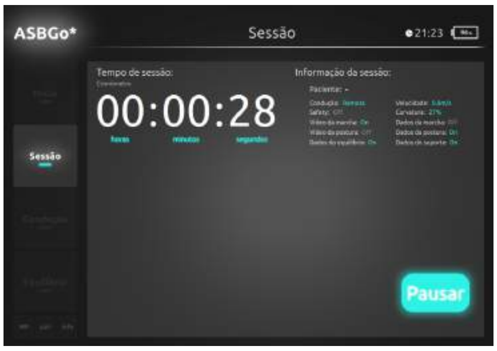


Figure 137: Local GUI *'Sessão'* page configurations example.

Figure 138 and 139 illustrate session pages in different states. The session page presents a clock that can be in 'chronometer' or 'timer' mode, the information of the session, the *'Configurar'* button that allows to return to the configuration page and a start button called *'Iniciar'*. Below the clock, there is an empty space reserved for the interaction functionalities that are currently not implemented in the new prototype.



(a)

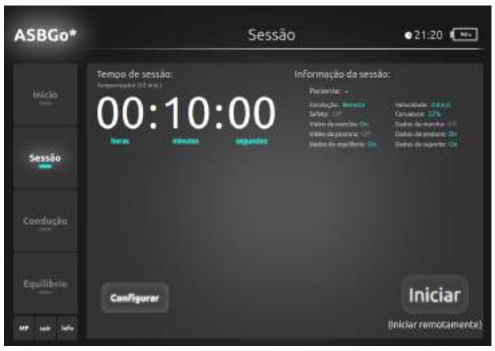


(b)

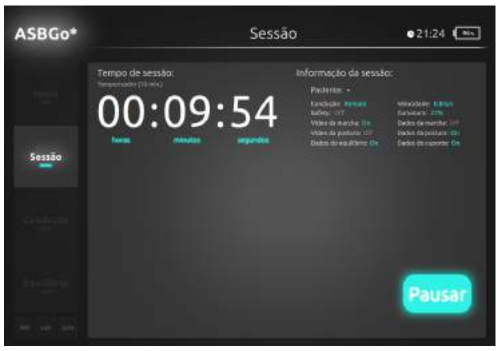
Figure 138: Local GUI session page with remote drive mode and 'chronometer' mode: (a) Idle state; (b) Running state.

Figure 138 presents a session whose time is in 'chronometer' mode. In this example it is chosen the remote drive mode and so, the session can only be started using the remote controller. Contrarily, when it is used the local drive mode, the session must be started from the local GUI . Once the session is initiated (in this case, using the remote controller) no more configurations can be done and so, the '*Configurar*' button disappears (see figure 138b). Also, being the session in running mode, all the menus get in an inactive state and the chronometer starts counting.

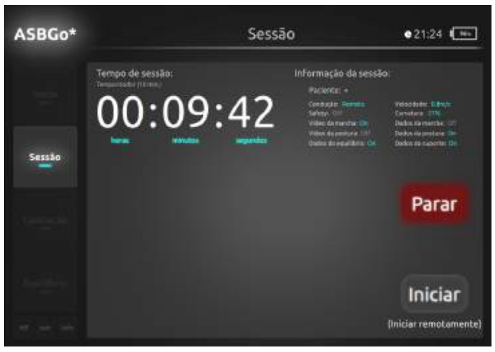
Figure 139 presents a session whose time is in 'timer' mode. When the session is running, it can be paused (figure 139b and 139c). Being in a paused state, the session can continue by being pressed the '*Iniciar*' button (if in local drive mode) or it can be stopped by being pressed the '*Parar*' button. When a session is stopped the session configuration page re-appears. On the other side, when the timer expires, the session finishes, a warning message appears (see figure 139d) and it is returned to the session configuration page.



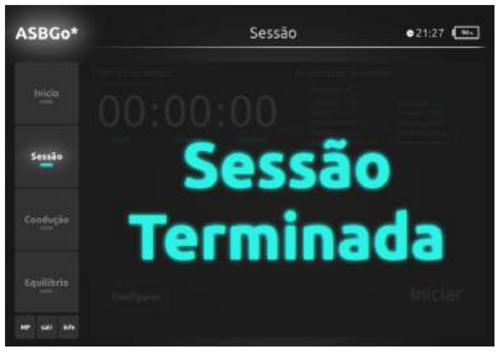
(a)



(b)



(c)



(d)

Figure 139: Local GUI session page with remote drive mode and 'timer' mode: (a) Idle state; (b) Running state; (c) Paused state; (d) Finished.

'Condução' page

The *'Condução'* page allows the user to only drive the walker. It can be relevant e.g. to test and choose the most comfortable velocity for the patient. Figure 140 shows the *'Condução'* page.

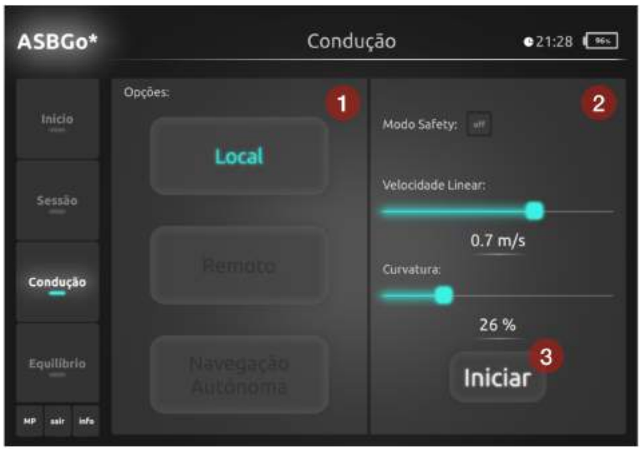


Figure 140: Local GUI *'Condução'* page.

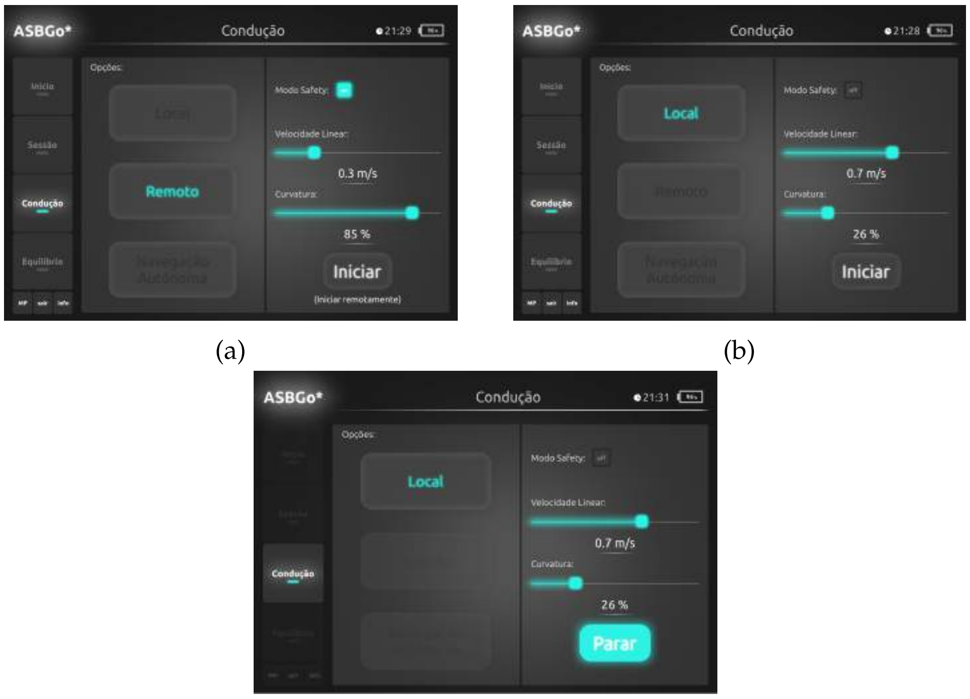


Figure 141: Local GUI *'Condução'* page: (a) Remote drive mode; (b) Local drive mode; (c) Running in local drive mode.

