UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI TECNICA E GESTIONE DEI SISTEMI INDUSTRIALI
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA MECCATRONICA

TESI DI LAUREA MAGISTRALE

DEVELOPMENT OF THE COMMUNICATION SYSTEM FOR A LOWER LIMB HUMAN HEXOSKELETON USING THE ROS MIDDLEWARE

Relatore: Monica Reggiani
Correlatore: Elena Ceseracciu

Laureando: Marco Matteo Bassa
1058764-IMC

ANNO ACCADEMICO: 2014-15

ABSTRACT

Human exoskeletons are complex systems whose analysis and control requires the interaction of a large number of sensors, devices and consequently the integration of heterogeneous programs. Being these devices under development, and consequently often reconfigured, a flexible, modular way of interfacing the software building the control architecture is required. In this text the problem is faced using the Robot Operating System (ROS) middleware. In particular is presented the implementation of a communication system for a human exoskeleton developed by the BioMot Project. This work does not include the full-development of the middleware based structure, but describes the beginning of its implementation from both a software and a hardware point of view. The created communication interface is useful for data acquisition in the preliminary experiments and for the future control of the system. The interface between the middleware and the exoskeleton's hardware is made trough a BeagleBone Black board; the problematic of the set-up of this board and its interface with the other elements of the system is also presented.

SOMMARIO

Gli esoscheletri per umani sono sistemi complessi, il loro controllo richiede l'interazione di un grande numero di sensori, dispositivi e conseguentemente l'integrazione di programmi eterogenei. Essendo questi dispositivi in continuo sviluppo e quindi soggetti a continue riconfigurazioni, è necessario utilizzare un modo flessibile e modulare per interfacciare i software che compongono l'architettura di controllo. In questo testo il problema è affrontato utilizzando il Robot Operating System (ROS) middleware. In particolare è presentata la implementazione di un sistema di comunicazione per un esoscheletro umano sviluppato nel progetto BioMot. Questo lavoro non copre l'intera realizzazione della struttura basata sul middleware, ma descrive l'inizio della sua implementazione sia da un punto di vista hardware che da un punto di vista software. L'interfaccia di comunicazione creata è funzionale sia all'acquisizione di dati durante gli esperimenti preliminari sul sistema, sia al futuro controllo del robot. Per interfacciare l'hardware dell'esoscheletro con l'architettura middleware è stata utilizzata una Beagle Bone Black board; anche la problematica dell'installazione di questa scheda sull'esoscheletro e il suo interfacciamento con gli altri elementi del sistema sono presentati.

Live as if you were to die tomorrow.
Learn as if you were to live forever.-Mahatma Ghandi

ACKNOWLEDGEMENTS

I wish to thank my family, and particularly my parents, for all the support and the motivation transmitted during this work.

I thank Monica Reggiani and Elena Ceseracciu for the supervision, the numerous advices and the patient revision of my code and of this text.

My gratitude also goes to all my coworkers from the Cajal institute, in particular to Guillermo Asín Prieto and to Juan C. Moreno that wisely supervised my work and to Andrea Órtiz, that continuously provided me with Colombian coffee.

Finally, I want to thank all the people that made my period in Madrid not only a great professional experience, but also a fantastic life experience.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 WEARABLE ROBOTS AND EXOSKELETONS

Wearable robots (WR) are person-oriented devices, usually in the form of exoskeletons. These devices are worn by human operators to enhance or support a daily function, such as walking. WRs find applications in the enhancement of intact operators or in clinical environments, e.g. rehabilitation of gait function in neurologically injured patients. Current WRs are extra body structures inducing fixed motion patterns on its user. Most advanced WRs for human locomotion still fail to provide the real-time adaptability and flexibility presented by humans when confronted with natural perturbations, due to voluntary control or environmental constraints. The driving motivation behind the increasing funding to stimulate innovation in Wearable robots for the medical field can be easily tracked to the need of industrialized countries to reduce the economic burden on the health care system of an aging population and a low birth rate. Since this work is focused on exoskeletons, the terms "Wearable robots" and "Exoskeletons" will be considered equivalent from now on.

Some examples of already-existing (powered) exoskeletons are:

- HULC[1](Human Universal Load Carrier), developed by Professor H. Kazerooni and his team at Ekso Bionics, is intended to help soldiers in combat to carry a load of up to 200 pounds at a top speed of 10 miles per hour for extended periods of time.

1 http://www.lockheedmartin.com/us/products/hulc.html]

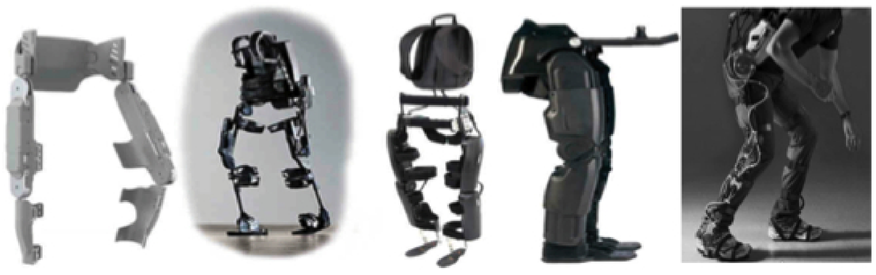


Figure 1: Examples of human exoskeletons

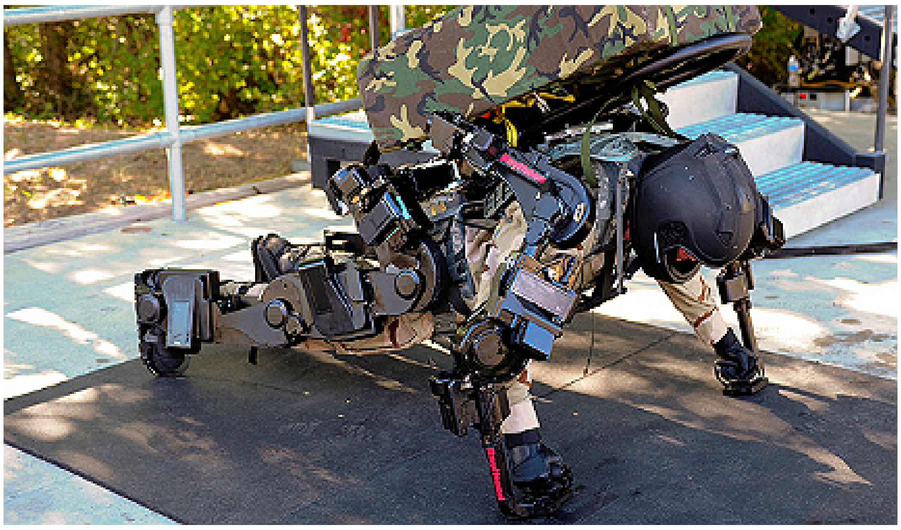


Figure 2: The XOS2 exoskeleton

- XOS 2 [2](Figure 2), a suit developed by DARPA, allows the users to lift heavy objects at an actual-to-perceived-weight ratio of 17:1.
- HAL exoskeleton [3], is a powered suit developed by Japan's Tsukuba University and the robotics company Cyberdyne. It has been designed to support and expand the physical capabilities of its users, particularly people with physical disabilities.
- X1[3] Robotic Exoskeleton, developed by NASA, should help astronauts stay healthier in space maintaining their muscular tone and at the same time can be used for assisting paraplegics walking.

A big number of the developed devices are aimed for enhancing the performance of healthy users, mainly for military applications, allowing to carry/lift an high amount of weight. A detailed analysis of these devices is out of the scopes of this work, but a state of the art for lower-limb exoskeletons intended both for assistance and rehabilitation, can be found in [57]. This document makes a review of the current known actuation techniques: Hydraulic or pneumatic actuators, electrical motors, SEA (Series Elastic Actuators), Pneumatic Muscle Actuators. The possible control algorithms are then classified according to the human-robot interaction mode: the signals collected from the human body(EMG, EEG and muscle hardness), the interaction force signals measured between the human and the exoskeleton, and the signals that are only collected from the exoskeletons or active orthoses. The possible performance assessment methods for com-

2 http://www.army-technology.com/projects/raytheon-xos-2-exoskeleton-us/
3 http://www.nasa.gov/offices/oct/home/feature_exoskeleton.html

paring different exoskeleton solutions are then described, in particular: Metabolic Cost, Gait Biomechanical Analysis and Muscle Activity Analysis.

The review points out the limitations of current exoskeletons, in particular the deviation between the wearers' real motion intention and that estimated by actual human-robot interfaces, still leads to an increase in the wearers' metabolic cost and, in conjunction with actual biomechanical solutions, introduces certain constraints on their motion. Hence, the accuracy of current human-robot interfaces and human biomechanics and mechanism designs needs to be further optimized.

1.2 THE BIOMOT PROJECT

The work presented in this thesis is part of the BioMot project, an internetional project that involves eight institutions from five countries (Spain,Italy,Belgium,Iceland and Japan). The aim of BioMot is "to improve the efficiency in the management of human-robot interaction in overground gait exoskeletons by means of mixture of bioinspired control, actuation and learning approaches" [39]. The objective is to show how the embodiment of bioinspired and architectural mechanisms can allow a user to conveniently alter the behaviour of WRs for walking. The final goal of the project is to deliver novel ambulatory wearable exoskeleton technology that exploits neuronal control and learning mechanisms and provides more energy efficient cooperative (human-robot) performance and adaptive assistance based on the user's residual and voluntary action. These systems will not be designed and will not be applied for military applications.

