RAM ROBOTICS AND MECHATRONICS

Increasing the Autonomy of the Pipe Inspection Robot PIRATE

G.A. (Gisela) Garza Morales

MSc Report

Committee:
Prof.dr.ir. S. Stramigioli
Dr.ir. E.C. Dertien
Dr.ir. J.F. Broenink
Dr.ir. G. van Oort

August 2016

036RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

Summary

The Pipe Inspection Robot for AuTonomous Exploration, or for short PIRATE, is an inspection robot for gas pipes. It aims to determine the location of a leak more accurately so that workers know where they need to dig a hole and thereby making the digging more precise (and cheaper). Another advantage that might be offered by such robot, is that not only leaks can be detected, but also the quality of the network can be inspected and weak spots can be detected and be repaired even before gas leakage might occur. Currently the efforts put into the robot have been focused on the mechanical, electronics, communication and vision areas, nonetheless, the movement is still dependent on manual operation by a skillful inspector. This can provide a safe inspection, however, it can be labor intensive (costly) and time consuming. Ideally and in order to be usable in a commercial (industrial) environment a certain level of autonomy is required.

The present thesis aims at increasing the current level of autonomy for the PIRATE by designing and implementing autonomous behaviors. At first an analysis of the possible layered control software structure is done. From this, the scope of the thesis is limited to the Sequential layer (control of small/simple sequences). Based on this, the goal becomes to design and implement these simple sequences (Partially Autonomous Behaviors - PABs) in a modular and scalable manner, which can also be integrated with the current manual control. To demonstrate the potential of the design a simple mission that combines various PABs (namely clamp straight, drive and home) has been developed.

The design part of the project includes the development of the software framework where the simple autonomous sequences can be constructed. This is done using the object oriented approach, where different classes are developed according to specific functionality. The basic functionalities that are covered in the design are: communicating with the user interface, translating its commands and send them as robot instructions, creating a centralized structure for coherent message sharing, developing and managing operation modes (manual and partially autonomous), designing and implementing the Partially Autonomous Behavior (PAB) routines and receiving the current robot state, among others.

For the implementation of the software framework the Robot Operating System environment is chosen, because it has been previously used for the implementation of the feedback state reception and a visual interface, as well as for the wide variety of packages for robot implementation it offers. During the development, the transition conditions (guards) were experimentally defined. In the final experiments, these conditions and the sequences are tested covering three different situations: clamping when no re-orientation is needed, clamping when slight re-orientation is required and clamping starting from a lying position (complete re-orientation).

The results show that, in each of these situations, the proposed sequence and transition conditions effectively clamp the robot in the desired orientation. Apart from these experiments, the simple mission was tested and it successfully shows that different PAB classes could be combined to create more complex sequences or plans. It was possible to see the importance of the software control layers approach, which greatly facilitated the design process and accomplished modularity and re-usability, especially in the pattern followed to design the PAB classes, which can be reused and adjusted for future PABs. Some general recommendations are to include, apart from the angle feedback, torque measurement in the transition conditions to increase robustness. Another strong suggestion is to improve the wiring and in general the prototype material to reduce fragility. Overall the present study successfully presents a suitable software framework for further development of other PABs in the Sequential layer, as well as potential for creating more complex missions by integrating them.

Preface

To my lovely parents, Rosario and Ernesto, without whom this would not have been possible. Thank you for always encouraging me, believing in me and above all for the wonderful life that you gave me and all the opportunities I had. To my little brother, who has always been a support in my life. He is true to himself as I want to be. To the love of my life, Armando, thank you for everything, for being in my life, for inspiring me everyday, for being cheerful and calming me during my stress days (always). This marks the end of one of our adventures. Hopefully many more will come.

I would like to dedicate this thesis specially to my aunt Mague (RIP) and my grandfather Ernesto(RIP). God has called both of you during these two years that I was away. I know that you were and still are watching over me and I hope this makes you proud. I will forever miss you.

To the rest of my lovely family who has always supported me, specially my cousins Greta, Edgar, Eduardo, Caty, Juan Carlos, Tommy, Male, Beto, Luis, Roberto, Daniel, Paola, to my aunts and uncle Maria Elena, Alicia, Tomas, Norma, Sandra, Chiquis, Lalo and Sergio. Thanks to my nephews Pepe, Erik, Emilio and Esteban and to baby Dania, who I haven't met yet. You guys were the first that showed me what loving something so innocent and perfect was about.

Thanks to all the great people that I have met through these wonderful (and sometimes painful) two years: Hengameh, Mohamed, Evyatar, Giuseppe, Nelson, Charalambos, Abishek, Helena, Carlos, Shamel, Adel, Samer, Vivian, and many more. Also to the wonderful people that I met way before and that are also here living their dutch adventures: Cynthia and Cobos, thanks for your support, your advice and joining my adventures. To my best group of friends: Vero, Johana, Joyce, Loren, Marce, Chaba, Franky, Borre, Chihuis, Axel, Cesar, Moros. Thanks to all of you for your great support and friendship.

From RAM, special thanks to Mark, I really appreciate all the time you spent explaining me and all your patience, without you this wouldn't have been possible. To Douwe, for introducing me to MakerSpace, for challenging me and for his great advice in a time that I really needed it. To Zhou, for all your great advice and ideas. To my supervisor Edwin, for all his support, inspiration and guidance. To Jan, for pointing me to the right direction, your overall support and apart from this project, all your teaching. To Gijs, for helping me when I needed the most during this project, thanks for your support. To Jolanda, for calming me down when I went to see her, I really needed to hear that it was going to be okay.

I am probably missing more people in the list, but in general thanks to all the people who in one way or another have touched my life and stayed with me. I promise I won't let you down.

Gisela A. Garza Morales
Enschede, August 2016

Contents

1 Introduction

1.1 Context

The gas distribution network in the Netherlands has a length of roughly 100.000 km in urban areas. For the whole network there exists a constant need for monitoring in order to maintain the system running. By (Dutch) law, segment of the gas pipe network has to be inspected every 5 years[1]. An important remark is that pipe replacement is expensive, so it is important to have accurate data on the locations of leaks or damaged pieces [1]. Currently the inspection of the network is done by leak searching ('sniffing') above ground. The worst case accuracy of above ground detection is several meters [1]. For this reason, a more precise predictive monitoring can be helpful to determine a more accurate location of the leakage as well as provide information about how long can a segment still offer reliable service and when exactly it is recommended to replace it.

In response to this pipe inspection problem, the PIRATE project was first initiated in 2006 by KIWA[2], and continued since by the Robotics and Mechatronics (RaM) Department at the University of Twente. The project was given the acronym *'PIRATE'* which stands for Pipe Inspection Robot for AuTonomous Exploration. The idea can be considered as a response to a report by the Dutch Transportation Security Council chaired by Mr. Pieter van Vollenhoven [2] in which the details and figures of safety of the gas-transportation in the Netherlands have been given.

The PIRATE robot aims to determine the location of a leak more accurately so that workers know where they need to dig a hole in these urban areas and thereby making the digging more precise (and cheaper). Another advantage that might be offered by such robot, is that not only leaks can be detected, but also the quality of the network can be inspected and weak spots can be detected and be repaired even before gas leakage might occur.

The project is a result of a cooperation between University of Twente and DEMCON[3]. Since the beginning, vast efforts have been put into the mechanical design such as the ones started by Dertien et. al [3] in 2006, continued by Vennegoor [4] in 2007, Spijksma [5] and Burkink [6] in 2009, and Borgerink [7] in 2012 . Moreover, the development of the electronics was described by Dertien [8] and Ansink [9] in 2007 and the local control by Reemeijer in 2010 [10]. To complement the inspection characteristics, research has been done to integrate measurement and vision capabilities to the robot by Dorst [11] in 2010, Brilman [12] in 2011 and finally by Reiling [13] in 2014. On the self-localization side it is possible to mention Meennink's [14] work from 2010 as well as the development of wireless communication by Doggen [15] in the same year. All efforts were culminated and summarized with a PhD thesis by Dertien [1] in 2014. Furthermore, after 2014, there has been more work done by Reiling, which has made the local (real-time) control software more robust and reliable. This has achieved a very competent manual operation control for the robot.

In a finished state, the robotic system is expected to make money within the first year of operation; a gain of 11M euro per year is thought to be possible [1] with a full autonomous inspection system. Because of this, special attention needs to be paid to this, and invest in the development of more abstract software levels that can have more autonomous behaviors.

1.2 Problem statement

As mentioned before, currently the efforts put into the robot have been focused on the mechanical and electronics design, nonetheless, the movement is still dependent on manual operation

[1] Besluit externe veiligheid buisleidingen, http://wetten.overheid.nl
[2] KIWA (gastec), Apeldoorn, http://www.kiwa.nl
[3] DEMCON Advanced Mechatronics B.V. www.demcon.nl

by a skillful inspector. This can provide a safe inspection, however, it can be labor intensive (costly) and time consuming. Ideally and in order to be usable in a commercial (industrial) environment a certain level of autonomy is required. Development of fully autonomous behaviors for the robot, requires a complex layered software architecture that offers more abstraction levels than the currently (manual) local closed loop controllers. Because of this required complexity, full autonomy is out of scope of the present work due to time limitation. However, the first steps towards this ambitious goal will be taken. For the present project, the experience obtained from manual control will be formalized into a more abstract software level, which although not yet fully autonomous, it will provide the software framework necessary to implement partially autonomous behaviors.

1.3 Goals

The goals for this project are formulated as follows:

1. Design a centralized software framework that can be used for implementing the possible control layers for the PIRATE.
2. Incorporate the current manual system to the new autonomous control so that both modes can coexist in the system (with the necessary restrictions).
3. Design and implement (at least one) modular autonomous behavior routines based on hybrid state-machine structures.
4. Design and implement reusable standard motion functions (defined as *motion primitives* by Dertien [1]) that can be used in the autonomous routines.

1.4 Approach

In the latest control development of the PIRATE by Reiling, an Arduino MEGA board was the centralized master of the system. It was responsible of the interpretation of the user interface's commands, as well as the bi-lateral communication with local PID controllers (ATmega M328 boards located inside the robot). Figure 1.1 shows the original architecture, where the Arduino MEGA acts as the master.

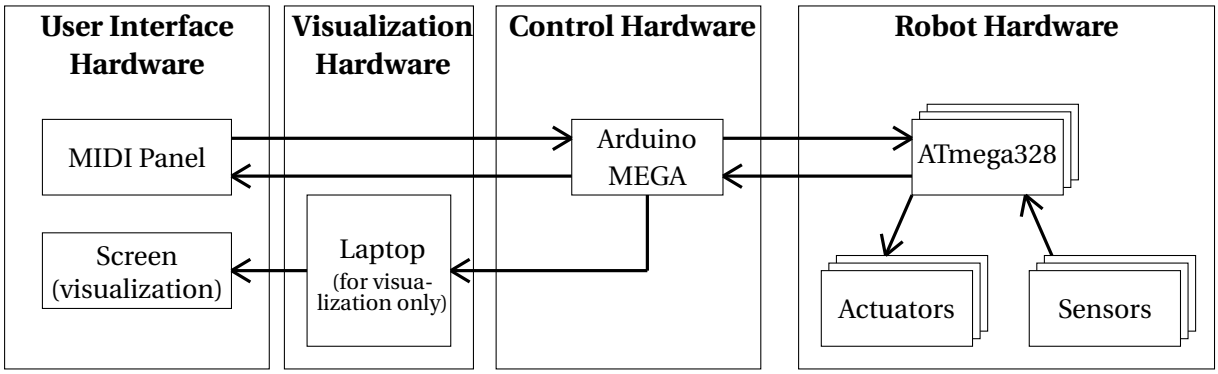


Figure 1.1: General previously-existing hardware architecture

The first step towards increasing the autonomy of the PIRATE would be to migrate the Arduino MEGA's functionalities to a new software framework (in a laptop). Figure 1.2 shows that in the desired architecture to be implemented, a laptop will now be the control hardware. Currently, apart of the manual control, a visual interface for the PIRATE has been developed by Reiling using the Robot Operating System (ROS). This can be seen in Figure 1.1. Because ROS has already been used successfully, it has been chosen for the development of the software framework.
Due to the robustness that the current communication between the Arduino Mega and the ATmegas, it is desired, when possible, to keep this part of the current implementation intact. In this way, the Arduino Mega's role will be as a transparent bridge between the framework and

the local PID control (ATmega328) boards. Because we want the software to be modular and reusable, the object-oriented approach will be taken for the design. In this way, there will be message-related classes that will handle the data structures to be sent to the PIRATE, for example setpoint, control modes and limits. Other classes shall handle the manual commands and autonomous behaviors (for these last ones individual classes will be considered to implement each of the state machines). The design will also include a class that contains the reusable standard functions (motion primitives) that will be used in all of the autonomous routines. The aforementioned visual interface integrates the joint state feedback (sent by the ATmegas) and uses the Robot Operating System (ROS) environment to output a visualization in a laptop. This feedback, which is already in ROS, will be retrieved in a class from which it will be distributed where required. Finally, and as it can be seen in Figure 1.2, the framework will also include the design and implementation of communication capabilities with the user interface. The developed autonomous behaviors will be evaluated with a graphic assessment of the state-transition conditions and the overall test of the performance of the state-machines (time, torque consumption, etc). This will also provide information about the potential of the developed software framework as well as the capabilities of the reusable functions (motion primitives) within the sequences. Apart from an individual test of the autonomous behavior, this thesis will yield a simple demonstrable mission where some autonomous behaviors will be tested. The mission will consist of *clamping, driving forward, driving backward and unclamping.*

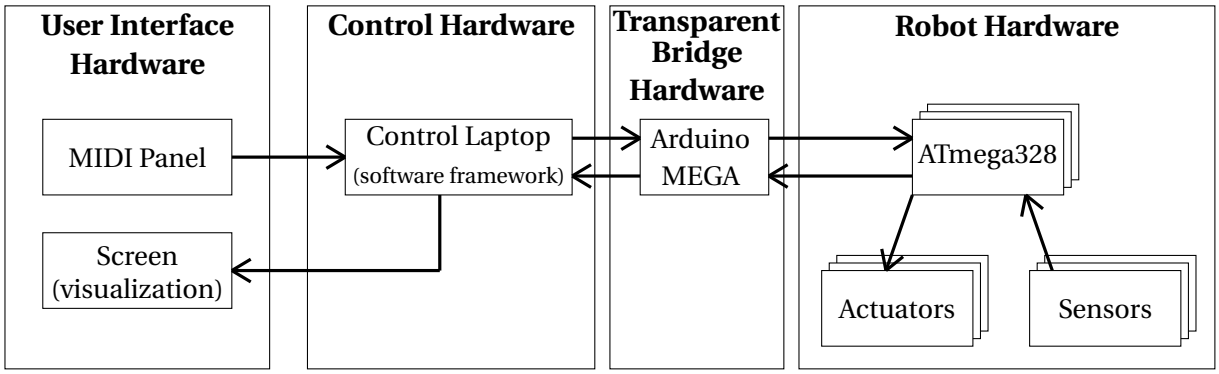


Figure 1.2: General Implemented Hardware Architecture- Difference lays in the fact that the user interface is now connected to the laptop (where the software framework sits) and the Arduino MEGA is now a *transparent* bridge between ROS and the ATmega328 boards (local PIDs)

1.5 Organization of the report

In chapter 2 background information will be given, including the basics of the Robot Operating System (ROS) and a discussion of the software abstraction levels applicable for similar robots. Moreover, also in this chapter, a detailed description of the PIRATE robot will be given. In chapter 3, we will analyze key aspects necessary for functionality of the software framework, such as the operation modes needed, etc. An analysis of a possible control software structure for the PIRATE will be presented and from this we will identify the level of autonomy that can be reached within the scope of this thesis. From this we will derive a set of requirements for the present project. In chapter 4, the software framework design is discussed. In this part we will present the architecture of the software framework, including all the developed messages and classes. In chapter 5 the implementation of the software framework for the system is described in detail. In chapter 6 the results of the implementation are shown. Finally chapter 7 concludes this thesis and presents recommendations for future work.

2 Background

To place the design of the software framework into perspective, background information is given on the ROS and on software control architectures. Also, some related work is presented.

2.1 Robot Operating System (ROS)

The Robot Operating System (ROS) is an open-source flexible 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 across a wide variety of robotic platforms [1].

ROS supports a wide range of sensors and algorithms. For the user it is also easy to add new parts, since Python and C++ (amongst others) are supported as programming languages and it can be used with the most popular IDEs [2].

In ROS, a running instance of a program is called a *node* [3], which is basically an executable to communicate with other nodes through a network structure.

In order for several nodes to run at the same time, they must be able to communicate with one another. The part of ROS that facilitates this communication is called the ROS *master*.

The primary mechanism that ROS nodes use to communicate is to send *messages*, which have particular data type(s) and organized into named *topics*. Message exchange is based on the *publish/subscribe* pattern: a node that wants to share information will publish messages on the appropriate topic or topics; a node that wants to receive information will subscribe (listen) to the topic or topics that it's interested in. The ROS master takes care of ensuring that publishers and subscribers can find each other [4]. Figure 2.1 shows the publish/subscribe pattern used in ROS.

In ROS, all messages on the same topic must be of the same data type. Inside ROS, these messages are transformed to header files, where the user defined fields are placed in *structs*, making the communication more robust. Another benefit of using a message passing system is that it forces the implementation of clear interfaces between the nodes in the system, thereby improving encapsulation and promoting code reuse.

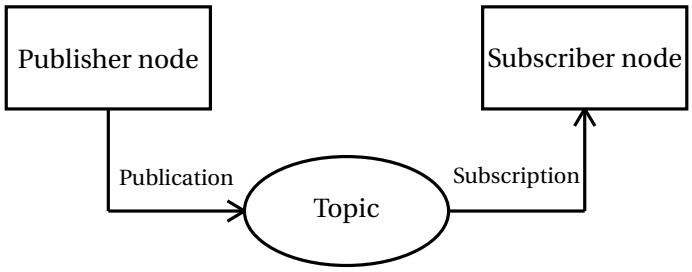


Figure 2.1: Publish/Subscribe structure in ROS.

Because the publish/subscribe system is anonymous and asynchronous, the data can be easily captured and replayed without any changes to code. Furthermore, if synchronous request/response interactions are required between processes, the ROS provides this capability using *services*.

[1] ROS.org - About ROS http://www.ros.org/about-ros/
[2] ROS.org - IDEs http://wiki.ros.org/IDEs
[3] ROS.org - Understanding Nodes http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes
[4] ROS.org - Understanding Topics http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

While topics (anonymous publish/subscribe) and services (remote procedure calls) cover most of the communication use cases in robotics, sometimes it is necessary to initiate a goal-seeking behavior, monitor its progress, be able to preempt it along the way, and receive notification when it is complete. ROS provides *actions* for this purpose. Actions are like services except they can report progress before returning the final response, and they can be preempted at any time by the caller [5].

For more detailed information about ROS, its components and features, the reader is advised to visit its website [6] or to take a look at [16] and [17].

2.2 UML basics

In this section, the basic definitions and most common features of UML will be discussed. This does not aim to be in any way a comprehensive summary on UML, but rather provide the base for the reader to understand the diagrams presented in this thesis.

The *Unified Model Language (UML)* is a language for expressing the constructs and relationships of complex systems [18]. This modeling approach is more complete than other methods and is particularly efficient for modeling real-time, embedded systems. The basic features of UML discussed in the present thesis are:

1. Class diagrams
2. Use cases

2.2.1 Class diagrams

In object-oriented programming, the term *class* is used to describe an abstraction of the common properties from a set containing many similar objects [18]. In this sense, a class can be though of as the *type* of object. In UML, classes are shown using rectangles with the name of the class inside a rectangle [18]. A very common variation of this uses a three-segment rectangle, from which the top mentions the name of the class, in the middle the attributes (properties) and in the bottom the member functions (behaviors) related to such class.

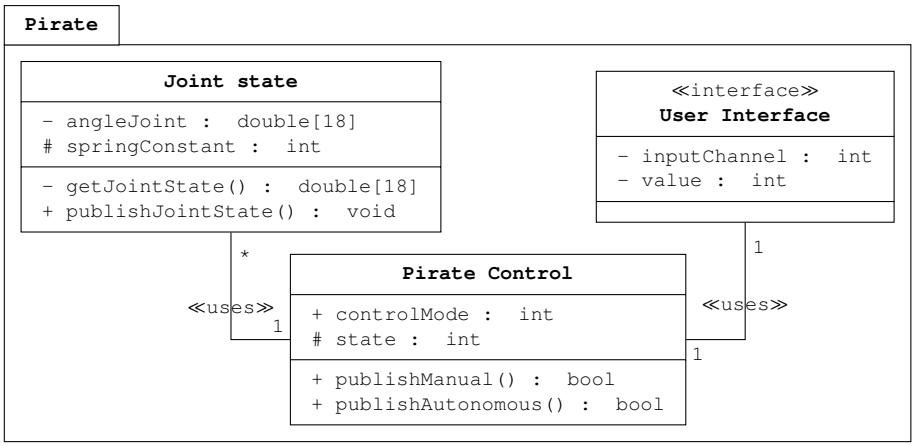


Figure 2.2: Example of a UML class diagram for PIRATE control

Figure 2.2 shows an example of a simple class diagram. It is possible to find three classes, namely Pirate Control, User Interface and Joint State. Inside the classes, it is possible to see the attributes, member functions (behaviors) and the access modifiers. The access modifiers restrict the accessibility to the attributes and member functions and can be *public (+), private(-)* or *protected(#)*.

[5] ROS.org - Core components http://www.ros.org/core-components/
[6] ROS.org - http://www.ros.org

Another important UML concept that can be seen in Figure 2.2, is the number at the end of the relationship line, which denotes the number of instances that participate in the relationship at each end. This is known as *multiplicity* of the role. This means that the number 1 in the side of the Autopilot class means that there is one single instance of this class. On the other hand, the symbol $*$ on the side of the Joint state class, means that the instances on that side of the relationship are "unspecified but greater than or equal to 0" [18].

Inevitably, when modeling a large system, there will be many different classifiers in the model. UML provides an organizing element called a *package*. Normally each package represents a specific part of the system.

2.2.1.1 Relations among Classes

In order for an instance of a class (also known as *object*) to send messages to another, they must associate with each other in some way [18]. For this, there exist different types of relationships. According to [18] and [19], the basic relationships are:

- **Association**: indicates when one object uses the services of another but does not own it and is normally bi-directional. This bi-directional relationship is represented by a single solid line between the two classes. In an unidirectional relation only one class knows that the relationship exists. This is drawn as a solid line with an open arrowhead pointing to the known class.
- **Aggregation**: applies when one object physically or conceptually contains another. The aggregation class is the *owner*, but each of them have their own life-cycles. The representation is a solid line from the owner class to the part class, with an unfilled diamond shape on the owner's end.
- **Composition**: is a strong form of aggregation. In this case, the child class's instance life-cycle is dependent on the owner object's life-cycle. The composition relationship is drawn like the aggregation relationship, with a filled diamond shape.
- **Generalization**: is also known as *inheritance* and refers to the ability of one class (*child class*) to inherit the identical functionality of another class (*super class*), and then add new functionality of its own. Represented by a solid line drawn from the child class with a closed, unfilled arrowhead pointing to the super class.

2.2.2 Use Case diagrams

According to Powel[18], a use case is a function that returns an observable value to an *actor* (object outside its context), without revealing the design of the function. They associate with the actors by receiving and sending messages between them. Their representation is usually as ovals with solid borders, while the icon for an actor is a stick figure.

Figure 2.3 shows an example of a simple use case diagram.

2.3 System Control Architectures

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators [20]. Thus, a robot is also an agent. These components need to interact with each other, therefore, the architecture needs to structure the system in a way that helps managing this complexity [20].

Modern intelligent robots to assist in different challenging environments require advancement in robotic services. In order to provide those services, it is required to have sophisticated intelligent software in the robot systems [21]. This software should recognize uncertain surrounding environments, plan for tasks based on knowledge, and execute behaviors such as navigation or manipulation [21].

In the next subsections, an overview of the most important paradigms is presented.

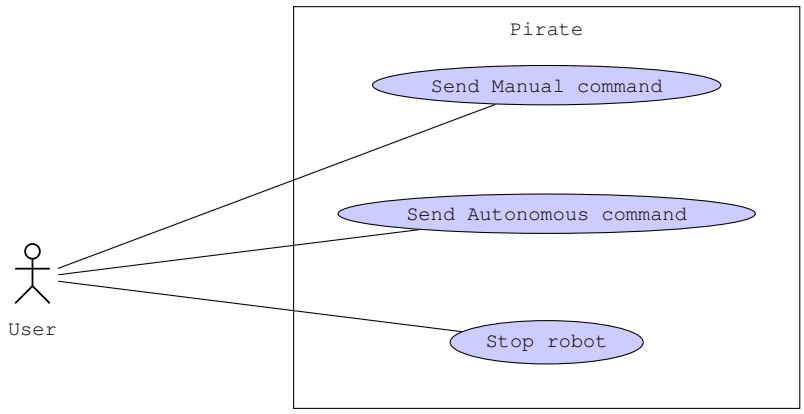


Figure 2.3: Example of a Use Case diagram for the PIRATE

2.3.1 SPA- The Hierarchical Paradigm

This is the oldest approach which has been used to compose robot systems. It is often referred to as the *Sense, Plan, Act (SPA)* approach, which is traditionally sequential and orderly [20].

This approach has two important architectural features [22]:

1. *Control flow is unidirectional and linear*: Information flows from sensors to world model (perception and modeling) to plan to effectors (task execution and motor control).
2. *Execution of an SPA plan is analogous to the execution of a computer program*: Both are built of primitives composed using partial orderings, conditionals, and loops and the intelligence of the system resides in the planner, not the execution mechanism.

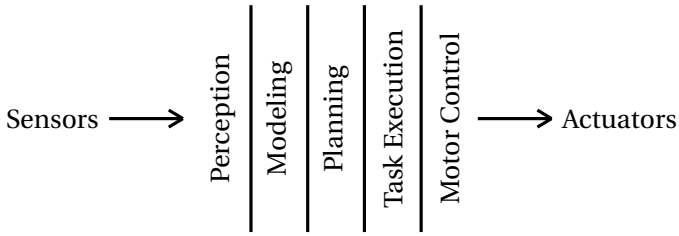


Figure 2.4: SPA Approach. Figure from [23]

The main disadvantage of this approach results precisely from the sequential flow shown in Figure 2.4. In this case, the plan usually takes more time because of the need to compute the actions. Because of this, the output signal to the actuator is very delayed and results in "jittery" behavior. Furthermore, it faces the fact that, when the plan is ready, the environment may have changed already and thus, when the planner is finished it may have generated an obsolete plan.

2.3.2 The three-layer architecture

The three-layer architecture consists of three components: a *reactive feedback control mechanism, a reactive plan execution mechanism, and a mechanism for performing time-consuming deliberative computations.* These components run as separate computational processes. The first layer is called *Controller*, the second one *Sequencer* and the deliberative one is referred to as *Deliberator*.

The main characteristics of the three layers according to Gat[22] are:

1. *Controller*: consists of one or more feedback control loops, tightly coupling sensors to actuators. Contains functions (called primitive behaviors or skills) that are called to ac-

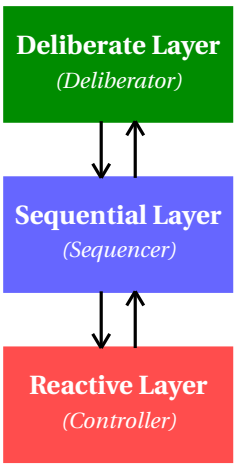


Figure 2.5: Three layer architecture. Figure from [21]

tion by the Sequencer, to form a more complex task-achieving behavior. Main constraint is that the computing of the algorithm must be fast (real-time).

2. *Sequencer*: its job is to select which primitive Behavior the controller should use at a certain time, and to supply parameters for the Behavior. By changing primitive Behaviors at strategic moments the robot can perform useful tasks and sometimes cope with non-idealistic situations. Because of this, the sequencer must be able to respond conditionally to the current situation, by adding steps, performing additional checks, etc.
3. *Deliberator*: takes care of the time-consuming computations. Usually tasks are planning, search, vision or any complex processing algorithms. Several Behavior transitions can occur between the invoking and producing a result. It produces plans or responds queries for the sequencer.

2.3.3 Layered control structure

In *mechatronic* systems, the dynamic behavior of the plant/machine to be controlled is essential for the functionality of the system.

According to this, the control processes should be divided over the range of hard and soft real-time. To make this apparent, a layered priority structure is proposed by Broenink et al. in [24], which is based on Bennet[25]. This structure has since then been modified, and the existing approach was published by Broenink[26] in 2014 and is depicted in Figure 2.6. It is an architecture used for robotic systems, containing the physical/mechanical plant (robot) at the right and the (embedded) control software at the left. In the middle sit the parts that transform the signals between the two domains. Again in this architecture, similar to the one shown in subsection 2.3.2, the control layers are used to indicate the priority (and abstraction) of the control processes.

The embedded control software part is structured in layers according to Broenink et al. [26] in 2012:

1. *Loop control*: contains the control algorithms for controlling the actuators. Since the actuators require an update of their setpoint every sampling period, this layer is implemented as hard real-time.
2. *Sequence control*: can be described as a task level controller, which enables and feeds the loop controllers with setpoints and necessary parameters. This part can be implemented in soft real-time.
3. *Supervisory control*: is a strategy controller. It performs calculations, which take considerable computing time (compared to the sampling period). The result of these cal-

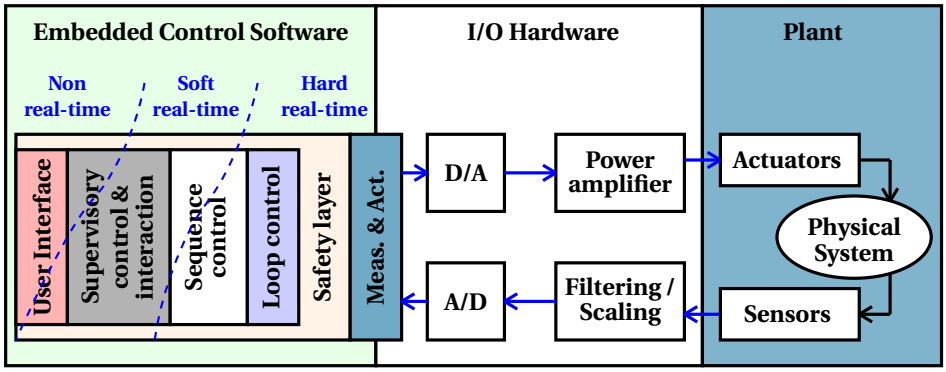


Figure 2.6: Layered software architecture for embedded systems. Figure from [27]

culations to determine the tasks to be sent to the Sequence controller. Typical supervisory control tasks include path planning, vision algorithms and environment mapping, among others.

4. *Safety layer*: checks for safety issues on all control levels (this is why it is drawn behind the other layers). In this level, all signals going to the hardware are reviewed.
5. *Measurement & Actuation*: this layer implements different techniques for signal conditioning in order to adapt the variable ranges to the signal levels of the hardware.

Comparing with the three layered approach, discussed in subsection 2.3.2, it is possible to see the similarities between the Supervisory control with the Deliberator, the Sequence control with the Sequencer and the Loop control with the Controller. However, an important difference present in this layered control structure is the definition of the real-time requirements for each of the layers, as well as the inclusion of the Safety and the Measurement and Actuation layers. This serves to give a more realistic view of a software architecture for robots.