As to the *'Condução'* page:

(1) It can be chosen a drive mode, local or remote mode (the autonomous navigation is currently not developed);

(2) It can be defined the desired velocity and curvature percentage for the walker's locomotion. Also there are the possibility to choose a safety locomotion mode, represented as *'Modo Safety'* in figure 140, that is currently not available remaining for future development;

(3) The locomotion can be initiated by pressing the *'Iniciar'* button.

As it can be seen in figure 141a, when it is selected the remote drive mode, the locomotion must be initiated using the remote controller. Also, being the *'Condução'* page in running mode, all the menus get disabled as well as the selection of the drive modes. In the other hand, the desired velocity and curvature percentage can be changed (see figure 141c).

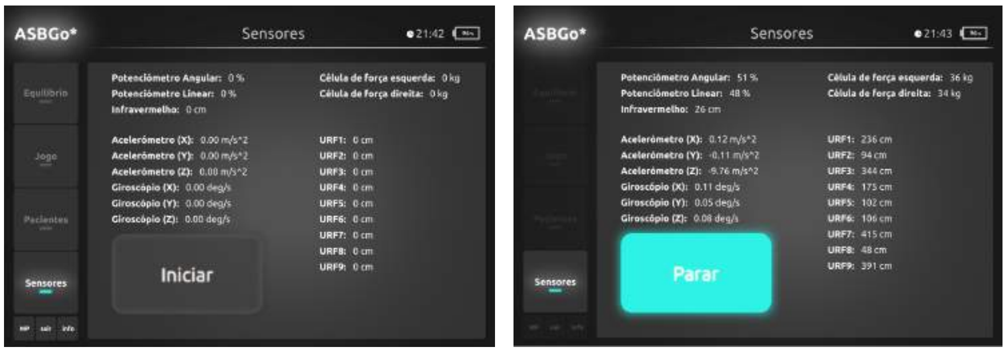
'Equilíbrio', *'Jogo'*, *'Pacientes'* pages

The three present pages are currently not implemented. As to the *'Equilíbrio'* page, it will be possible to verify, in real-time, the balance of the patient by the acquisition of the IMU 's data and by using an algorithm that outputs the behaviour of the patient's center of mass over time. As to the *'Jogo'* page, it will present the visual/auditive multitasking game that, as referred before, is also used in rehabilitation sessions. Finally, the *'Pacientes'* page will wrap the embedded database that will allow to add or delete a patient from the database and it will also allow to edit and visualize all the patients' information.

'Sensores' page

With the *'Sensores'* page, it is possible to visualize the embedded sensors' data in real-time. It has no purpose for the rehabilitation sessions, but serves to prove and check the acquisitions of the sensors.

As it can be seen in figure 142, the page presents a start button, *'Iniciar'* button, that, when pressed, enables a local locomotion with a static velocity and curvature percentage enabling also all the embedded sensors. When the page is in a running state, all the menus get disabled.



(a)

```
Potenciómetro Angular: 0 %          Célula de força esquerda: 0 kg
Potenciómetro Linear: 0 %           Célula de força direita: 0 kg
Infravermelho: 0 cm

Acelerómetro (X): 0.00 m/s^2        URF1: 0 cm
Acelerómetro (Y): 0.00 m/s^2        URF2: 0 cm
Acelerómetro (Z): 0.00 m/s^2        URF3: 0 cm
Giroscópio (X): 0.00 deg/s          URF4: 0 cm
Giroscópio (Y): 0.00 deg/s          URF5: 0 cm
Giroscópio (Z): 0.00 deg/s          URF6: 0 cm
                                    URF7: 0 cm
                                    URF8: 0 cm
                                    URF9: 0 cm
```

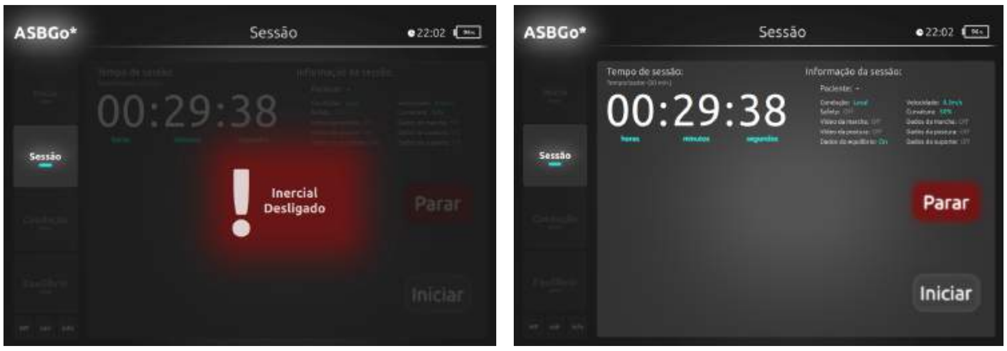
(b)

```
Potenciómetro Angular: 51 %         Célula de força esquerda: 36 kg
Potenciómetro Linear: 48 %          Célula de força direita: 34 kg
Infravermelho: 26 cm

Acelerómetro (X): 0.12 m/s^2        URF1: 236 cm
Acelerómetro (Y): -0.11 m/s^2       URF2: 94 cm
Acelerómetro (Z): -9.76 m/s^2       URF3: 344 cm
Giroscópio (X): 0.11 deg/s          URF4: 175 cm
Giroscópio (Y): 0.05 deg/s          URF5: 102 cm
Giroscópio (Z): 0.08 deg/s          URF6: 106 cm
                                    URF7: 415 cm
                                    URF8: 48 cm
                                    URF9: 391 cm
```

Figure 142: Local GUI *'Sensores'* page: (a) Idle state; (b) Running state.

5.1.7.5 *Warnings*

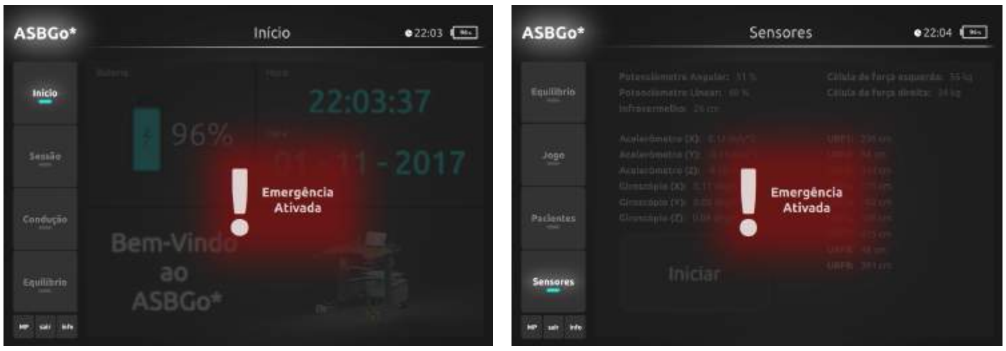
Figure 143 shows a possible warning related to the IMU . Since the acquisition board for this sensor is removable, in pages where the IMU is enabled, it is taken into account if the board is connected to the system or if it was disconnected (if the system is in running mode). When the warning is displayed, it disappears after a few seconds.



(a) (b)

Figure 143: Local GUI IMU 's warning: (a) Warning triggered pausing the session; (b) Warning disappears and session remains paused.