BioMot's exoskeletons will apply adaptive assistance as a function of real-time estimation of human effort provided by a detailed neuromusculoskeletal model that computes neuromuscular activity (surface electromyography, EMG) to predict joint moments and hence prescribe the exoskeleton function. Gait detection algorithms based on human performance (brain signals, EEG) and embedded sensors (kinematic and kinetic) are developed for decision making, handling transitions or volitional changes in the task (such as gait speed). Local reflex-based joint controllers are designed to allow for automatic adaptation when confronting changes in the interaction. At the physical level, intrinsically compliant actuators are developed to exploit natural dynamics of movement, orchestrated by the control system for economy and stability. A global learning scheme modules joint compliance as a function of gait efficiency and semantic signals infered from user demand.

These things together will form a cognitive system for the exoskeleton, which will be able to processes biomechanical and electrophysiological signals to adaptively assist the human movement exploiting the natural dynamics of overground walking.

To reach these goals, a system composed by a potentially high number of sensors and devices needs to be created and consequently a communication architecture allowing the interaction and the synchronization of different programs has to be developed.

1.3 ROLE AND STRUCTURE OF THIS WORK

This work faces the problem of interfacing the large number of heterogeneous software that forms the exoskeleton's environment with a structured communication architecture. The problem is solved using the *Robot Operating System* (ROS) middleware. ROS is an open-source, meta-operating system for robots. It provides services including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers [13]. This frame is becoming more and more popular in robotic application and is already being employed by a large number of complex robots [16].

The use of a middleware allows to avoid the definition of a large amount of point to point communication channels (for example socket connections), that would lead to a complex and not flexible communication structure. In ROS, such connections are handled automatically and can be changed dynamically, making the connection of software developed by different groups a lot easier and the general structure more elegant.

This thesis presents the required steps for the application of the ROS middleware to an already existing human exoskeleton: the H2 exoskeleton. The development is intended to be flexible in order to make the middleware adaptable to a newer exoskeleton that is under development by the BioMot's group (BioMot exoskeleton); a part of the code is also intended to be compatible with other robots, as the H2R humanoid robot, developed by other groups collaborating with BioMot. An insight of the integration of the newer actuation systems (MACCEPA actuators) in the middleware architecture will also be provided.

The text begins describing the hardware and software building the H2 exoskeleton system and the adaptations required for the application of the middleware architecture, in particular the set-up of a Beagle Bone Black board (chapter 2) and its communication with the already existing hardware. The features provided by the ROS middleware to the communication architecture and the steps required for its set-up are described in chapter 3. In chapter 4, the set-up of a CAN communication library on the Beagle Bone Black, the ROS nodes designed to acquire data from the exoskeleton and those for sending command messages, are described. The results of an acquisition experiment, made with a walking exoskeleton, are reported in chapter 5. A preliminary interface between the newly designed MACCEPA actuators and the ROS middleware is presented in chapter 6. Finally, chapter 7 describes the problematic of interfacing the middleware with the future hi-level controller of the system; it was in particular investigated the possibility of applying the ROS-Control framework to exoskeletons.

2 H2 EXOSKELETON SYSTEM

2.1 INTRODUCTION

The system in which the exoskeleton works is not only made by the exoskeleton itself, but also by auxiliary sensors and computation systems. These are necessary for making experiments in order to better understand the human-robot interaction, the human gait process and to elaborate new high-level control strategies. Following is a description of the elements actually forming the system; newer elements might be added in the future.

2.2 THE H2 EXOSKELETON

In order to conduct experiments and gather information about the development of symbiotic performance of subject-specific robotic units, the H2 exoskeleton was selected by the BioMot group as a testing device [41]. The H2 exoskeleton is a lower limb exoskeleton designed for rehabilitation of adult people between 1.50 and 1.90 m in height, with a maximum body weight of 100 kg, such as stroke or SCI (Spinal Cord Injury) patients following neurological insults. It is conceived for assist-as-needed over ground gait and balance training in a clinical environment. It is also useful for maintaining the muscular tone of the lower limbs and to activate the circulation in persons that have lost mobility in the legs or suffer from muscle weakness. The hardware was built by "Technaid S.L.", who has an exclusive license for the design, manufacturing and commercial explotatioin of the system, while CSIC is the proprietary of the Know-How rights.

2.2.1 *Mechanical description*

"The exoskeleton is a wearable device (about 10 kg) with six degrees of freedom, in which hip, knee and ankle are powered joints. Front and back pictures of the structure can be seen in figure 3. Aluminium and stainless steel are primarily used in the mechanical structure in consideration of mechanical resistance and lightweight. The exoskeleton frame has bilateral uprights for the thigh and the shank, hinged hip, knee and ankles and articulated footplates (distally) and waist area (proximally). The mechanical structure is designed to allow active and passive movements in the sagittal plane. In the frontal plane, small passive movements are possible in hip joint. The joints range of motion is mechanically limited to avoid damages to patients. The maximum values are: 100 deg. flexion, 20 deg. extension for the hip; 100 deg. flexion and 0 deg. extension for the knee; and 20 deg. flexion and 20 deg. extension for the ankle. For the ankle, plantar-flexion is shown as extension and dorsi-flexion as flexion. The length of the thigh and the shank can be adjusted by a mechanism of two telescopic bars that are pushed one inside the other and are fixed in different positions. The same mechanism is used to change the position of the foot relative to the exoskeleton's ankle. The size and positions of adjustable rounded carriers with Velcro straps allowed for customization to individual requirements. Each joint is equipped with a brush-less DC motor, coupled to a Harmonic Drive gearbox. The design and selection of the actuators was based on typical identification of torque and power of each joint during normal gait at normal speed. This motor has a rated voltage of 24 V (DC) and nominal torque of 220 mNm. Furthermore, a gearbox is coupled to the motor shaft in order to reduce speed and increase torque. Harmonic Drive gearboxes are selected to reduce the weight and size of the final actuators. A Harmonic Drive with gear ratio of 160:1 gives to each joint a continuous net torque of 35 Nm. In H2 an average torque of 35 Nm for the hip actuator is presumed to be adequate enough for most patients. Energy is provided by a Lithium-ion battery pack of 22.5 VDC nominal and 6.8 Ah to power the exoskeleton. The battery pack weighs 960 grams and measures 180 x 70 x 40 mm. The H2 actuator technology for wearable robotic gait exoskeletons will constitute the reference standard to test the performance that is achieved with the novel compliant actuators developed in BioMot's WP2"[41].

2.2.2 *Electronic description*

The electronic part of the exoskeleton is constituted by an arm-based H2-HAL board, 6 H2-Joint boards and a CAN-bus network connecting the boards. The arm board (Figure 4) is in charge of executing a real-time control of all the joints; it interacts with H2-Joint boards acquiring sensors information and controlling the actuators. The small size of this board (only 56 x 44 mm) and its very low power consumption, allows it to be placed on the exoskeleton structure, reducing the bulk, complexity and difficulty of wiring, in addition to minimize connections. Moreover, it eliminates the need of a backpack been carried by the user. The board can access two CAN-bus lines: one for reading information and sending commands to the joints and the other for extern communication.The computational power of the board relies on a STMicroelectronis ARM microcontroller STM32F405RG running at 168 MHz. The joint-boards (Figure 5) are in charge of data acquisition from the different joint's sensors: angular position, interaction torque, joint speed and foot-ground contact. H2-Joint boards contain all the circuitry of the analog filters for each joint sensor and also the ampli-