2.4 The PIRATE robot

2.4.1 Design background

The PIRATE design proposed by E. Dertien's PhD thesis [1] allows for a separation in modules where each has its specific function. Advantages of modularity are: interchangeability for repairs and easiness to remove or add modules depending on the application. Furthermore, manufacturability of the system increases when the modules have identical shapes and parts. In early designs, the PIRATE had seven modules for which reductions have been discussed and proposed by Reemeijer in 2010 [10]. Nevertheless, the design has changed over time [28] and the current design, differs from the one presented in Dertien's PhD thesis[1] which consisted in five modules. The latest model of the PIRATE has again seven physical modules. This consists of the former five plus one module in the front and one in the back to integrate vision and illumination capabilities to the robot. Amongst these seven modules, four different types, all of which have the possibility to be driven by the wheels, can be distinguished (see Figure 2.7):

1. Bend module
2. Rear module
3. Front module
4. Rotational module

2.4.1.1 Bend Module

The bend module is perhaps the most crucial part in the robot's design. To drive in a stable manner through a pipe, the robot always needs to constantly exert a certain force on the

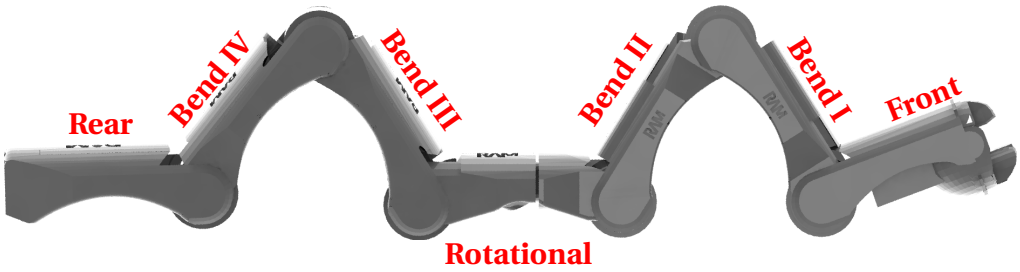


Figure 2.7: Different module types in the PIRATE

pipe wall in order to keep itself in the center. This force (called *clamping/preloading force*) is generated through a torque between the first two modules and the last two modules, hence the current necessity for four bending modules (from Figure 2.7 Bend I, II, III and IV). These modules are positioned in a V-shape through two geared motors with a spring connected acting as series elastic actuator or SEA [29]. This operation is known as *clamping*. Clamping is necessary to make sure the wheels stay in contact with the pipe wall and provide required traction. Due to space requirements the motor used in these modules, a Faulhaber 1016 motors, lacks an encoder, so feedback comes from two magnetic hall-effect sensors AS5055 from AMS, one for the spring and one for the angle. These two magnetic absolute encoders are used as displacement sensors, one to measure the angle between the modules and the other to dimension the spring deflection directly, independent of the angle between the modules [1]. Figure 2.8 shows the PCB designed for mounting the AS5055 sensors in the PIRATE. The motor does not need to be powered to maintain a certain clamping torque which is an important concern when considering battery powered operation, especially since the clamping torque needs to be provided continuously [1]. Another function of these motors is to bend the entire robot shape along the curve of a bend or T-joint[1].

(a) (b)

Figure 2.8: (a) AS5055 Position sensor PCB. Figure from [1] (b) CAD drawing of gearing in bending module. Figure from [4]

To understand better how this clamping force acts, it is convenient to look at a geometric relations between two bending modules (which is where the torque T will be generated). This torque results in a clamping force F_c on the wall. Independent on the orientation of the robot, the configuration for clamping will have on one side one wheel and on the other two wheels (forming the V-shape). The wheel that is alone will posses twice the clamping force of the other two wheels. This is explainable since the reaction force for each of the two wheels will sum up at this point. These geometric relations are shown in Figure 2.9.

The amount of friction (clamping) force *(F_c)* that can be generated using this V-shape depends on the diameter of the pipe and is given by [1]:

$$F_c = \frac{T}{r} = \frac{T}{\sqrt{(l^2 - (D_p - D_\omega)^2)}} \tag{2.1}$$

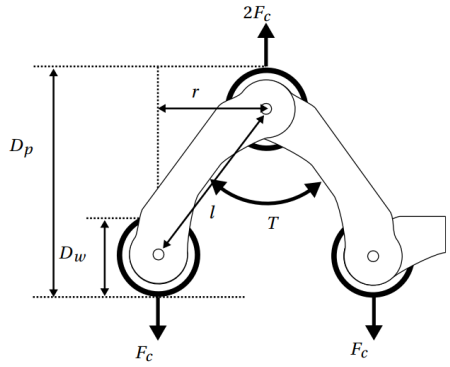


Figure 2.9: Geometric relations causing the clamping forces and torque. Figure from [1]

where l is the length of a module, D_ω is the wheel diameter, D_p the pipe diameter. The current version of the robot has a D_ω= 0.046m and the modules have a l=0.09m. Also for the present work, a standard pipe diameter D_p=0.09m was used, since it presented the average case scenario.

The bending module requires a gear train to transfer power from the self-locking worm gear via the spring to the joint in order to clamp [1]. An IPM (Ideal Physical Model) of the system (without controller) is given in Figure 2.10. The CAD design of the system is given in Figure 2.11. Note that each mechanism drives the geared edge on the next module.

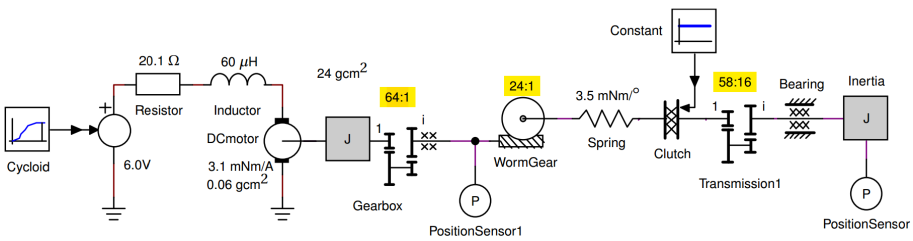


Figure 2.10: IPM of the bend drive.Figure from [1]

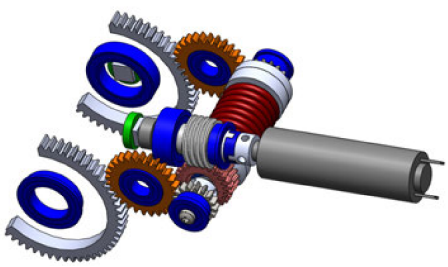


Figure 2.11: CAD drawing of complete bending gear system showing worm gear, spring and clutch. Figure from [1]

2.4.1.2 Rear Module

The rear module is the one that currently holds the connection from the exterior to the robot (using a tethered cable). This is possible through a standard Ethernet plug (RJ45 connector) that carries both communication and power buses. It contains also vision capabilities (the *rear camera*) that is used to obtain images from the pipe from the back point of view of the robot. This camera is fixed, so it has no pan or tilt capabilities. Currently this module is not actively controlled, but its position is measured by the same kind of sensors as the ones in the bend module. This module also has LEDs which provide illumination for the back part of the robot.

2.4.1.3 Front Module

The front module's function is to provide vision from what stands in front of the robot through a camera (*the front camera*), which can tilt and pan by itself to provide better vision for exploration. The front of this module has a semi-spherical plastic dome to protect the camera without taking the vision possibilities. For the pan and tilt of the camera motors along with a linear guidance are used, and can be controlled using position or PWM control. The front module itself is also capable of tilting, which can be controlled also by position or PWM mode, and which uses a Pololu motor (which currently has a gear ratio that makes it more "sensitive" than the bending modules). The same kind of sensors for angle measurement are used in this module. It also has LEDs which provide illumination for the front part of the robot.

2.4.1.4 Rotational Module

The rotation module is located in the middle of the robot. It consists of two parts which are connected by a Faulhaber 1516 motor with incremental encoder. The motor is positioned in a longitudinal direction and is able to rotate all front modules compared to the back modules, clockwise as well as anticlockwise. Normally in a rotation procedure, either the front (front and first two bend modules) or the back part (rear and last two bend modules) of the robot will be clamped. When commanded the part that offers the least resistance will rotate, so in this case the part that is unclamped (either rear or front) will rotate. The motor and encoder of the module are controlled using the attached motor controller circuit.

2.4.1.5 Wheels - Drive Mechanism

Wheels are attached between each module at the joint and at the end of the robot and the current design has 6 wheels in total (see Figure 2.7). In previous iterations of the design, it had only two driven wheels, whereas the current version, every wheel has a motor mounted inside - a so called *"in wheel drive"*. A coupling between motor shaft and wheel has to be implemented which decouples five of the degrees of freedom while connecting only one (traction). Figure 2.12 shows the wheel with decoupling, bearing, connector shaft and geared DC motor. The O-ring which is used as tire is not shown in the figure.

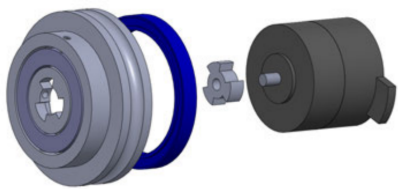


Figure 2.12: Exploded view of in wheel drive mechanism showing (from left to right) wheel with rotational decoupling, bearing, linear decoupling and motor - *image from* [1]

2.4.2 Electronics

The electronic system design uses a modular approach rather than a centralized system. This decision reduced considerably the amount of wiring and benefited the real-time capabilities of the control system. The robot has been controlled using a Master-slave node approach. The communication is possible through an *Arduino Mega*[7] board as *master*. This board is encapsulated in a box, where the connectors for power (type DC barrel power jack) and communication (RJ-45 that connects to the PIRATE robot, USB type-A that originally connected with the user interface [now is unused] and a USB type-B that connects to the laptop) are attached. The board for the Arduino Mega is referred to as *Pirate Bay* and it is depicted in Figure 2.13. The Pirate Bay also holds the connections for the rear and front cameras of the PIRATE robot.

[7] Arduino Mega board - https://www.arduino.cc/en/Main/arduinoBoardMega

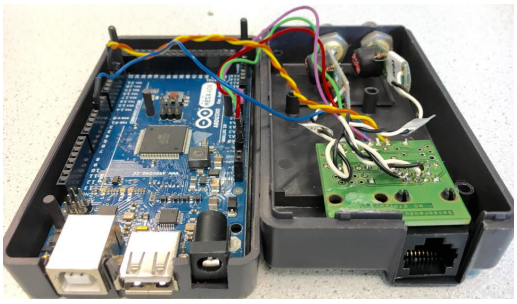
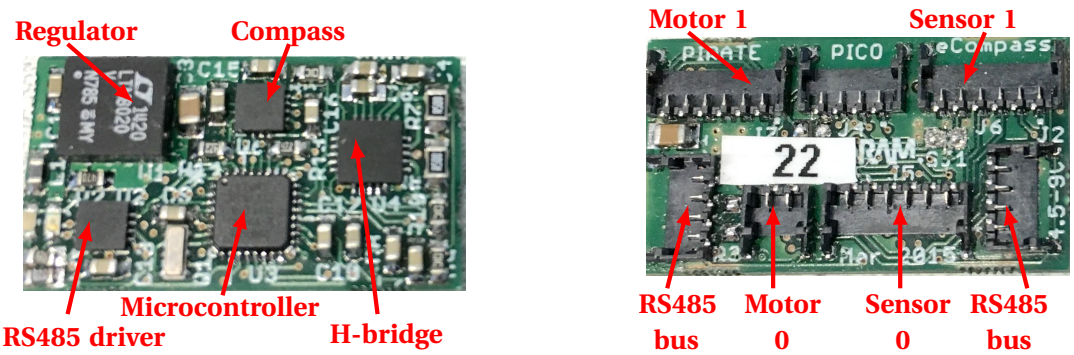


Figure 2.13: Pirate Bay

The modules mentioned in subsection 2.4.1, have one (or two in case of the rotational and front module) small board measuring 15 × 27 mm, shown in Figure 2.14. The board includes as microcontroller the *ATmega328p* [8], which is the most common controller on Arduino boards. Each of these boards are referred to as *PICO boards*. Apart from the microcontroller, the boards contain a H-bridge (model A3906), a regulator (model LTM8020), a RS485 transceiver (model LTC2850) and a compass (model FXOS8700CQ). The PICO board is shown in Figure 2.14. As it can be seen, each PICO board can control up to two motors and receive feedback from up to two connected sensors. In the back part of the PICO board, DF57 connectors were added. This choice was based on the fact that they are smaller and come with pre-crimped wires, which improves the reliability. Each of the PICO boards can connect up to two sensors and two motors and they provide the RS485 communication using a daisy chain structure (so each board has one input and one output connection) throughout the robot, which effectively reduces the wiring length and makes it easy to maintain.



(a) PICO Board front

(b) PICO Board back

Figure 2.14: PICO board -front side (a) and back side (b)

2.4.3 Control

The robot currently contains a total of nine PICO boards (one per bending module, one in the rear, two in the rotational one and two in the front). The boards are numbered starting from 20 (at the front) to 28 (in the rear module). The need for having two boards in the rotational one is because this module is composed of two parts and has control of two drive (wheel) motors and the rotation motor. Since each PICO board can control up to two motors, a second board had to be added. This added board was also connected to the IMU sensor. In the front module, the same situation applies, it controls the motor that tilts the module, but also two other motors for pan and tilt of the camera, which required again two PICO boards. The LED in the front is attached to the spare motor connection of one of these boards.

[8] ATmega328p board - http://www.atmel.com/devices/atmega328p.aspx

Depending on the function of the module (and excluding the rear module which is not actively controlled), the corresponding PICO board can be configured for different types of control. By default *all* the modules can be controlled using direct PWM to the motor. Table 2.1 shows the connections of the boards and the type of control they can have (for the module's names see Figure 2.7).

Table 2.1: Connections and type of control in the PICO boards

Slave No.	Located in Module	Motor 0		Motor 1		Sensor 0	Sensor 1
		Connection	Control	Connection	Control		
20	Front	Tilt camera	PWM/ Position	Pan camera	PWM/Position	Tilt camera	Pan Camera
21	Front	Tilt front	PWM/ Position	LEDs	PWM	Angle	N/A
22	Bend I	Bend	PWM / Torque	Drive	PWM/ Velocity	Angle	Spring
23	Bend II	Bend	PWM/ Torque	Drive	PWM/ Velocity	Angle	Spring
24	Rotational	N/A	N/A	Drive	PWM/ Velocity	IMU	N/A
25	Rotational	Rotate	Position	Drive	PWM/ Velocity	Angle	N/A
26	Bend III	Bend	PWM/ Torque	Drive	PWM/ Velocity	Angle	Spring
27	Bend IV	Bend	PWM/ Torque	Drive	PWM/ Velocity	Angle	Spring
28	Rear	N/A	N/A	LEDs	PWM	Angle	N/A

2.4.3.1 Torque control

As seen in Table 2.1, in the four bending modules torque control is implemented using the measurement of the spring deflection. An IPM of the drive train developed for 20sim simulation in [1] is shown in Figure 2.15. More recently, a PID control loop was implemented in the corresponding PICO boards using the spring deflection as controller feedback signal. Calculating the spring deflection is elaborate because the software needs to track the amount of revolutions in the gearing. In this case, the first gear stage in the motor (1:64) is never taken into account. The second stage (worm gear) is 24:1 and the third stage (from spring to the joint) is 58:16, which gives a total gearing of 87:1. The spring position is calculated in raw end-stage angle units (i.e. with a 87:1 ratio) which means that the spring constant felt is multiplied by $3.625^2 \times 3.168 = 41.6 mNm/deg$.

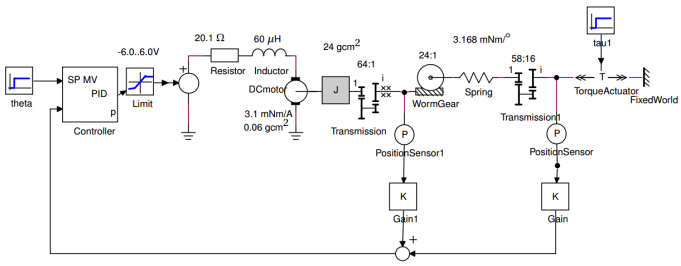


Figure 2.15: Schematic overview of the clamp control setup in 20sim. Figure from [1]

2.4.3.2 PWM

As we have seen from Table 2.1, the predominant option to move the motors is by using PWM, as this gives the flexibility to control the motor directly, which in some cases can be extremely useful. The case of the wheels (drive) is a good example. To move the drive motors PWM is preferred, as this will allow wheels to spin at different velocities. In reality PWM is actually feed forward velocity control, as voltage is directly related to speed (considering an ideal motor).In

the sampling routine (every 50Hz) the encoder position is differentiated in order to yield a measure of the velocity. The maximum setpoint that can be sent by the user interface is equivalent to approximately 170mm/s.

2.4.3.3 Position control

For the front module's tilt as well as for the front camera's tilt and pan position control is available. This is again implemented with a similar PID controller in the respective PICO boards. For this, the current spring angle's position is used as feedback.

2.4.4 Motion primitives

In [1], the concept of *motion primitives* is introduced. They refer to *the smallest functionally meaningful action that the PIRATE robot can perform.* According to this, each robot action can be broken down to a series of these motion primitives which are essentially: *clamp, unclamp, drive, bend* and *rotate.*

Also in the PhD thesis, these motion primitives are parameterized: *drive* has one parameter: the velocity. Whether wheels should turn clockwise or counter clockwise (depending on which side of the pipe they touch) can be determined based on the joint angle information. *clamp* has also one parameter: the desired clamping force on the pipe wall. For the V-shape both bending modules are used which get the same setpoint, only in opposite directions. The clamping torque required should be compensated for the current pipe diameter which can be measured with the module angles. rotate has one parameter: the angle ϕ between both sides of the robot. bend has also one parameter: the desired radius γ for a certain module to curve along a gradual or sharp bend. Table 2.2 shows these parameters.

Table 2.2: Motion primitives. *Table from [1]*

Motion primitive	Parameter
drive	ω [rad/s]
clamp	τ [N]
rotate	ϕ [rad]
bend	γ [rad]

It is important to mention that not all these motion primitives may be required or even that they might need to be more precisely defined into functions for the new software framework.

3 Analysis

In this chapter, the current system control architecture is discussed and the implications of moving to a more abstract layer are presented. Based on this, the requirements for the present work are listed.

3.1 Current system control architecture

As discussed in subsection 2.3.3, there are different layers for the embedded control software. Based on their description, it is possible to notice that the current state of the PIRATE robot is located in the so-called *Loop control* layer. This is given by the fact that the implementation of the control loop is done locally in the ATmega328p microcontrollers (PICO boards) in real-time. The setpoints to these controllers are sent every sampling period using the Arduino MEGA (Pirate Bay). The system works as follows: the user interfaces with the system via a fader panel. In this, setpoints for the different modules can be given and also control modes (i.e. torque, velocity, position or PWM mode depending on the type of module) can be changed. Additionally, there is a laptop used only for running a visualization in ROS (called Rviz[1]) which communicates via RS232. The control setup is shown in Figure 3.1.

In the current state, all functionality (motors, sensors) are readily accessible through an interface for direct control. However, this *manual* operation is now the whole potential of the current architecture. This, because the control is implemented in the Arduino MEGA microcontroller and there is a limitation in available memory and processing capabilities, would at some point be exceeded if more complex control was implemented in the same manner.

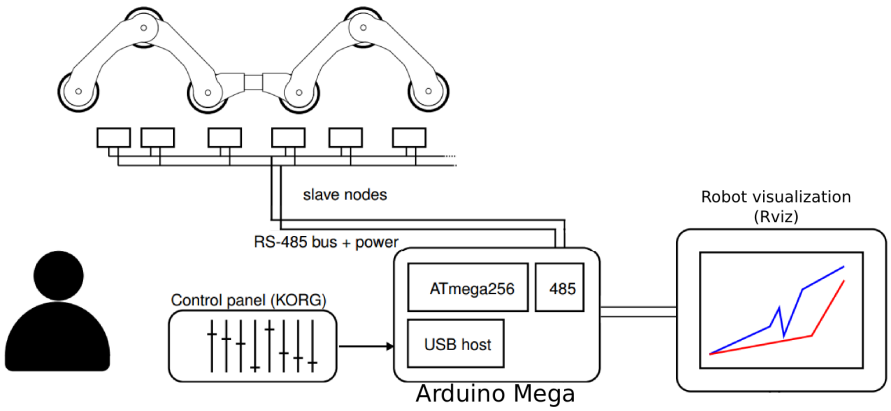


Figure 3.1: Schematic overview of control setup implementation. *Derivative of* [1]

3.2 Analysis of a layered software architecture for the PIRATE

The overall fully autonomous control functions desired for the PIRATE, can be mapped into the layered controller structure mentioned in Section 2.3.3.

This results in a structure as depicted in Figure 3.2. In it, it is important to notice that the *Safety Layer*, just as in shown in Figure 2.6, expands throughout all the software layers. This because each of them have their own safety methods to prevent damage to the robot. The loop control and its related safety are hard-real-time, because missed deadlines may result in unstable control actions for the joints or in a system failure. In this level there are two function blocks: *Joint control* and *State estimation*. Going towards more abstraction, the sequence control includes the so-called *Partially Autonomous Behaviors (PAB)* and *Robot state*. Partially Autonomous Be-

[1] ROS.org - rviz http://wiki.ros.org/rviz

haviors are sequential movements that execute by commanding the loop control and using the feedback that the Robot State gets from the joints and IMU sensor. In this stage, because the result has utility even after the deadline has passed, this is classified as soft-real-time. Finally, the supervisory control consists of the function blocks Mission Planning and World modeling. These functions do not influence the stability of the robot and therefore they are non-real-time. When implemented, this control layered structure would provide a *fully autonomous* behavior for the PIRATE.

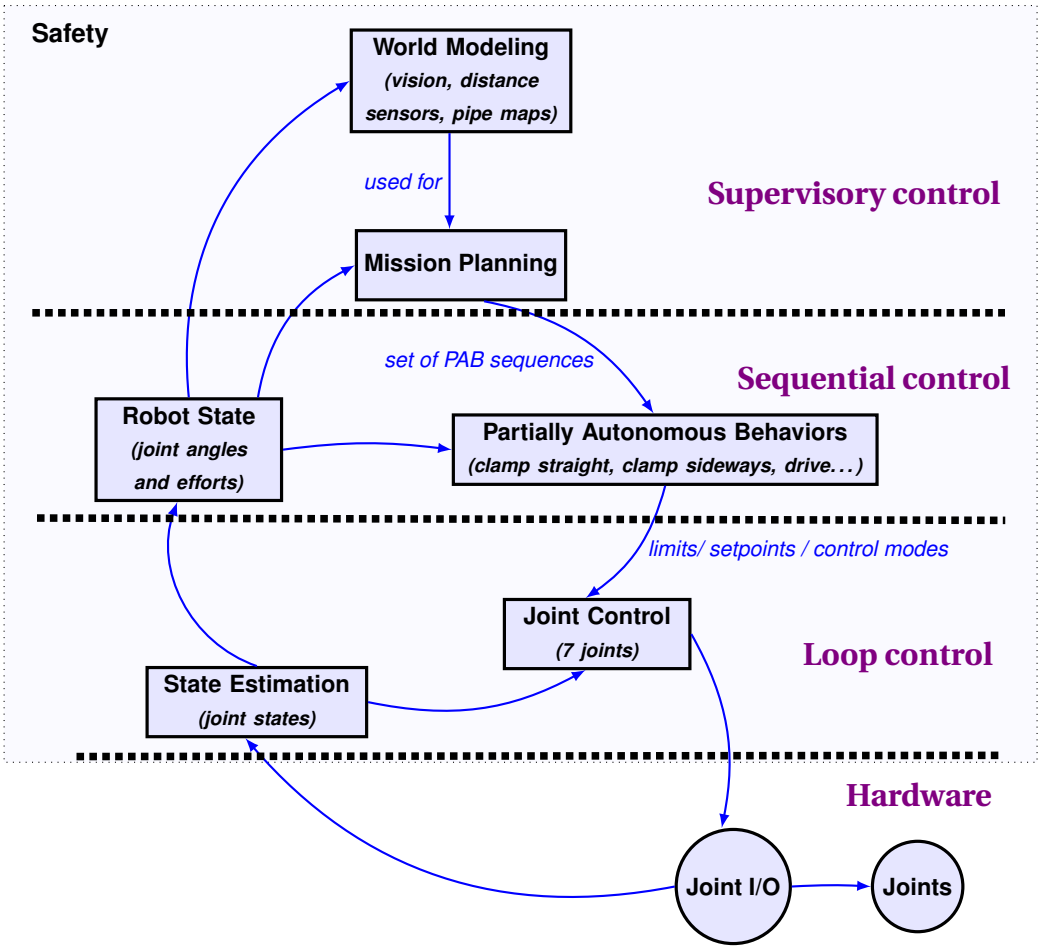


Figure 3.2: Functional structure for the PIRATE mapped onto the layered control software architecture Adapted from [30]

The analyzed possible structure consists of eight function blocks, which are described as:

- **World modeling**: The PIRATE is supposed to inspect pipes with different topologies and diameters. In order to successfully perform autonomously, it is required that the map of the inspected pipe is available on forehand, with the adequate level of detail such as: pipe junctions location, pipe diameters throughout the pipeline, etc. Information about the active environment can also be collected in this function block, by using vision or distance sensors, which in combination with the supplied world map will create an up-to-date model of the environment.
- **Mission Planning**: Consists of an AI planning algorithm, which normally takes two inputs: user input for the mission (the specific tasks that are to be completed) and the internal database of the pipe lines where the robot will be working in (world model). The output of this algorithm is called the *mission plan* which is a sequence of Partially Autonomous Behaviors which are executed successively. If an error occurs, the PAB level

will inform the Mission Planning, so that it can react accordingly by changing the world model and the PAB sequence to create a new plan.

- **Partially Autonomous Behaviors (PABs)**: In order for the PIRATE to be able to move and inspect the pipe lines, it needs to move in various ways and perform different tasks (e.g.: enter the pipe, and inspect the different paths). These movements are split up in simple movements (e.g.: clamp straight, drive, take joints), which are called partially autonomous behaviors. The name comes from the idea that they are sequences by themselves which by using feedback can perform small "autonomous" tasks. In the big scope of autonomy for the PIRATE, however, these are merely partially autonomous as not much planning is involved. To perform motions, the PABs should be able to manipulate the joint controllers by being able to send:
  - Control modes
  - Setpoints
  - Limits

  Based on the description of the tasks from Dertien's PhD thesis from 2014 [1], it is possible to define mainly six different partially autonomous behaviors in the PIRATE:

  (a) Clamp Straight
  (b) Clamp Sideways
  (c) Home (Unclamp)
  (d) Take T-joint
  (e) Take Mitre-bend
  (f) Drive
- **Robot State**: This function block collects the information provided by the sensors on the status of the robot and transforms it to standard units (angles in radians, orientation in degrees, etc.). This information can be requested by other function blocks.
- **Joint Control and State estimation**: These two blocks are closely related because together they form the PID loop controllers that are present in each of the PICO Boards. The Joint control part, sends the desired setpoints to the motor actuators and the State estimation gathers the information of the sensors and uses it as feedback for the PID control loop.
- **Joint I/O**: Each of the joints' sensors (inputs) and actuators (outputs).
- **Joints**: currently the PIRATE robot has 7 joints, which are referred to as *modules* and they are discussed in detail in Subsection 2.4.1.

Designing and implementing these software architecture for full autonomy is too ambitious for the time frame and scope of this thesis. Nonetheless, the present work proposes to take the first steps in the design of the framework that can allow the control software to advance towards the Sequential control layer. The considerations for this are analyzed in Section 3.3.

3.3 Considerations for moving towards the Sequential control layer

If more autonomy is desired for the robot, it is required, as mentioned before, to change the way the hardware elements interact with each other, as the Arduino MEGA shouldn't be the master of the system anymore. For this, a possible solution is to move the tasks that the Arduino MEGA currently does to a computer based system. The Robot Operating System (ROS) is a suitable framework for the implementation, because of the readily available packages it has, such as the ones for serial communication and the visualization part, from which this last is already implemented and would be reusable if the control is migrated.

As mentioned before, the developments that have been done from 2014 onwards have rendered a relatively stable system for manual commands. This is because a lot of effort has been put into the communication routines between the PICO boards and the Arduino MEGA. For this

reason, it is possible to contemplate that the Arduino MEGA can act as a transparent bridge between the PICO boards and the new computer-based master, while still performing these robust communication routines.

In this way the computer can further become an intelligent master controller, which generates the setpoints based on sensory information and stored mission data. The transparent bridge will send these signals to the local motor control boards as well retrieve the sensor's data it receives from a the internal serial bus on the robot to send feedback to the host computer.

Currently the Arduino MEGA's task is to listen to the user interface inputs and generate setpoints for the PICO boards, accordingly. Since this is a relatively easy task, migration to a ROS node should not present complications. However, when more autonomous behaviors are desired, it implicates a necessary distinction of the different modes of operation that should be present in the PIRATE. Examples of these modes could be *Manual* and *Partially Autonomous Behaviors*, (*PABs*).

Having a system with different modes of operations has several implications on defining the correct interaction between them. The most important property of such a system, is the fact that only one of the modes should be active at a time, and the implementation should keep careful track of which is the current mode.

Another situation to be taken into account when switching modes, is the fact that the PIRATE has a camera in the front which can be adjusted in pan and tilt position. Furthermore, there are also LEDS both in the front and the rear modules. Since these functionalities wouldn't, in principle, be automated for now, and the fact that the user might need them at any time, it would be desired that they remain manually controllable during the execution of the more autonomous operations. This could potentially be facilitated by separating the execution of the manual and PAB operation modes into two ROS nodes, which are independent and can run concurrently. In this way, if a PAB mode is active it could, for example, restrict the access of the manual mode to all the necessary modules, but the ones that control the camera and the LEDs.

Another important fact for correct mode interaction is that the coherence between the messages (setpoint, control, etc.) when switching the operation modes should be assured, especially if the modes are implemented in separate programs (nodes). This means that, for example, if the last operation mode was PAB and the last setpoint sent to one of the modules was equal to 256, then if the mode is switched to manual, this setpoint remains until it is manually modified, thus it should not be restarted (this is undesired because, for example, sending a 0 setpoint would cause the robot to move).

To protect the integrity of the robot, one of the most crucial necessities when dealing with a partially autonomous sequential program is the possibility to cancel the execution at any time, which should also be considered in the design of the system.

Even though the system will not reach the Supervisory layer directly, the programmed sequences shall be able to cope with non-nominal situations. Such events can happen because of the rough environment in which the robot must work. An example of this would be that if the robot falls on its side, it should be able to recuperate the desired position.

3.4 Requirements

The basic requirements for the present work's development are derived from the above discussion and are differentiated into technical (related to the physical system and the environment) and functional (related to the desired behavior).

3.4.1 Technical requirements

1. The system must be able to interface with the current hardware.
2. The system shall be running in a laptop for transportability.
3. The design shall be modular and standard, to promote re-usability.
4. The design shall be scalable such that more autonomous levels can be implemented in the future.

3.4.2 Functional requirements

1. The robot will have two operation modes and can only be in one at the time: manual or PAB (where partially autonomous operations can be requested).
2. Partially autonomous behaviors will be performed upon operator request (i.e. pressing a button in the user interface) and only when the robot is in PAB mode.
3. Operator can stop the execution of a manual command or cancel partially autonomous behavior at any time.
4. Operator can manually maneuver the front camera and the LEDs (front and rear) at all time in any of the current operation modes (manual or PAB).
5. During PAB mode, all manual action to the joints will be disabled (excluding the ones mentioned in 6). In order to regain manual control of the robot the PAB command must be first canceled and the robot mode switched from PAB to manual.
6. If the sequence requires a change in control mode, it will be done automatically, so no operator intervention is needed.