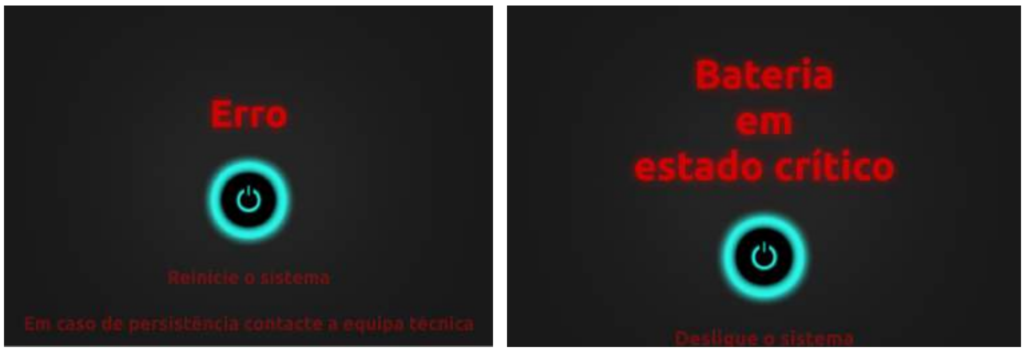
Another warning that appears in the local GUI is the emergency warning. When the emergency button is pressed, the GUI is notified and the warning is displayed until the emergency button is released. From here, it is possible to navigate through the menu's options, being the pages loaded, but being in a disabled state (see figure 144).



(a) (b)

Figure 144: Local GUI emergency's warning. It is possible to navigate through the menu's options but the pages are disabled until the emergency button is released.

As it can be seen in figure 145, when the communication with the *Low-Level Controller* gets lost or when the battery's voltage gets into a critical state, the *Main Controller* stops, the local GUI is notified and a warning is displayed for the user, requesting to shutdown the system.



Erro

Reinicie o sistema

Em caso de persistência contacte a equipa técnica

Bateria em estado crítico

Desligue o sistema

(a) (b)

Figure 145: Local GUI shutdown warnings: (a) Communication with the *Low-Level Controller* lost; (b) Battery's voltage in critical state.

5.1.7.6 *Demonstration*

Figure 146: Local GUI demonstration (3:29).

5.1.8 *Final Product*

Starting from the mechanical frame presented in figure 107 at the beginning of this chapter (5) and passing through all the design and development process, it resulted the ASBGo* smart walker, the more reliable, robust and extensible version of the ASBGo (see figure 147).

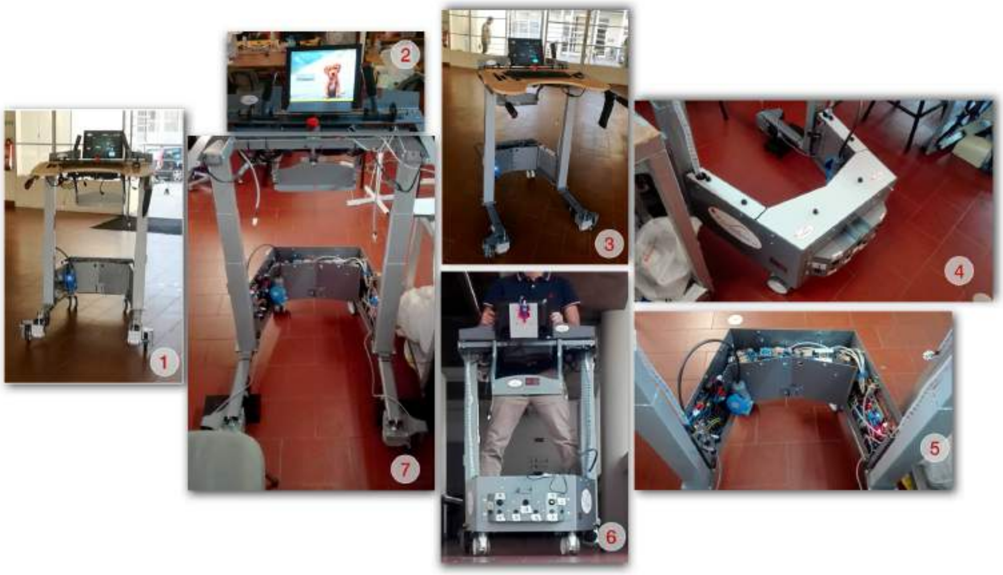
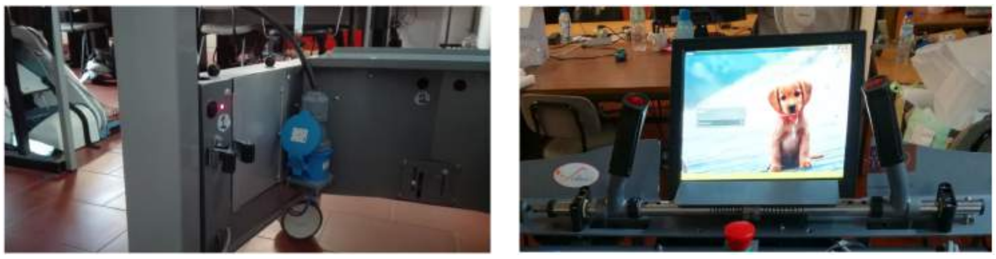


Figure 147: The ASBGo* in different perspectives.

Figure 148 illustrates how simple is to start the ASBGo* . First, as it can be seen in figure 148a, the male power plug connector of the system must be connected to the battery i.e., the female plug connector of the system, and the system must be powered ON using the switch labeled with a ′1′ (note that without being the male power plug connector connected to the battery, the switch has no effect). Then, it is necessary to turn ON the *Main Controller* by pressing its power button located where there is a ′2′. Being the *Ubuntu 14.04* OS booted, a login is required and after that, the local GUI is immediately initiated.



(a) (b)

Figure 148: Starting the ASBGo* : (a) Connect the male power plug connector of the system to the battery, turn the system ON using the switch (represented as ′1′) and turn the on-board computer on using its power button (represented as ′2′); (b) Login.

5.2 system validation

It will be now exhibited several videos that show some features of the new ASBGo* . In order to be fully understood, relevant information is displayed during the videos. (*Again, it is advised to open this document with the Adobe Acrobat Reader to allow the visualization of videos.*)

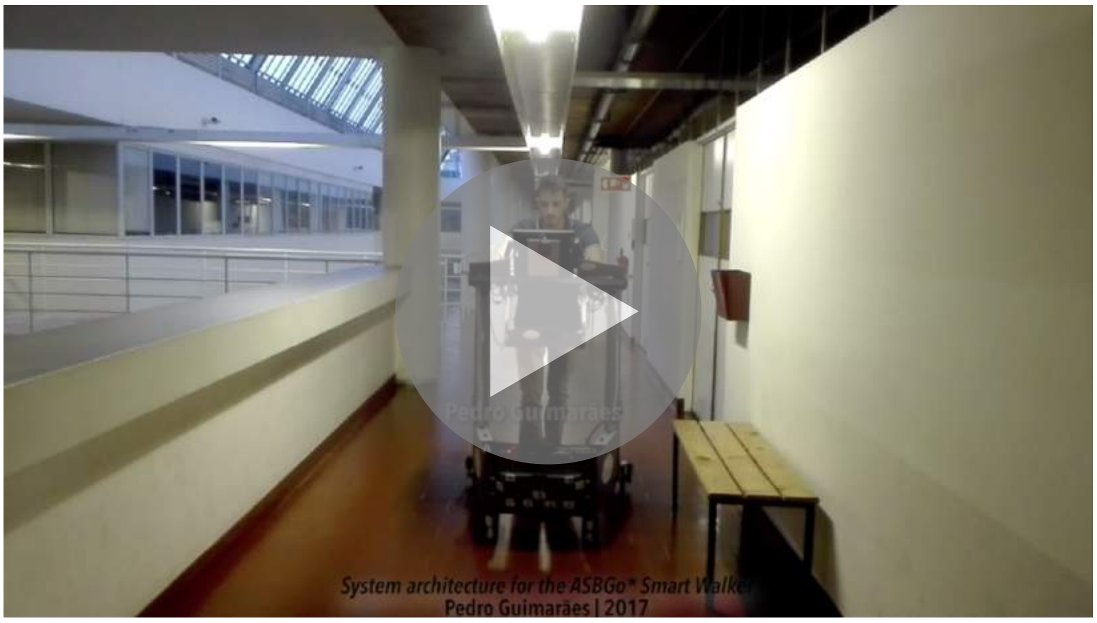


Figure 149: (1) Validation of the walker's locomotion in local drive mode (using the handlebar); (2) System in a preliminary stage running with ROS (using the command line). (*0:37*)