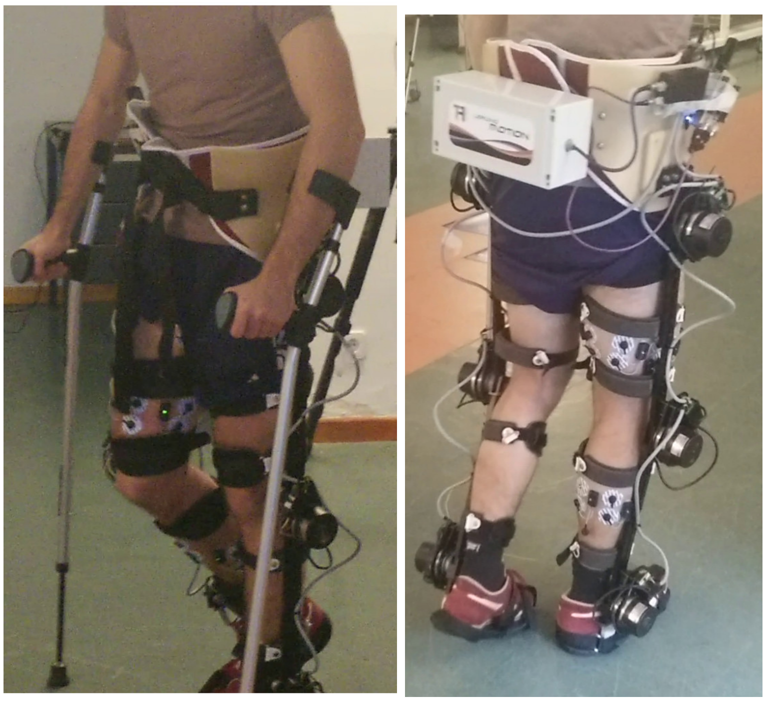


Figure 3: The H2 exoskeleton from the front and the back side

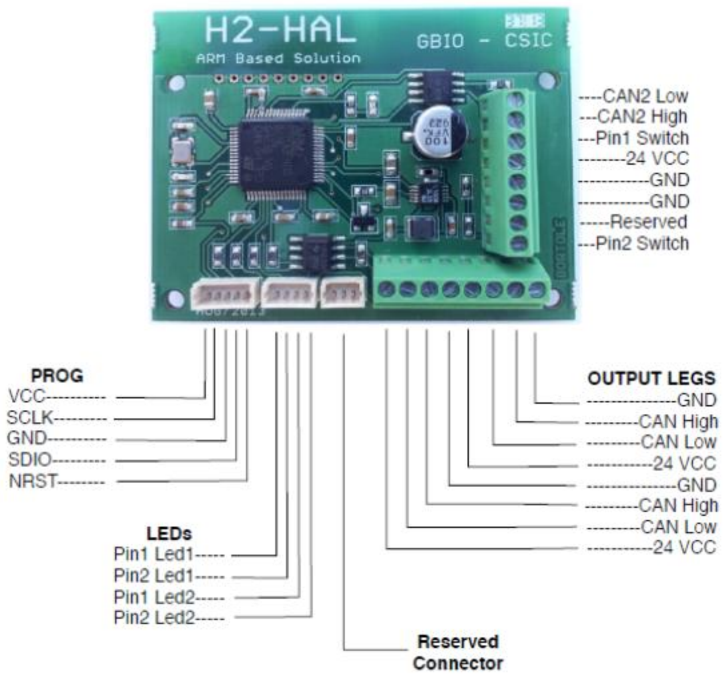


Figure 4: H2 ARM board description

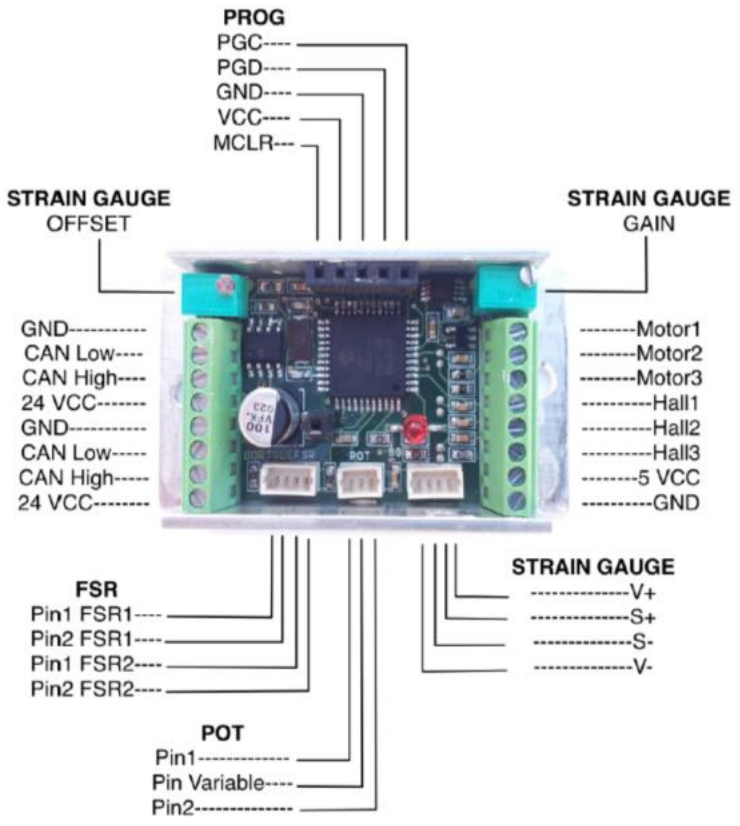


Figure 5: H2 Joint-board description

fiers for the strain gauges. After filtering and amplifying processes, the signals are digitalized by a DSP microcontroller (DsPIC30F4011). A small data packet of six bytes aggregates the sensor's information on each joint and these data are sent to H2-ARM controller every one millisecond. The boards also include a specifically designed driver for the brushless motors.

These parts together form the so called HAL (Hardware Abstraction Layer). The main processor runs an algorithm that receives commands to control each joint independently. Three types of control are possible for each joint: position, stiffness and torque. The commands are received either by Wi-Fi or CAN. Moreover, H2-ARM runs an algorithm for walking, based on a pre-recorded gait pattern.

2.2.3 *Communication and user interface*

Three ways of communication are available for the H2-ARM board: bluetooth, wi-fi and CAN-bus.

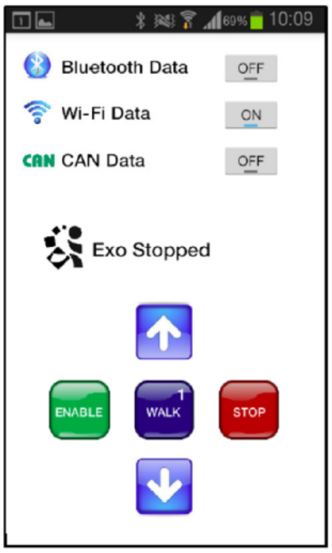


Figure 6: Android user interface for bluetooth comunication with H2

2.2.3.1 *Bluetooth communication*

The gait can be controlled using Bluetooth commands; in particular byte commands from 0 to 10 can stop or start the replication at different speeds. The decimal value 11 disables all H2 joints. Values from 13 to 18 start or stop sending data via Bluetooth, Wi-Fi and/or CAN. When sending data with bluetooth is activated, H2 will send 5 bytes every 500 milliseconds regarding its internal data (gait speed value, type of control, battery voltage).

Figure 6 shows the user interface of an app that can be installed on a smartphone running Android OS. With 5 main buttons the user can control the gait speed in 10 different levels. Other 3 secondary buttons are used to start or stop the data stream sent by H2 via Bluetooth, Wi-Fi or CAN. Before trying to command H2 by Bluetooth, the user has to pair H2 Bluetooth interface with the smartphone (or laptop, etc.). This interface can be used by the therapist in order to adjust the exoskeleton performance to the user's ability.

2.2.3.2 *Wi-fi communication*

H2 also has a wi-fi port for communication with other devices like for example a laptop. When turned on, H2 creates an open Wi-Fi spot named H2H (or H2M). Any computer connected to this network can send/receive data via UDP to/from H2. Each motor can be specifying a position, torque or stiffness control in a UDP message. The data sent by H2 if wi-fi communication is activated are : Right hip angle, Right knee angle, Right ankle angle, Left hip angle, Left knee angle, Left ankle angle, Right hip torque, Right knee torque, Right ankle torque, Left hip torque, Left knee torque, Left ankle torque, Right

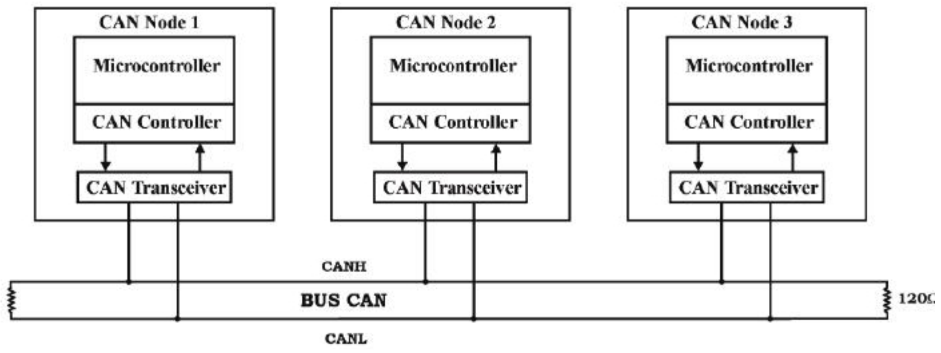


Figure 7: CAN bus configuration scheme

heel foot switch, Right toe foot switch, Left heel foot switch, Left toe foot switch, Battery voltage.

The wi-fi module has proved to be unreliable for both reception and sending of messages; consequently the CAN communication is preferable for control-tasks.

2.2.3.3 *CAN communication*

Since CAN bus is the chosen communication interface between the H2-arm board and the later described Beagle Bone Black, a brief description of its working principles is given.

CAN bus (for controller area network) is a vehicle bus standard designed to allow micro-controllers and devices to communicate with each other within a vehicle without a host computer. CAN bus is a message-based protocol, designed specifically for automotive applications, but is now also used in many other applications, particularly in automatic and robotic. "The CAN bus was developed by BOSCH as a multi-master, message broadcast system that specifies a maximum signaling rate of 1 megabit per second(bps). Unlike a traditional network such as USB or Ethernet, CAN does not send large blocks of data point-to-point from node A to node B under the supervision of a central bus master. In a CAN network, many short messages like temperature or RPM are broadcast to the entire network, which provides for data consistency in every node of the system" [21]. The complexity of the node can range from a simple I/O device up to an embedded computer with a CAN interface and sophisticated software.

Each node requires a:

- **Central processing unit**, microprocessor, or host processor (decides what the received messages mean and what messages it wants to transmit).
- **CAN controller**, often an integral part of the microcontroller, it stores the received serial bits from the bus until an entire mes-

sage is available, which can then be fetched by the host processor; when sending it transmits the bits serially onto the bus if it is free from other communications.