3.5 Design choices

As part of the analysis discussion presented earlier in this chapter, a number of design choices have been made. They are listed as follows:

1. *The laptop will act as a master to all ATmega microcontrollers (slaves)*: Due to the processing and memory capabilities, the Arduino Mega is no longer a good choice for being the central master of the system. Because of this, a laptop is chosen as a more appropriate hardware to be the master control.
2. *ROS will be chosen as the framework development environment*: In Reiling's implementation, ROS has already been used to process the feedback from the state of the PIRATE. It has shown promising results and its readily available packages and functionality for robotics present a suitable option for the development of the software framework.
3. *Arduino MEGA will be a transparent bridge between the laptop-based control and the ATmegas*: As discussed earlier, the communication between the Arduino mega and the slaves has proven to be robust. Because of this the Arduino Mega will stay as a transparent interface (bridge).

3.6 Project scope and limitations

As discussed in Section 3.2, reaching full autonomy falls greatly out of the scope for this thesis. Because of this, the present project will develop the software framework for the Sequential control layer. In this layer, there are also some limitations based on the available time. Within these we can mention:

1. Only one of the PABs must be fully designed and implemented. This means that this behavior should take into account some non-ideal scenarios to cope with. The selection has been made for the *Clamp Straight* behavior.
2. Other PABs could be *basically* designed and implemented. This means that they will not consider possible non-ideal scenarios. These will be the *Drive* and *Home* behaviors, chosen to complete the mission mentioned in Section 1.4.

4 Software Framework Design

This chapter describes the steps followed to obtain a competent software framework that integrates all the requirements defined previously in order for the general software architecture to reach the Sequential control layer. First of all, an overview of the general design made is shown. After this, each of its parts is discussed in detail, along with the reasoning behind the design choices.

4.1 General framework design

The general software architecture is composed of different classes which perform specific functions. The considerations taken to define the required classes, were based on the desired goals and requirements.

The main functions that must be covered are:

1. Communicate ROS with user interface (read/write).
2. Translate user interface commands to instructions (setpoints, control modes or limits) for the robot.
3. Assure coherent and centralized message sharing among the different parts of the software.
4. Send the translated instructions (commands) from ROS to the robot.
5. Create and manage the control modes between manual and PAB (Partially Autonomous Behaviors).
6. Implement sequences for partially autonomous behaviors (PABs).
7. Receive and handle in ROS the current robot state.

The software framework and the overall designed classes are shown in Fig. 4.1.

In Figure 4.1, which is a UML-like diagram, we can distinguish three *packages*, which indicate the overall function of the classes inside of it. The packages are: *Pirate Control, Pirate Robot Interfaces* and *User Interface*. The first one, as the name suggests, contains classes that provide the control functionality for the robot. The *Limit, Setpoint* and *Control* classes, will be collectively referred to as *message-related* classes because they contain the corresponding messages (according to their name) to be sent to the robot, as well as the required functions for accessing and modifying them. The *Pirate Manager* shares an unilateral association with the *Limit, Setpoint* and *Control* classes with a multiplicity of one, which means that only one object of each class is created. This is because the *Pirate Manager* exists as a *centralizing class* which will allow consistent message sharing (function (3) of the list). The relationship is based on the fact that a reference of each of the message-related classes will be passed to the constructor of the *Pirate Manager*. This centralizing class is, in turn, composed of other classes, namely the *Mapper* and *Pirate Server*. When the objects for both these classes are constructed in the *Pirate Manager*, they are also passed a reference of each of the message-related classes (the association of both these classes to each the message-related classes is omitted from Figure 4.1 for visibility). The *Mapper* and *Pirate Server* classes are in general a client-server structure that translates the user interface commands into manual or partially autonomous instructions to the robot, which means that they are performing functions (2), (4), (5) and (6) of the aforementioned list. Being that the *Pirate Server* is in charge of the PAB execution, its task was divided into smaller classes. Currently, the *Pirate Server*, includes the *Motion Primitives* class, which comprises the most basic functionalities that the robot can perform and the *Clamp Straight* class, whose main function is to execute this sequence. The *Clamp Straight* class has a unilateral association with the *Motion Primitives* class, because it will make use of its member func-

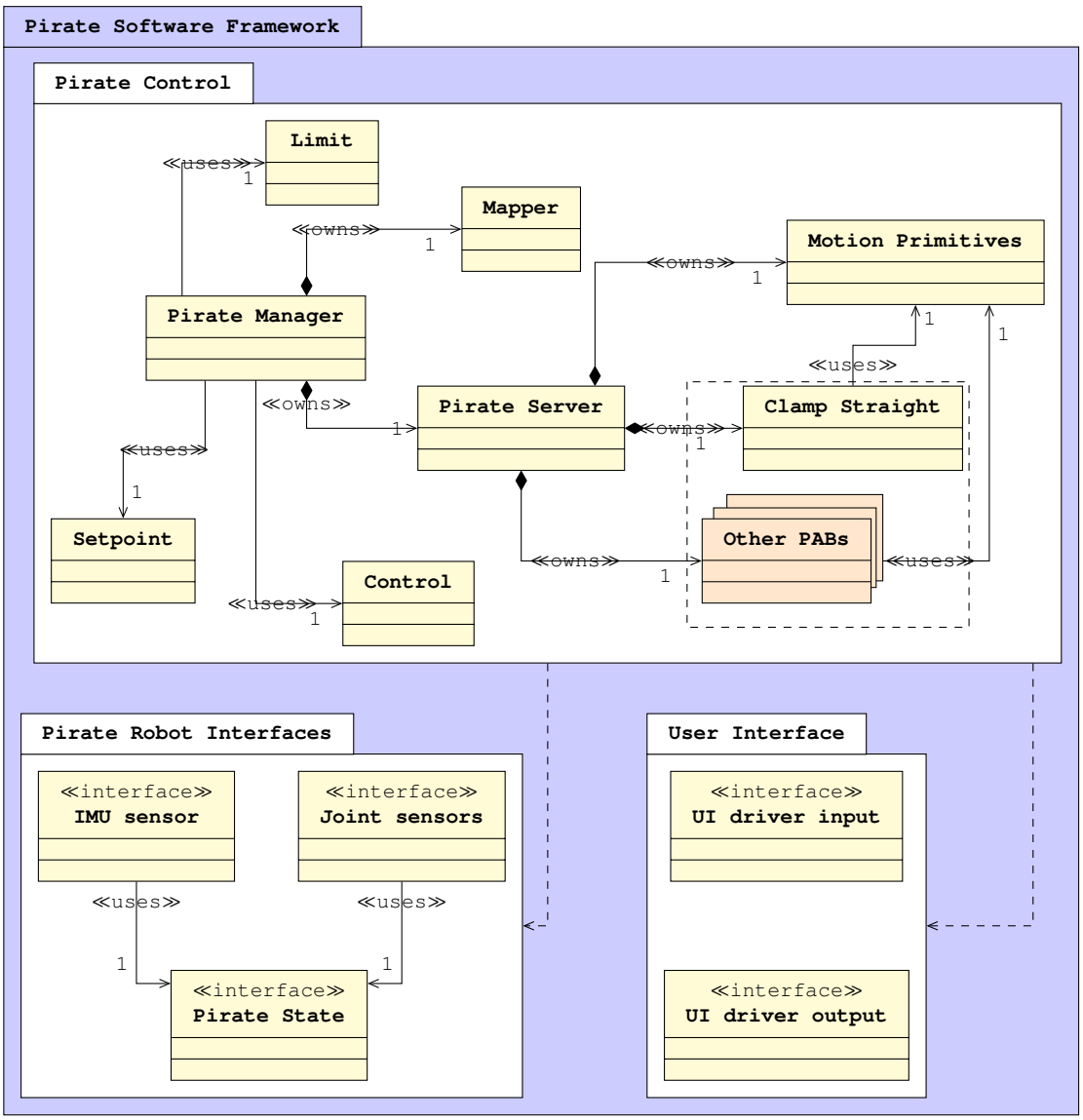


Figure 4.1: UML-like diagram showing Pirate Software Framework

tions for its own sequence execution, which can be considered as made up of several motion primitives. Separating the functionality into individual classes for each PAB, provides modularity and makes the addition of other PABs a matter of implementing another individual class for each, and just making minor changes to the *Pirate Server* and *Mapper* classes. This can be observed in the additional class *Other PABs* in Figure 4.1. This class is there only to represent the fact that in the future more PAB-like classes could be easily added to the current structure. As mentioned in Subsection 3.6, from the other PAB classes, basic design will be done for the *Drive* and *Home*, as part of the demonstrable mission. These will be briefly described in Subsubsection 4.7.4.2, and in Figure 4.1 can be considered to be inside the Other PABs classes. Both the *Clamp Straight* (and all other PAB) and *Motion Primitives* classes require for their construction an reference of the message-related classes as well (the association of both these classes to each the message-related classes is again omitted from Figure 4.1 for visibility).

The second package, called *Pirate Robot Interfaces*, contains three interfaces that are used to retrieve and publish the robot state. The *Pirate State* class is the one that gets the information from the robot and publishes it in ROS (function (7) from the list). The classes *IMU sensor* and *Joint sensors* have each a unilateral association with the *Pirate State* class, because they use the

information that it retrieves and distribute it to other classes in the Pirate Control package. The final package *User Interface*, makes communication with the user interface (function 1 from our list) possible through two classes which in Figure 4.1 can be seen as *UI driver output* and *UI driver input*.

Each class of the diagram presented in Figure 4.1 will be described in detail in the upcoming sections, starting with the message-related classes.

4.2 Setpoint class

As mentioned before, the robot contains a total of nine PICO boards which have the capability to connect up to two motors each. Because of this, it is required to first design a setpoint container message which the ROS node can publish to the Arudino Mega. One option was to make individual messages per board, indicating which motor the setpoint was meant for. However, when doing more autonomous behaviors, it would be more useful to be able to change more than one setpoint at a time. Furthermore, another consideration is the amount of data to be transferred between the computer and the Arduino Mega, which should be kept to a minimal to avoid communication issues. This resulted in deciding to create a setpoint array message, that contains only the necessary setpoints to be sent to the robot. Because of the possible connectivity with the actuators, the total required setpoints was found to be thirteen (for the six wheels' velocity only one setpoint is required, the other five are the same and can be copied in the Arudino Mega, thus $(2 \times 9) - 5 = 13$, ending up with a 13-element array). This array will be implemented as a so called custom ROS message, which we call *Pirate Setpoint Array*.

To allow other classes to interact with this message, the *Setpoint* class was developed. This class creates an instance of type *Pirate Setpoint Array*, and contains functions to set and get individual elements or the entire array, as well as to publish them. The full design of the *Setpoint* class is presented in Figure A.1 in the Appendix A.

4.2.1 Pirate Setpoint Array

As mentioned in Section 4.2, to send the setpoints to the robot, an array that contains thirteen elements was created. This array which represent the setpoints for each of the nine modules with the connection of the first motor in the PICO Boards. The second motor connection for six of the PICO Boards is used for the wheels, which will have a unified velocity and therefore only one setpoint is required. For the slaves (PICO boards) 20, 21 and 28 there are connections for the camera pan and both rear and front LEDs. The motor connections in the PICO boards are called *Motor 0* and *Motor 1* (for more information refer to Figure 2.14), so this terminology is used for the ROS software as well. Table 4.1 shows the mapping of the Pirate Setpoint Array.

Table 4.1: Pirate Setpoint Array mapping

Pirate Setpoint Array													
Motor	**Motor 0**									**Motor 1**			
Array Element number	0	1	2	3	4	5	6	7	8	9	10	11	12
PICO board ID	20	21	22	23	24	25	26	27	28	20	21	22	28
Physical element	Front Camera Tilt	Front mod.	Bend mod. I	Bend mod. II	IMU mod.	Rotation mod.	Bend mod. III	Bend mod. IV	Rear mod.	Front Camera Pan	Front LED	Wheels	Rear LED

4.3 Control class

Similarly to the Setpoint class, the Control class serves to create the custom message type that will include the control modes for the different PICO boards of the robot. Also because this class will provide the connection from other classes to this custom message, it has functions to get and set the individual elements or the whole array, as well as to publish them. The design of the Control class is depicted in Figure A.2 in the Appendix A.

4.3.1 Pirate Control Array

The *Pirate Control Array* is the custom message defined to contain all the different control modes of the PIRATE. The mapping is similarly defined as the one for the *Pirate Setpoint Array*. The difference lies in the size of the array, which is smaller (only twelve elements instead of thirteen) because the LEDs do not have control modes. The design decision was taken to keep the order of the elements as similar as possible with the mapping defined in Table 4.1, this meant that for the front LED (element 10 in the array), the element will be there but not used, and only the rear LED (element 12) will be eliminated from the array. This was done so that the element 11 which refers to the wheels, coincided with the one defined for the *Pirate Setpoint Array* message.

Table 4.2 shows the mapping of the Pirate Control Array.

Table 4.2: Pirate Control Array mapping

Pirate Control Array												
Motor	**Motor 0**									**Motor 1**		
Array Element number	0	1	2	3	4	5	6	7	8	9	10	11
PICO board ID	20	21	22	23	24	25	26	27	28	20	21	22
Physical element	Front Camera Tilt	Front mod.	Bend mod. I	Bend mod. II	IMU mod.	Rotation mod.	Bend mod. III	Bend mod. IV	Rear mod.	Front Camera Pan	Not used	Wheel drive

4.4 Limit class

Finally, one last message-related class was constructed. The Limit class creates the instance of the *Pirate Limit Array* message and provides functions to modify or retrieve its elements, such as set, get and publish. The class diagram is very similar to the ones provided for the other message classes and is presented in Figure A.3 from the Appendix A.

4.4.1 Pirate Limit Array

The *Pirate Limit Array* is the smallest of the custom messages designed for interaction with the robot. The reason for this is that currently only the bend modules have limits for the torque values they can reach. This means that this array shall contain only one element per bend module, so four in total. For this reason, this time the mapping had to be modified with respect to the standard followed for the *Pirate Setpoint Array* and *Pirate Control Array* messages, since to keep it would have involved an unnecessarily large quantity of elements that would not be used. Finally, the design of the *Pirate Limit Array* custom message contains only four elements and is ordered ascending according to the numbering of the bend modules.

Table 4.3: Pirate Control Array mapping

Pirate Limit Array				
Motor	**Motor 0**			
Array Element number	0	1	2	3
PICO board ID	22	23	26	27
Physical element	Bend mod. I	Bend mod. II	Bend mod. III	Bend mod. IV

4.5 Pirate Manager class

This is the centralized class that *manages* the other nested classes' creation. The reason for the existence of such a class is based on the constraint that the messages which would be sent to the robot required unification throughout the different levels of software as well as consistency even when changed from manual to PAB mode and viceversa. To obtain this desired centralization, a main container class was implemented. This approach provides a layered class structure where the manager creates an instance of the next level of classes and so on (i.e. composition). In the main node where the instances of the Setpoint, Control and Limit classes are created, the instance of the Pirate Manager class is generated too. The constructor of this class, takes as input a reference to each of the message-related classes. This references are later distributed throughout all the nested classes that are created within the Pirate Manager. At the next nesting level, it is possible to see in Figure 4.1 that inside the Pirate Manager two classes are created: the Mapper and the Pirate Server. The Pirate Manager class is shown in Figure A.4 from the Appendix A.

4.6 Mapper Class

The Mapper class has the dual function of handling the manual mode commands as well as, in case of Partially Autonomous mode, sending out PAB instructions to Pirate Server. The constructor of this class takes references to the message-related classes, namely Setpoint, Control and Limit. This is done so that it has access to all the public functions to set, get and publish the messages. From the *client-server* model between the Mapper and the Pirate Server, the first one serves the purpose of being the client, thus it requests the resources of the Pirate Server for the PAB execution.

Inside the Mapper class (for reference see Figure A.5 in the Appendix A), the client is created and the subscribers and publishers for the user interface topics will be generated as well (to publish to UI driver output and subscribe to UI driver input). The Mapper is able to send commands to change the status of the LEDs in the user interface. This is used, for example, to light on the button when the operation mode is PAB and turn it off when it is Manual it is directly visible, and also to distinguish when the control mode of the modules is PWM/Torque/Position (depending on the module type), etc. Furthermore, the goal variable is declared, along with the current operating mode variables, which will be used to control the permissions for each of the modes. Finally, the *map* member function will check if there are new setpoints, control modes or limits sent by the user, and if this is the case, it will update the message-related class references. Outside this, in the publisher function, the publishing commands to the message-related classes will be sent. In the case of the control messages, because we want immediate action for when they change, they are published as they are received inside the map function. The situation is different for the setpoints and limits, due to the fact that they are inputted from sliders and knobs in the user interface, which generate a lot of unnecessary intermediate values. Because of this and to avoid saturating the communication with the Arduino, these type of messages will only be published every $10Hz$, which is a reasonable rate considering that the robust communication between the Arduino Mega and the robot, has an approximately similar rate, so there is no need to be publishing faster than that.

Inside the map function, some of the inputs of the user interface will be used to send goals to the Pirate Server, when in PAB mode. Because of this, the Mapper has the possibility to request the status of the server and to store it in a variable (activeGoal), which is checked before sending a goal. If there is an active goal the Mapper will notify the user that it is not possible to send a new goal until the current one is completed or canceled. In this way, the program makes sure that if the user wants to send another goal, the current one must be canceled first.

4.7 Pirate Server class

The *Pirate Server* is in charge of handling (executing, aborting or preempting) the commands that the Mapper sends. Given the fact that it has to deal with very diverse partially autonomous behaviors (PABs) execution, its architecture is complex. This resulted in a composite class that uses the services of child classes, namely the *Motion Primitives* class and a class for each of the PABs that are required. In our case, only one of the PABs will be implemented, only one other class, the *Clamp Straight*, will be a child to the Pirate Server. More details on the Pirate Server class can be found Figure A.6 of the Appendix A. This composite class structure, provides the flexibility of adding more PABs by creating separate classes as desired. If this happens, only minor changes are required in the Pirate Server (to handle the request of the new PAB itself) and in the Mapper (to link the user interface command to send a new goal according to the new PAB).

It is important to notice that, apart from the child classes, once again the message-related classes (Setpoint, Control and Limit) are present as references. Inside the Pirate Server, the corresponding goal and its feedback, result and status variables will be created, which will be used to exchange information with the Mapper. These messages will be discussed in length in Subsection 4.7.1.

In general, the Pirate Server has its own internal state machine, where the states are mapped to a structure called *state_* (for reference see Figure A.6). This state machine keeps track of PAB being executed. The internal state machine of the Pirate Server can be seen in Figure 4.2.

Here we see the six main PABs (in green) that have been discussed in Section 1.3. The PAB represent, of course, temporal or transition states, which are active for a determined amount of time (for example, the time it takes to clamp straight, etc). When those states finish, they reach a so-called *idle* state (in white), where the robot doesn't produce any action, but rather waits until another command is received.

As previously expressed, the relationship between the Mapper and the Pirate Server class, is based on the *client-server model*. In order to communicate, the Mapper and the Server should share messages that contain the relevant information about the action that will be executed. In the subsection 4.7.1, the messages exchanged between the Mapper and the Pirate Server will be described in more detail.

4.7.1 Client-Server messages

In order for the server to start a PAB, the Mapper should send a message that contains which PAB is desired along some additional parameters. Because all of this information relates to the *desired action*, it will be referred to as the *goal*. Furthermore, according to the requirements mentioned in Section 3.4, it is necessary also that the Mapper can send a message to *cancel* the current goal on the Pirate Server. On the other hand, the server should send information back to the client, such as a *result* message, which should be sent at the end of a goal, a *feedback* message, which, if required, could supply the client with intermediate status information on the joint state of the robot, and finally a *status* message, which would be more general than the previous one, just presenting the current status of the goal itself (different from the joint information). The client-server model, is presented in Figure 4.3.

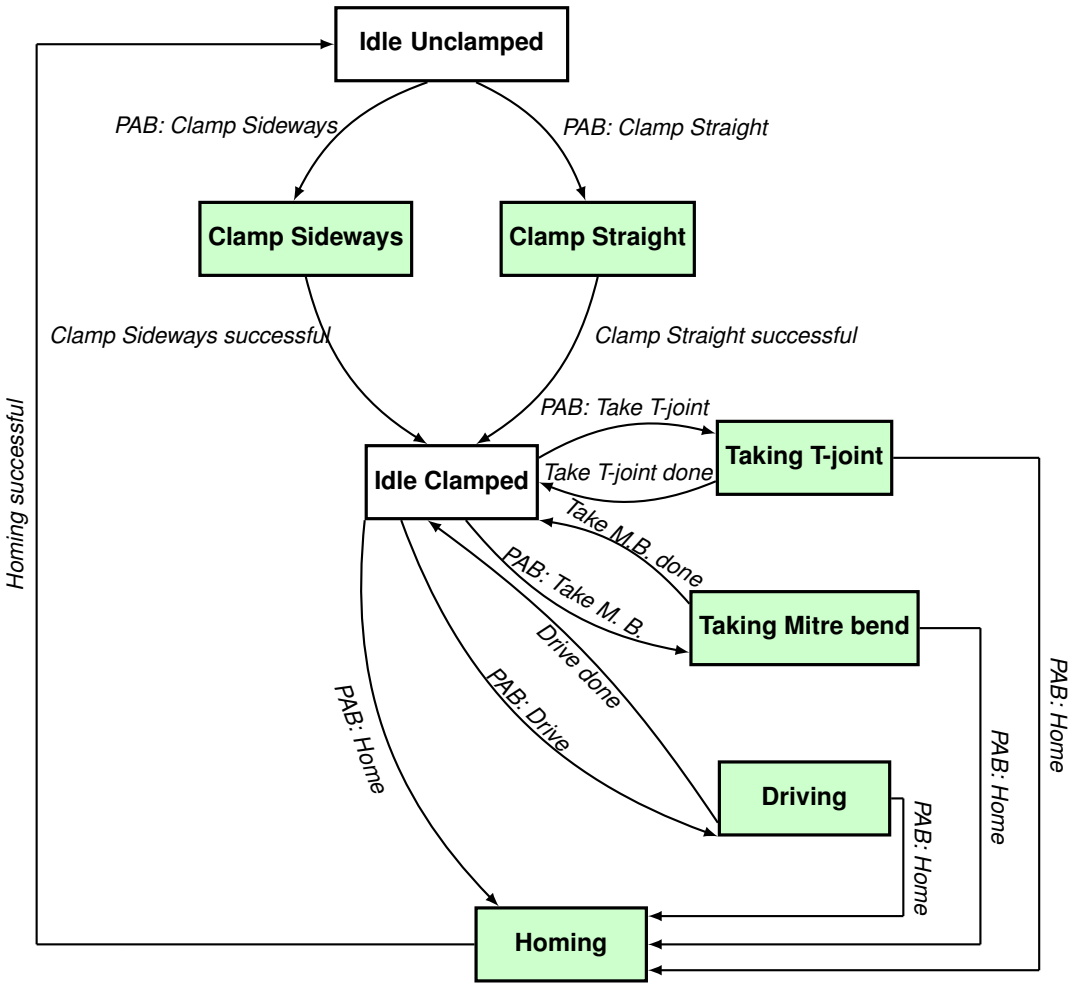


Figure 4.2: Internal state-machine of the Pirate Server

4.7.1.1 Goal definition

The goal has crucial information that the Mapper and Pirate Server require to function. It is basically a custom message (structure) that contains the relevant fields for the server to execute the action. In our case, the definition has to be more on the side of what kind of action it is, and some additional parameters to identify the possible options for each action. Since, in ROS, the goal is defined in the same way as a custom message, a structure is created that contains the PAB's name, the diameter of the pipe, and, depending on the PAB, additional parameters.

The description of the goal's fields is as follows:

1. *name*: This will be used to uniquely identify the commanded PAB. Possible values are listed in Table 4.4.
2. *pipe diameter*: This number indicates the diameter of the pipe in which the requested PAB will be carried out. The diameter of the pipe is necessary for all the PABs, because it renders important information for the *guards* (conditions) that are needed during the sequence execution.
3. *distance*: This is an optional parameter to complement for the *Drive* PAB. This represents the distance to drive and the sign defines the direction (positive- forward, negative- backward).
4. *additional parameters*: The design contemplated that this field can be used for the PABs that require more information than the pipe diameter to execute. The foreseen values for this field, depending on the PAB, are presented in Table 4.4.

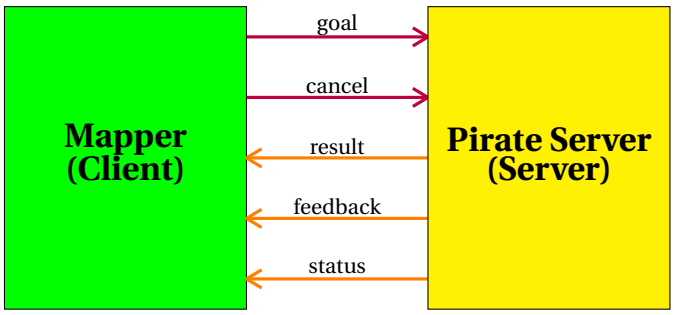


Figure 4.3: Client-Server model and the interaction messages.

The design of the goals for each of the possible partially autonomous behaviors is described in Table 4.4.

Table 4.4: Goal messages for each PAB, with corresponding possible options

Client-Server - Goal definition						
PAB	**Clamp straight**	**Clamp sideways**	**Home**	**Take T-joint**	**Take Mitre Bend**	**Drive**
name	CLAMP STRAIGHT	CLAMP SIDEWAYS	HOME	T_JOINT	MITRE_BEND	DRIVE
pipe diameter (mm)	90/110*	90/110*	N/A	90/110*	90/110*	90/110*
distance	N/A	N/A	N/A	N/A	N/A	Range should be determined when implementing and depending on the test piping. Must be a signed value, because direction will be indicated with a sign.
additional options	N/A	(Suggested) Direction: Right/Left	N/A	(Suggested) Direction: Right/Left /Straight	(Suggested) Direction: Right/Left	N/A

** There are more diameters that the PIRATE can work with, but for the scope of this thesis, only the two most common diameters are defined.*

It is worth noting that from the PABs in Table 4.4, only the Clamp Straight will be implemented. However, the proposed suggestions might be a good place to start future implementations.

4.7.1.2 Result, feedback and status definition

The messages for the result, status and feedback are simpler than the goal because the information is sent as structures, and thus each of these messages contains only one structure.

Feedback: could provide the server with an asynchronous way to send periodic updates to the client during the processing of a goal. Sending feedback is completely optional. In the present design, the handling of the sensor's feedback is done by either the Motion Primitives or the PAB classes. Because of this, the feedback message is currently not used.

Status: provides the client with status information about the goal that is currently being tracked by the server. The client uses this information to know if the goal is still active, aborted, etc. In this way, it can prevent the user from sending another goal while one is currently being processed (which is explained in detail in Section 4.6).

A goal is tracked by the server until it reaches a terminal state. The status message has one field:

- *goal state*: contains the current goal's status. The possible intermediate states are: active, pending and preempting (canceling). The terminal states are: rejected, succeeded, aborted and preempted (canceled).

Result: provides the server with a way to send information to the clients once a goal has been completed. Thus, a result must be sent on a transition to any terminal state. The result message is given by:

- *success*: is a boolean variable that returns true when the terminal state (refer to message *status*) of the server is succeeded, and false when it terminated in any other state.

4.7.2 Motion Primitives class

This class is one of the child classes from the composite *Pirate Server* class. According to the definition given in Section 2.4.4, its objective is providing the PAB classes with basic, modular functions that can be used repetitively throughout the sequences. To construct the instance of this class, the Pirate Server passes the reference it has of the message-related classes (Setpoint, Limit and Control). Furthermore, the Motion Primitives class makes use of one of the two interfaces that have the robot state, which is *Joint sensors*. The need for this is given by the fact that some of the motion primitives' functions are performing very small and modular sequences that, sometimes, require feedback from the robot. The design of the motion primitives class can be seen in Figure A.7 of the Appendix A.

Basically the types of motion primitives functions are defined as:
Basic functions - *These do not make use of the publish function of the message-related classes. This has to be done separately.*

- **Move Module**: This function will be used to send a setpoint to either the front module or any of the bend modules. To do so, it will write the desired setpoint using the functions in the Setpoint class.
- **Change Control Mode**: This functions allows to change the control mode (PWM, position, velocity, torque, etc.) to one of the modules of the PIRATE. It works by writing the desired control mode using the functions from the Control class.
- **Drive at velocity**: this enables to send a velocity setpoint to the wheels of the PIRATE. The function requires the desired velocity in mm/s and it checks if the requested velocity is smaller than the maximum permitted, which is $170mm/s$. The conversion to the counts-setpoint will be calculated inside the function and this will be written using the functions of the Setpoint class.

Advanced functions - *They integrate the use of the publish function of the message-related classes, as well the evaluation of so called guard conditions. This means that the return value of the function, a boolean, will indicate whether or not it was successfully completed.*
More information about these functions will be given starting from Subsubsection 4.7.2.2.

4.7.2.1 Feedback for the advanced functions

The clamp and unclamp motion primitives work using torque control. Because of this it made sense that the guards were the efforts (torques) in each of the bend modules. In practice, however, this was found to be not reliable. During handling of the robot, very different torques would be obtained, including considerable differences between measurements and even changes in the sign of the effort. Those effects could possibly be due to the spring deflections, the deformation of the wheels, play between the mechanical parts and the (possible) compliance of the prototype. All of this affects the torque control, so if it is desired to use it as guard conditions, the torque controller implementation should be improved for robustness. This, nonetheless, falls out of the scope of the present thesis. For this reason, it was found that angle feedback presented more reliability and repeatability, therefore it was chosen to use these positions as the guard conditions for both clamp and unclamp procedures. For the rotate and align functions, angle feedback was used.

4.7.2.2 Rotate

This motion primitive function allows to send a setpoint to the rotational module of the PIRATE. The designed function requires as input the desired rotation in $radians$. Inside the function it evaluates whether the requested value is within the limits, which are in the range of $-0.7rad$ to $+0.7rad$ from one part of the rotational module with respect to the other part. The conversion to the counts-based setpoint will be calculated inside the function and this will be written using the functions of the Setpoint class.

As mentioned in Sub-subsection 2.4.1.4, the rotational module is formed by two parts connected by a motor, and which are capable of rotating with respect to each other. For the explanation of the rotation, we will divide the robot into two parts: *front* and *rear* (not to be confused with front and rear modules of the robot). These parts are shown in Figure 4.4.

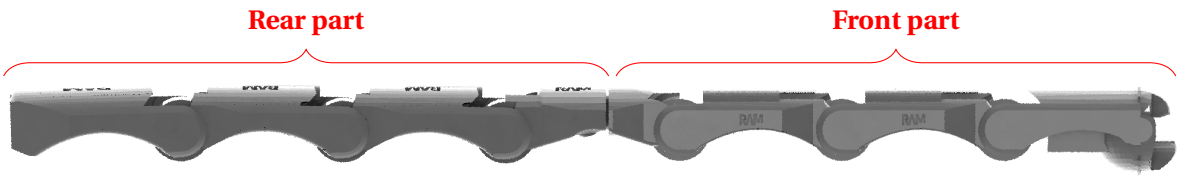


Figure 4.4: Division of the PIRATE from the rotational module's perspective

Normally, for the rotation procedure, either the front or the rear part of the robot will be clamped. This means that, for example, when the front part is clamped, the rear rotates and vice versa. It is important to notice that the sign in the setpoint of the rotation will indicate the direction (clockwise or counterclockwise) of the module. This is, however, dependent on whether it is the front or the rear part of the rotational module, the one that it is rotating. Table 4.5 shows the direction of the rotation with respect to the setpoint sign.