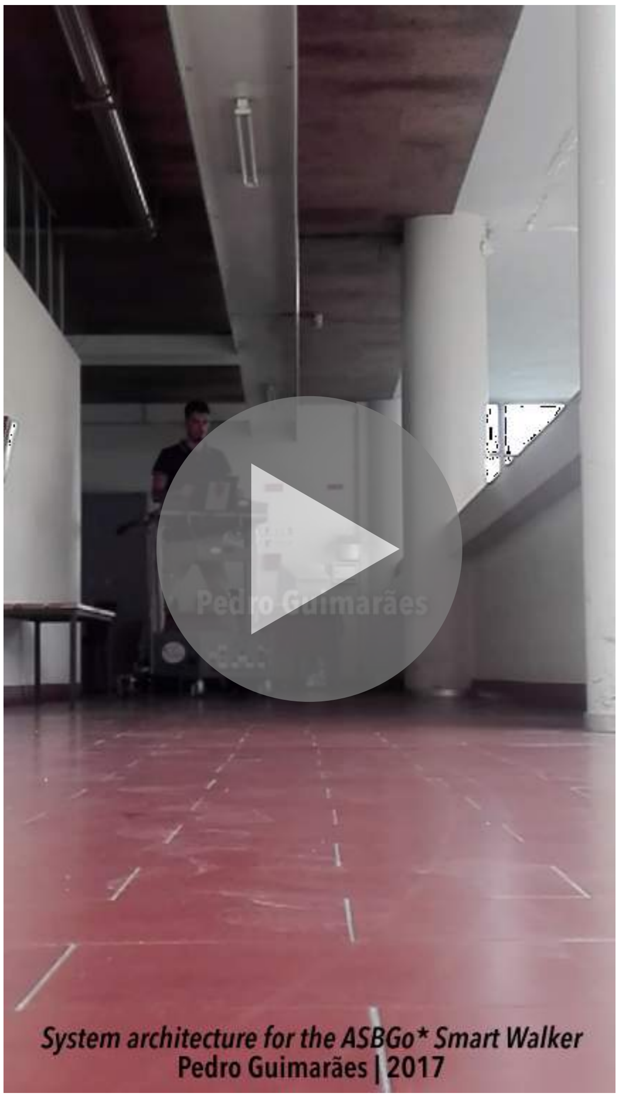


Figure 150: (1) Linear velocity achieved (PID) regardless of user's weight; (2) Two turning modes: defined curvature percentage and complete turns. (*0:34*)

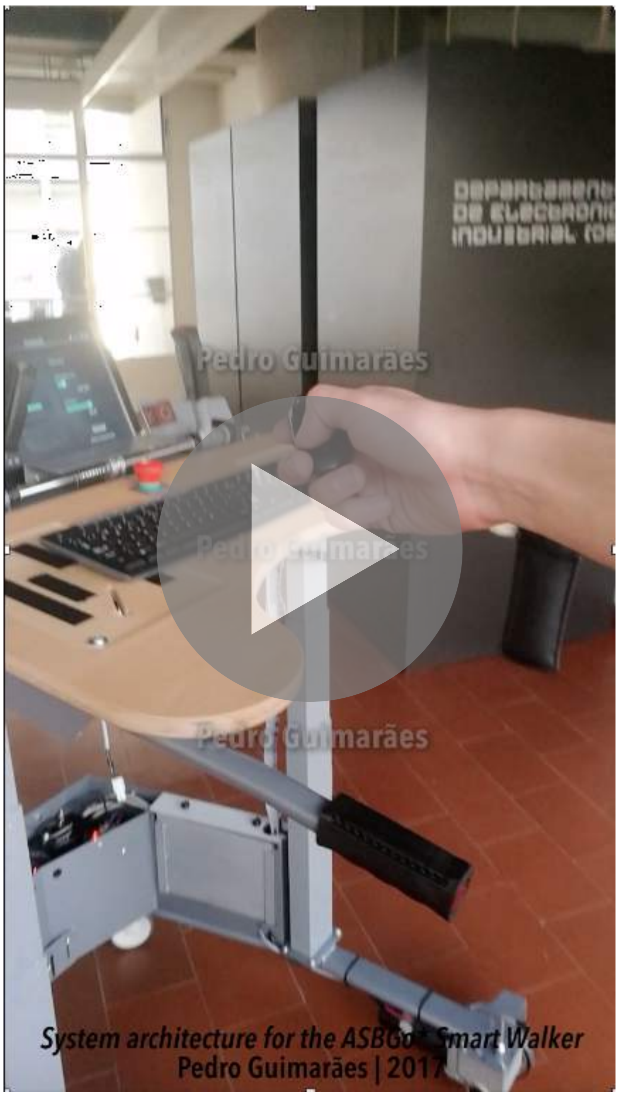


Figure 151: First remote control validation. (*0:43*)

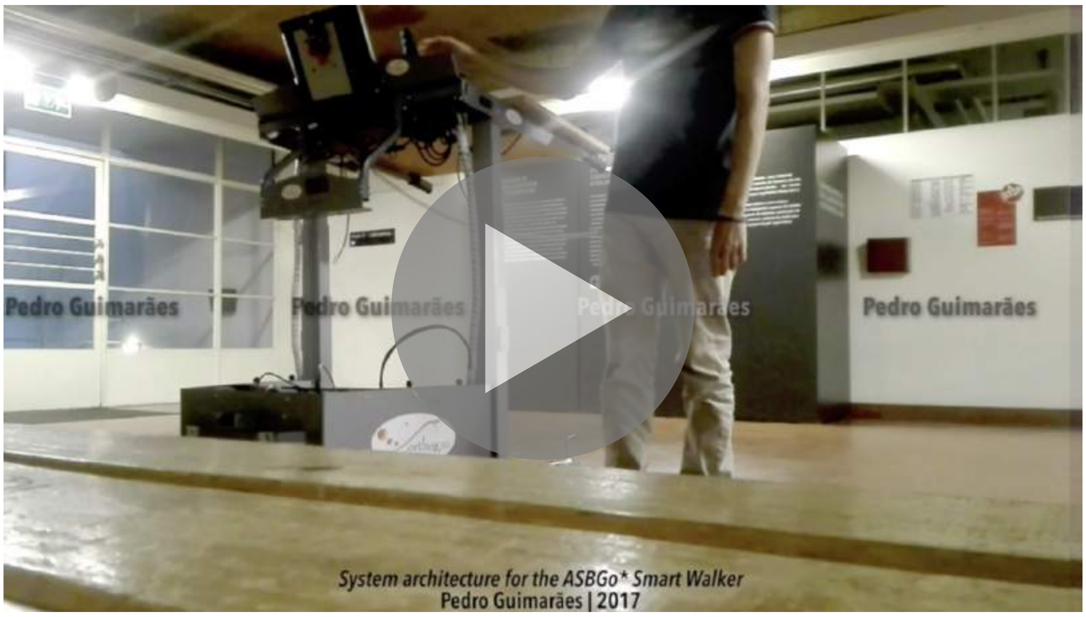


Figure 152: (1) Easy driving with smooth start and stop; (2) Deactivation of the motors' electromechanical break system allowing to manually push the walker; (3) Linear and angular velocities configurable ensuring suitability for different patients. (*1:41*)