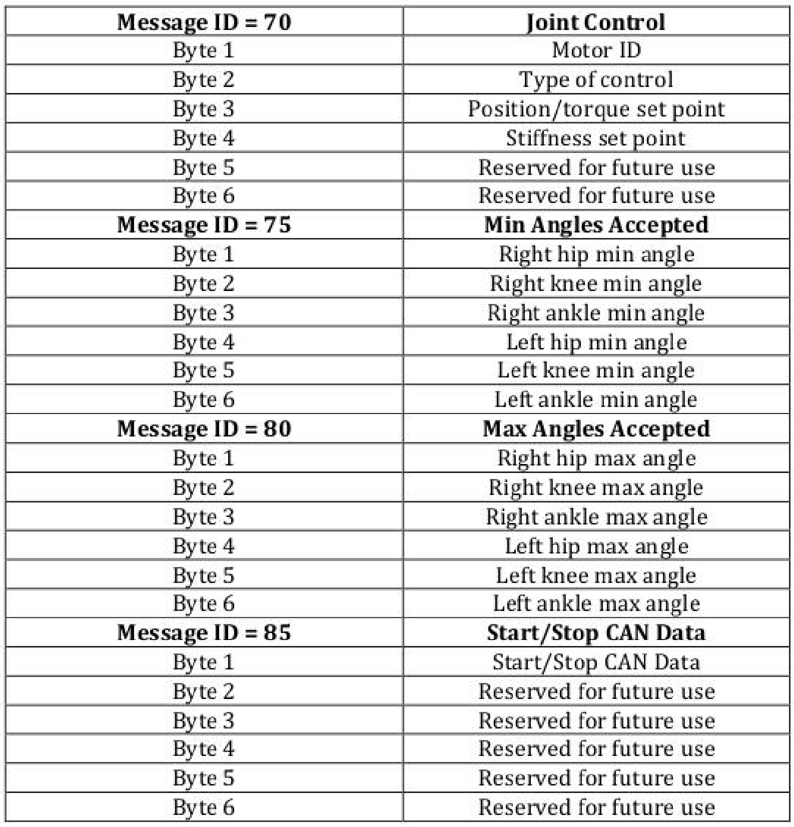
- **CAN transceiver**, it converts the data stream from CANbus levels to levels that the CAN controller uses. It usually has protective circuitry to protect the CAN controller. When transmitting it converts the data stream from the CAN controller to CANbus levels.

The H2-arm board is equipped with all these elements.
"A message or Frame consists primarily of the ID (identifier), which represents the priority of the message (but can also be used for other purposes), and up to eight data bytes. A CRC, acknowledge slot [ACK] and other overhead are also part of the message. The message is transmitted serially onto the bus using a non-return-to-zero (NRZ) format and may be received by all nodes. The network is flexible in terms of configuration, is highly immune to electrical interference, automatically avoids data collision and corrects errors regarding to data packets transmission."

"H2 CAN port is intended to communicate with external devices in real time. The bus speed is set to 1Mbps and it accepts messages in standard format with packets of 6 bytes. Figure 8 summarizes accepted commands and their functions. Byte values are in decimal format.

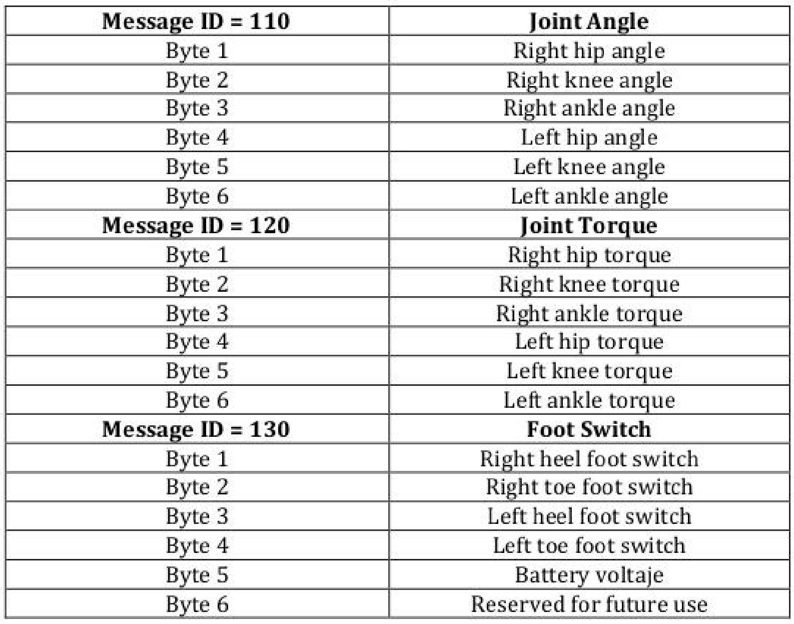
The command Joint Control can be used to control each one of the six joints independently. Motor ID values are: 1 = Right Hip; 2 = Right Knee; 3 = Right Ankle; 4 = Left Hip; 5 = Left Knee; 6 = Left Ankle. For type of control, it has the following values: 1 = Position control; 2 = Stiffness control; 3 = Torque control; 4 = Motors disabled; 5 = Motors stopped. When Position Control is used, byte 3 is the set point for that joint and bytes 4, 5 and 6 are not used. For Torque Control, byte 3 is the set point for that joint and bytes 4, 5 and 6 are not used. Stiffness Control uses byte 3 as the set point for position and byte 4 as the percentage of stiffness for that joint (where the value 0 means no stiffness and the value 100 means the maximum possible stiffness). The commands Min Angles Accepted and Max Angles Accepted can be used to set the minimum and maximum angles accepted as set point for Position Control. The command Start/Stop CAN Data is used to start or stop sending data via CAN (byte 1 = 1 starts data; byte 1 = 0 stops data). If CAN data has been started, H2 will send every 10 milliseconds 3 messages with 6 bytes each, regarding its internal data. Figure 9 summarizes the data sent by H2-arm.

The whole communication architecture (HAL) is described in figure 10. Each communication cycle in the network protocol involves passing a message from the H2-ARM node to all H2-Joint nodes in the network. As the message travels through the bus, each H2-Joint



Message ID = 70	**Joint Control**
Byte 1	Motor ID
Byte 2	Type of control
Byte 3	Position/torque set point
Byte 4	Stiffness set point
Byte 5	Reserved for future use
Byte 6	Reserved for future use
Message ID = 75	**Min Angles Accepted**
Byte 1	Right hip min angle
Byte 2	Right knee min angle
Byte 3	Right ankle min angle
Byte 4	Left hip min angle
Byte 5	Left knee min angle
Byte 6	Left ankle min angle
Message ID = 80	**Max Angles Accepted**
Byte 1	Right hip max angle
Byte 2	Right knee max angle
Byte 3	Right ankle max angle
Byte 4	Left hip max angle
Byte 5	Left knee max angle
Byte 6	Left ankle max angle
Message ID = 85	**Start/Stop CAN Data**
Byte 1	Start/Stop CAN Data
Byte 2	Reserved for future use
Byte 3	Reserved for future use
Byte 4	Reserved for future use
Byte 5	Reserved for future use
Byte 6	Reserved for future use

Figure 8: CAN messages accepted by H2-arm



Message ID = 110	**Joint Angle**
Byte 1	Right hip angle
Byte 2	Right knee angle
Byte 3	Right ankle angle
Byte 4	Left hip angle
Byte 5	Left knee angle
Byte 6	Left ankle angle
Message ID = 120	**Joint Torque**
Byte 1	Right hip torque
Byte 2	Right knee torque
Byte 3	Right ankle torque
Byte 4	Left hip torque
Byte 5	Left knee torque
Byte 6	Left ankle torque
Message ID = 130	**Foot Switch**
Byte 1	Right heel foot switch
Byte 2	Right toe foot switch
Byte 3	Left heel foot switch
Byte 4	Left toe foot switch
Byte 5	Battery voltaje
Byte 6	Reserved for future use

Figure 9: CAN messages sent by H2-arm

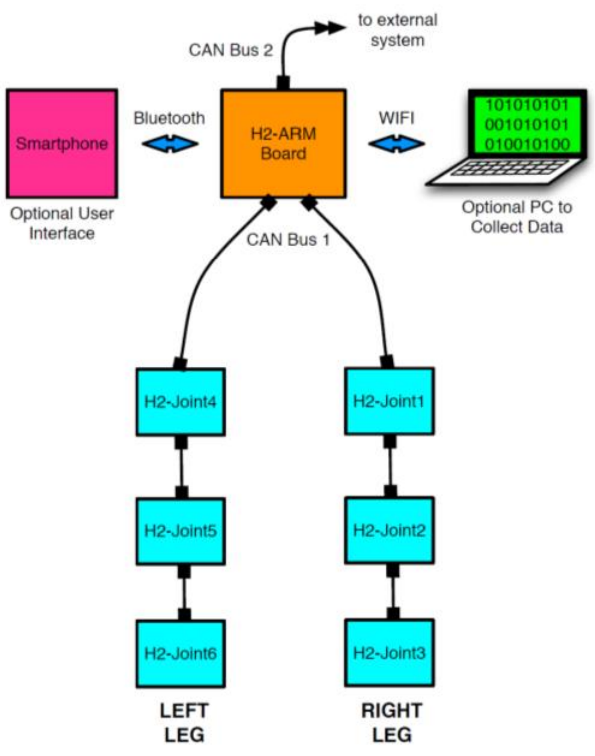


Figure 10: H2's communication archiecture

reads its assigned actuator command data byte (by looking for its own ID number and message byte sequence). Then, each H2-Joint returns one message back to H2-ARM node with its locally collected sensor's data. Because the communication cycles occur at a fixed rate (1 kHz) set by the control scheme, this protocol allows for deterministic control. Also, it provides built-in network error detection because at every message received, each H2-Joint has to return data information to H2-ARM. In this way, H2-ARM has a strong way to determine the integrity of the network and the correct operation of the joint's actuators"[41].

2.3 THE BEAGLEBONEBLACK

BeagleBoneBlack (figure 11) is a low-cost (at time of writing about 50 euros), open source, community-supported development platform for ARM Cortex-A8 processor developers and hobbyists. It runs a linux arm-distribution which by default is a Debian OS, but can be changed to an Ubuntu, Angstrom, Fedora distribution or with an Android OS.