Table 4.5: Rotation direction according to setpoint sign

Sign of setpoint	***Direction of rotation***	
	Front clamped	***Rear clamped***
Negative	Clockwise	Counterclockwise
Positive	Counterclockwise	Clockwise

As we can see in Table 4.5, the directions the same sign produces are opposite depending on the part of the robot that will be rotating. The presented direction is taken from a front point of view as depicted in Figure 4.5.

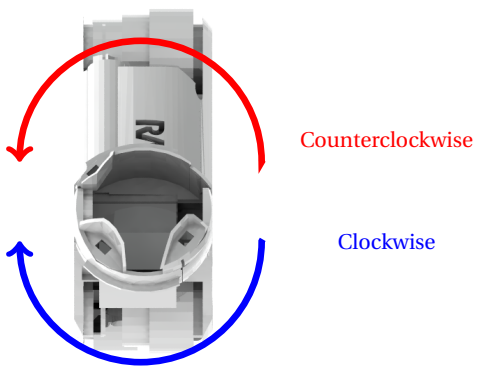


Figure 4.5: Rotation directions from front point of view

Since the rotate motion primitive is an advanced function, apart from publishing the setpoint, it includes the evaluation of whether or not the desired condition (guard) has been reached. The angle feedback of the rotational module is coming from a relative encoder, which at start-up is reset to zero value. This will be known as *aligned* condition and it will be further explained in Sub-subsection 4.7.2.6. This makes possible to check if the rotation has been completed by actually checking whether or not the encoder's feedback is equal (within an experimental tolerance to be determined) to the desired rotation setpoint in radians.

4.7.2.3 Clamp rear / front parts

As mentioned in Sub-subsection 2.4.1.1, independent on the orientation of the robot, the configuration for clamping involves two of the bending modules forming a V-shape, which when in contact with the pipe wall creates the so-called *clamping force*. In the Motion Primitives, we will define as *clamp front* the one formed between the Bend Modules I and II, and *clamp rear* as the one formed between Bend Modules III and IV.

The clamp rear or front motion primitive considers that the involved bend modules are in torque control mode. Then, according to the supplied pipe diameter argument, it sends the necessary setpoints to be able to clamp (either the rear or the front part of the robot). For this, the definitions of rear and front part of the PIRATE (from Figure 4.4) are applied again. Figure 4.6, shows the result of the clamp rear and clamp front motion primitives.

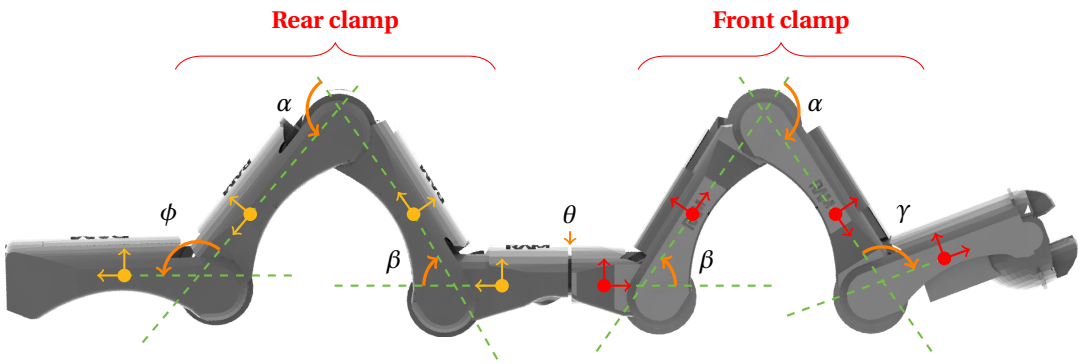


Figure 4.6: Illustration of both *clamp front* and *clamp rear* as well as all the angles used in the PIRATE

The function returns success only if the robot state indicates it has been clamped. To know this, the angles of each of the bend modules are evaluated, as shown in Figure 4.6. Once these angles (referred to as *guards*) are reached ($angle_{BendIV,I} \approx \alpha$ and $angle_{BendII,III} \approx \beta$), the function sends a zero setpoint to the Bend Module III or Bend Module II in the case of rear and front clamp, respectively. This is done to release some of the torque in these modules. If this weren't done, the torque would make it very hard (and sometimes impossible) to clamp or unclamp the remaining part of the PIRATE. Another important detail that the sequence takes into account is moving the front module (represented by angle γ). The rotational angle might be used in case orientation adjustment is required and is represented in Figure 4.6 as θ.

4.7.2.4 Custom clamp of rear / front parts

The services provided in the custom functions are similar to the ones described in Sub-subsection 4.7.2.3, with the difference that the guards to consider a successful clamping procedure can be customized and thus are part of the input arguments of the function. The reason for the creation of such customizable functions lies in the fact that they will be used when a *full clamp* cannot be performed. We refer to as *full clamp* as the one that can be performed in positions straight or sideways, i.e. in the points where the robot can reach the full diameter of the pipe (i.e. where previous guards defined as α and β can be reached). For the non-ideal

situations, where the robot will have to get up from a side position, sometimes this full clamp cannot be completed, and thus different guards are needed.

4.7.2.5 Unclamp and partial unclamp of rear / front parts

Unclamping can be thought of as the opposite action of clamping, which basically means that instead of moving the bend modules so that they form a V-shape, they are moved towards a position where they form a straight line. There ar two variations for the unclamp which are:

1. *Full*: The outcome of this function is that the concerned modules (Bend I and II or Bend III and IV), independent of the robot orientation are in a complete straight line. Because of this, the guards will be equivalent to having $\alpha \approx 0rad$ and $\beta \approx 0rad$.
2. *Partial*: The procedure is similar to the full one, however the guards in this case are greater than $0rad$, which means that the robot is unclamped, but the modules are not forming a straight line but rather a very open V-shape.

In the same manner, the guards for the successful completion of these routines consist on checking both α and β angles to be consistent with the desired value. This condition will not have to be changed depending on the pipe diameter, because it is designed for one of the smallest ones, 90mm. Because of this, in the other diameters the same guard will be more than enough.

4.7.2.6 Align rear and front part

This function serves the purpose of counteracting the effects generated by the rotate function. In this last one, we look to generate a relative angle between the front and rear part of the robot (θ). In the align routine, this relative angle is brought close to zero radians (within a tolerance to be defined in the implementation). This function is required because only the rear part of the robot has the IMU sensor information, so without alignment it is a good way to locate and "home" the front part.

4.7.3 Error handling - Timeout for Motion Primitives Advanced functions

Finally, all the advanced motion primitive functions, being that they continuously evaluate whether or not their condition for success has been reached, include a *timeout*, which serves as an error handling method. The functions receive a configurable timeout in seconds. If at any point the sequence is not completed before the timeout, this will trigger an error and a message for the user so that it can check the robot. Normally if the timeout is reached, it indicates that the robot has met an obstacle (or presents a mechanical failure, but this is less likely). At the currently developed stage of autonomy, and for the scope of this thesis, only this type of error handling will be considered. In the future, such a response can be sent to a more abstract control software layer (Supervisory) that can evaluate the error and act accordingly.

4.7.4 Partially Autonomous Behaviors (PABs) classes

As it has been mentioned, the software framework will integrate programmed sequences that are able to perform basic tasks of the PIRATE. These basic tasks have to be carefully delimited and are referred to as *Partially Autonomous Behaviors*. For the purposes of this report they are defined as:

an action that performs a specific simple functionality of the robot. It consists of a sequence of motion primitives (simplest motion commands) and is able to cope with the most basic types of disturbances for each. By itself, it does not make the robot autonomous but together constitutes the base for more complex (fully) autonomous behavior.

Keeping this definition in mind, and based on [1], the basic partially autonomous behaviors for the PIRATE are defined (according to the orientation shown in Figure 4.7) as:

1. **Clamp Straight**: Movement of the robot where two V-shapes (upside down) are formed between modules Bend III and IV and Bend I and II, respectively. The orientation of the robot shall be $\pi/2$ radians from the base of the pipe (bottom). For reference see Figure 4.7.
2. **Clamp Sideways**: Just as the other clamp this involves two V-shapes (upside down) formed between modules Bend III and IV and Bend I and II, respectively. The difference lays in the orientation of the robot which could be either 0 radians (sideways-right) or π radians (sideways-left).For reference see Figure 4.7.
3. **Drive**: This action allows the robot to navigate inside a pipe. The precondition to this is that the robot must be clamped (in any of the possible forms) for better stability. As a proposition, to use this PAB the user must supply a desired distance in mm to drive. The distance value should include a sign to indicate the direction of the drive (negative sign - drive backwards).
4. **Take Mitre-Bend** (for short referred as Mitre-bend): it is one of the complex PABs. It requires a long sequence where the robot must start clamped, then detect when the front part loses traction (entering the T-joint), unclamp and rotate according to the desired direction to take and reclamp the front part of the robot (Bend I and II). This process has to be repeated so that the rear part of the robot (after the rotational module) pass the bend in the same fashion. Depending on the direction desired (left or right) and the orientation of the bend itself, the clamping required beforehand could be straight or sideways (left or right).
5. **Take T-joint** (for short referred as T-joint): Has the same description as the mitre-bend. Nonetheless, compared to the Mitre bend, an added complexity for the sequence is the open side of the T-joint with which no contact is possible.
6. **Home (unclamp)**: this routine shall bring the PIRATE to a resting position on the bottom of the pipe. The orientation of the robot in this case will be $\pi/2$ radians.

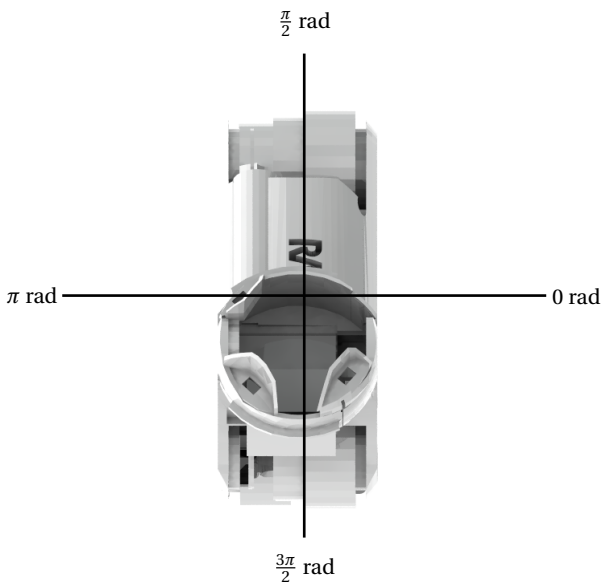


Figure 4.7: General orientation of the PIRATE in radians used in the design

Due to the time constraint for this project only one of the partially autonomous behaviors shall be fully designed and implemented in software using a state machine architecture, which will be the *clamp straight*. This should yield a proof of concept of the developed software frame-

work, which should be designed such that other routines for PABs can be implemented without the need to do major modifications to the developed software.

4.7.4.1 Clamp straight class

The clamp straight class is part of the PAB-related classes. It includes the state-machine sequence for the robot to clamp-straight per the definition given in Section 4.7.4. This class, once again and according to our structure, gets a reference from the message-related classes (Setpoint, Control and Limit). The difference now is that it also gets a reference from the Motion Primitives class. The sequence makes use of the motion primitives' functions (presented in Subsection 4.7.2) inside the *execute* member function, because this is where the state machine sequence lies. The class also gets information (i.e. is a client) from the two interfaces *Joint sensors* and *IMU sensor*. This because for the correct execution of the sequence, the feedback of both the robot's joints and the orientation sensor are required.

The execution of the clamp straight routine is managed by a state machine. The states take the motion primitives' functions and check some additional parameters, which are related to the robot's orientation.

In the state machine it is possible to distinguish two main scenarios:

- **Nominal**: This case is defined as when the robot is found with an orientation of around $\pi/2$ (range to be experimentally defined), according to Figure 4.7.
- **Non-Nominal**: This can be considered the *non-ideal* case for the Clamp Straight (and possibly for some of the other PABs). It is defined as when the robot's orientation is outside of the *nominal* range. These ranges will be experimentally defined in Chapter 5.

As it is possible to see in the state machine depicted in Figure 4.8, the states are basically the motion primitives mentioned in Subsection 4.7.2. More details about the transition conditions can be found on Table D.1 in the Appendix D.

Figure 4.8: Internal state-machine of the Clamp Straight class

It is possible to see in the state machine from Figure 4.8, that a very important step to do before trying to clamp the robot's front part is to move the front module upwards (lift it). This is due to the fact that when clamping the modules Bend I and II, the front one is also moved and if it is not lifted it will be forced and possibly damaged.

4.7.4.2 Other PAB classes

The name PAB-classes refers to the other classes that are capable of executing a state machine for each of the Partially Autonomous behaviors. In the scope of this thesis, the base is developed for their design and implementation. Because already the design of one of the PABs is presented in this thesis, the design of the other PAB classes should follow the same structure. Create a class that receives in the constructor a reference to the message-related classes (Setpoint, Control and Limit) and also to the Motion Primitives class. This will allow to use the methods from this classes which already contain communication and basic movements for the PIRATE. Also each of the PAB classes should contain a public member function *execute*, where the designed state machine can be implemented. The complexity of the sequence depends directly on the complexity of the PAB itself, so perhaps in more complex classes such as taking a T-joint or a Mitre bend, additional functions will need to be designed. All in all, to develop other PAB classes, the basics are already present in the current development. Therefore, it is a matter of designing the required state machine and integrating it to the Mapper and Pirate Server classes so that it is possible to call the *execute* member function for the corresponding PAB-class. From this other PAB-classes, two have been partially designed with just the basics

to show the possibilities of missions as a result of the present thesis. They are the *Drive* and the *Home* and both will be shortly described next.

4.7.4.2.1 Home class

Per the definition, homing is the activity to unclamp the robot and leave it in a 0 radians orientation. The home class was developed to complement the Clamp Straight function and show the potential of implementing other PAB classes and executing them sequentially for a mission. This, however, was only done basically and in no way is a comprehensive design of this behavior. This design can perform a simple state machine that can unclamp the PIRATE *assuming* that is clamped at a straight position. The design of the class is based on the recommendations from Subsubsection 4.7.4.2. Inside the execute member function lays the state machine for the basic behavior of the homing. The class is shown in Figure A.9 from the Appendix A. The basic state-machine developed only to test can be seen in Figure 4.9.

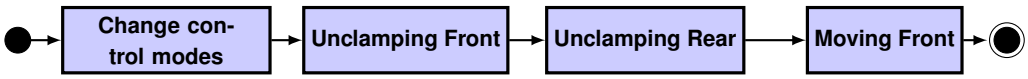


Figure 4.9: Basic Internal state-machine of the Home class *assuming* that the PIRATE starts in a straight position

As it can be seen in Figure 4.9, the sequence is very basic and linear. In the future, proper design should be done so that this function has the capability of detecting what the current orientation is and take the decisions to home the robot based on that.

4.7.4.2.2 Drive

As mentioned before, driving is the behavior that allows the robot to move throughout a pipe in both forward and backward directions. The pre-condition as seen in Figure 4.2 is that the robot is clamped. The class design also follows the standard of the other PAB-classes and is shown in Figure A.10 from the Appendix A.

For the basic development this PAB only has one state: Driving. For future (complete) design, possible non-ideal situations should be taken into account (possible obstacle or getting a bit stuck), so this sequence can become more robust. This was made simple, because it is part of the demonstration mission, but a complete design is out of scope for this thesis.

4.8 Interfaces

The interfaces are designed based on the need to retrieve feedback information from the robot. Basically there is a main interface class that obtains the status of all the sensors of the robot called *Pirate state*. From this state the information is passed to other two interface classes, depending on the type of information, namely one for the joint status called *Joint sensors* and other for the orientation information of the robot called *IMU sensor*.

4.8.1 Pirate state interface

The Pirate State interface is the main connection from ROS to the robot's hardware through the Arduino Mega (Pirate Bay board). The goal of this interface is to retrieve the robot's feedback (efforts, angles, velocity, odometry, etc.) for all the joints as well as the IMU sensor. The logic behind this node was already implemented by Reiling, prior to the start of this thesis, however, it was not using the publish-subscribe model from ROS. In the present design, the data exchange is designed based on the ROS publishing and subscribing capabilities. In this way the Pirate state class will subscribe to the topic published by the Arduino Mega. The details for the implementation are discussed in Chapter 5.

4.8.1.1 Pirate Small State message

Depending on the type of module that the PICO board is attached to, there are variations with respect to which kind of feedback is obtained and sent to the *Pirate state* interface class. Table B.1 shows the feedback obtained from the Arduino Mega per PICO board.

As it is possible to see, the longest message, from PICO board 25, contains 9 elements. The reason for this is that this board contains the message of the IMU sensor which serves in our design to obtain the orientation of the robot. Because of this, a message was designed so that the Arduino Mega could publish the feedback in ROS and the *Pirate state* class could subscribe to it. It is important to mention that this was chosen to be sent per each PICO board and not as a "full" state, because of the limits of the amount of information exchange in the Arduino side. The design from Reiling (mentioned earlier) suggested that this was an efficient way to send the feedback from the robot.

This message to be exchanged as feedback per PICO board was named *Pirate Small State* and contains an array of 9 elements. Details of the mapping of the Pirate State per PICO Board can be found in Table B.1 from the Appendix B.

4.8.2 IMU and Joint sensor interfaces

The design of these two interfaces is based on the *blackboard design pattern*, which is used in software engineering, to coordinate separate, disparate systems that need to work together, or in sequence. The blackboard consists of a number of "global variables", like a repository of messages, in our case the robot state (namely IMU and Joint data), which can be accessed by separate processes, such as the Motion Primitives class or the Clamp Straight (or any other PAB). This was chosen over the traditional *publish-subscriber* approach from ROS, because even though the robot status will be constantly received from the Arduino Mega, it is not necessary that it notifies the PAB or the Motion Primitives class every time this happens. This means that they do not require to be subscribed to all the feedback received, but rather *request* for it when they are performing a specific sequence and they need the sensor data to validate if the guard has been achieved. Figure 4.10 shows the communication between the IMU and Joint blackboards and the client classes.

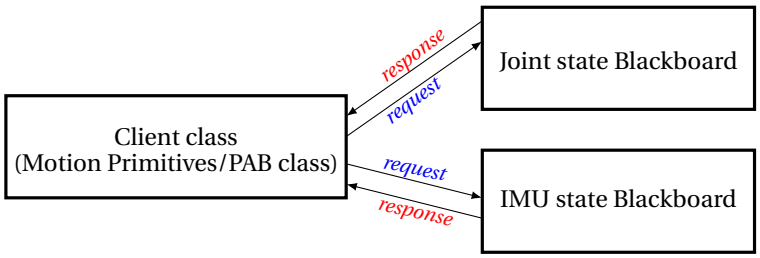


Figure 4.10: Blackboard-like design for the IMU and Joint Sensors classes

4.8.3 User interface Drivers

The user interface drivers are two small ROS nodes that serve the purpose of communicating ROS with the user interface. In our case the selected interface device is a MIDI fader panel. It was chosen because of its native communication capabilities via USB and of the intuitive with which several modules can be commanded at a time.

In general, there are two kinds of drivers present in the design:

- *Input driver:* has the function of retrieving the inputs of the user interface and publishing them to ROS.

- *Output driver:* has the function of subscribing to a the topic that sends the LED status for the user interface (found in the Mapper class) and sending it via USB to the user interface device.

4.8.3.1 User interface input and output messages

To make possible the communication through ROS topics, two simple messages were designed to send the input and output to the user interface. They are called *MidiInput* and *MidiOutput*. The elements of the messages are shown in Table 4.6.

Table 4.6: Midi Input and Output Messages

MidiInput	***MidiOutput***
channel ID	button ID
value	boolean value

As it is possible to see, the input message has the channel ID which is a unique number that identifies the element of the fader panel that was activated and in this case it's corresponding value. The output message only serves for the illumination of the LEDs so it only takes into account the button's ID (not all the channels) and the corresponding value is a boolean (on/off) value.

5 Implementation

This chapter covers the implementation of the design described in Chapter 4. Based on this, first, we will introduce and describe the ROS packages or tools used for the implementations of the communication, client-server models and blackboard design patterns. Next, the design's functionality will be described based on the basic use cases. After that, more detail on the implementation will be given of the most relevant classes, namely the Mapper, Pirate Server, Motion Primitives and Clamp Straight and Pirate State classes. Finally, pertinent information about the implementation of a simple test mission scenario, will be given.

5.1 Serial communication between Arduino Mega and ROS

As stated before the communication from ROS to the PIRATE will be implemented using the Arduino Mega as a transparent bridge. This microcontroller has 4 UARTs (hardware serial ports), from which one is currently used to establish the communication to/from the PICO boards. The development from Reiling, previous to the start of this thesis, had used one of the serial ports to send the feedback information to ROS via serial commands. This implementation, however, did not make use of the publishing/subscribing capabilities of ROS and the readily available resources for implementing serial communication with Arduino microcontroller as client. Furthermore, in the original development by Reiling, the Arduino Mega was in direct communication with the user interface. Because this was no longer the case (see Section 1.4), it was necessary to implement a way to send the setpoints, control modes and limits to the Arduino Mega via serial port. In the present work, the *rosserial* package was used to replace the existing communication for the feedback (using the designed *Small State* message described in Sub-subsection 4.8.1.1) and also to implement the ROS-side of the communication. Basically *rosserial* is a protocol for wrapping standard ROS serialized messages, topics and services over a character device such as a serial port or network socket.

The implementation of rosserial requires two parts:

- **Serial Client:** The client libraries allow to easily get ROS nodes up and running on various systems. Among the readily available supported libraries it is possible to find one for Arduino microcontrollers (called *rosserial_arduino* [1]). In this way, the library is downloaded and installed to the Arduino development software and allows to use ROS publishers and subscribers inside the microcontroller's code.
- **Serial Server / ROS-side interface:** Is a node in the host machine that allows devices running rosserial code to bridge the connection from the serial protocol to the more general ROS network. ROS already provides fully functional host nodes for the rosserial code. The one selected for our development is called *rosserial_python* [2], which is the most recommended for PC usage.

Figure 5.1 shows the message exchange between the ROS network and the Arduino Mega using rosserial. It is possible to see that the message-related classes serve as publishers for the control, setpoint and limit messages for which the Arduino is a subscriber. Moreover, the microcontroller publishes the Small state message which was discussed in Sub-subsection 4.8.1.1 and that contains the feedback state of the robot per PICO Board ID.

[1] ROS.org - rosserial_arduino http://wiki.ros.org/rosserial_arduino
[2] ROS.org - rosserial_python http://wiki.ros.org/rosserial_python

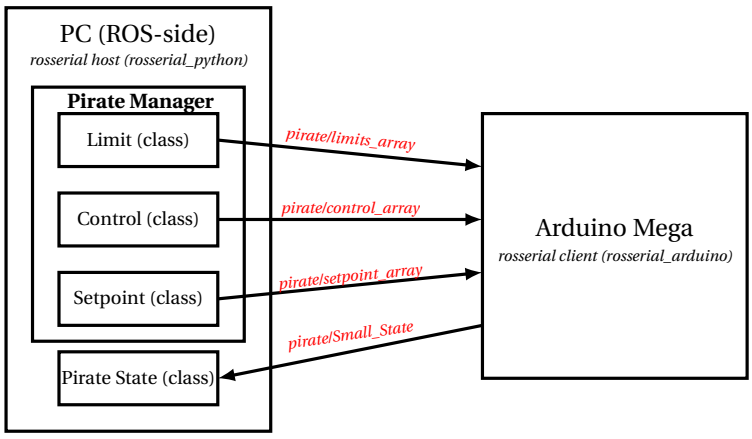


Figure 5.1: Communication using rosserial between Arduino Mega and ROS

5.2 Actionlib from ROS

The actionlib [3] stack provides a standardized interface for interfacing with preemptable tasks in ROS. A preemptable task is one that can be canceled at any time, as per our requirements for the PAB sequences.

The actionlib package provides tools to create the client-server model that we have designed as the Mapper and the Pirate Server classes. The actionlib is composed of:

1. *Action server*: executes long-running goals that can be preempted. Since it is created in a separate node, it is therefore thread-independent and thus non-blocking to the client.
2. *Action client*: provides an interface to send requests to the server.

Their implementation of the Action Server and Action Client might be tailored made using the full functionality of the actionlib package, however ROS does provide simple templates to use, called *Simple Action Server* and *Simple Action Client*. For the development presented in this thesis, the level of descriptiveness provided by these simple templates was enough for our desired functionality.

Basically the client is in charge of sending goal or cancel requests, and the server, on the other hand, handles the goal execution, providing the client with status, feedback and result. As it is possible to see, the definition of the actionlib package fits perfectly to our necessities and requirements, so it was used for the implementation of both the Mapper and Server communication structure.

5.3 ROS Services

For the implementation of the blackboard design pattern for the IMU and Joint sensor classes we will use ROS services. The reason not to use the publisher/subscriber scheme of ROS was discussed in Subsection 4.8.2, but it is mainly based in the fact that the Motion Primitives and the Clamp Straight classes need to get the robot state information at specific time during the sequence, which means that a request/reply model is more appropriate. This type of communication pattern is done via a *service*, which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction as if it were a remote procedure call. When doing the implementation of the system, the necessity raised for several classes, namely the Clamp Straight class and the Motion Primitives class, to be able to *request* for the server status, to know if it had been preempted. The easiest way to achieve this was to include a service server inside the Pirate Server

[3] ROS.org - Actionlib Detailed Description http://wiki.ros.org/actionlib/DetailedDescription

class. In this way, the both Clamp Straight and Motion Primitives can periodically ask during their loops for the status of the server. If it has been preempted they will stop their functions, as it is expected after cancellation of a goal.

5.3.1 IMU and Joint service messages

Like topics, services have an associated service type that depends on the desired type of request and response. In all of our cases, the request can be empty since the response of the service does not depend on any particular argument to be supplied. Furthermore the response of the service can vary depending on the required feedback. Table 5.1 shows the information contained in the response of the implemented services.

Table 5.1: Type of responses for the implemented services

Service	***Respose***
IMU sensor	Roll, pitch and yaw
Joint sensors	Position and effort of each joint (module)
Pirate Server status	Active / Cancelled

5.4 User interface drivers

For the user interface two nodes were implemented in Python using the online available resource called *pygame.midi*[4]. One node is for communication from the current user interface (i.e. a MIDI fader panel - *input driver*) to ROS and the other to deliver messages from ROS to the panel (in this case, to turn on the LEDs of the buttons as feedback - *output driver*).

5.5 Use Cases

As mentioned in Subsection 2.2.2, use cases are a good way of showing the functionality of the software system. In Figure 5.2 the Main Use Case (Pirate) diagram, which describes the most relevant functions for this development, is shown.

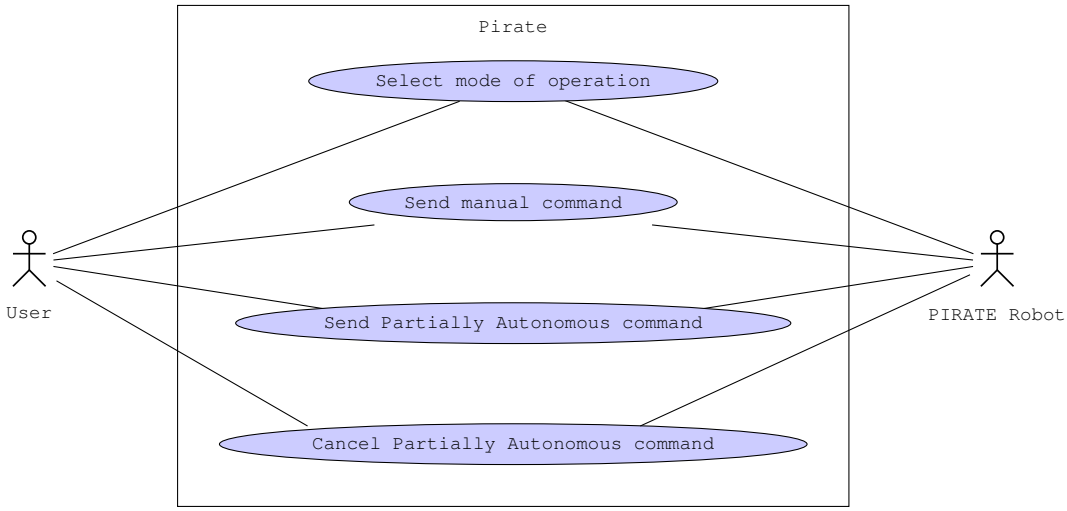


Figure 5.2: Main Use Case Diagram for the PIRATE.

It is possible to see that the main use case is composed of four use cases, which show the main functionalities of the PIRATE. Each of these will be briefly described below.

[4]pygame Midi - Documentation http://www.pygame.org/docs/ref/midi.html

5.5.1 Select mode of operation

A robot can operate in different ways depending on the application. The simplest way to define an operation mode is as:

a way or manner in which the robot does a certain action and how it interacts with the user

In our case, the present work develops two possible modes of operation for the PIRATE:

1. *Manual mode*: Mode in which the operator must trigger and intervene at each instant in order for the robot to execute an action.
2. *Partially Autonomous mode*: Mode in which the user has the possibility to trigger (pre-programmed) partially autonomous behaviors (PABs). It is considered partially (and not fully) autonomous because, for this robot, there is still possibility to build more abstract software levels that can handle more complex mission-planned routines as well as cope with more diversity of disturbances.

The precondition for this use case consists on the robot being in *"stop"* state (i.e. when the user interface is stopped). Furthermore the default start-up condition for the robot is manual operation. In order to change the mode of operation of the robot one of the unused buttons of the MIDI panel has been chosen. The selected button is the *REWIND*, as it can be seen in Table E.1 in Appendix E. According to the implementation, if the button is pressed once it will change the mode from PAB to Manual, this will turn on the LED in the button, as an indication of the fact that the system is in PAB mode. If the user presses the button once more, the mode will be reverted to manual, and the LED will turn off. This was designed so that in the future, when other available buttons are required to trigger the PAB sequences, there is availability. As complement to this information, Figure 5.3 shows a diagram for this use case.

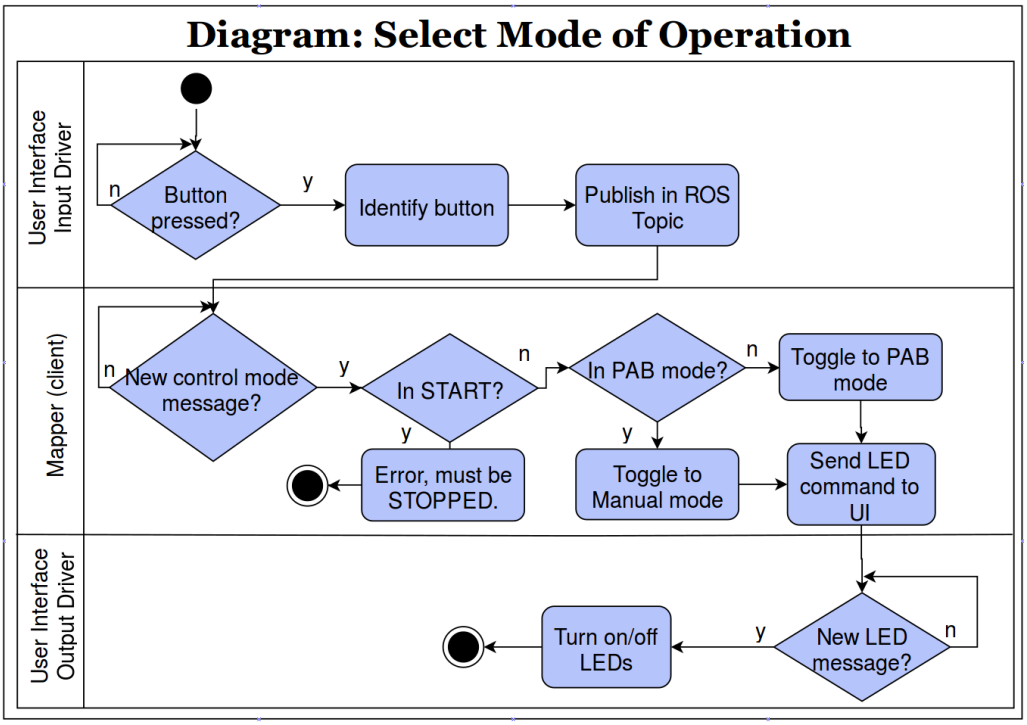


Figure 5.3: Diagram for "Select Mode of Operation" use case.