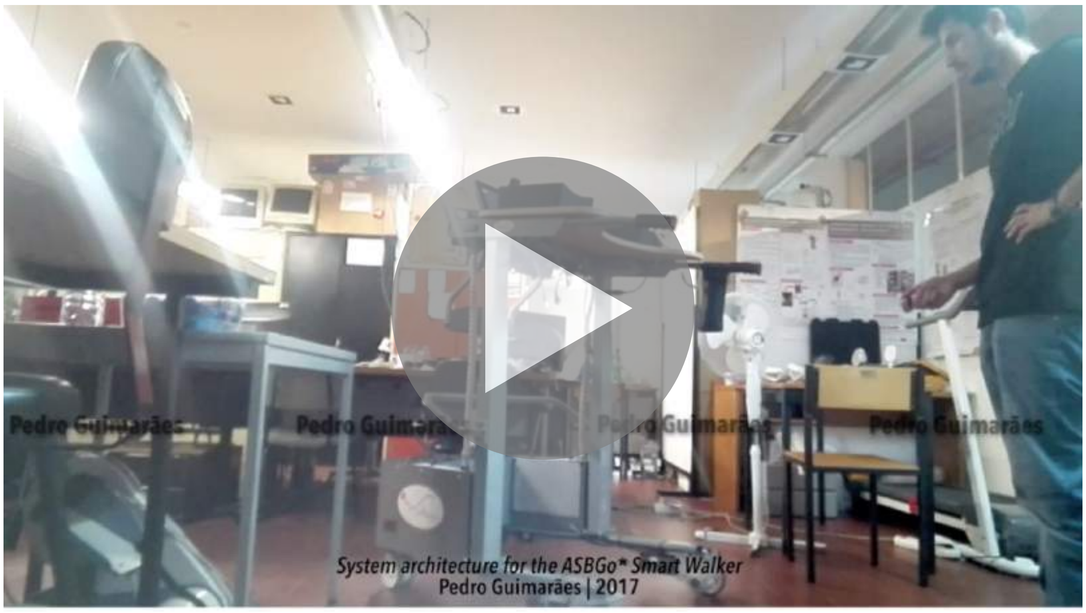


Figure 153: (1) Remote drive mode using the remote controller; (2) Instantaneous and precise response to commands; (3) Easy to maneuver in tight spaces even with higher velocities; (4) User-friendly interface to configure and use the ASBGo* smart walker; (5) The programmer mode stops all the applications of the system and gives access to the *Ubuntu 14.04*. (*2:49*)

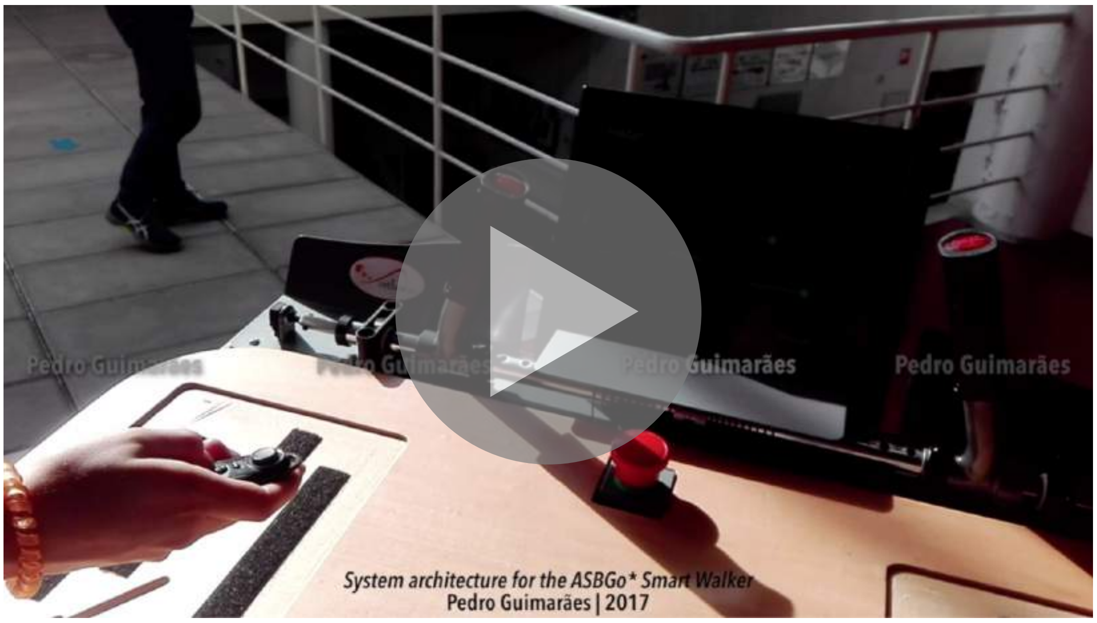


Figure 154: (1) A member of the ASBGo team, which will be responsible for the insertion of the ASBGo* in clinical environment, in its first interaction with the new device; (2) Emergency triggered and emergency button released; (3) New linear velocity defined in real time; (4) Member of the ASBGo team using the device with body supported. (*1:26*)

6

CONCLUSION

The work developed allows to answer the research questions (RQ) outlined in the sub-chapter 1.2.2 - *Goals and research questions*.

RQ1: "How many ASBGo prototypes exist, what functionalities they have and how their hardware and software solutions are implemented ?".

As presented in Chapter 3, there are four versions of the ASBGo that precede the new ASBGo* . For all the versions, both the hardware and the software allowed to prove the concept however, the lack of reliability, extendability, usability and the existence of poor and academic electronic solutions, never facilitated the continuous use and development. Also, although the contributions for the old versions were valuable, it is difficult to gather every functionality when there are *ad hoc* implementations and so, it gets difficult to create a global unified system in order to be expanded. The *Prototype I* presented a rudimentary structure and a certain delay and hysteresis in its locomotion. Its main functionalities were: (1) measure the user's proximity using an IR sensor; (2) detect the user's drive intentions using a potentiometer; and (3) detect the user's support using load cells. The *Prototype II* was easier to transport and store and it was tested with real patients. However, although better than *Prototype I*, it turned out to be a little uncomfortable and revealed some mechanical limitations in terms of adjustment. Its main functionalities were: (1) a first proof-of-concept obstacle avoidance feature; (2) user's balance assessment by the use of accelerometers; and (3) a walker-assisted gait assessment method arisen from the use of IMU s in the patient. The ASBGo III was the first version used in continuous and regular rehabilitation and it was more stable and ergonomic. Its new features were: (1) feet pose estimation by the use of an active depth camera; and (2) a legs tracking feature using a laser range finder. In the ASBGo++ version, an improved mechanical structure was outlined and created, new high-level software features were developed and a deep study of all the versions of the ASBGo was accomplished. However, the ASBGo++ was an incomplete version due to the absence of electronics. As to its new features it had: (1) a body tracking feature using an active depth camera; (2) a biofeedback feature and a multitasking game; and (3) a database that gathers all the information of the patients and of the gait analysis.