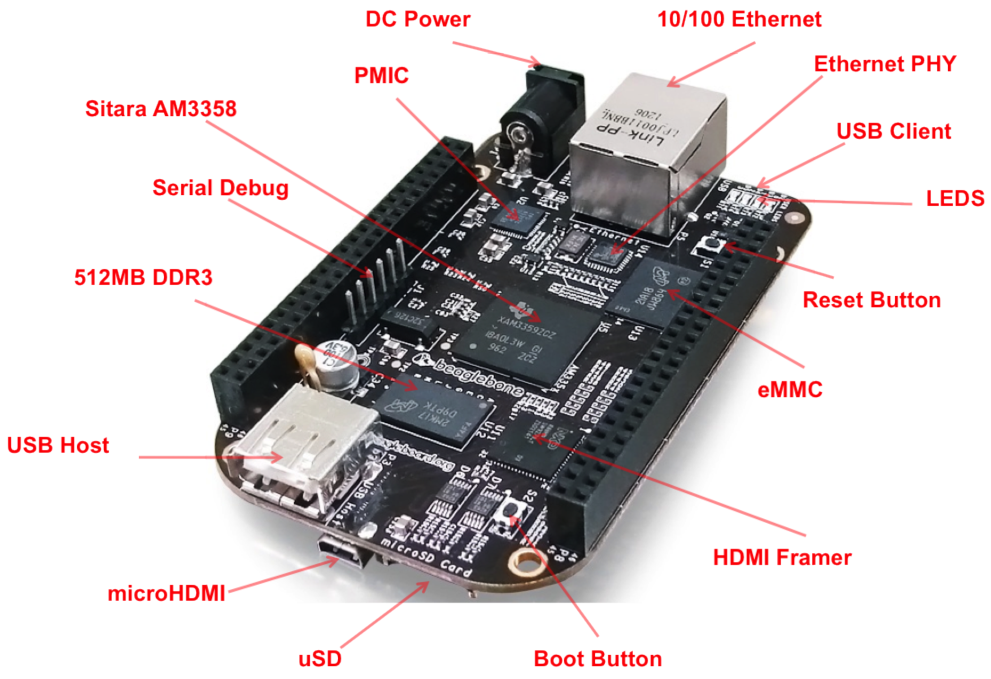


Figure 11: The BeagleBoneBlack board

2.3.1 *Features of the board*

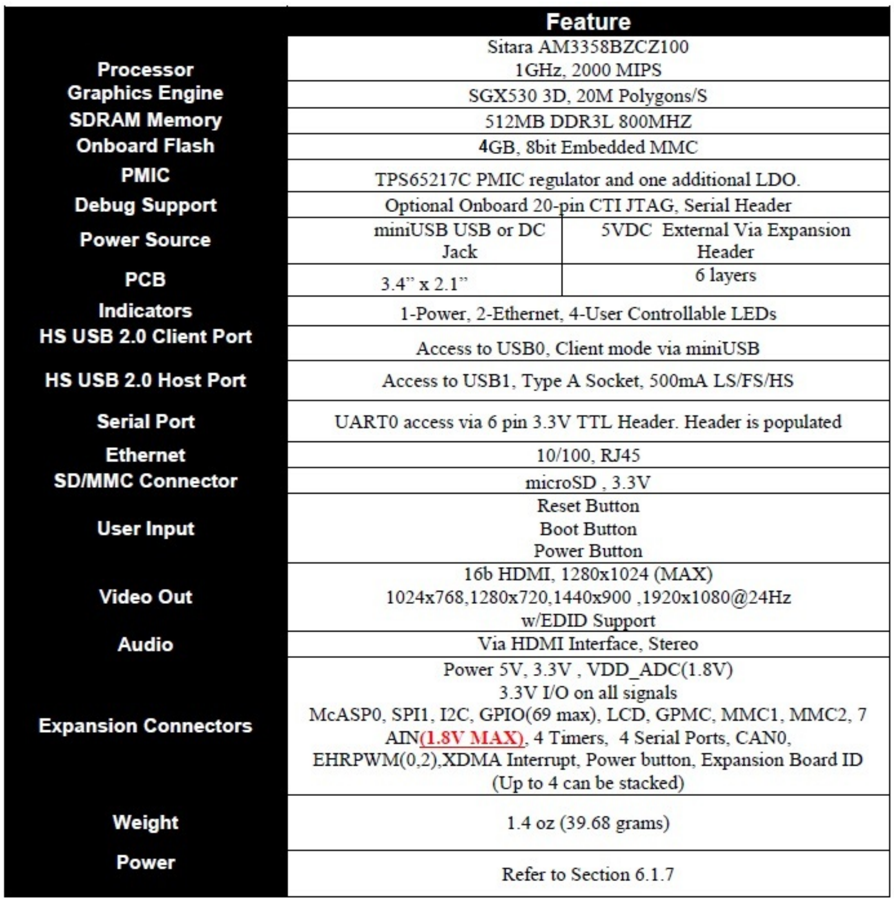
The board is equipped with an AM335x 1Ghz ARM cortex-A8 processor, 512MB of DDR3 RAM, 4 Gb 8-bit on-board flash storage, a 3D graphic accelerator and two programmable real time units running at 200Mhz. The internal memory can be easily expanded with a micro-sd card. The connectivity is guaranteed by a USB host, an Ethernet host, a micro-HDMI port, a CAN controller and 2x46 headers for I-O, expansion and cape connectivity. A resumen of the board's features is in figure 12.

2.3.2 *Reasons for choice and comparison with other boards*

One of the main goals for the chosen board is the integration between all the devices forming the data acquisition system.

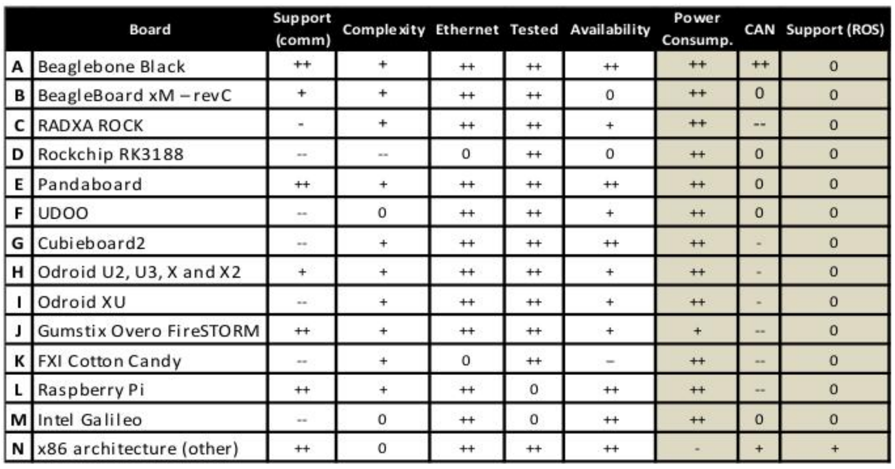
In order to achieve such objective, the device needs to support a large amount of I-O ports; particularly critical is the availability of a CAN interface since it's the connection used by H2-arm for real-time communication.

Another important requirement is the support for ROS (Robot Operative System), since it is the chosen middleware for the integration of the devices.



	Feature	
Processor	Sitara AM3358BZCZ100 1GHz, 2000 MIPS	
Graphics Engine	SGX530 3D, 20M Polygons/S	
SDRAM Memory	512MB DDR3L 800MHZ	
Onboard Flash	4GB, 8bit Embedded MMC	
PMIC	TPS65217C PMIC regulator and one additional LDO.	
Debug Support	Optional Onboard 20-pin CTI JTAG, Serial Header	
Power Source	miniUSB USB or DC Jack	5VDC External Via Expansion Header
PCB	3.4" x 2.1"	6 layers
Indicators	1-Power, 2-Ethernet, 4-User Controllable LEDs	
HS USB 2.0 Client Port	Access to USB0, Client mode via miniUSB	
HS USB 2.0 Host Port	Access to USB1, Type A Socket, 500mA LS/FS/HS	
Serial Port	UART0 access via 6 pin 3.3V TTL Header. Header is populated	
Ethernet	10/100, RJ45	
SD/MMC Connector	microSD , 3.3V	
User Input	Reset Button Boot Button Power Button	
Video Out	16b HDMI, 1280x1024 (MAX) 1024x768,1280x720,1440x900 ,1920x1080@24Hz w/EDID Support	
Audio	Via HDMI Interface, Stereo	
Expansion Connectors	Power 5V, 3.3V , VDD_ADC(1.8V) 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO(69 max), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 4 Serial Ports, CAN0, EHRPWM(0,2),XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)	
Weight	1.4 oz (39.68 grams)	
Power	Refer to Section 6.1.7	

Figure 12: Features of the BBB



	Board	Support (comm)	Complexity	Ethernet	Tested	Availability	Power Consump.	CAN	Support (ROS)
A	Beaglebone Black	++	+	++	++	++	++	++	0
B	BeagleBoard xM – revC	+	+	++	++	0	++	0	0
C	RADXA ROCK	-	+	++	++	+	++	--	0
D	Rockchip RK3188	--	--	0	++	0	++	0	0
E	Pandaboard	++	+	++	++	++	++	0	0
F	UDOO	--	0	++	++	+	++	0	0
G	Cubieboard2	--	+	++	++	++	++	-	0
H	Odroid U2, U3, X and X2	+	+	++	++	+	++	-	0
I	Odroid XU	--	+	++	++	+	++	-	0
J	Gumstix Overo FireSTORM	++	+	++	++	+	+	--	0
K	FXI Cotton Candy	--	+	0	++	–	++	--	0
L	Raspberry Pi	++	+	++	0	++	++	--	0
M	Intel Galileo	--	0	++	0	++	++	0	0
N	x86 architecture (other)	++	0	++	++	++	-	+	+

Figure 13: Comparison between embedded boards

Using a well-developed and open-source operative system as Linux, allows for costs reduction and makes a large amount of software and libraries available for development.