The change of operation mode is also shown as feedback in the main ROS screen, which can be found in Figure C.1 from Appendix C.

5.5.2 Send manual command

This use is executed when the system is in Manual operation mode. The user interface (currently MIDI panel) has the mapping for all the different modules, limits, control modes, cameras, LEDs, etc. This mapping to the MIDI panel is shown in the Appendix E. According to the value inputted in the UI, the *Mapper* will generate the setpoints and send them to the Arduino MEGA. This is shown in detail in the diagram from Figure 5.4. How the Mapper class does this will be described later in Section 5.7.

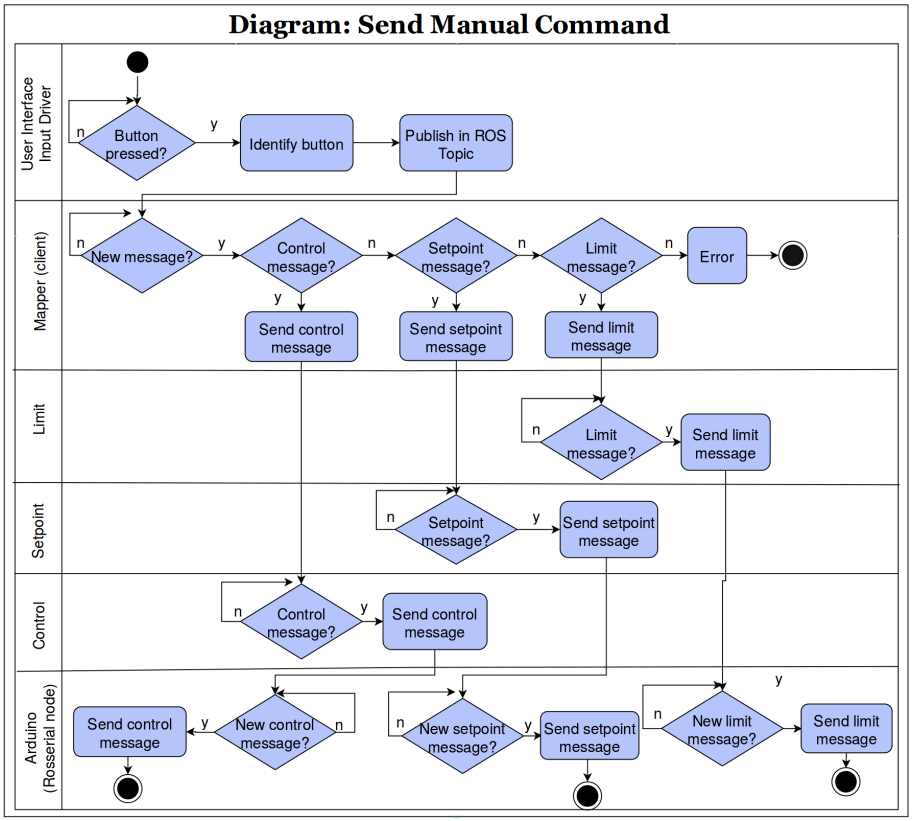


Figure 5.4: Diagram for "Send manual command" use case.

5.5.3 Send PAB command

As mentioned before, the manual commands are mapped to several buttons, sliders and knobs in the user interface. A few of these buttons don't work for manual operations, but serve to trigger PAB sequences when the operation mode of the system is PAB. Figure 5.5 shows the functionality of this use case.

As it is possible to see, most of the designed classes from Chapter 4, are used during a PAB command execution. Figure 5.5 shows the interaction of all the designed classes in a normal flow of sending a PAB command. In the end, it is possible to see two outcome states, namely the Aborted (when timed-out) and the Succeeded. There is one additional possibility which is Preempted (Canceled) and this will be discussed next. Besides the PAB commands, the diagram in Figure 5.5 shows that if an allowed manual command (moving cameras, turning on LEDs, etc.) is sent while in PAB mode, it will be executed, as if it were on Manual mode. It is worth noting that the Mapper's and Pirate Server's implementation is done in independent nodes and they both work in a non-blocking way. This means that if the Mapper first received a "Clamp Straight" PAB command, once it has requested it to the Pirate Server, it can keep listening to other commands. The condition not to send a new goal when another one is being executed

is in the *"goal currently executing?"* condition. If there is one running, the new PAB command will not execute, and the user will be alerted.

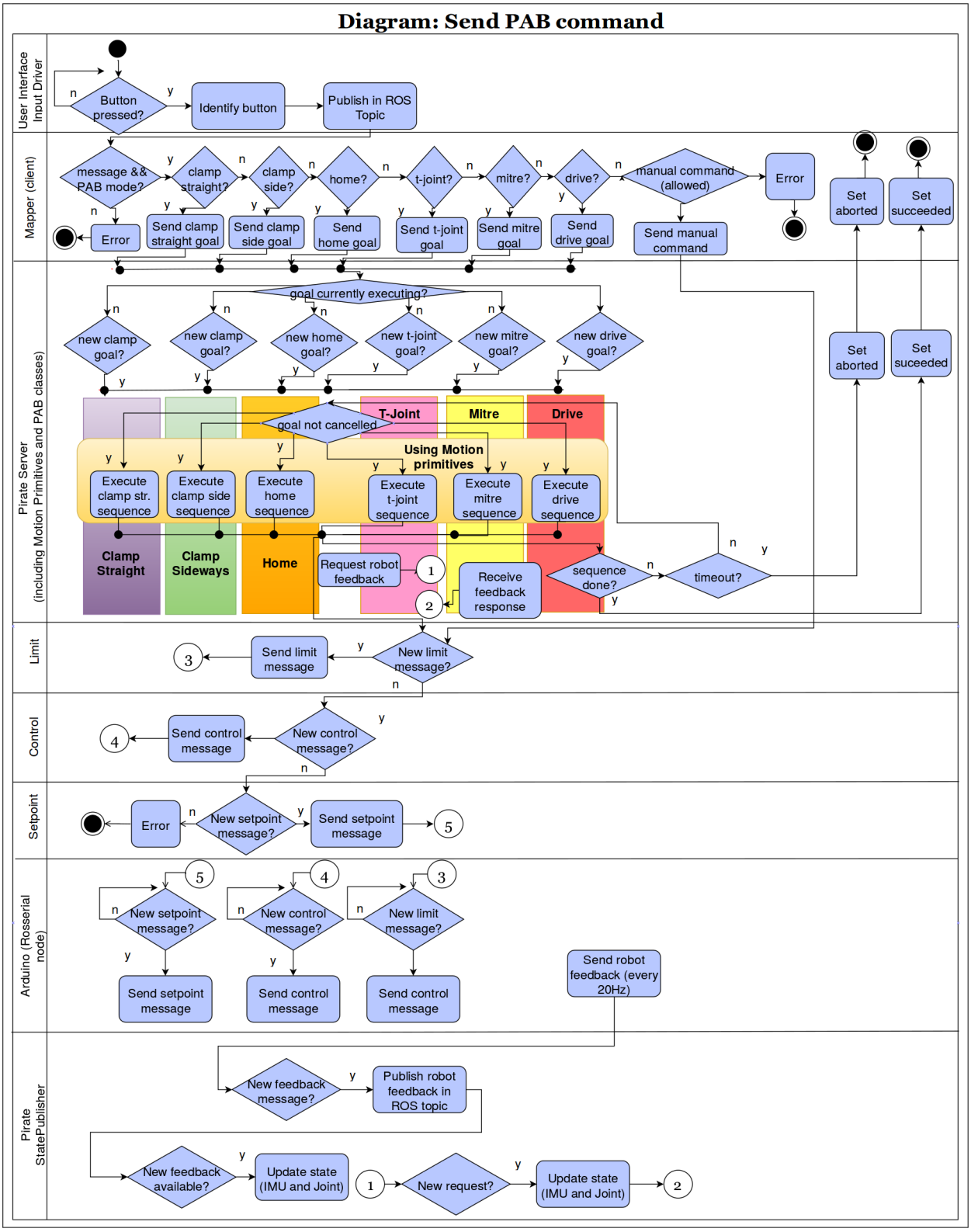


Figure 5.5: Diagram for "Send PAB command" use case.

5.5.4 Cancel PAB command

One of the most important features to implement in our software framework, was the ability to cancel a partially autonomous command independently of the stage of its execution. This capability is obtained by using one of the features of the *Action Server* class from the *Actionlib* in ROS. This standard class from ROS, has a method that is triggered when the server receives a *preempt* signal from the client. This is used to alert the other dependent classes (Motion Primitives and Clamp Straight) that the user has canceled the PAB. This and the other functionalities of the Pirate Server class will be explained in more detail in Section 5.8.

Figure 5.6 shows the functionality of this use case.

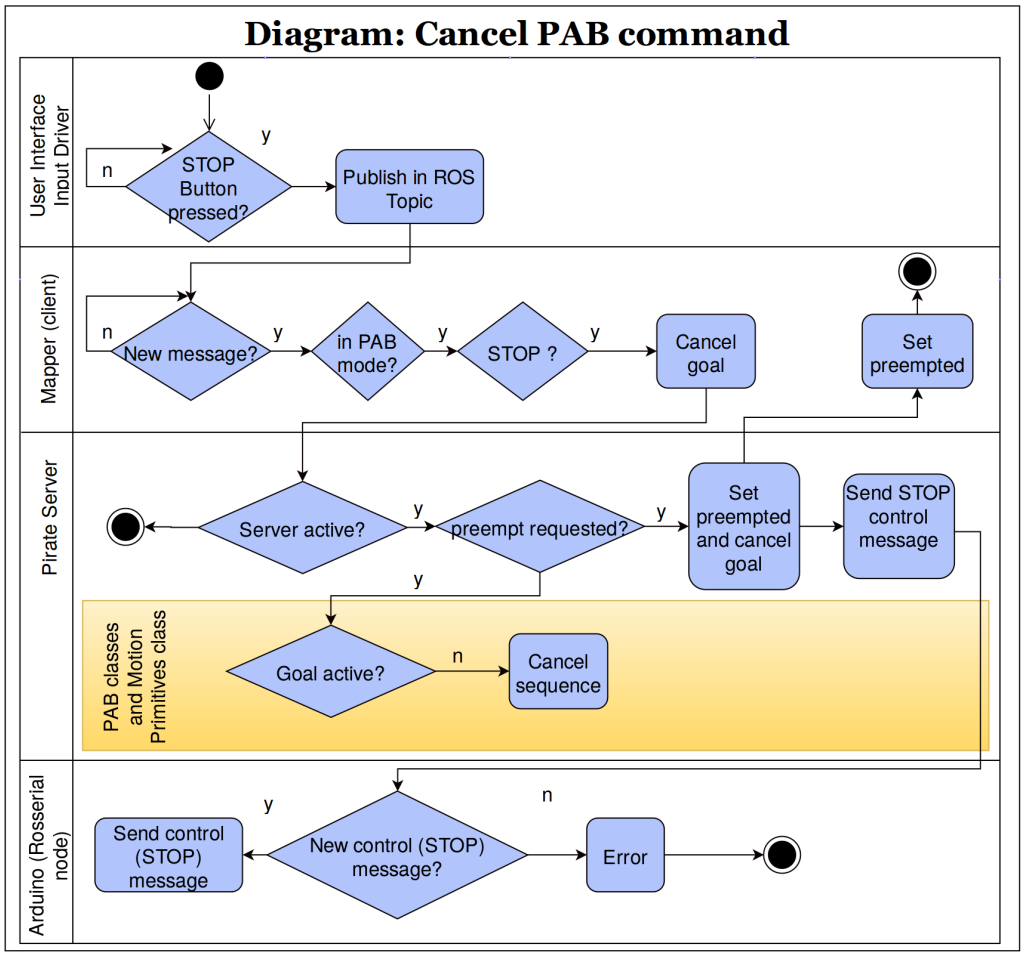


Figure 5.6: Diagram for "Cancel PAB command" use case.

5.6 Generalized ROS software implementation

As mentioned in Chapter 4, several classes were designed, each with a specific task to fulfill. In general, the implementation of the classes is done using C++. From all the classes contained in the Pirate Manager class, there is only one executable in ROS running, which is called *Pirate Manager node*. This node uses as libraries all the other classes contained, i.e. Limit, Setpoint, Mapper, Pirate Server and Motion Primitives as described in the proposed software framework design in Figure 4.1. For this reason in the ROS computation graph, only this node is shown, as seen in Figure 5.7.

In general terms, it is possible to see on the right of the image that the input of the *Pirate Manager node* is the topic that contains the input of the user interface (Midi panel). This topic has the information as described in Table 4.6. As stated previously, the *Pirate Manager node*

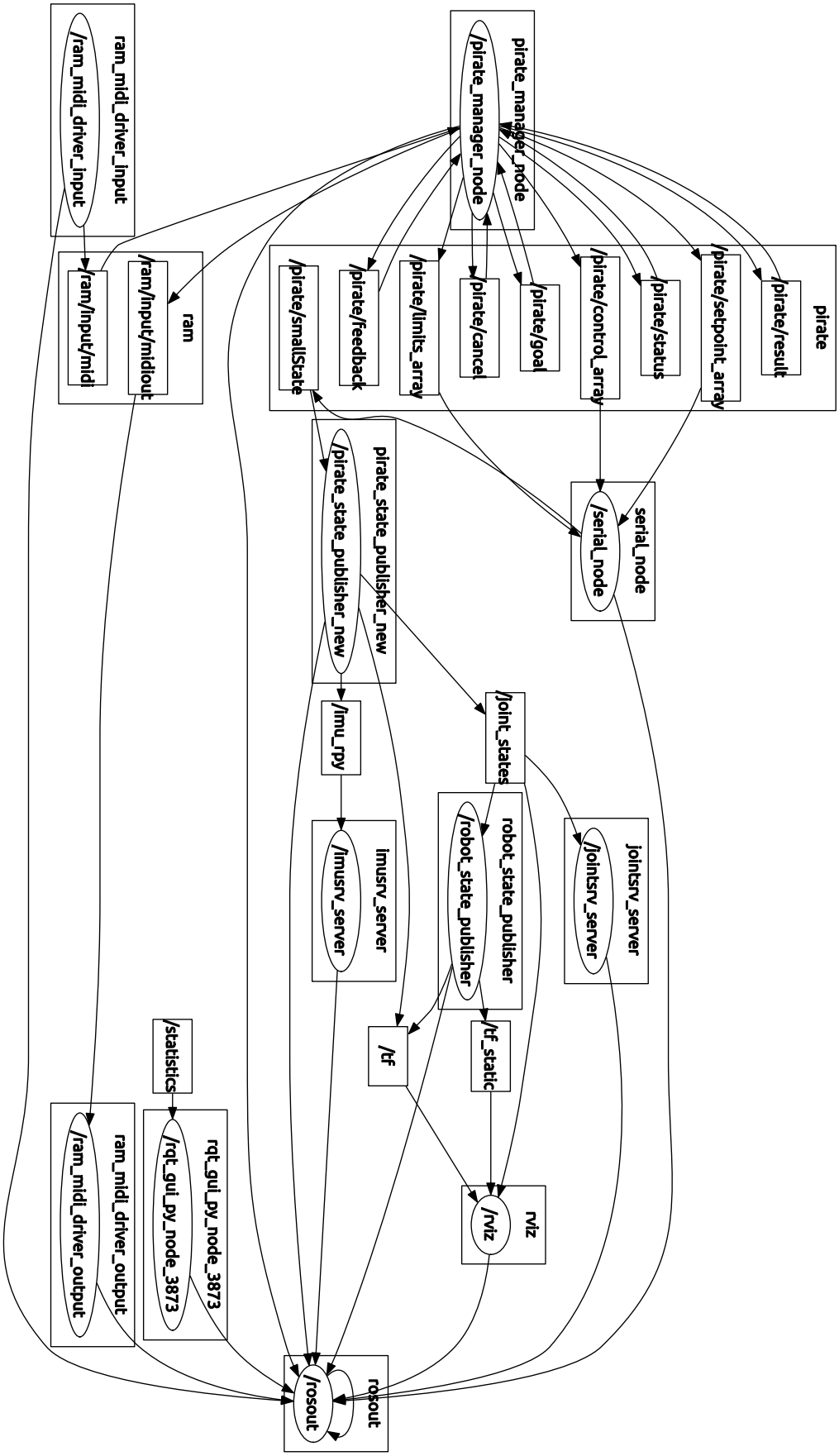


Figure 5.7: ROS computation graph for the developed software framework.

is the instance of the centralized class Pirate Manager, this is the one that interconnects most of the other topics. As outputs of the node, are the message-related topics for the setpoint, control modes and limits. Furthermore, the action topics are shown grouped together and the relationship is both coming in and out of the pirate manager node. This can be explained, as mentioned before, due to the fact that it contains inside both the Mapper and the Server, which exchange information (goal, status, result, etc.) through these topics. More towards the center the *serial node* can be observed. This is generated using the rosserial interface. The serial node receives the information by subscribing to the message related topics. In this way, the Arduino (which is the client to the rosserial) receives this information when it is published and updates the robot accordingly. Besides receiving the setpoint, control mode and limit commands the serial node publishes the Small State message discussed in Sub-subsection 4.8.1.1, which contains the feedback of the robot's status.

On the other side of the graph, the Pirate State node is found. This node subscribes to the Small State topic thus it gets the feedback for each PICO board in the robot. Then it distributes it to the implemented services, separating it in the joint service (whose node is called *jointsrv_server*) for the feedback of the joints and the IMU service (called *imu_rpy* node) for the orientation status. From the joint state obtained in the Pirate State node, a Robot publisher is created to use in an *rviz* node for the visualization. The visualization part was implemented by Reiling, during the year prior to the start of this thesis.

Finally the implementation of the communication with the user interface can be found in the lower right corner. The topics are */ram/input/midi* and */ram/input/midiout*, for the input and output to the Midi panel, respectively. It is possible to see that this topics enter the *pirate manager node* because inside it they are handled by the *Mapper* class.

5.7 Mapper class implementation

As mentioned in Section 4.6, one of the functions of the Mapper is to keep track of the operation modes, as described in the use case from Subsection 5.5.1. The feedback screen for changing the operation mode is shown in Figure C.1 in the Appendix C.

The Mapper class' main function is to translate the user interface commands. This is done by subscribing to the topic published by the User Interface Input driver node. In ROS, a subscription uses a callback function to handle whenever new data has been published. The general implementation of this callback function (named *map*) is based on a *switch case* structure which is shown in Figure 5.8.

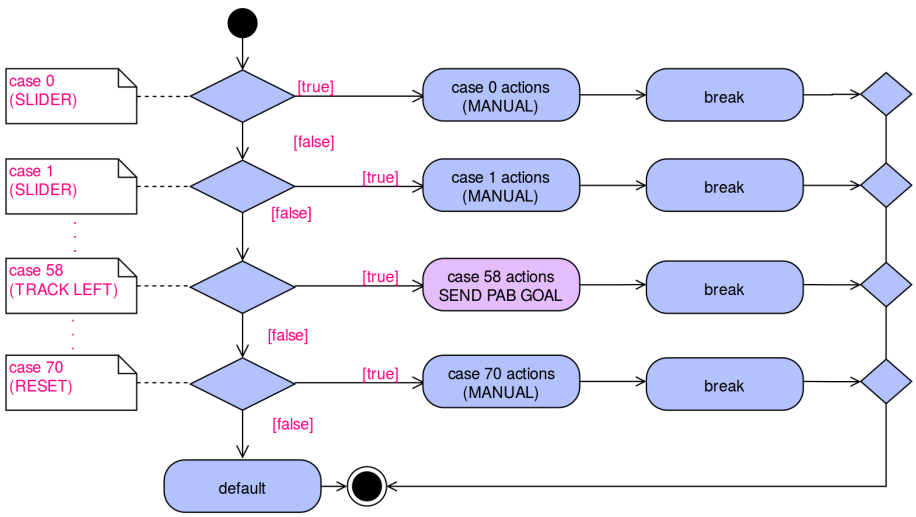


Figure 5.8: Generalized implementation of the callback function in Mapper class

In Figure 5.8 it is possible to see the different cases numbered between 0 and 70, this is because those are the channel numbers that are sent (according to Table E.1). These numbers come directly from the manufacturer of the Midi fader panel and are shown in the Appendix E. The statement "case X actions" in Figure 5.8, refers to the setpoint/control mode/ limit generation from the raw Midi value obtained from the topic. This translation was designed originally by Reiling and is the same that used to be implemented in the Arduino Mega software, before the start of this thesis.

Previous development only included manual commands, nevertheless, the current implementation of the Mapper includes now the possibility to act as a Client from the Actionlib package in ROS. To do so an instance of the *Simple Action Client* is created inside the Mapper class. The *Simple Action Client* contains ready to use member functions such as the ones for sending the goal to the server, canceling it, etc. In the example shown in Figure 5.8, it is possible to notice that case 58 corresponds to one of the buttons that triggers a PAB command. Inside this case's statements the goal parameters are configured according to what was discussed in Table 4.4 and the goal is sent. To cancel a goal, if one is active and the system is in PAB operation mode, then the *STOP* button cancels the sequence, just like in manual operation mode it would stop the motion.

5.8 Pirate Server class implementation

As we know from Section 4.7, the server handles the execution, tracking and preempting of the goals. The way it does this is by using an instance of the *Simple Action Server* class from the ROS Actionlib. This class offers some readily available functions (such as the one to preempt a goal). It is also important to notice that this supplied class of the Actionlib from ROS has an internal state machine (not to be confused with the state machine of the Pirate Server class itself, shown in Figure 4.2). The end states of this state machine can be *Succeeded, Aborted* or *Preempted*. The call for the PAB sequence is processed in the member function *execute*, where also a *switch-case* structure is implemented. A basic representation of the execute member function of the Pirate Server class is depicted in Figure 5.9.

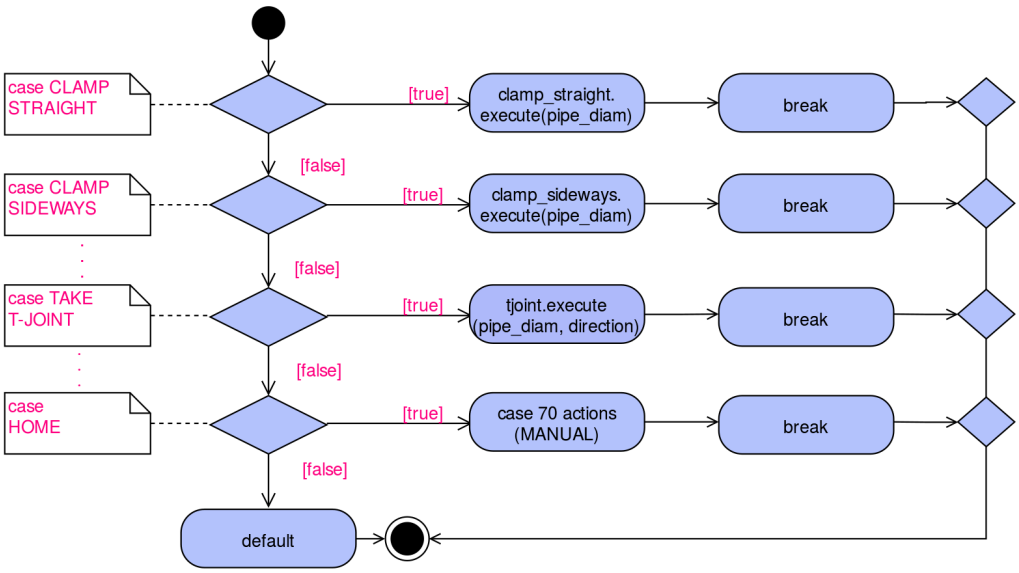


Figure 5.9: Generalized implementation of the execute member function in Pirate Server class

In Figure 5.9, it is possible to see that inside each of the cases, the member function *execute* of each of the PAB-classes will be called and, in case it is needed, the parameters will be passed. If the execute method of the corresponding PAB class returns a success value, then the server will set its internal state to *Succeeded*, when something went wrong the return value of the PAB

execute method will be false (not succeeded) which will send the internal state machine of the Simple Action Client to a *Aborted* state. This state is presented when, for example, the modules didn't respond during clamping and the sequence timed-out. At start-up in the feedback screen it is shown that the Pirate Manager, the Mapper, the Server and all the component classes are being constructed. This is depicted in Figure C.2 from the Appendix C.

Furthermore, the preempting process, as mentioned before, is handled by the supplied function of the Simple Action Server class (called *setPreempted*). This callback function will be called whenever the server receives a *cancel goal* command. In this way the status of the internal state machine of the Simple Action Server instance will change to *preempted*. This will trigger the dependent classes, in our implementation, Motion Primitives and Clamp Straight (and other PAB classes) to stop their functions as well. The feedback screen from this situation is shown in Figure C.3 from the Appendix C.

5.9 Motion Primitives class implementation

As discussed in Subsubsection 4.7.2.1, the first choice for the guards for the clamping and unclamping functions was the effort feedback. However, during design testing as well as throughout the implementation, the torque measurements obtained from the robot were inconsistent and thus did not provide a suitable guard option. This might be possible to correct by improving the robustness of torque control implementation in the PICO boards, however this was out of the scope for the present thesis. This can be observed in the torque results from Chapter 6. Because of this lack of repeatability, it was decided to use only angle feedback for all the advanced motion primitives as well as for the implementations in the PAB classes. Due to the fact that in the design stage of the Motion Primitive class, presented in Section 2.4.4, most of the functionality of the member functions was already discussed at length, in this section we will add the guards values obtained experimentally. The guard values are used inside the routines that have feedback, referred to as the Advanced routines, such as clamp front/rear, unclamping (full and partial versions) and align. Due to the limited time for the present work, the implementation was focused on the 90mm and 110mm pipes (this last one only for a nominal case). The guards for the relevant motion primitives are presented in Table 5.2. It is important to mention that all the guards used are angles because during the implementation the torque measurements were not so constant and the most repeatability was found when using the guards with angles instead of torques. In this way by supplying the diameter of the pipe, it is possible to know what angles to expect when the PIRATE has been clamped, when it has unclamped, etc. This is also the case for the guards implemented in the Clamp Straight class, which will be discussed next.

Table 5.2: Guard values used in the implementation of the Motion Primitives class

Motion Primitive	***Module***	***Angle***	***Setpoint (counts)***	***Guards (in radians)***	
				Pipe diameter	
				90mm	***110mm***
Clamp Rear / Front	Bend Module I / IV	α	256	-0.77	-1.25
	Bend Module II / III	β	-256	0.35	0.65
Unclamp Rear	Bend Module IV	α	-100	-0.15	-0.15
	Bend Module III	β	150	0.07	0.07
Unclamp Front	Bend Module I	α	-100	-0.2	-0.2
	Bend Module II	β	150	0.03	0.03
Unclamp Rear / Front Partial	Bend Module II / III	β	150	0.29	0.45
	Bend Module I / IV	α	-100	-0.48	-0.9
Align	Rotational Module	θ	0	± 0.0425	± 0.0425

In order to let the user know if there has been any errors, for every advanced motion primitive function (that includes feedback), there is a timeout. This was standardized to 15 seconds, because according to timings the motion primitives take between 5-8 seconds to complete, so this gives room before an error is registered. The feedback screen with a timeout is presented in Figure C.4 from the Appendix C.

5.10 Clamp Straight class implementation

The Clamp Straight class is possibly the most important and challenging implementation of the project, because of all the details that the execution routine has to take into account. The state machine presented in Figure 4.8, already showed how the system would process the sequence. For the implementation of this state machine, we chose a *switch-case* structure once again, where it would be easy to keep the actions corresponding to each state in each one of the cases. This structure is depicted in Figure 5.10. Furthermore, the feedback screen from the implementation, with a successful clamp carried out is shown in Figure C.5 from the Appendix C.

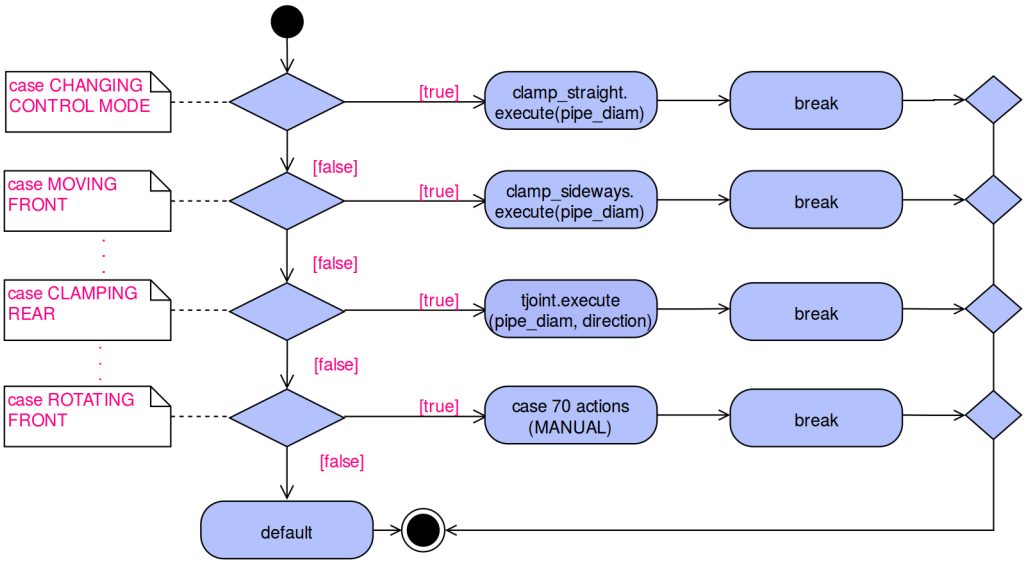


Figure 5.10: Generalized implementation of the state machine of the Clamp Straight class

One of the most important parts of the implementation of the clamp straight was to clearly define the range for considering a nominal case. According to tests, this was defined to be between *1.4* and *1.74* radians (this is mentioned in Table 5.4), according to the orientation defined in Figure 4.7.

The first state in which the clamp straight routine starts is by changing the control mode of some of its modules. This is done to take maximum advantage of the fact that the bend modules can be controlled using torque (even if as feedback we check the reached angle). By using this setting and sending the maximum setpoint, the modules clamp very fast and at the moment that they hit the way the angle is reached. This proved to be a good method during the implementation. For this reason, the first step in the sequence changes the modes of Bend modules I, II, III and IV to torque mode.