RQ2: "What are the main goals and the requirements that the new prototype should fulfill, in terms of behaviour and features ?".

As presented in Chapter 4, section 4.1, the new ASBGo* must be capable to be used in rehabilitation sessions and must present such an architecture that allows and eases future development. This way, electric and electronic circuits were designed and implemented and the following behaviours were introduced: (1) acquire data from a set of sensors in real-time. Updated information of the sensors must be disposed, in specific units, in an high-level layer and must be easily accessed in order to be developed/explored algorithms and software features; (2) monitor the battery's voltage; (3) freeze in case of critical battery; (4) Stop in case of emergency; (5) move multi-directionally, more specifically, in eight directions; (6) allow its locomotion through the use of the handlebar or the remote controller; (7) control the velocity of its locomotion using a control algorithm; (8) detect some possible errors (e.g. processing unit failure, motors errors); (9) provide a local graphical user interface allowing the configuration and the use of the device (e.g. perform rehabilitation sessions);

RQ3: "What is the most proper robotic software platform to develop the new system architecture and what strategy should be followed to engineering it ?".

The more high-level software developed was built using ROS (detailed in Chapter 2, section 2.2). This robotic software platform is a precious asset in the new system in a way that future development is now a much more easier task. Its inference on modular code development allows to divide the software in logical components (processes) that communicate with each other. That communication is easily achieved by the use of a message-passing infrastructure of ROS . It is very well documented and presents several tutorials that improve the learning curve of developers. However, if questions emerge, ROS has an active online community that support any possible doubt and also allows to install ready-to-use functionalities, implemented and tested by other developers, that are totally configurable, present clear interfaces and can easily be integrated in a robot software.

To engineer the new system architecture all the previous versions had to be studied (Chapter 4). Gathering the functional requirements for the new device, a conceptual model in blocks was delineated to obtain an overview of the global system. The necessary material was explored, specified and purchased and by following a co-design method, hardware and software for the ASBGo* was developed respecting the requirements defined.

RQ4: "In how many conceptual blocks the system will be organized and how can they be specified ?".

In Chapter 4, section 4.2 it is revealed an overview of the global system and also the two current conceptual blocks of the ASBGo* : *Real-time sensors data acquisition and locomotion* and *Human-Machine interface (HMI)*. The first one consist in the collection and the processing of information from a series of embedded sensors and components that allows to the system to understand the surrounding environment, present an intelligent behavior and act accordingly.

The second block is responsible for the interaction between the users and the device. In order to be easily used in the medical environment by medical specialists, the interaction with the system must exist and must abstract all the technical concepts. So, by the use of user-friendly interfaces it is possible to offer a pleasant, simple and attractive way to use the smart walker.

RQ5: "What material had to be acquired and/or purchased to develop the new system architecture ?".

Detailed in Chapter 4, subsection 4.3.1, a list of material, to develop the new architecture, was gathered. This way, it was collected a series of (1) sensors such as: a linear potentiometer, an angular potentiometer, an infrared sensor, load cells, an inertial measurement unit, encoders, ultrasonic range finders, RGB-D cameras (currently not in use), a laser range finder (currently not in use); (2) powerful processing units: a development board and an on-board computer; (3) USB-to UART serial converter module; (4) actuators and motors drivers; (5) HMI devices: touchscreen monitor, bluetooth remote controller; (6) 12V batteries; (7) step-downs modules; (8) safety components; and (9) cables.

RQ6: "Does the mechanical structure requires physical modifications ?".

At the beginning of this work, there was already a physical structure of the walker (developed in the ASBGo++ version) in where all the development of this thesis evolved to in the end originate the ASBGo* . However, some mechanical details were not predicted and they were only detected and performed when a proper system architecture was developed. This way, (1) many holes had to be made: to couple the power switch and the indicator LED, to allow passing the wires, to couple the PCBs in order to get stable and to couple some components to the walker's frame ; (2) a mechanical strategy to easily couple and stabilize the URFs was achieved; (3) two windows, at each side of the walker's compartment, were cut in order to improve accessibility to the components inside the walker; (4) holes and little paths were made in the wood lids of the ASBGo* 's compartment in order to the wires pass to the upper part of the walker and be easily handled; (5) new shafts for the motors were designed to ensure compatibility with the encoders; (6) a stopper was applied in the handlebar to avoid the destruction of the angular potentiometer; (7) a mechanical piece was coupled in front of the walker (above the URFs) and it will be where the laser range finder will be placed; (8) a big cut was performed in the upper part of the ASBGo* and a mechanical piece was developed in order to couple the touchscreen monitor; (9) mechanical pieces, for the new battery charging system, were created, acquired and applied in the new device.

RQ7: "The implemented solution was successfully validated and responds to all the demands that were initially proposed ?".

The new ASBGo* present the requirements proposed in the Chapter 4, section 4.1. It does not gather all the desired functionalities that were planned by the ASBGo team (during the last years) by the fact that it is impossible to implement all of them in the scope of a thesis.

However, it already wraps many features that confers it the ability to be used in rehabilitation sessions and so, help patients to get confident in their gait and pleasantly assist the medical technicians. Also, although not completely enriched with functionalities, the current state of the new device and the way that it was developed, allows to future developers to not concern about certain and enormous details that, in the past, the lack of them led to the collapse of older versions. Since a proper system architecture was developed, the future contributions will not branch and diverge from the development line created in this thesis but rather, will go along with it.

RQ8: "What are the next steps that should be taken and what future work possibilities are available in order to improve the developed system ?".

Besides the validation of the new prototype in clinical environment, the next steps for the new smart walker are just the addition of new algorithms. Since the more low-level software is concluded and abstracted and since it is used ROS to build high-level software, it is only necessary to use the resources of the new system to design, implement, test, simulate and add new software components to the existing software system. More specifically, the next steps for the ASBGo* is the development of the rest of the requirements listed in Chapter 4, section 4.1 and validate them in a real medical context.

6.1 conclusions

The work carried out during this thesis allowed to create a new sophisticated, more professional, reliable and expandable version of the ASBGo , the ASBGo* . Starting only with a new physical frame, outlined in the ASBGo++ version, all the development process involved in this thesis addressed many layers, starting from the hardware, passing through low-level software and finishing in high-level features.

To raise this new device, it took place a phase of integration in the context of the project. By the study of all the previous versions, considerations for the new prototype started to get shape. Also, the considerations and requirements resulted from the deep study performed by ASBGo members in the last version, the ASBGo++ , contributed as a remarkable help for the outline of guidelines for the new development.

The old versions of the ASBGo presented several *ad hoc* functionalities, which contributed to prove the concept of the different algorithms as well as their purpose and need in the medical context. However, the functionalities could be use, but always with the intervention of an ASBGo technician or many protocols and limitations had to be followed due to the academic nature of the prototypes. Additionally, none presented a well-constituted system architecture and so, they were never classified as a unified and global system. With this type of "systems", the further the development goes, more the system starts to get confused and messy for new developers and it gets extremely difficult (or impossible) to add new features along with previous ones implemented and so, the version collapses.

To tackle the problem of the previous prototypes, a new system was developed from root, following a Top-Down strategy, thus creating a logic and unified architecture. This way, by collecting the functional requirements of the new device, an overview of the system architecture was defined and two blocks were highlighted and proposed to be develop.