Since the board needs to be set-up onto the exoskeleton, low power consumption is essential for achieving a long time duration of system's battery.

Finally, the possibility of finding the board on the market is, of course, crucial.

Figure 13 shows a comparison between BBB and other similar boards that were available on the market at the moment of the choice by the BioMot group (the newly released Rashpberry Pi 2 could not yet be taken into consideration) [41].

2.3.3 *Set-up of the operative system*

The BBB is shipped with a Linux-Debian OS (Operative System) already set-up in the internal eMMC (embedded Multi Media Card) memory. However, in order to use a more widely known Linux distribution and to make the set-up of the ROS environment and of other libraries easier, it was chosen to set-up an Ubuntu distribution on the board. Many pre-built images specifically designed for the BBB can be found on-line, for example in the official distribution page [7]. An Ubuntu OS image could be found on the *elinux* site [34]. When the OS was set-up, the kernel 3.8.13 with ubuntu 14.04 was available, but new versions are continuously being released and updates might solve eventual driver/software-incompatibility problems. Note that, being the BBB based on an ARM processor, an ARM-distribution of Linux must be used. Also, the newest versions of the Linux kernel don't support certain board functionality as pin muxing trough the *device-tree-overlay*, so only tested kernel distributions should be used.

The OS can be flashed on the eMMC, replacing the existing Debian distribution, otherwise the system can be set-up onto a micro-sd card and booted directly from there. Even if the micro-sd card can be easily ejected from the system, leading, of course, to a system crash if the OS is located there, it allows for a bigger disk space (the eMMC has a size of only 4 Gb, of which 2 are occupied by the OS) and, above all, for an easy backup of the entire system: once all the drivers, scripts, kernel modifications and the software are set-up on the memory, an image of the entire system can be easily done by making an image of the micro-sd card. If the hardware or the kernel fails to boot the board, all saved data can be easily accessed through a simple card reader, and a previous stable image can be written on the micro-sd.

2.3.4 *Set-up of ROS*

As will be described in chapter 3, ROS (Robot Operating System) is the middleware software that needs to be set-up on the board in order to interface the exoskeleton with the other devices in the system. Even if, at the moment of writing, there isn't yet an official release for the selected ARM-distribution of Ubuntu, an experimental release could be found on the ROS distribution page [18]. The last edition of ROS (ROS Indigo) was chosen since it's the only one compatible with the selected OS. All the set-up instructions could be found in the cited page; the download of the software, of course, requires an internet connection that can be easily gained connecting the board via the ethernet port to an internet-provided network.

Even if the selected one is an experimental ROS release, it is already widely used and no bugs/problems could be detected during my work.

2.3.5 *Set-up of the CAN network*

Even if the BeagelBoneBlack is natively equipped with a CAN controller integrated in the microprocessor (as described in [4]), it is neither equipped with a CAN transceiver nor with a connector (a 9 pin connector is the standard choice even if only 2 wires are needed). Still, thanks to the BBB expansion headers, it is possible to use an expansion cape in order to provide the missing elements. As previously said, the BBB is equipped with 2 sets of 46 pins that can be set-up by the user in order to provide I/O function or connectivity with additional devices. Each pin can provide up to 8 functions depending on the mode set by the user (see figure 14 from [9]) and works with a maximum tension level of 3,3V. In order to set the desired pin configuration, a proper *device tree overlay* needs to be loaded.

The Device Tree is a data structure for describing hardware. Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time. For a peripheral driver to work, the correct muxing configuration must be applied to the affected pins; since there are many, and configuration is complex, a special omap specific driver has been written and has been used on the kernel provided by TI (Texas Instruments). It is usually possible to find application-specific device-trees in the web; sometimes some of them are already existing in the operative system and only need to be loaded in order to make a specific device work.

As shown in figure 14, CAN1 network can be enabled by setting pins P9-24 and P9-26 to mode 2; CAN0 network could be enabled by setting pins P9-19 and P9-20 to mode 2, but such operation is known to generate compatibility problems with other capes and therefore was not tried.

A device tree-overlay was found on-line [58]. In order to compile it on an Ubuntu system, the device tree compiler had to be patched with a script provided by Robert C. Nelson [58] .

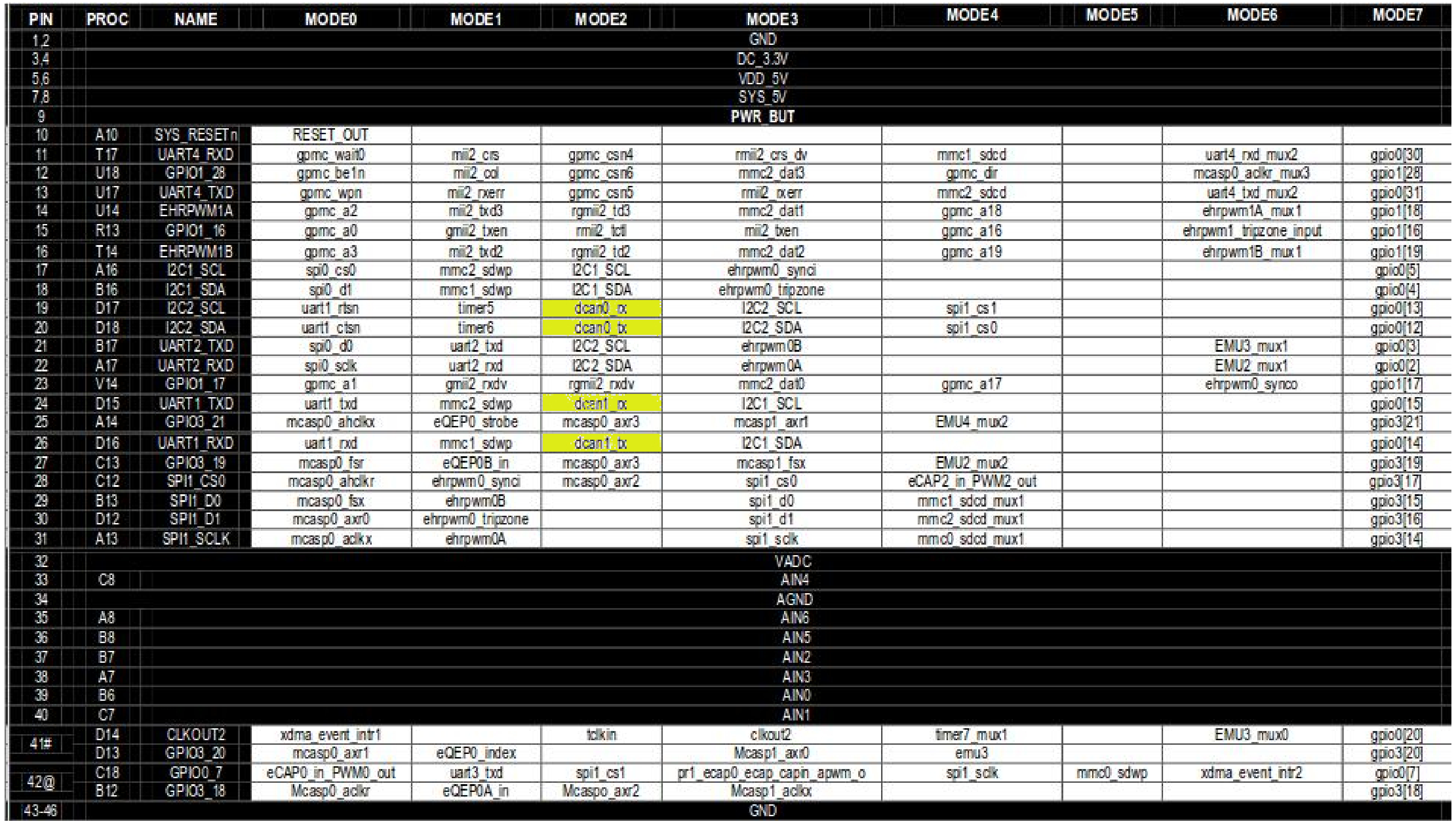
In newer ubuntu images released by TI, the device-tree related to the CAN network is already available in the operative system and is automatically loaded at boot time. Using this device tree, by the way, it was not possible to make the device work.

The required expansion cape was built using a VP230 transceiver from TI (Texas Instruments), as described in [53]. The BBB with the CAN expansion cape can be seen in figure 15.

In order to understand if the device was working properly, a software named *can-utils* was set-up on the system [35]. This utilities, based on the socket-can libraries (that will be described in 4.2.1), allow the user to receive data from a selected can device (*candump* function) and print them on std::out, or to send a specified frame to a chosen can-network (*cansend* function).