Another important step to take according to the designed state machine (from Figure 4.8), would be to move the front module. As discussed previously, this step needs to be carried out before trying to clamp the front part of the robot. If this is not done then when the bend modules move they exert force in the front module and it is possible to damage it, as well as the fact that it makes it even harder for the modules to clamp (because it is opposing their movement). The guard for this translation depends on the pipe diameter and in the non-nominal case the movement is divided into two steps (the sequence passes twice in this state), because the first

Table 5.3: Guards for the clamp custom and move front functions

Motion Primitive	***Module***	***Angle***	***Setpoint (counts)***	***Guards (in radians)***	
				Pipe diameter	
				90mm	***110mm***
Clamp Front Custom (first time)	Bend I	α	256	-0.69	Not implemented
	Bend II	β	-256	0.30	Not implemented
Clamp Front Custom (upcoming times)	Bend I	α	256	-0.72	Not implemented
	Bend II	β	-256	0.30	Not implemented
Clamp Rear Custom	Bend III	β	-256	0.33	Not implemented
	Bend IV	α	256	-0.68	Not implemented
Move Front Nominal case	Front	γ	32	0.4	0.7
Move Front (first time - Non-nom.)	Front	γ	32	0.105	Not implemented
Move Front (upcoming times - Non-nom.)	Front	γ	32	0.3	Not implemented

time a slight movement is done, and when the robot is approaching the straight position, the angle is increased for better clamping. The guards for the movement of the front module are listed in Table 5.3.

In a non-nominal sequence, the use of the motion primitive functions called *clamp custom* is necessary. This is because robot starts lied on either side and thus the (full) clamping guards do not apply because the clamp is done in a length equivalent to the chord of the pipe and not the full diameter. Because of this during the implementation the optimal values to provide a clamping in such situations were found. According to the design during the implementation, the first time it clamps the front is while the robot is still fully laid on the side. During the implementation it was seen that this clamping specifically requires different parameters, than the upcoming times when it is closer to making a full clamp. For this reason, there is a guard specially for the first time clamping the front in case the robot is starting from the side (unknown position). The upcoming times clamping the front have another guard, before it can be considered full clamp, for which the guards are mentioned in Table 5.2. The guards for the custom clamp can be found in Table 5.3.

In general, due to the time limitation for the present thesis, the non-nominal sequence was only implemented in the 90mm diameter pipe. It is worth noting that even though the implementation is not done for 110mm, the only difference would be to get appropriate guard values for the front module to move and for the custom clamp. The designed sequence, nonetheless, would be fully applicable to the other diameters without any expected modification.

5.10.1 Rotation procedure while clamping

The rotation motion primitive can be called whenever there is need to re-adjust the orientation of the robot in order to be considered *Straight*. Within the so-called *nominal* case which is defined as mentioned in Section 5.10, there may still be the necessity to adjust. In order for the robot to be considered *Straight* the orientation must be in the range of 1.52 to 1.62 radians, which is a ± 0.05 radian tolerance for the true straight position at $\pi/2$ radians. Table 5.4 shows the tolerances defined for the nominal and straight regions and they are depicted in Figure 5.11.

If for some reason the robot is not straight when clamping, it will unclamp the rear part and then rotate and re-clamp. After this, the front part will unclamp and first align (at this point the current orientation is obtained because the front part does not have IMU sensor). After aligning it will check if it is still outside the tolerance of Straight position. If it is outside, it will rotate

Table 5.4: Tolerances for the different cases of the PIRATE Clamp Straight

Region	*Lower Limit (rad)*	*Upper Limit (rad)*
Nominal	≥1.4	≤1.74
Straight	≥1.52	≤1.62
Non-nominal	≥0	<1.4
	>1.74	$\leq \pi$

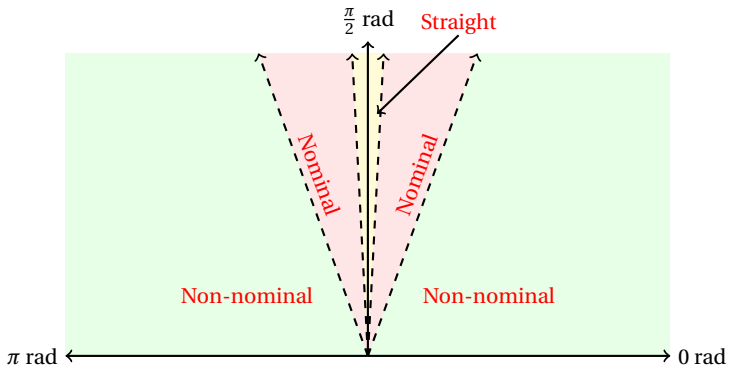


Figure 5.11: Straight, nominal and non-nominal tolerances defined for the implementation.

further, if not it will stay aligned and clamp again. The setpoint is, of course, variable in every case, as it depends on the difference of the current orientation to the true straight position. This is calculated differently for the front and rear modules, due to the fact that their rotation setpoints have to have opposite sign in order to rotate in the same direction. The setpoints are calculated using the following equations:

$$setpoint_{front} = current\ orientation - straight\ position \tag{5.1}$$

$$setpoint_{rear} = straight\ position - current\ orientation \tag{5.2}$$

There are some safety limits implemented for the rotation, as the we want to protect the mechanical parts of the robot, which could be damaged if the parts twist too much. The safety limit was set to be a rotation of ±0.7 radians.

The feedback screen of the implementation, showing the rotation steps is depicted in Figure C.6 in the Appendix C.

5.11 Pirate State implementation

As commented in Subsection 4.8.1, the goal of the Pirate State interface class is to retrieve the robot's feedback (efforts, angles, velocity, odometry, etc.) for all the joints as well as the IMU sensor. The logic behind this node was already implemented by Reiling, prior to the start of this thesis. During the implementation we based on his development and just adjusted it to be a class and use the publish-subscribe model from ROS.

The feedback of the robot states comes from a topic published in the Arduino using the Small State message, discussed in Sub-subsection 4.8.1.1. Because of this, the Pirate State class will subscribe to this topic published and in turn divide the information into two topics for the *services* for the joint sensors and the IMU sensor. After the services get the most updated data, they can send it every time they get a request from the clients. This is shown in Figure 5.12. Apart from the services, we can see that the Pirate state feeds the visualization node.

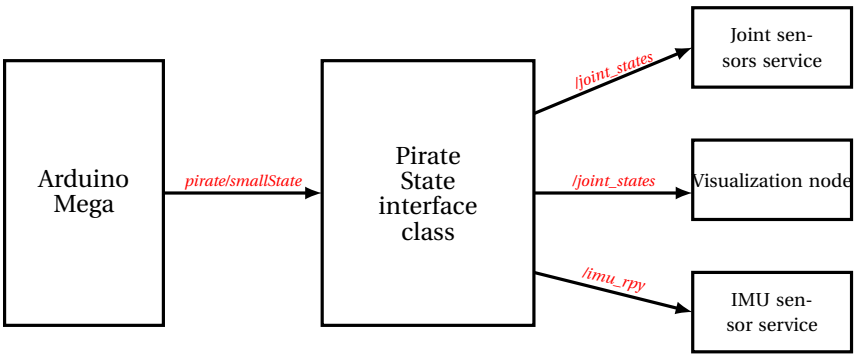


Figure 5.12: Communication of the feedback from the Arduino Mega through the Pirate State and the services.

5.12 Simple test mission implementation

As described in Section 1.4, part of the aims of the present thesis is to demonstrate the capabilities of the software framework by developing a simple example mission. The mission should shown how some of the possible partially autonomous tasks can be executed sequentially. This is done so that it is possible to see that a deliberative layer, where planning for such sequences, can be done in the future.

The mission was defined as the following steps:

1. Clamp Straight
2. Drive forward
3. Drive backward
4. Home

Because of this, also a very simple verion of the Drive and Home classes was implemented. For the Home, inside the execute member function, a switch case structure was implemented, which follows the designed state machine showed in Figure 4.9. As mentioned previously, this implementation is not through an has the assumption that the robot starts clamped in a straight orientation. In the future, this should be complemented for unknown starting orientations. For the Drive, as it only has one state, the implementation of the execute member function takes in a desired distance and velocity, and calculates the *ideal* time it should take for the robot to achieve this distance at a certain velocity. This calculation is made according to:

$$x = \frac{(v_0 + v_f)t}{2} \tag{5.3}$$

In equation 5.3, v_0 represents the initial velocity which for the test mission will be 0rad/s. The v_f represents the desired (constant) velocity, t the required time and x the desired distance. This equation is then used to solve for the time and in this way achieve a desired distance. This distances, in practice, were however not achieved, which might be due to friction between the wheels and the pipe (stiction) and some other non-idealities. This was then adjusted manually for a chosen velocity of 150rad/s, and worked acceptably. However when really focusing on this implementation, it is perhaps a better option to use the odometry to measure the distance and not the time. This, however, was not done because the idea was to test the drive functionality, without to much consideration for the actual distance accuracy. For the scope of this thesis, an in-depth implementation of the Drive function, is out of reach. Nonetheless, this proves the potential of implementing the drive motion primitive for PAB-sequences.

With this, the implementation of the simple test mission designed for this thesis was completed.

6 Results

This chapter will show the results obtained from the designed and implemented software framework. First the tests and their aim will be described. Then for each of them, the results will be presented and discussed.

6.1 Tests

In order to prove the effectiveness and the potential of the developed part of the software framework, various tests were designed. They will be comprehensive experiments and will be carried out to check the functionality at the PAB-level (which will prove also the effectiveness of the motion primitives because at some point they are all used inside the sequences).

The aim of this project focused heavily on the design and implementation of the *Clamp Straight* routine. The tests for this sequence are given by:

- *Simple clamp straight*: The robot starts with an orientation *within* the tolerance range of the *straight position*. It is expected that the robot clamps and does not require to re-adjust the orientation (no rotation).
- *Nominal re-adjust clamp straight*: The robot starts in an orientation within the tolerance of the nominal case, but outside of the straight range. It is expected that the robot clamps and re-adjusts slightly (small rotations) until it reaches the straight position.
- *Non-Nominal clamp straight*: The robot starts with an unknown orientation (either lying on the right or left side normally) outside of the nominal case. It is expected that the robot clamps and re-adjusts (rotates) until it reaches the straight position, i.e. *"gets up"*.

Because the 90mm pipe was the one that had all the implementations complete, the tests are carried out in this pipe diameter. The tests were run 10 times for each of the scenarios. For each of the tests, a graphical assessment of the guards will be done, to observe and evaluate the way the sequence works. A summary of the results for the guards evaluation is given in Section 6.6. Also some details on the torque measurements will be given, as proof of some of the inconsistencies found, reason for which they were not used as guards in the present work. The resolution of the angle measurement during the tests, is 0.01 radians for each of the bend and front angles and for the orientation angle it is around 1×10^{-9} radians.

As mentioned in Section 1.4, apart from evaluating the clamp straight behavior, a simple test mission forms part of the assessment. The mission consists on: clamping straight, driving forward, driving backward and homing the robot. The mission was run 10 times as well. The aim of this mission is to show the overall capacity of combining different PABs in sequences, which would be, in the future, generated by a supervisory layer.

To evaluate all routines, a log was obtained during the experiments which collects the measured angles, efforts and orientation of the PIRATE robot.

6.2 Simple clamp straight in 90mm pipe

As mentioned previously, the simple clamp straight routine only consists of the steps: changing the control modes to torque for the bend modules, move up the front one before clamp, and clamping both rear and front modules. As in this case the start-up orientation is within the straight range, there will be no need to rotate to re-adjust.

The results of the experiment can be seen in Figure 6.1. It is worth noting that the logged information has been obtained in a live test using the robot, however for better understanding, graphics of the 3D CAD PIRATE are shown along.

Table 6.1 summarizes the milestones achieved at each instant of time by the robot along with the control mode used. It is possible to see that the guards are reached as desired, and that the end result is a robot that has an straight orientation. This fact that the orientation is achieved can bee seen as it is very close to the ideal straight position, marked with a pink dotted line in the graph of the rotation (angle 25) and orientation. From Figure 6.1 it is also possible to estimate the duration of this routine of around 10.77 seconds (average).

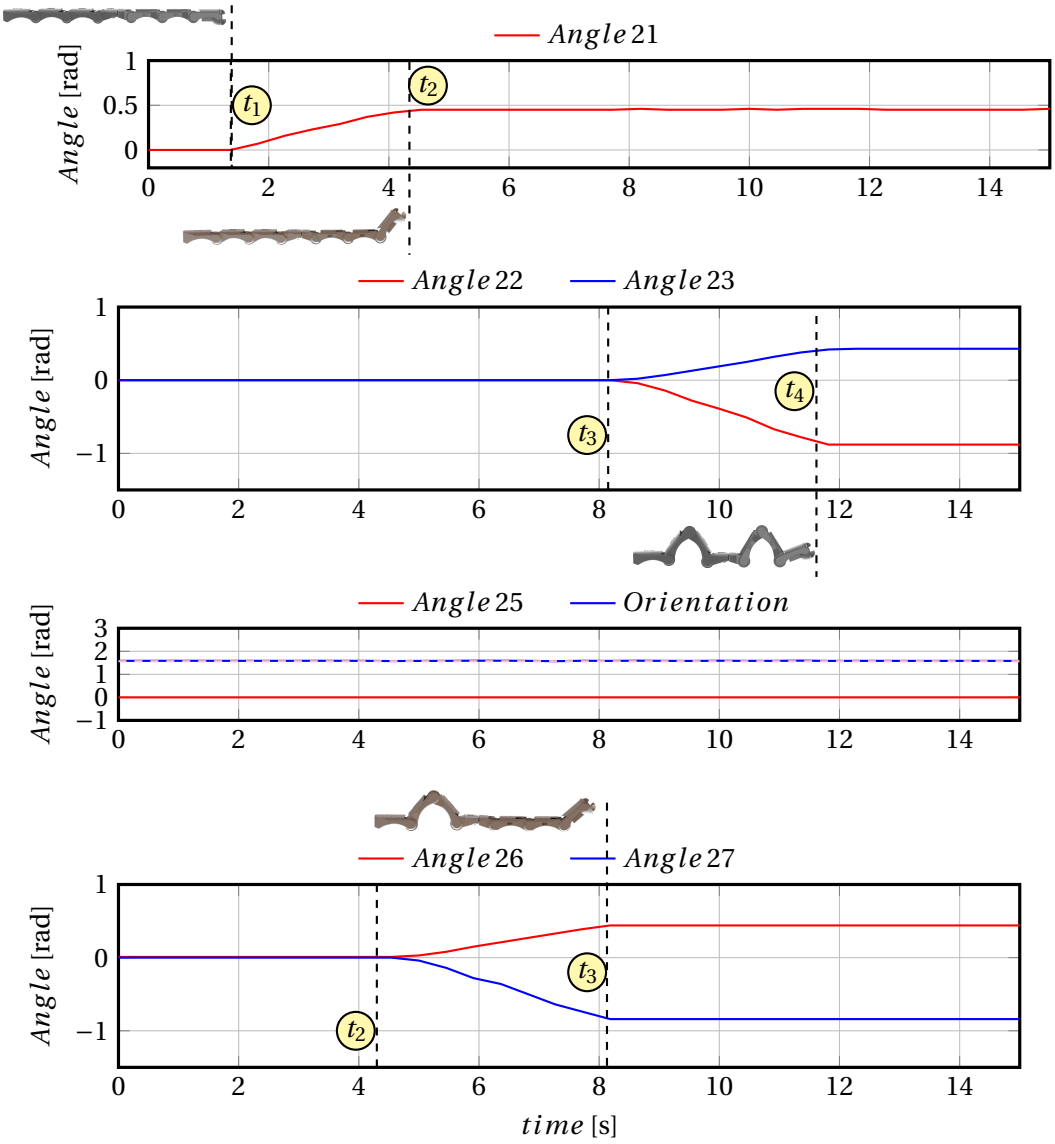


Figure 6.1: Results showing the achievement of the guards for the test Simple Clamp Straight in 90mm pipe

Table 6.1: Sequence of steps of motion primitives with respective control modes and guards for simple clamp straight routine in 90mm pipe

Start Time	***End Time***	***Motion Primitive***	***Control mode***	***Module***	***Guard (rad)***	***Module***	***Guard (rad)***
t1	t2	Move	PWM	Front	0.4	-	-
t2	t3	Clamp rear	Torque	Bend III	0.35	Bend IV	-0.77
t3	t4	Clamp front	Torque	Bend II	0.35	Bend I	-0.77

Figure 6.2 now shows the torque measurements during the operation, as it is possible to see, there are indeed peaks of torque when clamping, however as explained in both Chapters 4 and 5, these were observed to be not very reliable as the magnitudes varied from a few mNm to much more considerable quantities. Here it is possible to see that the effort obtained from the slave 23 (Bend II) is considerably lower in magnitude as the other ones. This slave is expected to behave like slave number 26 (Bend III) because of the mirrored design, however it is clearly different. It is important to note that both slave 22 (Bend I) and 27 (Bend IV) present an increase of torque during clamping, while 23 and 26 are supposed to have negative values. It will be seen in the other experiments that this is not always the case, and was therefore found unreliable as guard values.

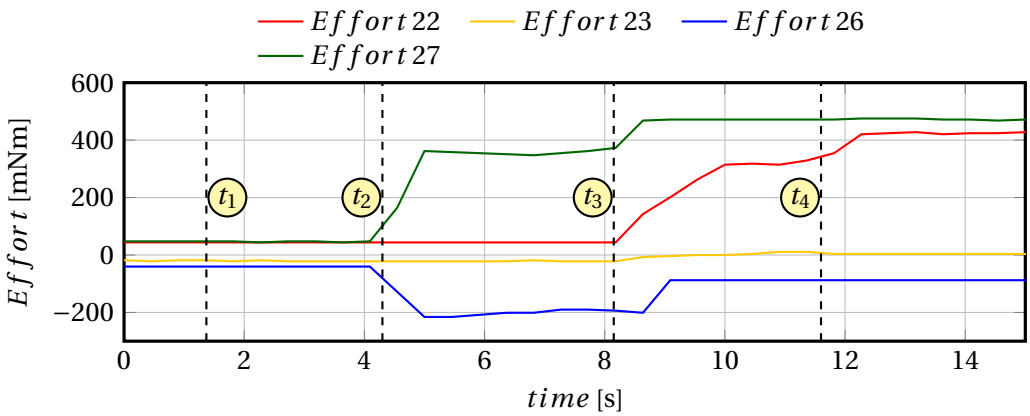


Figure 6.2: Effort measurements for bend modules for the simple clamp straight test in 90mm pipe.

6.3 Nominal re-adjust clamp straight in 90mm pipe

In this scenario, the PIRATE starts with an orientation within the defined nominal range, but outside the straight tolerance range. According to the designed state machine, the sequence of steps to take are the following:

1. Move front module to be ready to clamp
2. Clamp rear part
3. Clamp front part
4. Unclamp rear part
5. Rotate the rear part according to the calculated difference with respect to the straight position
6. Clamp rear part
7. Unclamp front part
8. Rotate the front part according to the calculated difference with respect to the straight position
9. Clamp front part

This behavior can be observed in detail in the graphs of Figure 6.3. Again, the measurements have been done with the robot, and the visual aids of each step are taken from the 3D CAD PIRATE. Again in Figure 6.3 it is possible to see that the final position is very close to the value indicated by the pink dotted line in the graph for rotational angle and orientation. This means that the sequence has finished within the straight position tolerance. A summary of each step's guards and control modes is shown in Table 6.2.

This routine contains more steps than the previous one which is ready with one repetition and requires no re-adjustments. Because of this, it is natural that the time this one takes (around 25

seconds in average for this type of experiment) is greater (more than twice the time) than the one of the simple clamp.

The torque measurements from this experiment can be seen in Figure 6.4. It is possible to appreciate that once again slave 23 (Bend II) shows very smooth changes, which could be hard to identify from non-clamping behaviors. Also changes in torque for slaves 22 (Bend I) and 27 (Bend IV), which should be comparable in magnitude because of the mirrored design are very different. Slave 22 shows a magnitude of around half of that of 27, which once again for guards makes it unreliable.

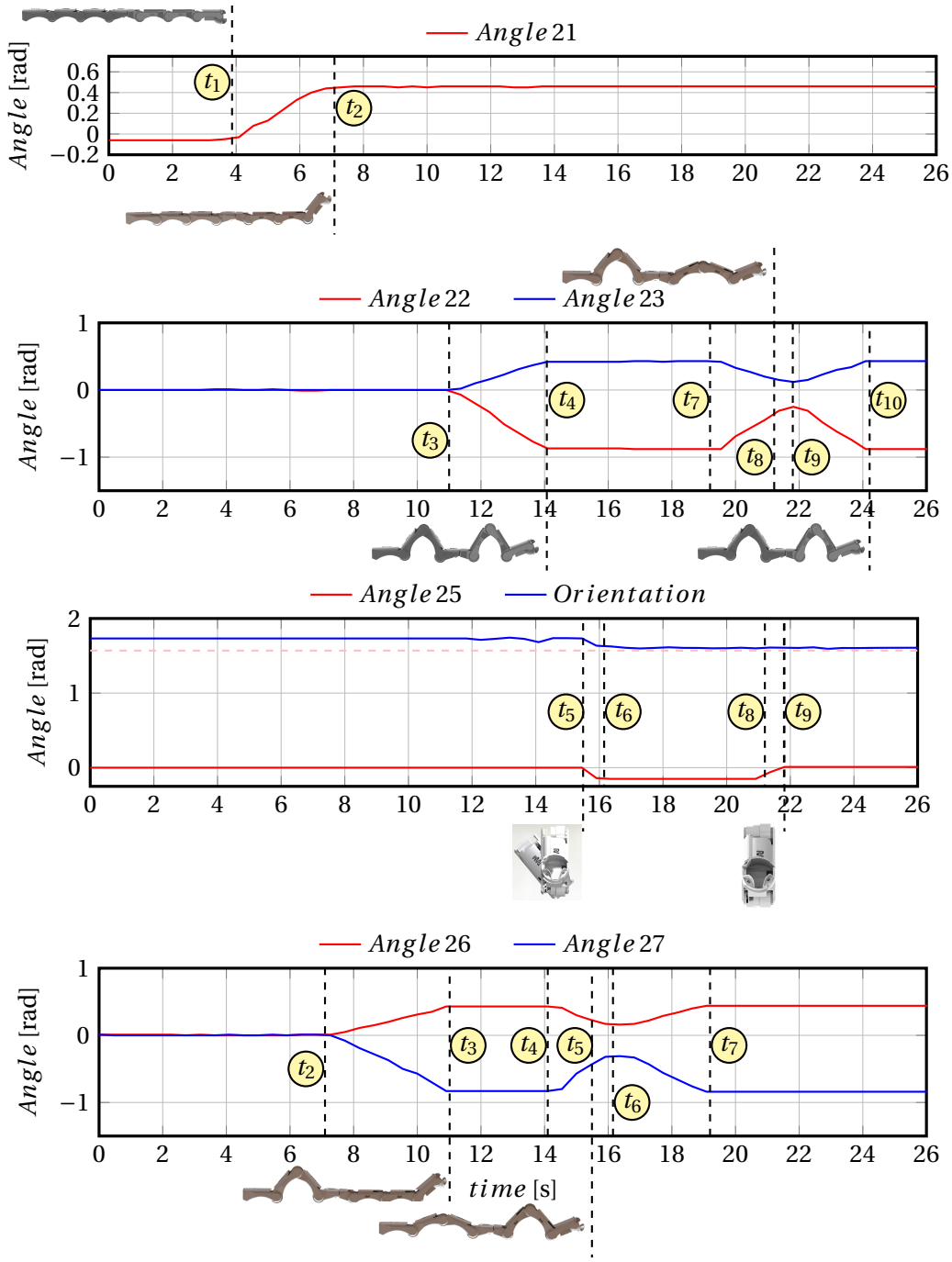


Figure 6.3: Results showing the guards for the Nominal re-adjust Clamp Straight in 90mm pipe

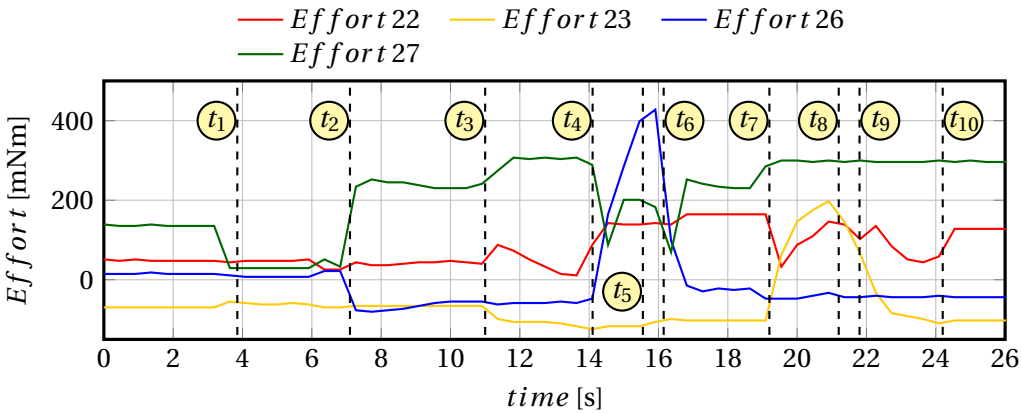


Figure 6.4: Effort measurements for bend modules for the nominal re-adjust clamp straight in 90mm pipe.

Table 6.2: Motion primitives, control modes and guards for nominal re-adjust clamp straight (90mm)

Start Time	***End Time***	***Motion Primitives***	***Control mode***	***Module***	***Guard (rad)***	***Module***	***Guard (rad)***
t1	t2	Move	PWM	Front	0.4	-	-
t2	t3	Clamp rear	Torque	Bend III	0.35	Bend IV	-0.77
t3	t4	Clamp front	Torque	Bend II	0.35	Bend I	-0.77
t4	t5	Unclamp rear partial	Torque	Bend III	0.29	Bend IV	-0.48
t5	t6	Rotate rear	Position	Rotational	θ_{diff}	-	-
t6	t7	Clamp rear	Torque	Bend III	0.35	Bend IV	-0.77
t7	t8	Unclamp front partial	Torque	Bend II	0.29	Bend I	-0.48
t8	t9	Rotate front	Position	Rotational	0	-	-
t9	t10	Clamp front	Torque	Bend II	0.35	Bend I	-0.77

This test was also implemented and tested for a 110mm pipe. The result graphs of this can be seen in the Appendix D, Section D.2

6.4 Non-nominal clamp straight in 90mm pipe

This experiment represents the worst-case scenario for the clamping straight sequence. It parts from an unknown position outside the nominal tolerance region (in the non-nominal region). Because of this, the order of the routine in the state machine has to be altered, which yields the following sequence of steps:

1. Clamp rear part
2. Rotate front part according to difference with respect to straight position
3. Move slighly front module to prepare for clamping front part
4. Clamp front part
5. Move the front module more to be ready for a full front clamp.
6. Unclamp rear
7. Rotate rear part according to difference with respect to straight position
8. Clamp rear part
9. Unclamp front part
10. Rotate front part according to difference with respect to straight position
11. Clamp front part
12. Unclamp rear part
13. Rotate rear part according to difference with respect to straight position

14. Clamp rear part
15. Unclamp front part
16. Rotate front part according to difference with respect to straight position
17. Clamp front

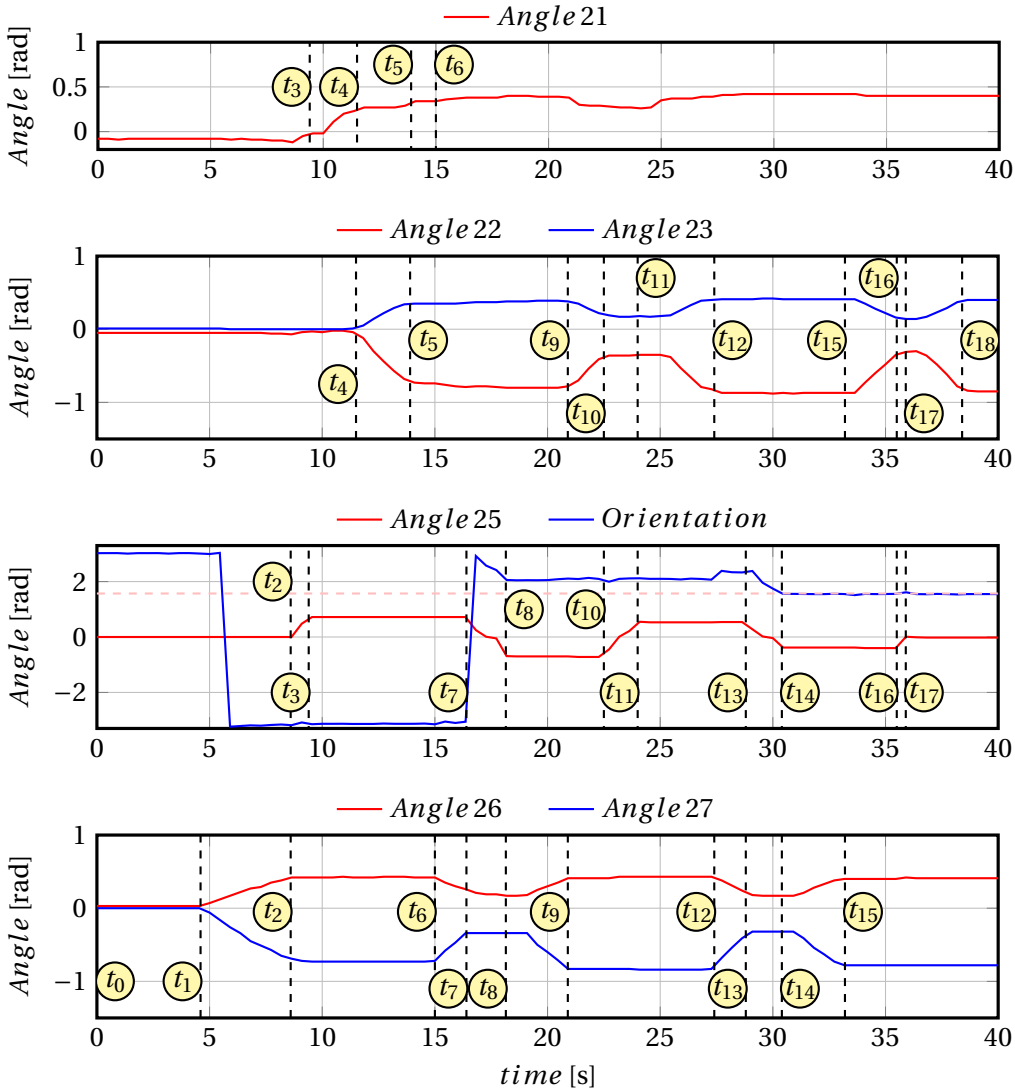


Figure 6.5: Results showing the achievement of the guards for the test Nominal re-adjust Clamp Straight in a 90mm pipe

Table 6.3 summarizes the steps of the sequence, together with the control modes and the guards that allow the transition between the states.