A profound research for new material was made. Many had to be purchased, some were acquired and more important, compatibilities had to be taken into account. The material gathered passes by powerful processing units, such as a development board, the *STM32F4 Discovery*, and an on-board computer, the *NUC6i7KYK*, several sensors, power supplies, motors, hardware modules, safety components and even cables. In this new system, the *STM32F4 Discovery* is known as the *Low-Level Controller* and the on-board computer, the *Main Controller*.

To interface with sensors, motors and other components, a real-time low-level application was developed in the *Low-Level Controller*. Built upon *FreeRTOS*, this application acquires and manages the data from a group of embedded sensors and sends the information to the *Main Controller*. Besides that, it handles emergency events, monitors the battery's voltage of the system and even detects if the battery reached a critical discharge state. It receives duty cycle values from the *Main Controller* and generates PWM signals for the motors' actuation.

At the hardware level, the new ASBGo* counts with a robust and complex electric system that also involves a complete set of electronic circuits that were designed, tested and implemented resulted in PCBs for the sensors' acquisitions, protections boards, and a main PCB , the *Shield*. This main PCB couples to the *Low-Level Controller* and is where are present the circuits to detect an emergency and to monitor the battery's voltage and is where the boards for the sensors' acquisitions are connected.

As referred, the *Low-Level Controller* and the *Main Controller* exchange data through a safe serial communication protocol. The *Low-Level Controller* acknowledges the commands from the *Main Controller* and acts accordingly. In the other hand, the *Main Controller* receives data and warnings in order to process the information and execute/command actions.

The high-level software was developed in the on-board computer and it was built upon ROS . This robotic software platform induces the development of granular software i.e. the development of different software components that communicate with each other through clear interfaces. The communication system is assured by a middleware provided by ROS and so, an inter-process communication is ensured. This way, the system gets unified and it gets easier to add new functionalities that can perform an independent task or can consume resources outputted from already existing processes in order to perform its task.

The *Main Controller* receives the sensors' data acquired from the low-level, at a configurable frequency, and disposes that resources that are constantly updated. With this, it is only necessary to develop new processes that consume that data, performs an algorithm and outputs information. This feature is one of the keys of the new ASBGo* that none of the previous versions had. Thereby, some high-level processes (ROS nodes) are implemented and are very organized in the ROS file-system in order to ease the life of future developers. It was implemented nodes that handle drive intentions of the users, either locally through the use of the walker's handlebar either remotely through the use of a wireless remote controller, calculate the current velocity of the motors, evaluate the battery's voltage, examine the motors and nodes responsible for the locomotion of the walker through the use of a PID control strategy. Also, a user-friendly, attractive and high-tech GUI was designed to allow the easy interaction between the device and the medical specialists.

At the mechanical level, several modifications were performed in order to suit the mechanical frame to the new system architecture developed. Such modifications allowed to couple and dispose the new material, improve the accessibility to certain parts of the walker and allowed the protection of certain components of the device.

In summary, the ASBGo has now a more advanced prototype that is prepared to get easily enriched with more functionalities providing also the means to accomplished that. More important, it provides the facilities to be used in rehabilitation sessions in order to help patients to recover from their gait difficulties.

6.2 prospect for future work

Since the new ASBGo* provides the means for a continuous development, the future perspective is to implement and include the rest of the proof-of-concept functionalities of the older versions in the new system. Besides implemented, they have to be studied to get conceptually well divided and logical, in order to be developed the necessary ROS nodes grating, this way, the continuous modularity of the new system.

One of the functionalities is the patient's gait and posture assessment. By the use of two RGB-D cameras, one for the gait and another for the posture, and by developing trustworthy computer vision algorithms it is possible to obtain spatio-temporal parameters that allows to help the patient to correct his posture and gait through the biofeedback that is provided. Also, that data can be stored in a future embedded database that will allow to check the evolution of the patients as well as store the patients' information. In addition, all this information can be shared, accessed an analyzed through the workstations of the medical team through an exclusive cloud of the walker.

Another interesting feature for the new device is a sophisticated autonomous navigation functionality. ROS offers several tools to develop and explore a great autonomous navigation algorithm. Using the laser range finder, it is possible to scan the environment in order to obtain a grid map. Then, it could possible to select a point in the map and by the use of the encoders' data, for the odometry, the laser, for obstacle avoidance, and a path planner algorithm, the walker would go by itself to the desired position. Additionally, a shared control, which takes in count the drive intentions of the user and the autonomous intentions of the walker to output a locomotion command, would enhance the autonomous navigation functionality.

Many more algorithms can now be explored. The sensors' data are disposed and updated at a certain frequency and can be easily accessed and so, the *sky is the limit*. It is relevant to say that since it is already developed a local GUI , future algorithms must be configurable through the interface and also, if their outputs offer important information for the users, that information must be represented in the GUI .

At the hardware level, there are a protection board that protects the on-board computer, the touchscreen monitor and the USB HUB from short-circuits. However it would be relevant to develop a circuit that protects this components against over-voltage by the combination of voltage dividers, zener diodes, operational amplifiers and relays. Besides being very unlikely to occur over-voltages, the circuits would confer a little more robustness to the electric system.

BIBLIOGRAPHY

Joana Alves. Dynamic model of a transtibial prosthesis. Master's thesis, Universidade do Minho, 2016.

Amazon. DROK DC Car Power Supply Voltage Regulator Buck Converter 8A/100W 12A Max DC 5-40V to 1.2-36V Step Down Volt Convert Module , 2017. URL https://www.amazon.com/DROK-Voltage-Regulator-Converter-1-2-36V/dp/B00C4QVTNU.

Itsuki Noda Noriaki Ando and Davide Brugali James J Kuffner. Simulation, modeling, and programming for autonomous robots. 2012.

Beetronics. I12 inch touchscreen (4:3), 2017. URL https://www.beetronics.co.uk/12-inch-touchscreen.

BestApplePrice. Apple iMac MF883HN/A All-in-One Desktop Computer, 2017. URL http://www.bestappleprice.com/apple-imac-mf883hn-a-all-in-one-desktop-computer.

Inês Caetano. Introduction of a motorized smart walker on rehabilitation of ataxic patients. Master's thesis, Universidade do Minho, 2017.

ColdFireElectronica. Acelerometro / Giroscopio 6-DOF IMU GY-521 MPU-6050, 2017. URL http://www.coldfire-electronica.com/esp/item/154/55/acelerometro-giroscopio-6-dof-imu-gy-521-mpu-6050.

SparkFun Electronics. Ultrasonic Range Finder - LV-MaxSonar-EZ1 (Retail), 2017. URL https://www.sparkfun.com/products/retired/10427.

ElekTRICKS. STM32F4-Discovery + ChibiOS = Data Acquisition System, 2016. URL http://www.elektricks.net/stm32f4-discovery-chibios-data-acquisition-system/.

Transfer Multisort Elektronik. PIHER PC-16 SH10IP06 50KB, 2017. URL https://www.tme.eu/gb/details/6pmo-50k/carbon-single-turn-axial-potentiometers/piher/pc-16-sh10ip06-50kb/.

Vitor Faria, Jorge Silva, Maria Martins, and Cristina Santos. Dynamical system approach for obstacle avoidance in a smart walker device. In *Autonomous Robot Systems and Competitions (ICARSC), 2014 IEEE International Conference on*, pages 261–266. IEEE, 2014.

FindIC. HEDS-5500 A06, 2017a. URL http://www.findic.us/image/en-re5GDGVxQ.

FindIC. W54-XB1A4A10-30, 2017b. URL http://www.findic.us/w54-xb1a4a10-30-datasheet-pdf-en-1Q22M3gpQ.html.

Phidgets Inc. Micro Load Cell (0-5kg) - CZL635, 2016. URL https://www.phidgets.com/?tier=3&catid=9&pcid=7&prodid=224.