Testing the device showed that, in order to make the can transceivers work, the BBB needs to be powered with an extern power supply since the power/tension level provided by a usb cable is not high enough. With such configuration the BBB was able to send/receive data from other devices as the H2-ARM board; here is an example of data from H2-ARM board:

```
can0  078   [6]  FE FC FE 01 CD EF
can0  082   [6]  00 00 00 00 42 64
can0  06E   [6]  4C 5E 00 4C 5D 0B
can0  078   [6]  FE FC FE 01 CD EF
can0  082   [6]  00 00 00 00 42 64
can0  06E   [6]  4C 5E 00 4C 5D 0B
```



PIN	PROC	NAME	MODE0	MODE1	MODE2	MODE3	MODE4	MODE5	MODE6	MODE7
1,2						GND				
3,4						DC_3.3V				
5,6						VDD_5V				
7,8						SYS_5V				
9						PWR_BUT				
10	A10	SYS_RESETn	RESET_OUT							
11	T17	UART4_RXD	gpmc_wait0	mii2_crs	gpmc_csn4	rmii2_crs_dv	mmc1_sdcd		uart4_rxd_mux2	gpio0[30]
12	U18	GPIO1_28	gpmc_be1n	mii2_col	gpmc_csn6	mmc2_dat3	gpmc_dir		mcasp0_aclkr_mux3	gpio1[28]
13	U17	UART4_TXD	gpmc_wpn	mii2_rxerr	gpmc_csn5	rmii2_rxerr	mmc2_sdcd		uart4_txd_mux2	gpio0[31]
14	U14	EHRPWM1A	gpmc_a2	mii2_txd3	rgmii2_td3	mmc2_dat1	gpmc_a18		ehrpwm1A_mux1	gpio1[18]
15	R13	GPIO1_16	gpmc_a0	gmii2_txen	rmii2_tctl	mii2_txen	gpmc_a16		ehrpwm1_tripzone_input	gpio1[16]
16	T14	EHRPWM1B	gpmc_a3	mii2_txd2	rgmii2_td2	mmc2_dat2	gpmc_a19		ehrpwm1B_mux1	gpio1[19]
17	A16	I2C1_SCL	spi0_cs0	mmc2_sdwp	I2C1_SCL	ehrpwm0_synci				gpio0[5]
18	B16	I2C1_SDA	spi0_d1	mmc1_sdwp	I2C1_SDA	ehrpwm0_tripzone				gpio0[4]
19	D17	I2C2_SCL	uart1_rtsn	timer5	dcan0_rx	I2C2_SCL	spi1_cs1			gpio0[13]
20	D18	I2C2_SDA	uart1_ctsn	timer6	dcan0_tx	I2C2_SDA	spi1_cs0			gpio0[12]
21	B17	UART2_TXD	spi0_d0	uart2_txd	I2C2_SCL	ehrpwm0B			EMU3_mux1	gpio0[3]
22	A17	UART2_RXD	spi0_sclk	uart2_rxd	I2C2_SDA	ehrpwm0A			EMU2_mux1	gpio0[2]
23	V14	GPIO1_17	gpmc_a1	gmii2_rxdv	rgmii2_rxdv	mmc2_dat0	gpmc_a17		ehrpwm0_synco	gpio1[17]
24	D15	UART1_TXD	uart1_txd	mmc2_sdwp	dcan1_rx	I2C1_SCL				gpio0[15]
25	A14	GPIO3_21	mcasp0_ahclkx	eQEP0_strobe	mcasp0_axr3	mcasp1_axr1	EMU4_mux2			gpio3[21]
26	D16	UART1_RXD	uart1_rxd	mmc1_sdwp	dcan1_tx	I2C1_SDA				gpio0[14]
27	C13	GPIO3_19	mcasp0_fsr	eQEP0B_in	mcasp0_axr3	mcasp1_fsx	EMU2_mux2			gpio3[19]
28	C12	SPI1_CS0	mcasp0_ahclkr	ehrpwm0_synci	mcasp0_axr2	spi1_cs0	eCAP2_in_PWM2_out			gpio3[17]
29	B13	SPI1_D0	mcasp0_fsx	ehrpwm0B		spi1_d0	mmc1_sdcd_mux1			gpio3[15]
30	D12	SPI1_D1	mcasp0_axr0	ehrpwm0_tripzone		spi1_d1	mmc2_sdcd_mux1			gpio3[16]
31	A13	SPI1_SCLK	mcasp0_aclkx	ehrpwm0A		spi1_sclk	mmc0_sdcd_mux1			gpio3[14]
32						VADC				
33	C8					AIN4				
34						AGND				
35	A8					AIN6				
36	B8					AIN5				
37	B7					AIN2				
38	A7					AIN3				
39	B6					AIN0				
40	C7					AIN1				
41#	D14	CLKOUT2	xdma_event_intr1		tclkin	clkout2	timer7_mux1		EMU3_mux0	gpio0[20]
	D13	GPIO3_20	mcasp0_axr1	eQEP0_index		Mcasp1_axr0	emu3			gpio3[20]
42@	C18	GPIO0_7	eCAP0_in_PWM0_out	uart3_txd	spi1_cs1	pr1_ecap0_ecap_capin_apwm_o	spi1_sclk	mmc0_sdwp	xdma_event_intr2	gpio0[7]
	B12	GPIO3_18	Mcasp0_aclkr	eQEP0A_in	Mcaspo_axr2	Mcasp1_aclkx				gpio3[18]
43-46						GND				

Figure 14: Expansion Header P9 Pinout

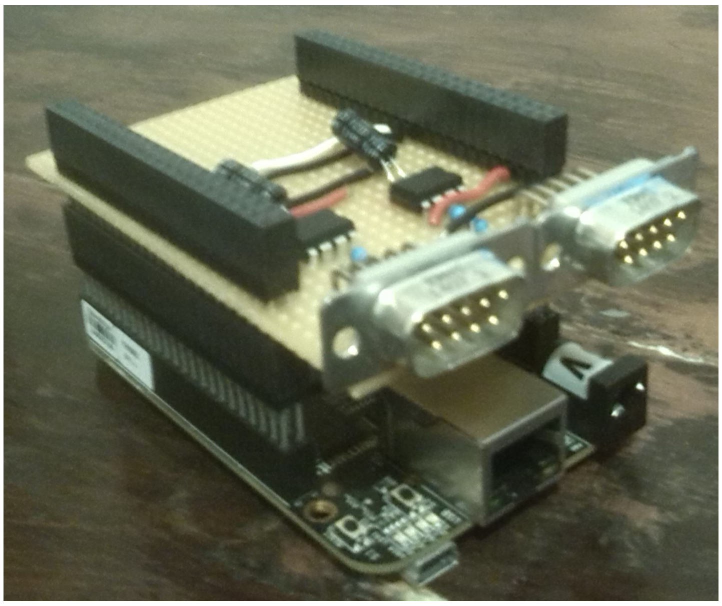


Figure 15: Beagle Bone Black with a CAN-cape expansion

The first value shows the name of the device receiving the data, the second specifies frame's identifier, the third shows the number of information bytes available and finally the transmitted data (on a hexadecimal base) are displayed.

To load the device-tree overlay and the required can-modules automatically at start-up, the following simple script was written (the compiled device tree BB-DCAN1-00A0.dtbo needs to be previously copied in /lib/firmware):

```
#launchCAN.sh
echo BB-DCAN1 > /sys/devices/bone_capemgr.*/slots
sudo modprobe can
sudo modprobe can-dev
sudo modprobe can-raw
sudo ip link set can0 up type can bitrate 1000000
sudo ifconfig can0 up
echo "Interface CAN0 on, bitrate 1000000, use candump to read
    messages"
```

And since the echo command needs to be executed as superuser, the script was added in the /etc/init.d directory. The init.d directory contains a number of start/stop scripts for various services on the system, each script needs to be linked in the proper run-level directory (/etc/rcrunlevel.d/). When linking, the script's name needs to be preceded by an S or a K and a number; scripts starting with a K have a higher execution priority than scripts starting with an S, and scripts with a lower number have higher priority. Since it is not essential to enable the CAN-network as soon as possible, the script was linked to the init.d directory preceded by an S99 as follows:

```
ln -s /etc/init.d/launchCAN.sh /etc/rc2.d/S99launchCAN.sh
```

Note that, even if we enable the *can1* device, linux shows it as the can0 device since it is the first to be loaded. Also note that channel's bit-rate is set to the highest value possible (1Mbps) in order to get the maximum performances from the system.

The development of a structured library for CAN communication will be exposed in section 4.2.2.

2.3.6 *Set-up of the Wi-fi network*

As previously said, connection to the board's control terminal is possible using an *ssh* key; the easiest way of doing it is with the provided usb-cable since the board will automatically create an *"ethernet-like"* connection, setting its IP to 192.168.7.2 and the PC's IP to 192.168.7.1. Alternatively the board can be connected using a longer ethernet cable. Once the board is installed onto the exoskeleton by the way, such form of connection becomes problematic, of course, since wires might interfere with its movements.

The board doesn't have a built-in wi-fi interface, but it allows for the installation of a wi-fi adapter through the USB host. Since the BBB's OS uses a quite old kernel (the used edition is 3.8.13), many wi-fi adapters are not supported, so the device was chosen from a list of tested devices ([36]). Considering the price, reliability and availability of the products, a *Belkin N150* adapter was chosen.

Such an adapter was used in combination with a *TP-Link TL-WR841N* wireless Router. Since the communication between machines running ROS is made by specifying the IP of the machine running the ros-master (see 3.3.4.2), the IP of the machines connected to the network needs to be uniquely associated to the MAC address of their network cards. This objective can be easily achieved accessing the router configuration and properly setting it's DHCP service.