Table 6.3: Sequence of steps of motion primitives with respective control modes and guards for non-nominal clamp straight routine in 90mm pipe

Start Time	***End Time***	***Motion Primitive***	***Control mode***	***Module***	***Guard (rad)***	***Module***	***Guard (rad)***
t1	t2	Clamp rear custom	Torque	Bend III	0.33	Bend IV	-0.68
t2	t3	Rotate front	Position	Rotational	θ_{diff}	-	-
t3	t4	Move front (first time)	PWM	Front	0.105	-	-
t4	t5	Clamp front custom (first)	Torque	Bend II	0.3	Bend I	-0.69
t5	t6	Move front	PWM	Front	0.3	-	-
t6	t7	Unclamp rear partial	Torque	Bend III	0.29	Bend IV	-0.48
t7	t8	Rotate rear	Position	Rotational	θ_{diff}	-	-
t8	t9	Clamp rear custom	Torque	Bend III	0.33	Bend IV	-0.68
t9	t10	Unclamp front partial	Torque	Bend II	0.29	Bend I	-0.48
t10	t11	Rotate front	Position	Rotational	θ_{diff}	-	-
t11	t12	Clamp front custom	Torque	Bend II	0.3	Bend I	-0.72
t12	t13	Unclamp rear partial	Torque	Bend III	0.29	Bend IV	-0.48
t13	t14	Rotate rear	Position	Rotational	θ_{diff}	-	-
t14	t15	Clamp rear	Torque	Bend III	0.35	Bend IV	-0.77
t15	t16	Unclamp front partial	Torque	Bend II	0.29	Bend I	-0.48
t16	t17	Rotate front	Position	Rotational	0	-	-
t17	t18	Clamp front	Torque	Bend II	0.35	Bend I	-0.77

Finally, during this test, as well as the previous ones, the torque measurements were also recorded and are depicted in Figure 6.6. It is possible to observe that this time the feedback from slaves 26 and 27 behaves as in the simple clamp case, having opposite directions, as expected. In this case slave 23 showed greater magnitude changes than in the previous tests, but it is possible to appreciate a "smooth" increase, which would be harder to detect if the magnitude is not consistently located in a range. Most of the magnitudes and even sometimes the sign of the torque measurements changes from experiment to experiment (see results for 110mm pipe in a non-nominal case with re-adjust in Appendix D, Section D.2). If an appropriate direction and magnitude cannot be checked, it is possible that the guards will not work consistently. For this reason, they were not used as guard conditions, since most of the attempts to do so proved to be hardly reliable.

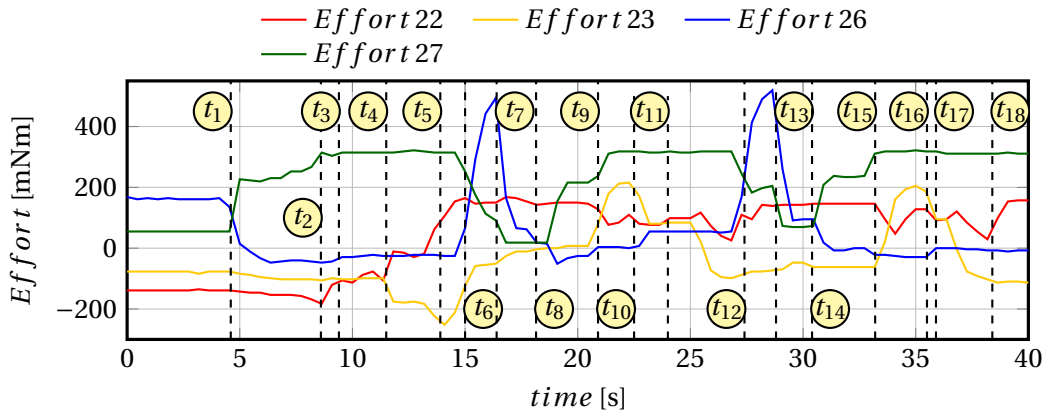


Figure 6.6: Efforts for bend modules for the non-nominal clamp straight in 90mm pipe

6.5 Simple test mission in 90mm pipe

Since the Chapter 1, it was mentioned that one of the aims of the present thesis is to demonstrate the capabilities of the software framework by integrating different PABs into a sequence (called *mission*). The scope of this thesis included the full design and implementation of only one PAB, the Clamp Straight. However, a simple implementation of the Home and Drive PABs was included with the intention of creating the test mission.

The test mission consists of the following sequence:

1. Move front to prepare for clamping front
2. Clamp rear part
3. Clamp front part
4. Drive forward
5. Drive backward
6. Home

The intention is to see that it is possible to call different PABs execute functions sequentially. This provides a proof that the developed software framework for the Sequential control layer, can indeed be used when implementing a more abstract supervisory software layer.

The angle measurements and orientation, just like in the previous tests is shown in Figure 6.7. During the first four time events it is possible to see the same behavior and guards as the simple clamp straight class presented in Section 6.2. Between the fourth and fifth time events, the drive occurs. The time of the drive depends, of course, on the desired distance and velocity that is supplied to the functions. In this case the distances were forward 5cm and backward 3cm and the velocity was set to 150mm/s. The final part of the graph corresponds to the execution of the Home PAB. There it is also possible to see the use of the unclamp front/rear motion primitive which hadn't been used in the Clamp Straight routine (where the partial one was used).

Table 6.4 summarizes the steps of the sequence, together with the control modes and the guards that allow the transition between the states.

Table 6.4: Sequence of steps of motion primitives with respective control modes and guards for simple mission routine in 90mm pipe

Start Time	***End Time***	***Motion Primitive***	***Control mode***	***Module***	***Guard (rad)***	***Module***	***Guard (rad)***
t1	t2	Move	PWM	Front	0.4	-	-
t2	t3	Clamp rear	Torque	Bend III	0.35	Bend IV	-0.77
t3	t4	Clamp front	Torque	Bend II	0.35	Bend I	-0.77
t4	t5	Drive forward and Drive backward	PWM (Velocity)	Bend I, II, III, IV, Rotational and Rear	Time according to distance	-	-
t5	t6	Unclamping front	Torque	Bend II	0.03	Bend I	-0.2
t6	t7	Unclamping rear	Torque	Bend III	0.07	Bend IV	-0.25
t7	t8	Moving front	PWM	Front	-60	-	-

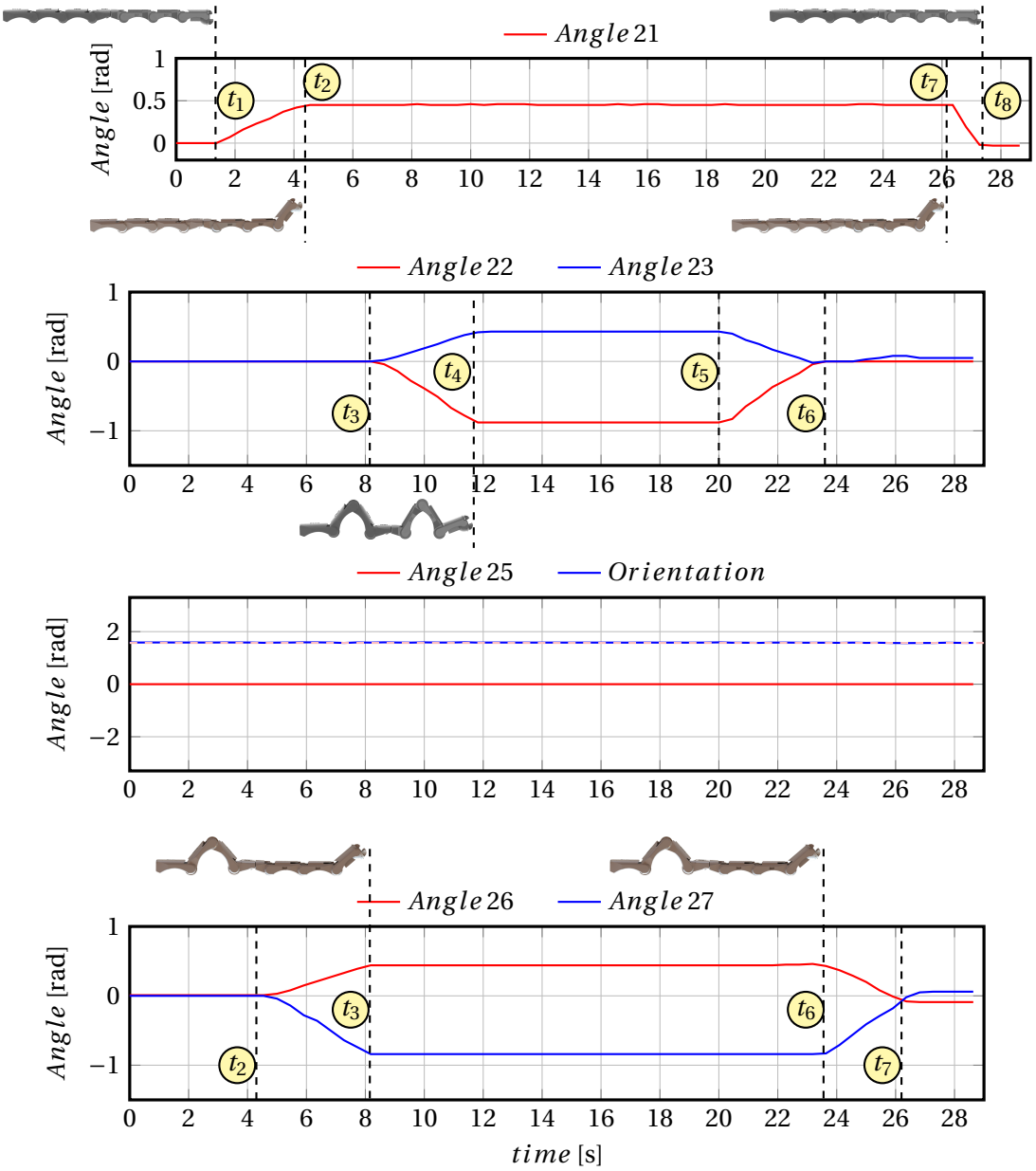


Figure 6.7: Results showing the achievement of the guards for the simple test mission in a 90mm pipe

6.6 General guards evaluation

Having carried out the necessary tests, Table 6.5 presents a summary of all the guards' results. The table contains the worst case value seen during the testing, which refers to the closes value to the guard. Also an approximate average of the value for each motion primitive over all the tests. With these results, it is possible to see that the guards chosen during the implementation have been effective throughout the different tests and scenarios.

Table 6.5: Summary of the results of the guards throughout the experiments

Motion Primitive	Module	Angle	Worst case (rad)	Average (rad)	Guard (rad)
Clamp Front	Bend I	α	0.4	0.42	0.35
	Bend II	β	-0.82	-0.85	-0.77
Clamp Rear	Bend III	β	0.42	0.44	0.35
	Bend IV	α	-0.79	-0.82	-0.77
Unclamp Front partial	Bend I	α	-0.37	-0.34	-0.48
	Bend II	β	0.21	0.2	0.29
Unclamp Rear partial	Bend III	β	0.2	0.2	0.29
	Bend IV	α	-0.35	-0.33	-0.48
Unclamp Rear	Bend III	β	-0.01	-0.09	0.07
	Bend IV	α	0.02	0.06	-0.15
Unclamp Front	Bend I	α	-0.09	0.00	-0.2
	Bend II	β	0.01	0.00	0.03
Clamp rear custom	Bend III	β	0.42	0.43	0.3
	Bend IV	α	-0.78	-0.79	-0.68
Clamp front custom (first time)	Bend I	α	-0.75	-0.77	-0.69
	Bend II	β	0.33	0.34	0.3
Clamp front custom (upcoming times)	Bend I	α	-0.77	-0.77	-0.72
	Bend II	β	0.35	0.36	0.3
Move front (nominal)	Front	γ	0.43	0.445	0.4
Move front (non-nom first time)	Front	γ	0.25	0.265	0.105
Move front (non-nom upcoming times)	Front	γ	0.37	0.38	0.3
Align	Rotational	θ	0.03	0.01	± 0.0425

7 Conclusions and Recommendations

7.1 Conclusions

The aim of this assignment was, in broad terms, to increase the level of autonomy of the PIRATE robot. The reasons for this are related to the fact that handling the robot manually makes the inspection labor-intensive and requires a specialized operator to control the robot.

The major issue relates to the robot's autonomy, nonetheless, this is a complex task, and it normally requires an elaborated system control architecture. By means of a layered control approach, several levels of autonomy for the PIRATE were defined, amongst which this project's scope was delimited to constructing the Sequential layer. In this way, the project sought to develop a software framework for this layer, which in general handles partially autonomous behaviors and corresponding robot feedback.

The design and implementation of the software framework and corresponding elements are presented in detail in Chapters 4 and 5, respectively. The development was subjected to tests in different scenarios (nominal and not-nominal) from which the results are discussed in Chapter 6. In synthesis, the most relevant findings were:

- **Message size and frequency limitation**: Because serial communication between the laptop and the Arduino Mega was used, a limitation to the size and frequency of the sent messages was necessary to avoid communication problems. Therefore the maximum array size that is transmitted at a time contains 13 integers (16-bit) and the maximum frequency for sending messages to the Arduino was set to a maximum of 10Hz (which is similar to the setpoint update rate from the Arduino Mega to the local ATmega boards). This resulted in a relatively stable communication.
- **Centralized system for message sharing**: One of the key aspects for the design was based on a centralized (composite) class. In this way coherency and consistency between the different operation modes (handled in individual classes) was achieved.
- **Client-server model**: Another relevant aspect of the design is the use of a client server model to separate the request and handling of the partially autonomous behaviors. In this way, the program that listened to the user interface's requests, could run in parallel with the server which executed the behavior. This allowed the desired non-blocking functionality, which meant that some requests of the user interface could be executed at the same time as a partially autonomous behavior.
- **Using the blackboard design pattern for sensor information**: Like in any other controlled system, feedback played a major role for the development of the PABs. Because of this, it was important that sensor information was available upon request, to avoid the need of constantly updating at moments when it is not required. To do this ROS provides a blackboard-like communication paradigm called *services*, where a server is constantly updating the variables, and can provide a client with the information upon request. This proved to be a very efficient method for distributing the sensors' feedback among different classes.
- **Motion primitives concept**: This concept was first introduced by Dertien in 2014 [1] and was observed to be a very good option for describing the steps to perform during a PAB. By thinking in motion primitives, the sequences were designed such that each state represented a motion primitive. This proved to be quite intuitive which made simpler to break down complex routines.
- **Using joint angles as feedback for the bend modules**: Given the fact that the bend modules are controlled in the sequences using torque, it made sense to use the torque measurements as feedback. This, however, proved to be hard to accomplish in practice. It was

found that the torque magnitudes varied greatly, that the signs changed sometimes and in general that they were not very consistent during the tests. This could be due to the deformation of the wheels, play in the mechanical parts, the lack of rigidity of the printed prototype, or overall lack of robustness of the torque controller, etc. Angle feedback was later tested and yielded better results and more repeatability than the torque, therefore this was chosen as a feedback for the bend modules.

- **Handling of non-nominal situations**: A robust partially autonomous behavior should be able to handle at least the most significant situations in which the robot can be. A good delimitation of the handled cases as well as the related assumptions, helped during the design and implementation process. In this way, during the design phase special attention was put to this situation, for which some additional states and new guards where integrated to the sequences.
- **Possibility to carry out missions**: In the same manner that the motion primitives were supposed to encapsulate small functions of the PIRATE, the PAB concept sought to enclose more complex sequential functionality that can be called and reused when needed by a supervisory (planning) software layer. Based on this and as a proof of concept, a simple mission was designed and implemented. During the testing, the mission was successfully completed.

In this study it was possible to see the importance of the software control layers approach, which greatly facilitated the design process and accomplished modularity and re-usability. Once the layers were defined, it was easier to limit each class' boundaries and functions. Because of the time limitation, the scope of the study was limited to the design and implementation of one PAB and just the basics of others (to be able to combine them into a mission). Nonetheless, the pattern followed to design the Clamp Straight PAB, can be reused and adjusted for the future creation of other PABs. A relevant point to discuss is that the present study encountered some limitations during implementation and testing. The limitations have to do mainly with the robot, which was found to be sometimes unstable and mechanically too fragile. This reduces the reliability of the developed software because of unexpected failures due to wiring problems, mechanical play, etc. In spite of this, the study was able to provide the first steps towards autonomous architecture for the PIRATE. It presents a suitable software framework for further development of other PABs in the Sequential layer, as well as potential for creating more complex missions by integrating them.

7.2 Recommendations

In general, the purpose of this study was to take the first steps towards more autonomous behaviors for the PIRATE. In that sense, the software framework presented offers a good base for future development. The centralized structure works very well in keeping the information consistent through all the different programs, so it is recommended to keep this approach. The modularity of each classes is also worth mentioning as it allows modifications to be very local, and to spread across the different levels. One example of this is that if the User interface is changed in the future, the parts like UI Input Driver and possibly the switch case inside the Mapper won't be applicable anymore. Nonetheless, all the logic starting from the Pirate Server class would be unaffected by the change, as long as the format for sending the goals is kept.

Even though the developed software works well as a first prototype of building more autonomous conditions there is still room for improvement. As the scope of this study aimed at the construction of only one PAB class, the design did not make use of the inheritance possibility for classes. For future design and implementation of the other PAB classes, it is recommended to take advantage of the inheritance and create a PAB parent class (super-class), and make specializations of for each of the PABs themselves. Another aspect that must be further evaluated is the necessary accuracy of the orientation guard (in straight position). The current

accuracy for this is around ±0.0425 radians. This sometimes showed to be a little too strict, which causes too many re-adjustments (longer sequence). This should be assessed, so that perhaps the tolerances can be widened.

In general, the communication between ROS and Arduino Mega worked acceptably, however there were times were the feedback would stop being sent from the Arduino Mega, and thus the sequences would fail. Another thing that happened less frequently was that when just starting the robot, it would start moving unrequested and uncontrollably. Tests with the oscilloscope did not show particularly obvious problems. Furthermore, the feedback from the Arduino, did not present any setpoints being sent from ROS. This suggests that it might be due to connector or wiring (hardware) problems. Such situation could bring problems in the future, so it is strongly suggested that more attention is put into improving the robustness of the wiring and checksums in the PICO boards. Due to the time limitation, thorough failure root cause analysis fell out of the scope of the present study.

In general the Arduino Mega microcontroller has some processing limitations that should be considered when having a wider scope for the autonomy of the PIRATE. When increasing the complexity on the system, there is a possibility that the communication through the Arduino might not be sufficient or that more failures like this would keep happening. It is, therefore, suggested to look into other options to communicate the laptop with the local controllers, possibly using a RTOS (Real-Time operating system).

During the design and implementation of the motion primitives, the torque measurements, which were the originally intended source of feedback for the bend modules, were found to be considerably inconsistent and could not be used. Theoretically, since the clamping process is based on exerting a force on the pipe wall, the best option for feedback should be to use torque measurements. For this reason, it is strongly suggested to pay attention to improving the reliability of the torque measurements and integrate these as guards along with the found angle guards. Having a combination of position and torque feedback as conditions should provide more repeatability and overall improve the logic's robustness.

Also, with the experience from implementing a very basic Drive PAB, it was seen that perhaps for exact measurements of distance and robot location, sensor fusion techniques should be implemented. This because, only the odometry or the velocity might not be enough to take into account possible movements that result from clamping, unclamping, and other PABs, etc.

In general for the PIRATE, the current prototype of the robot, fabricated using 3D printing, has some evident limitations. Repairs had to be frequent and this reduced a lot of time for the design and implementation, as well as decreasing the overall reliability of the developed software, because of external unexpected failures. It is, therefore, strongly recommended to move towards a more robust prototype, with less fragile and less deformable materials. This would reduce the possibilities to break and also the mechanical play present in the robot, which can highly affect the sensor measurements.

Finally, improving the visualization was also out of scope for the present study. Since nominal and non-nominal cases depend heavily on the orientation of the robot, it would be desired integrate the IMU data so that the visualization provides an accurate image of the orientation of the robot.

Acknowledgements

I would like to thank my amazing parents and brother for their wonderful support throughout this adventure. To my lovely grandfather and my aunt that witnessed this from heaven. To my wonderful family and friends, the new and the old, I am forever in your debt for all your incredible support. To Armando, who has been my rock for the last 8 years, thank you darling.

To all the wonderful people from RAM from which I learned much more than science.

Last but not least, this thesis wouldn't have been possible without the full support of the scholarship granted by the *Instituto de Innovacion y Transferencia de Tecnologia de Nuevo Leon (http://i2t2.org.mx)* together with the scholarship of *Consejo Nacional de Ciencia y Tecnologia (http://www.conacyt.gob.mx)*

Gisela A. Garza Morales
Enschede, August 2016

A Appendix: Designed classes

```
Setpoint
- nh_ : ros::Node
- setpoint_msg : PirateSetpoint_Array
- pub_setpoint : ros::Publisher
+ setElement(uint8 PICOboardID, uint8 Motor, int16 setpoint) : void
+ setArray(int16 setpoint_array[13]) : void
+ getElement(uint8 PICOboardID, uint8 Motor) : void
+ publish() : bool
```

Figure A.1: Setpoint class

```
Control
- nh_ : ros::Node
- control_msg : PirateControl_Array
- pub_control : ros::Publisher
+ setElement(uint8 PICOboardID, uint8 Motor, int16 controlMode) : void
+ setArray(int16 control_array[12]) : void
+ getElement(uint8 PICOboardID, uint8 Motor) : void
+ publish() : bool
```

Figure A.2: Control class

```
Limit
- nh_ : ros::Node
- limit_msg : PirateLimit_Array
- pub_limit : ros::Publisher
+ setElement(uint8 PICOboardID, uint8 Motor, int16 limit) : void
+ setArray(int16 control_array[4]) : void
+ getElement(uint8 PICOboardID, uint8 Motor) : void
+ publish() : bool
```

Figure A.3: Class diagram for Limit class

```
Pirate Manager
- nh : ros::Node
- setpoint_ : Setpoint&
- limit_ : Limit&
- control_ : Control&
- mapper : Mapper*
- server : Pirate_Server*
+ periodic_manual_publish() : void
```

Figure A.4: Pirate Manager class

```
Mapper
- nh : ros::Node
- setpoint_ : Setpoint&
- limit_ : Limit&
- control_ : Control&
- ac : Client
- sub_UI : ros::Subscriber
- pub_UI : ros::Publisher
- goal : PirateGoal
- inPAB : bool
- inManual : bool
- UIout : UIOutput
- activeGoal : bool
- goal_status : Goal_status_service
+ publisher() : void
- map(ui_msg : UI_msg&) : void
```

Figure A.5: Class diagram for Mapper

```
Pirate Server
- nh : ros::Node
- setpoint_ : Setpoint&
- limit_ : Limit&
- control_ : Control&
- motion_primitives : Motion_Primitives*
- as_ : PirateActionServer*
- state_ : PirateState
- server_state : Server_State
- clamp_straight : Clamp_Straight*
- feedback_ : PirateFeedback
- result_ : PirateResult
+ execute(goal : PirateGoal&) : void
- preemptCB() : void
- send_status(req : Request&, res : Response&) : void
- isNewGoalAvailable( req : Request&, res : Response&)
```

Figure A.6: Class diagram for Pirate Server

```
Motion Primitives
- nh : ros::Node
- setpoint_ : Setpoint&
- limit_ : Limit&
- control_ : Control&
- joint_client : JointClient
- status_client_mp : StatusClient
+ moveBFModule(uint8 moduleID, int16 setpoint) : void
+ changeControlMode(uint8 moduleID, uint8 control_mode) : void
+ changeLimitBFModule(uint8 moduleID, uint16 limit) : void
+ driveVel(int velocity) : bool
+ rotateRad(double radians, ros::Duration timeout) : bool
+ clampRear(uint8 pipeDiam, ros::Duration timeout) : bool
+ unclampPartialRear(uint8 pipeDiam, ros::Duration timeout) : bool
+ unclampFullRear(ros::Duration timeout) : bool
+ clampFront(uint8 pipeDiam, ros::Duration timeout) : bool
+ unclampPartialFront(uint8 pipeDiam, ros::Duration timeout) : bool
+ unclampFullFront(ros::Duration timeout) : bool
```

Figure A.7: Motion Primitives Class

```
Clamp Straight
- nh_ : ros::Node
- setpoint_ : Setpoint&
- control_: Control&
- limit_ : Limit&
- motion_primitives_ : Motion_Primitives&
- joint_client : JointClient
- status_client : StatusClient
- imu_client : IMUClient
+ state_ : ClampStraightState
+ execute(uint8_t pipe_diameter) : bool - checkCase(void) : int
- checkOrientation(void) : double
- isAligned(void) : bool
- isStraight(void) : bool
- isNominal(void) : bool
- getJointPosition(int8 jointElement) : double
```

Figure A.8: Clamp Straight class

```
Home
- nh_ : ros::Node
- setpoint_ : Setpoint&
- control_: Control&
- limit_ : Limit&
- motion_primitives_ : Motion_Primitives&
- joint_client : JointClient
- status_client : StatusClient
- imu_client : IMUClient
+ state_ : HomeState
+ execute() : bool
- getJointPosition(int8 jointElement) : double
```

Figure A.9: Home class

```
Drive
- nh_ : ros::Node
- setpoint_ : Setpoint&
- control_: Control&
- limit_ : Limit&
- motion_primitives_ : Motion_Primitives&
+ state_ : DriveState
+ execute(int16 distance) : bool
```

Figure A.10: Drive class

B Appendix: Additional mapping information

B.1 Pirate State array mapping per PICO Board

Table B.1: Pirate state sent from the Arduino Mega per PICO Board

PICO Board ID	***Element number***								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
20	Odometry	Velocity	Drive Load	Angle (Bend)	Bend Load	Torque	-	-	-
21	Odometry	Velocity	Drive Load	Angle (Bend)	Bend Load	Torque	-	-	-
22	Odometry	Velocity	Drive Load	Angle (Bend)	Bend Load	Torque	-	-	-
23	Odometry	Velocity	Drive Load	Angle (Bend)	Bend Load	Torque	-	-	-
24	Odometry	Velocity	Drive Load	Angle (Bend)	Bend Load	Torque	-	-	-
25	Odometry	Velocity	Drive Load	X (IMU)	Y (IMU)	Z (IMU)	U (IMU)	V (IMU)	W (IMU)
26	Odometry	Velocity	Drive Load	Angle (Bend)	Bend Load	Torque	-	-	-
27	Odometry	Velocity	Drive Load	Angle (Bend)	Bend Load	Torque	-	-	-
28	Odometry	Velocity	Drive Load	Angle (Bend)	Bend Load	Torque	-	-	-

C Appendix: Feedback screen

C.1 Mapper class implementation

```
[ INFO] [1471536754.240824619]: Current operation mode is MANUAL mode
[ INFO] [1471536766.828252821]: Current operation mode is PAB mode
```

Figure C.1: Feedback screen when changing operation mode

C.2 Pirate Server class implementation

```
gisela@gisela-MacBookPro:~/catkin_ws$ rosrun pirate pirate_manager_node
[ INFO] [1471604573.518467232]: SETPOINT: Setpoint class constructed
[ INFO] [1471604573.519285980]: Control class constructed
[ INFO] [1471604573.520089577]: Limit class constructed
[ INFO] [1471604573.520807627]: MANAGER: Creating Pirate Manager
[ INFO] [1471604573.520852388]: MOTION PRIMITIVES: Creating Motion Primitives cl
ass instance
[ INFO] [1471604573.520908723]: MOTION PRIMITIVES: Service joint state establish
ed.
[ INFO] [1471604573.521024142]: SERVER: Action Server is being constructed
[ INFO] [1471604573.531037359]: SERVER: Action Server is now running
[ INFO] [1471604573.531095112]: SERVER: Creating Motion Primitives object
[ INFO] [1471604573.531130729]: Creating Clamp Straight class instance
[ INFO] [1471604573.531162876]: Service joint state, server status, and imu esta
blished.
[ INFO] [1471604573.531210149]: SERVER: Creating actions Clamp_Straight object
[ INFO] [1471604573.542145832]: MAPPER: Action server started. Mapper created.
```

Figure C.2: Feedback screen at start-up, showing the construction of the classes

```
gisela@gisela-MacBookPro:~/catkin_ws$ rosrun pirate pirate_manager_node
[ INFO] [1471604573.518467232]: SETPOINT: Setpoint class constructed
[ INFO] [1471604573.519285980]: Control class constructed
[ INFO] [1471604573.520089577]: Limit class constructed
[ INFO] [1471604573.520807627]: MANAGER: Creating Pirate Manager
[ INFO] [1471604573.520852388]: MOTION PRIMITIVES: Creating Motion Primitives cl
ass instance
[ INFO] [1471604573.520908723]: MOTION PRIMITIVES: Service joint state establish
ed.
[ INFO] [1471604573.521024142]: SERVER: Action Server is being constructed
[ INFO] [1471604573.531037359]: SERVER: Action Server is now running
[ INFO] [1471604573.531095112]: SERVER: Creating Motion Primitives object
[ INFO] [1471604573.531130729]: Creating Clamp Straight class instance
[ INFO] [1471604573.531162876]: Service joint state, server status, and imu esta
blished.
[ INFO] [1471604573.531210149]: SERVER: Creating actions Clamp_Straight object
[ INFO] [1471604573.542145832]: MAPPER: Action server started. Mapper created.
[ INFO] [1471608414.847109074]: Current operation mode is PAB mode
[ INFO] [1471608422.147266293]: Pirate: Executing, creating sequence for pirate with 90 mm diameter
[ INFO] [1471608422.147366658]: Clamping Straight
[ INFO] [1471608422.147405768]: Pirate: Executing, creating sequence for pirate with 90 mm diameter
[ INFO] [1471608422.148094788]: Goal just went active
[ INFO] [1471608422.251470770]: Changing control mode for module 21, 22, 23, 26 and 27.
[ INFO] [1471608422.251597360]: Control changed? 1
[ INFO] [1471608422.347070029]: CHECKING_ORIENTATION: Checking if we are in a norminal start position.
[ INFO] [1471608422.379759467]: Clamp case is: 0
[ INFO] [1471608422.380078377]: CHECKING_ORIENTATION: Moving front
[ INFO] [1471608422.470357094]: MOVING_FRONT: Moving front to prepare for clamping
[ INFO] [1471608425.647039341]: Pirate Preempted
[ INFO] [1471608425.647933447]: Goal finished.
[ INFO] [1471608425.648102707]: Finished in state [PREEMPTED]
[ INFO] [1471608425.648183357]: Success =: [0]
[ INFO] [1471608425.648572747]: Server was preempted. Cancelling clamp straight routine
Pirate has succeeded? 0
Value of success is 0
```

Figure C.3: Feedback screen when cancelling a PAB operation.