Lentin Joseph. *Mastering ROS for robotics programming*. Packt Publishing Ltd, 2015.

Ruijiao Li, Mohammadreza A Oskoei, Klaus D McDonald-Maier, and Huosheng Hu. Ros based multi-sensor navigation of intelligent wheelchair. In *Emerging Security Technologies (EST), 2013 Fourth International Conference on*, pages 83–88. IEEE, 2013.

LXbattery. BAT. ZENITH DEEP-CYCLE ZLS120-135 - 12V36 AH, 2017. URL https://www.lxbattery.pt/pt/bat-zenith-deep-cycle-zls120135-12v-36ah.html.

Lamec M. Intel NUC Kit NUC6i7KYK Mini PC Review, 2017. URL https://pcverge.com/intel-nuc-kit-nuc6i7kyk-mini-pc-review/.

Aaron Martinez. Concepts, 2016a. URL http://wiki.ros.org/ROS/Concepts.

Aaron Martinez. Ros architecture and concepts, 2016b. URL https://www.packtpub.com/books/content/ros-architecture-and-concepts.

M Martins, C Cifuentes, A Elias, V Schneider, A Frizera, and C Santos. Assessment of walker-assisted human interaction from lrf and wearable wireless inertial sensors. In *2nd International Congress on Neurotechnology, Electronics and Informatics-NEUROTECHNIX 2014*, volume 1, 2014a.

M Martins, A Frizera, R Ceres, and C Santos. Legs tracking for walker-rehabilitation purposes. In *5th IEEE RAS/EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 387–392. IEEE, 2014b.

Maria Martins, Cristina Santos, Eurico Seabra, Luis Basílio, and Anselmo Frizera. A new integrated device to read user intentions when walking with a smart walker. In *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*, pages 299–304. IEEE, 2013a.

Maria Martins, Eurico Seabra, Luís Basílio, and Cristina P Santos. Conceção, projeto e desenvolvimento de um guiador para um andarilho motorizado. In *International Conference on Engineering UBI (ICEUBI)*, pages 27–29, 2013b.

Maria Martins, Cristina Santos, Anselmo Frizera, and Ramón Ceres. Real time control of the asbgo walker through a physical human-robot interface. *Measurement*, 48:77–86, 2014c.

Maria Martins, Cristina Santos, Eurico Seabra, Anselmo Frizera, and Ramón Ceres. Design, implementation and testing of a new user interface for a smart walker. In *Autonomous Robot Systems and Competitions (ICARSC), 2014 IEEE International Conference on*, pages 217–222. IEEE, 2014d.

Maria Martins, Cristina P Santos, Anselmo Frizera, Ana Matias, Tânia Pereira, Maria Cotter, and Fátima Pereira. Smart walker use for ataxia's rehabilitation: Case study. In *2015 IEEE International Conference on Rehabilitation Robotics (ICORR)*, pages 852–857. IEEE, 2015.

Maria M Martins. Online control of a mobility assistance smart walker. Master's thesis, Universidade do Minho, 2011.

Maria Manuel Carvalho de Freitas Martins. *ASBGo: A smart walker for mobility assistance and monitoring system aid*. PhD thesis, Universidade do Minho, 2016.

Mediacode. Prémio eng° jaime filipe, 2017. URL `http://www.inr.pt/content/1/1149/premio-jaime-filipe`.

Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. Yarp: yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):8, 2006.

Bridget Butler Millsaps. From 3DS and Intel: 3D Scan and Print Customized Selfies from the Desktop with RealSense and 3DMe, 2015. URL `https://3dprint.com/53482/intel-3d-systems-3dme/`.

Nagares. RLAC/4-24, 2017a. URL `http://www.nagares.es/buscador/detalle_reles.php?codigo=0102792`.

Nagares. RLP/5-24, 2017b. URL `http://www.nagares.es/index.php/en/relays.html`.

OROCOS. The orocos project, 2016a. URL `http://www.orocos.org/`.

OROCOS. The orocos project, abstract, 2016b. URL `http://orocos.org/node/61`.

Solenne Page, Maria M Martins, Ludovic Saint-Bauzel, Cristina P Santos, and Viviane Pasqui. Fast embedded feet pose estimation based on a depth camera for smart walker. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 4224–4229. IEEE, 2015.

Karlsson Robotics. Slide Potentiometer - 10K, 2017. URL `http://www.kr4.us/slide-potentiometer-10k.html`.

RobotShop. Hokuyo URG-04LX-UG01 Scanning Laser Rangefinder, 2017. URL `http://www.robotshop.com/en/hokuyo-urg-04lx-ug01-scanning-laser-rangefinder.html`.

Ricardo Rodrigues. Projeto, desenvolvimento, otimização e construção de um andarilho multifuncional motorizado. Master's thesis, Universidade do Minho, 2014.

ROS. About ros, 2016a. URL `http://www.ros.org/about-ros/`.

ROS, 2016b. URL `http://www.ros.org/`.

Andrey Rusakov, Jiwon Shin, and Bertrand Meyer. Simple concurrency for robotics with the roboscoop framework. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1563–1569. IEEE, 2014.

Johan Rutgeerts. *Constraint-based task specification and estimation for sensor-based robot tasks in the presence of geometric uncertainty*. PhD thesis, KU Leuven - University of Leuven, 2007.

Isaac Saito. tf, 2016. URL http://wiki.ros.org/tf.

Daniel Serrano. Middleware and software frameworks in robotics–applicability to small unmanned vehicles–.

Jiwon Shin, Andrey Rusakov, and Bertrand Meyer. Smartwalker: an intelligent robotic walker. *Journal of Ambient Intelligence and Smart Environments*, 1:1–5, 2016.

Solarbotics. Sharp Analog Distance Sensor 10-80cm, 2017. URL https://solarbotics.com/product/35238/.

SuportLaser. Botão de Emergência, 2017. URL http://www.suportelaser.com.br/pd-374d12-botao-de-emergencia.html.

Cytron Technologies. Cytron 30A DC Motor Driver, 2017. URL https://www.cytron.io/p-md30c.

A Tereso, M Martins, CP Santos, M Vieira da Silva, L Gonçalves, and L Rocha. Detection of gait events and assessment of fall risk using accelerometers in assisted gait. In *Informatics in Control, Automation and Robotics (ICINCO), 2014 11th International Conference on*, volume 1, pages 788–793. IEEE, 2014.

Dirk Thomas. Introduction, 2016. URL http://wiki.ros.org/ROS/Introduction.

Emmanouil Tsardoulias, Fotis Psomopoulos, Ilias Trochidis, Alexandros Gkiokas, Stratos Arabatzis, Wojciech Szynkiewicz, Vincent Prunet, David Daney, and Włodzimierz Kasprzak. D3. 1 state-of-the-art report [certh]. 2013.

Andreas Wachaja, Pratik Agarwal, M Reyes Adame, Knut Möller, and Wolfram Burgard. A navigation aid for blind people with walking disabilities. In *IROS Workshop on Rehabilitation and Assistive Robotics, Chicago, USA*, volume 13, pages 13–14, 2014.

Andreas Wachaja, Pratik Agarwal, Mathias Zink, Miguel Reyes Adame, Knut Möller, and Wolfram Burgard. Navigating blind people with a smart walker. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 6014–6019. IEEE, 2015.

WAVESHARE. FT232 USB UART Board (Type A), 2017. URL https://www.waveshare.com/ft232-usb-uart-board-type-a.htm.

YARP. Welcome to yarp, 2016a. URL http://www.yarp.it.

YARP. What exactly is yarp?, 2016b. URL http://www.yarp.it/what_is_yarp.html.