C.3 Motion Primitives class implementation

```
gisela@gisela-MacBookPro: ~/catkin_ws   /home/gisela/catkin_ws/src/pirate_viz/launch/Manager.launch http://localhost:11311   gisela@gis
bash: cd: catkin_ws: No such file or directory
gisela@gisela-MacBookPro:~/catkin_ws$ rosrun pirate pirate_manager_node
[ INFO] [1471536441.398306427]: SETPOINT: Setpoint class constructed
[ INFO] [1471536441.398982588]: Control class constructed
[ INFO] [1471536441.399625739]: Limit class constructed
[ INFO] [1471536441.400189797]: MANAGER: Creating Pirate Manager
[ INFO] [1471536441.400222564]: MOTION PRIMITIVES: Creating Motion Primitives class instance
[ INFO] [1471536441.400245378]: MOTION PRIMITIVES: Service joint state established.
[ INFO] [1471536441.400323378]: SERVER: Action Server is being constructed
[ INFO] [1471536441.409736707]: SERVER: Action Server is now running
[ INFO] [1471536441.409775218]: SERVER: Creating Motion Primitives object
[ INFO] [1471536441.409804894]: Creating Clamp Straight class instance
[ INFO] [1471536441.409832484]: Service joint state, server status, and imu established.
[ INFO] [1471536441.409873587]: SERVER: Creating actions Clamp_Straight object
[ INFO] [1471536441.421113502]: MAPPER: Action server started. Mapper created.
[ INFO] [1471536672.126649028]: Current operation mode is PAB mode
[ INFO] [1471536680.726874508]: Pirate: Executing, creating sequence for pirate with 90 mm diameter
[ INFO] [1471536680.727076438]: Clamping Straight
[ INFO] [1471536680.727225569]: Pirate: Executing, creating sequence for pirate with 90 mm diameter
[ INFO] [1471536680.727559464]: Goal just went active
[ INFO] [1471536680.853273845]: Changing control mode for module 21, 22, 23, 26 and 27.
[ INFO] [1471536680.853418485]: Control changed? 1
[ INFO] [1471536680.944596897]: CHECKING_ORIENTATION: Checking if we are in a norminal start position.
[ INFO] [1471536680.948632128]: Clamp case is: 0
[ INFO] [1471536680.948859848]: CHECKING_ORIENTATION: Moving front
[ INFO] [1471536681.040920581]: MOVING_FRONT: Moving front to prepare for clamping
[ INFO] [1471536684.551639289]: MOVING_FRONT: Done moving front in 90 mm
[ INFO] [1471536684.641427685]: CLAMPING_REAR_FULL: Clamping rear full
[ERROR] [1471536699.729968581]: MP_CLAMP_REAR: Function clampRear timed out.
[ERROR] [1471536699.730147841]: CLAMPING_REAR_FULL: Couldn't clamp rear part of the robot. Please check the conditions. Cancelling sequence.
Pirate has succeeded? 0
Value of success is 0
[ERROR] [1471536699.730434166]: The requested goal could not be completed. Goal is Aborted.
[ INFO] [1471536699.730803276]: Goal finished.
[ INFO] [1471536699.730907541]: Finished in state [ABORTED]
[ INFO] [1471536699.730967521]: Success =: [0]
```

Figure C.4: Timeout during clamp routine

C.4 Clamp straight class implementation

```
[ INFO] [1471538922.257859252]: MP_CLAMP_FRONT: Current Position 22: -0.420180 and 23: 0.081929
[ INFO] [1471538922.357985687]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538922.358207822]: MP_CLAMP_FRONT: Current Position 22: -0.458662 and 23: 0.121616
[ INFO] [1471538922.456455902]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538922.457213152]: MP_CLAMP_FRONT: Current Position 22: -0.488567 and 23: 0.157558
[ INFO] [1471538922.551656921]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538922.551976106]: MP_CLAMP_FRONT: Current Position 22: -0.504900 and 23: 0.178382
[ INFO] [1471538922.644383888]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538922.644614528]: MP_CLAMP_FRONT: Current Position 22: -0.519258 and 23: 0.202072
[ INFO] [1471538922.744737844]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538922.744959399]: MP_CLAMP_FRONT: Current Position 22: -0.521328 and 23: 0.205488
[ INFO] [1471538922.840756648]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538922.840848318]: MP_CLAMP_FRONT: Current Position 22: -0.521328 and 23: 0.205488
[ INFO] [1471538922.935785296]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538922.935909706]: MP_CLAMP_FRONT: Current Position 22: -0.521328 and 23: 0.205488
[ INFO] [1471538923.034601746]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538923.034939796]: MP_CLAMP_FRONT: Current Position 22: -0.581566 and 23: 0.228801
[ INFO] [1471538923.132285521]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538923.132656926]: MP_CLAMP_FRONT: Current Position 22: -0.617990 and 23: 0.238755
[ INFO] [1471538923.231895371]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538923.232333641]: MP_CLAMP_FRONT: Current Position 22: -0.651367 and 23: 0.270387
[ INFO] [1471538923.332130177]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538923.332277167]: MP_CLAMP_FRONT: Current Position 22: -0.703131 and 23: 0.308380
[ INFO] [1471538923.431610372]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538923.431739752]: MP_CLAMP_FRONT: Current Position 22: -0.715033 and 23: 0.317096
[ INFO] [1471538923.557156493]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538923.557524108]: MP_CLAMP_FRONT: Current Position 22: -0.763775 and 23: 0.345438
[ INFO] [1471538923.654597153]: MP_CLAMP_FRONT: Setpoints are for 22 -0.770000 and 23 0.350000
[ INFO] [1471538923.654717343]: MP_CLAMP_FRONT: Current Position 22: -0.808415 and 23: 0.356629
[ INFO] [1471538923.655362823]: CLAMPING_FRONT_FULL: Successfully clamped front in 90 mm pipe
[ INFO] [1471538923.666534807]: CLAMPING_FRONT_FULL: CONGRATULATIONS! Finished clamping straight sequence with 90 mm diameter.
Pirate has succeeded? 1
Value of success is 1
[ INFO] [1471538923.667140813]: Succeeded
[ INFO] [1471538923.667708063]: Goal finished.
[ INFO] [1471538923.667798413]: Finished in state [SUCCEEDED]
[ INFO] [1471538923.667859133]: Success =: [1]
```

Figure C.5: Feedback screen of the implementation showing a successfully completed clamp routine

```
[ INFO] [1471536813.147620700]: CLAMPING_FRONT_FULL: Successfully clamped front in 90 mm pipe
[ INFO] [1471536813.152747997]: CLAMPING_FRONT_FULL: Orientation of rear module is still not correct. Going to unclampr rear
[ INFO] [1471536813.232940600]: UNCLAMPING_REAR: Unclamping rear
[ INFO] [1471536815.161441986]: UNCLAMPING_REAR: Unclamping rear done. Going to rotating rear.
[ INFO] [1471536815.248966512]: ROTATING_REAR: Rotating rear
[ INFO] [1471536815.260149651]: ROTATING_REAR: Rear and front part of the robot are aligned, proceeding with rotation.
[ INFO] [1471536815.260513161]: ROTATING_REAR: Checking orientation for rotating!
[ INFO] [1471536815.265122943]: ROTATING_REAR: Orientation is: 111.105169
[ INFO] [1471536815.265309443]: ROTATING_REAR: Difference in degrees is: -21.105169
[ INFO] [1471536815.265412438]: ROTATING_REAR: Desired orientation is 90.000000
[ INFO] [1471536815.352174379]: ROTATE_RAD: Position is 0.000971
[ INFO] [1471536815.352565084]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536815.448198017]: ROTATE_RAD: Position is 0.000971
[ INFO] [1471536815.448276408]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536815.541685475]: ROTATE_RAD: Position is 0.000971
[ INFO] [1471536815.541884015]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536815.638840225]: ROTATE_RAD: Position is 0.000971
[ INFO] [1471536815.639261915]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536815.733892138]: ROTATE_RAD: Position is -0.023281
[ INFO] [1471536815.734390698]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536815.831955738]: ROTATE_RAD: Position is -0.093748
[ INFO] [1471536815.832165603]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536815.932127194]: ROTATE_RAD: Position is -0.123618
[ INFO] [1471536815.932220614]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536816.030254403]: ROTATE_RAD: Position is -0.167410
[ INFO] [1471536816.030356633]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536816.131020689]: ROTATE_RAD: Position is -0.201624
[ INFO] [1471536816.131146999]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536816.232169060]: ROTATE_RAD: Position is -0.272647
[ INFO] [1471536816.232310625]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536816.355746300]: ROTATE_RAD: Position is -0.331023
[ INFO] [1471536816.355964240]: ROTATE_RAD: Tolerances are down -0.418355 and up -0.318355
[ INFO] [1471536816.356130230]: ROTATE_RAD: We have completed the desired rotation.
[ INFO] [1471536816.356285291]: ROTATING_REAR: Sending rotation setpoint of: -0.368355 radians
[ INFO] [1471536816.356476911]: ROTATING_REAR: We have completed the desired rotation of the rear part of the robot.
[ INFO] [1471536816.361738948]: ROTATING_REAR: Going to Clamping rear full...
[ INFO] [1471536816.449945799]: CLAMPING_REAR_FULL: Clamping rear full
[ INFO] [1471536819.239198549]: CLAMPING_REAR_FULL: Successfully clamped rear in 90 mm pipe
[ INFO] [1471536819.242883206]: CLAMPING_REAR_FULL: Orientation of rear module is still not correct. Continuing...
[ INFO] [1471536819.329104330]: UNCLAMPING_FRONT: Unclamping front
[ INFO] [1471536821.155637131]: UNCLAMPING_FRONT: Unclamping front done.
[ INFO] [1471536821.249005874]: ROTATING_FRONT: Rotating front
[ INFO] [1471536821.262050169]: ROTATING_FRONT: Aligning front to rear parts.
[ INFO] [1471536821.349593284]: MP_ALIGN: Checking if there is misalignment.
[ INFO] [1471536821.350857555]: Tolerances are up 0.042500 and down -0.042500
[ INFO] [1471536821.352144420]: Position is now: -0.374733
[ INFO] [1471536821.352765551]: MP_ALIGN: There is misalignment. Sending zero setpoint to front part for alignment
[ INFO] [1471536821.444909378]: MP_ALIGN: Checking if there is misalignment.
[ INFO] [1471536821.445021843]: Tolerances are up 0.042500 and down -0.042500
[ INFO] [1471536821.445092628]: Position is now: -0.374735
[ INFO] [1471536821.542525378]: MP_ALIGN: Checking if there is misalignment.
[ INFO] [1471536821.542638533]: Tolerances are up 0.042500 and down -0.042500
[ INFO] [1471536821.542711548]: Position is now: -0.374736
[ INFO] [1471536821.637531031]: MP_ALIGN: Checking if there is misalignment.
[ INFO] [1471536821.637640626]: Tolerances are up 0.042500 and down -0.042500
[ INFO] [1471536821.637693791]: Position is now: -0.374736
[ INFO] [1471536821.734817731]: MP_ALIGN: Checking if there is misalignment.
[ INFO] [1471536821.734938086]: Tolerances are up 0.042500 and down -0.042500
[ INFO] [1471536821.735011386]: Position is now: -0.374736
[ INFO] [1471536821.832597700]: MP_ALIGN: Checking if there is misalignment.
[ INFO] [1471536821.833224075]: Tolerances are up 0.042500 and down -0.042500
[ INFO] [1471536821.833742006]: Position is now: -0.374736
[ INFO] [1471536821.930278015]: MP_ALIGN: Checking if there is misalignment.
```

Figure C.6: Feedback screen's messages during rotations

D Appendix: Transition conditions and Results

D.1 Transition conditions for Clamp Straight state machine

Table D.1: Transition guards for the Clamp Straight state machine

State	***Transition***	***Transition conditions (guards)***
Change control modes	Moving front	*Nominal case*
	Clamp rear partial	*Non- nominal case*
Moving Front	Clamping rear full	*Nominal case*
	Clamping front Partial	*Non- nominal case*
Clamping Rear Full	Clamping front full	*Front not clamped AND nominal case*
	Unclamping front	*Front clamped AND Rear part not Straight*
	Rotating front	*Front not clamped AND non-nominal case*
	END	*Front clamped AND Rear part Straight AND Robot Aligned*
Clamping Front Full	Unclamping Rear	*Rear clamped AND Rear part not Straight*
	Clamping Rear Full	*Rear part not clamped AND Nominal case*
	Unclamping Front	*Rear clamped AND Rear part Straight AND Robot not Aligned*
	Clamping Rear Partial	*Rear part not clamped AND Non-nominal case*
	END	*Rear clamped AND Rear part Straight AND Robot Aligned*
Unclamping Rear	Rotating rear	*N/A*
Rotating Rear	Clamping rear full	*Scenario 1:Front clamped AND Rear not clamped AND Rear part Straight Scenario 2: Front clamped AND Rear not clamped AND Robot is Aligned AND Nominal case*
	Clamping rear Partial	*Front clamped AND Rear not clamped AND Robot is Aligned AND Non-nominal case*
Unclamping Front	Rotating Front	*N/A*
Rotating Front	Clamping Front Full	*Scenario 1: Rear clamped AND Front not clamped AND Rear part Straight Scenario 2: Rear clamped AND Front not clamped AND Robot is Aligned AND Nominal case*
	Clamping Front Partial	*Rear clamped AND Front not clamped AND Robot is Aligned AND Non-nominal case AND Front module lifted*
	Moving Front	*Rear clamped AND Front not clamped AND Robot is Aligned AND Non-nominal case AND Front module not lifted*
Clamping Front Partial	Unclamping Rear	*Rear clamped*
	Rotating Rear	*Rear not clamped*
	Moving Front	*First time clamping front*
Clamping Rear Partial	Unclamping Front	*Front clamped*
	Rotating Front	*Front not clamped*

D.2 Additional graphs for nominal re-adjust clamp straight in 110mm pipe

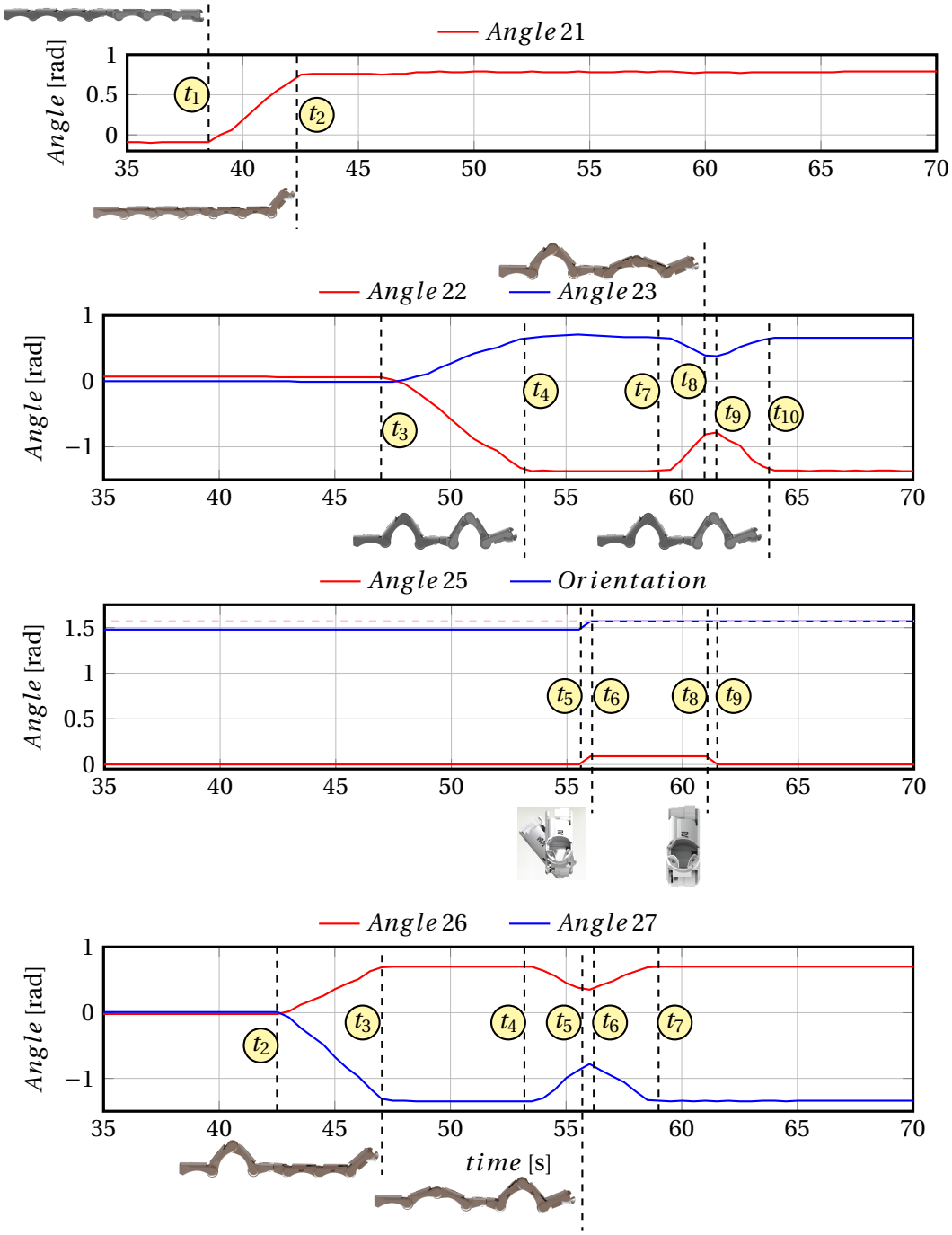


Figure D.1: Results showing the guards for the Nominal re-adjust Clamp Straight in 110mm pipe

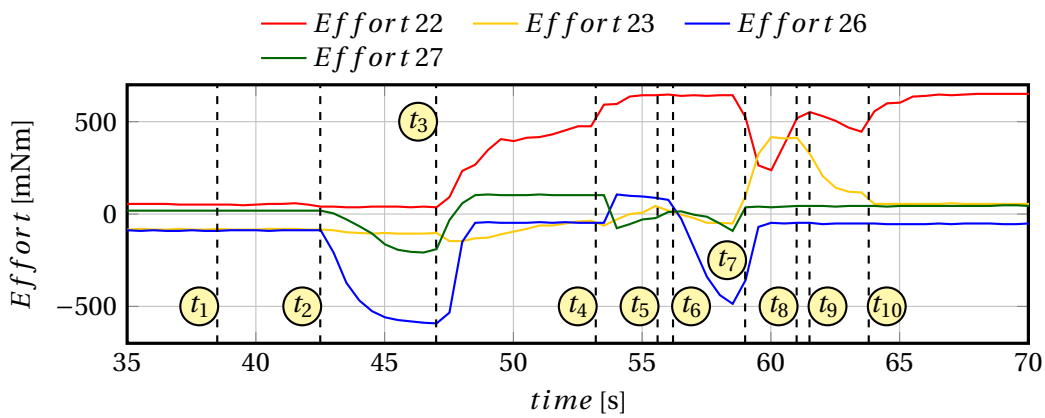


Figure D.2: Effort measurements for bend modules for the nominal re-adjust clamp straight in 110mm pipe.

D.3 Graphic sequence of non-nominal clamp straight

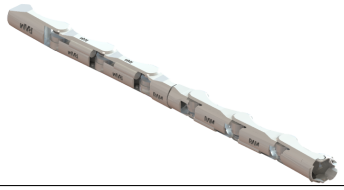
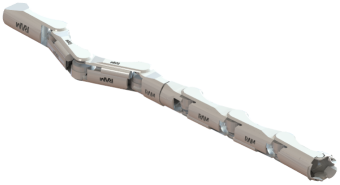
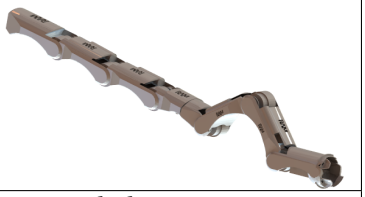
Lying on one side non-nominal case t_0 	Clamp Rear Partial t_1 	Rotating Front t_2 
Moving Front t_3 t_5 	Clamp Front Partial t_4 t_{11} 	Unclamping Rear t_6 
Rotating Rear t_7 	Clamp Rear Partial t_8 	Unclamping Front t_9 t_{15} 
Rotating Front t_{10} t_{16} 	Unclamping Rear t_{12} 	Rotating Rear t_{13} 
Clamp Rear Full t_{14} 	Clamp Front Full t_{17} 	Succeeded t_{18} 

Table D.2: Non-nominal case graphical sequence for a 90mm pipe

E Appendix: Mapping of the user interface

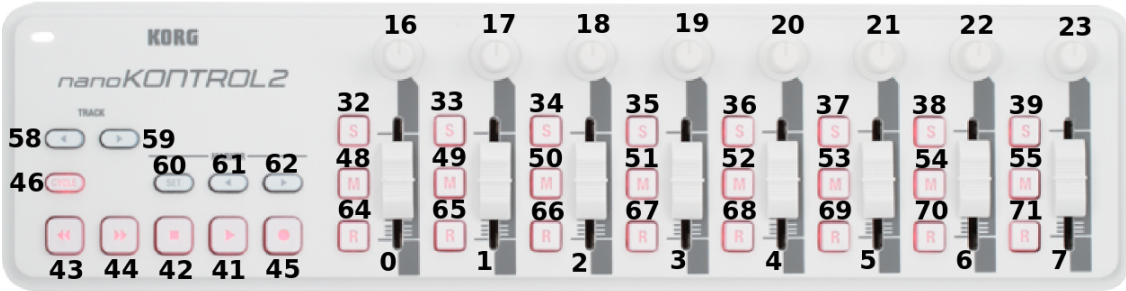


Figure E.1: Mapping of the user interface

Table E.1: Mapping per channel in MIDI panel

Channel	Description	Works in Operation Mode
0	Slider Rear Module	Not controlled
1	Slider Bend IV	Manual
2	Slider Bend III	Manual
3	Slider Bend II	Manual
4	Slider Bend I	Manual
5	Slider Front Module	Manual
6	Slider Tilt Front Camera	Manual / PAB
7	Slider Drive (wheels)	Manual
16	Rear LEDs	Manual / PAB
17	Knob Limits Bend IV	Manual
18	Knob Limits Bend III	Manual
19	Knob Limits Bend II	Manual
20	Knob Limits Bend I	Manual
21	Knob Pan Front Camera	Manual / PAB
22	Front LEDs	Manual / PAB
23	Knob Rotational Module	Manual
32	Not used (Control mode Rear module)	Manual
33	Control modes Bend IV (PWM - off / Torque - on)	Manual
34	Control modes Bend III (PWM - off / Torque - on)	Manual
35	Control modes Bend II (PWM - off / Torque - on)	Manual
36	Control modes Bend I (PWM - off / Torque - on)	Manual
37	Control modes Front M. (PWM - off / Position - on)	Manual
38	Control modes Tilt F. Camera (PWM - off / Position - on)	Manual / PAB
39	Control modes Drive (PWM)	Manual

Table E.2: Mapping per channel in MIDI panel - Cont.

Channel	Description	Works in Operation Mode
41	Start (Play)	Manual / PAB
42	Stop	Manual / PAB
43	Change Mode of Operation (Manual - off / PAB - on)	Manual / PAB
44	Not used	-
45	Used for calibration	Manual
46	Not used	-
48	Not used	-
49	Immediate Stop Bend IV	Manual
50	Immediate Stop Bend III	Manual
51	Immediate Stop Bend II	Manual
52	Immediate Stop Bend I	Manual
53	Immediate Stop Front M.	Manual
54	Immediate Stop Tilt Camera	Manual
55	Immediate Stop Drive	Manual
58	Not used (can be used for other PABs)	PAB
59	PAB Clamp Straight in 90mm	PAB
60	For calibration	Manual
61	Not used	-
62	Not used	-
64	Not used	-
65	Calibrate Bend IV	Manual
66	Calibrate Bend III	Manual
67	Calibrate Bend II	Manual
68	Calibrate Bend I	Manual
69	Calibrate Front Module	Manual
70	Calibrate Tilt Front Camera	Manual
71	Not used	-

F Appendix: Running instructions and Tips and tricks

F.1 Instructions to start the software

1. Run the *Manager.Launch* file, that starts the rosserial node, the UI input and output nodes. To run input the following to the terminal (within the catkin workspace):

```
roslaunch pirate_viz Manager.launch
```

2. Run the *pirate_manager_node*, by inputting the following to the terminal (within the catkin workspace):

```
rosrun pirate pirate_manager_node
```

3. Run the pirate state node and the visualization contained in the *pirate_manager_node* by inputting the following to the terminal (within the catkin workspace):

```
roslaunch pirate_viz pirate_gui.launch
```

In this software, the feedback screen is observed at the *pirate manager node*.

F.2 Sending Manual Commands

1. Once the system is running, press the *Start (Play)* button until the LED lights on. By default the system starts in Manual mode.
2. If the system is previously in PAB mode, first press the *Stop* button and then the *REWIND* button and toggle it until the LEDs are off. This means the system is in Manual mode (also check the feedback screen which will indicate the current mode of operation). Once it is correctly set, press again the *Start* button to be able to send commands again.
3. Once in manual mode, all the buttons, sliders and knobs mentioned in Table E.1 from Appendix E.
4. To stop at any time press the *Stop* button, or to individually stop each of the joints, press the corresponding *M*-button.
5. To change the control mode for each joint, press the *S*-button for the corresponding one. Normally PWM is when the LED is off, and the other mode when it is on.
6. To change the limits of the bending modules, move the corresponding knobs.
7. To turn on the LEDs also move the corresponding knobs.
8. To calibrate press the *Set* button and hold it while pressing the corresponding *R*-button of the desired joint to calibrate.
9. There is a special mode called *No Limits All* which overrides the current bending limits. This is called when pressing the *Set* button and holding it while pressing the *Rec* button.

F.3 Sending PAB commands

1. Press *Start (Play)*, to start the system.
2. Since the system is by default in Manual, first press the *Stop* button. Next press the *Rewind* button and toggle it until the LED turns on. This means that the system is in PAB mode. Once it has been successfully changed, press *Start* again to be able to send PAB commands (or the allowed manual commands in PAB mode).
3. The only PAB command developed in this thesis is mapped to the *Track Left* button. This one calls the Clamp Straight 90mm.
4. To cancel the PAB execution at any time, press the *Stop* button.

F.4 Tips and tricks

F.4.1 Required steps to be able to program PICO Boards

1. Install libusb (check latest version):

```
sudo apt-get install libusb-1.0-0-dev
```

2. Install avrdude (check latest version):

```
sudo apt-get update
sudo apt-get install avrdude
```

3. Check first if avrdude is working by using:

```
sudo avrdude -p m328 -c avrispmkII
```

4. If the previous instruction does not work without sudo, means there are administrator rights pending. Just add the user to the user group dialout using this link.
5. Create a file *avrisp.rules* using:

```
sudo nano /etc/udev/avrisp.rules
```

   and put the content similar to these instructions .
6. Modify the *boards.txt* in the Arduino folder in `/opt/` to add the Pirate boards (supplied as backup to this report, originally created by Reiling). 7. Add the Programs to the Sketchbook by adding the folders at the same level as libraries in the `user/home/Arduino/` folder. 8. Make sure the correct PICO board is selected and the correct programmer is selected. 9. To upload the PICO use menu *Sketch* → *Upload* using programmer the programmer.

F.4.2 Required instructions to program Pirate Bay (Arduino Mega) board

1. For programming the Arduino Mega, add libraries to Arduino looking for *"usb host 2.0"* and manually the other one which is supplied as backup (called *usb midi*). Do this by adding the .zip in *Sketch* → *Include Library* → *Add .zip library* (Sketch → Manage Libraries for the first one).
2. Select the board Mega ADK (and make sure the port is selected correctly, for example: *ttyACM0*)
3. Use menu *Sketch* → *Upload*.

F.4.3 Rules for the Arduino (standardized naming)

1. Copy the file *72-pirateViz.rules* in the PIRATE laptopn from *Documents* to `/etc/udev/rules.d` with the following command:

```
cd  /Documents/udev rules for pirate/sudo cp
    72-pirateViz.rules /etc/udev/rules.d
```

   Âă 2. This will name the ARDUINO MEGA ADK always as *"arduino0"* and the video cameras, as *"video0"* and *"video1"*.

F.5 Installing pygame (for Midi Drivers)

Install pygame (check latest version) via:

```
sudo apt-get install python-pygame
```

F.5.1 General tips and tricks

1. If when running the first time the Manager.launch file, there is an error, run it again. This happens sometimes when starting up.
2. If there is a communication error, either that the Arduino Mega is not sending the feedback, restart the system.
3. If one of the sensors/efforts is not responding (zero-valued or any value that doesn't change), restart the system.

Bibliography

[1] E. Dertien, "Design of an inspection robot for small diameter distribution mains. phd thesis," *University of Twente*, 2014. [Online]. Available: http://eprints.eemcs.utwente.nl/25317/01/Design_of_an_inspection_robot_for_small_diameter_gas_distribution_mains.pdf

[2] P. van Vollenhoven, "Gasexplosie na breuk van gasdistributieleiding. technical report," Raad voor de transportveiligheid, Tech. Rep., 2002.

[3] E. Dertien, "System specifications for pirate," University of Twente, Enschede, Tech. Rep., 2006.

[4] J. V. op Nijhuis, "Development of a pipe inspection robot. master thesis," *University of Twente*, 2007.

[5] S. Spijksma, "Redesign of the pirate bending module. internship report," University of Twente-DEMCON, Enschede, Tech. Rep., 2009.

[6] B. Burkink, "Pirate propulsion unit. bachelor thesis," *University of Twente*, 2009.

[7] D. Borgerink, "Development of the second pirate prototype using rapid prototyping. master thesis," *University of Twente*, 2012.

[8] E. Dertien, "Design of pirate mainboard," University of Twente, Enschede, Tech. Rep., 2007.

[9] J. Ansink, "Electronic design for a gas pipe inspection robot. internship report," Demcon, Enschede, Tech. Rep., 2007.

[10] H. Reemeijer, "Control of a pipe inspection robot. master thesis," *University of Twente*, 2010. [Online]. Available: https://www.ram.ewi.utwente.nl/aigaion/attachments/single/1129

[11] E. Drost, "Measurement system for pipe profiling. master thesis," *University of Twente*, 2009. [Online]. Available: https://www.ram.ewi.utwente.nl/aigaion/attachments/single/1018

[12] M. Brilman, "Embedded vision system for pirate. bachelor thesis," *Saxion - University of Twente*, 2011.

[13] M. Reiling, "Implementation of a monocular structured light vision system for pipe inspection robot pirate. master thesis," *University of Twente*, 2014.

[14] T. Mennink, "Self localization of pirate inside a partially structured environment. master thesis," *University of Twente*, 2010. [Online]. Available: https://www.ram.ewi.utwente.nl/aigaion/attachments/single/1140

[15] C. Doggen, "Wireless state feedback and control of a pipe inspection robot. master thesis," *University of Twente*, 2010.

[16] J. M. O'Kane, *A Gentle Introduction to ROS*. University of South Carolina, 2016.

[17] L. Joseph, *Mastering ROS for Robotics Programming*. Packt Publishing Ltd., 2015.

[18] B. P. Douglass, *Doing Hard Time: Developing Real-Time systems with UML, objects, frameworks, and patterns*. Addison Wesley Longman, Inc., 1999.

[19] D. Bell. The class diagram. [Online]. Available: https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

[20] R. Hartanto, *A Hybrid Deliberative Layer for Robotic Agents*. Springer-Verlag Berlin Heidelberg, 2011.

[21] J. K. et al., "Intelligent robot software architecture," *Recent Progress in Robotics*, 2008.

[22] E. Gat, "On three-layer architectures," *California Institute of Technology*, 1998.

[23] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, 1986.

[24] J. F. Broenink, M. A. Groothuis, P. M. Visser, and B. Orlic, "A model-driven approach to embedded control system implementation," in *Proceedings of the 2007 Western Multiconference on Computer Simulation WMC 2007, San Diego*, J. Anderson and R. Huntsinger, Eds. San Diego: SCS, San Diego, January 2007, pp. 137–144.

[25] S. Bennet, *Real-Time Computer Control: An Introduction*. Prentice-Hall, 1988.

[26] J. F. Broenink and Y. Ni, "Model-driven robot-software design using integrated models and co-simulation," in *Proceedings of SAMOS XII*, J. McAllister and S. Bhattacharyya, Eds., Jul. 2012, pp. 339 – 344.

[27] M. M. Bezemer, "Cyber-physical systems software development - way of working and tool suite," Ph.D. dissertation, University of Twente, Nov. 2013.

[28] K. P. Edwin Dertien, Mohammad Mozzafari Foumashi and S. Stramigioli, "Development of an inspection robot for small diameter gas distribution mains," *International Gas union Research Conference, IGRC 2014*, 2014. [Online]. Available: https://www.ram.ewi.utwente.nl/e13/files/pirate/articles/WP3-4_Dertien-1.pdf

[29] G. A. Pratt and M. M. Williamson, "Series elastic actuators," *MIT Artificial Intelligent Laboratory*, 1995.

[30] M. Lootsma, "Design of the global software structure and controller framework for the 3tu soccer robot. master thesis," *University of Twente*, 2008. [Online]. Available: http://essay.utwente.nl/58246/1/scriptie_M_Lootsma%5B1%5D.pdf