Since the ground and power planes of the BBB's HDMI port are right below the USB port, they can dampen the WiFi signals, leading to poor performances of the adapter. This problem can be solved by an extension cable that improves the distance between the adapter and the board. Since the system needs to be as compact as possible and the HDMI is never used, it was chosen to disable it trough the device tree overlay. To do this, the file */boot/uEnv.txt* on the BBB was modified adding the following line:

```
cape_disable=capemgr.disable_partno=BB-BONELT-HDMI,BB-BONELT-
    HDMIN
```

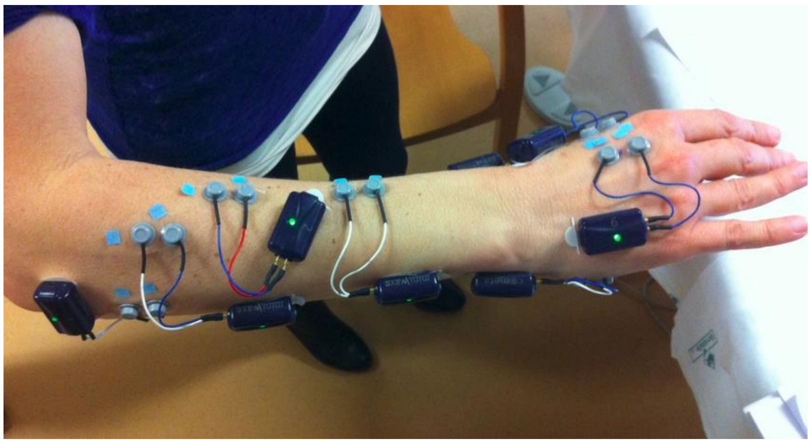


Figure 16: Example of Cometa's EMG application

Finally, to make the connection to the created network automatic, the file */etc/network/interfaces* was modified adding the lines:

```
auto wlan0
iface wlan0 inet dhcp
wpa-ssid "Name_of_network"
wpa-psk "Password_of_the_network"
```

2.4 OTHER ENVIRONMENT SENSOR

In order to better understand the gait-characteristics and the interactions between the user and the device, other sensors are used during the tests with the H2 exoskeleton.

2.4.1 *EMG sensors*

Electromyography (EMG) is an electrodiagnostic medicine technique for recording and evaluating the electrical activity produced by skeletal muscles. EMG is performed using an instrument called an electromyograph, to produce a record called an electromyogram. An electromyograph detects the electrical potential generated by muscle cells when these cells are electrically or neurologically activated.

To study muscle activity during the use of the H2-exoskeleton, a *"Cometa Wave Wireless EMG"* system was used. This device allows for readings from up to 32 EMG channels and 2 FootSwitch channels in a completely wireless mode. Signals can be read with a frequency up to 1MHz and recorded/forwarded thanks to a proprietary wireless protocol direct connection between electrodes/receiving unit and a dedicate software [20].

The device is equipped with a BNC trigger port that allows for registration when a 3,3 V constant input is given as input.

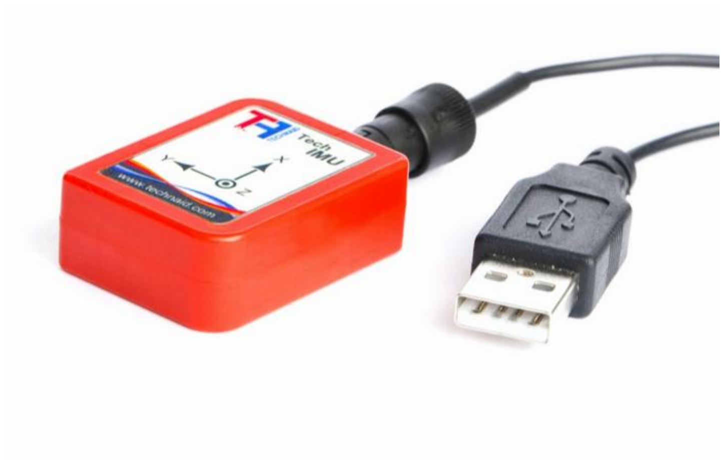


Figure 17: Technaid's IMU

2.4.2 *IMU sensors*

An inertial measurement unit (IMU) is an electronic device that measures and reports a structure's velocity, orientation, and gravitational forces, using a combination of accelerometers and gyroscopes, sometimes also magnetometers. IMUs are typically used to maneuver aircraft, but they also find application in mechanic and robotic.

To record and study the movements of a user wearing an exoskeleton, *Technaid Tech* IMUs are used. These sensors integrate three different types of sensors, i.e. an accelerometer, a gyroscope and a magnetometer. A sophisticated and robust algorithm, calibrated also taking into account changes in temperature, allows for a precise and robust estimation of 3D orientation, even in changing environmental conditions [55].

The device can connect to a PC or to the BBB through a USB cable and allows a sampling frequency up to 200 Hz.

2.4.3 *EEG sensors*

Electroencephalography (EEG) is the recording of electrical activity along the scalp. EEG measures voltage fluctuations resulting from ionic current flows within the neurons of the brain [33].

EEG measurements usually require the combination of a pre-amplifier, in the form of a helmet wore by the user, and of a second bio-signal amplifier, usually in the form of a box.

EEG devices from *g-tec* were used by the group during experiments with the H2 exoskeleton. No integration with the data coming from the middleware was provided during this work. For more information and technical specifications about the devices, see [40].

It is not yet defined how and if these measures will interact with the control of the exoskeletons.

3 THE ROS COMMUNICATION ARCHITECHTURE

3.1 INTRODUCTION

As described in the second chapter, an exoskeleton system is usually composed of a large number of sensors, devices, and consequently, many different programs that need to constantly interact with each other for achieving the final control. This is true also for the BioMot system, that, as was underlined in the introduction, is developed by distinct groups forming an international team. Finding a standardized way for interfacing all the different works done by each group would considerably fasten de integration process. At the same time, in order to make dissemination and knowledge transfer as easy as possible, and in order to re-use already built software, programs' structure should be as similar as possible to the current standards used in Robotic.

A possible way to achieve this objectives is using a middleware software. Following is a brief description of the ROS middleware and of how its architecture can be applied to the BioMot exoskeleton.

3.2 MIDDLEWARE SOFTWARE

A middleware is a computer software that provides services to applications beyond those available from the operating system. A middleware can be considered as a layer bridging the gap between applications and low-level constructs, a novel approach to resolve many of the open issues and drastically enhance application development. A schematic representation of the operational level of a middleware can be seen in figure 18: it adds a layer that glues together the network hardware, operating systems, network stacks, and applications. A complete middleware solution should contain a runtime environment that supports and coordinates multiple applications, and standardized system services such as data aggregation, control and management policies adapting to target applications, and mechanisms to achieve adaptive and efficient system resources use [43].

Since Autonomous robots are complex systems that require the interaction between numerous heterogeneous components (software and hardware), the use of a middleware architecture is common and particular types of middleware, known as **robotic middleware**, are specifically designed for this applications. This middleware need to manage the complexity and heterogeneity of the hardware and applications, promote the integration of new technologies, simplify soft-

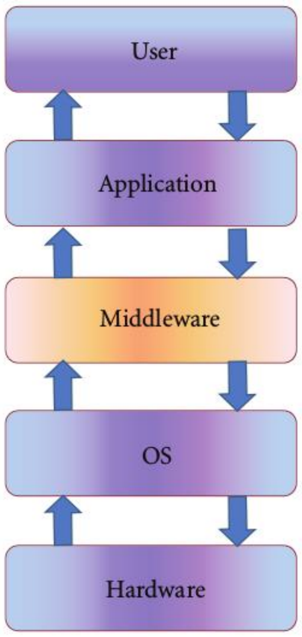


Figure 18: Middleware layers

ware design, hide the complexity of low-level communication and the heterogeneity of the sensors, improve software quality, reuse robotic software infrastructure across multiple research efforts to reduce development cost [38].

A developer needs only to build the algorithm as a component, after which the component can be combined and integrated with other existing components. Furthermore, if he wants to modify and improve his component, he needs only to replace the old one with the new one and the rest of the application doesn't need to be changed. Therefore, development efficiency will improve due to the high modularity of the system. Also other instruments, like a simulation environment, are usually provided (or can be easily integrated) within the middleware frame.

In [5] can be found a survey on twenty-one robotics middleware frameworks, evaluated and compared from various points of view: Player,CLARAty, ORCA, MIRO, UPNP, RT-Middleware, ASEBA,MARIE, RSCA, OPRoS, ROS, MRDS, OROCOS, SmartSoft,ERSP, Skilligent, Webots, Irobotaware, Pyro, Carmen, and RoboFrame. In particular *ROS*(Robot Operating System), is becoming one of the most utilized robotic middleware.

3.3 ROBOT OPERATING SYSTEM

3.3.1 *Description and History*

"The Robot operating System(ROS) is an open-source, meta-operating system for robots. It provides the services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers." [13] ROS is similar in some respects to other robot frameworks, such as the previously cited *Player, YARP, Orocos, CARMEN, Orca, MOOS, and Microsoft Robotics Studio*.

ROS was originally developed in 2007 under the name *Switchyard* by the Stanford Artificial Intelligence Laboratory in support of the Stanford AI Robot STAIR (STanford AI Robot) project. From 2008 until 2013, development was performed primarily at Willow Garage, a robotics research institute/incubator. During that time, researchers at more than twenty institutions collaborated with Willow Garage engineers in a federated development model. In February 2013, ROS stewardship transitioned to the Open Source Robotics Foundation.

3.3.2 *Objectives of the project*

The primary goal of ROS is to support code reuse in robotics research and development. ROS is a distributed framework of processes (also known as Nodes) that enables executables to be individually designed and loosely coupled at runtime. These processes can be grouped into Packages and Stacks, which can be easily shared and distributed. ROS also supports a federated system of code Repositories that enable collaboration to be distributed as well. This design, from the file system level to the community level, enables independent decisions about development and implementation, but all can be brought together with ROS infrastructure tools.

Other objectives include:

- Easy integration with other robot software frameworks.
- Light weight.
- Language independence(commonly used with C++ or Python).

- Easy testing.
- Multy-platform (it's actually available for UNIX and MAC systems, but efforts are being made to make it fully compatible with Microsoft Windows OS).
- Scaling: ROS is appropriate for large runtime systems and for large development processes.

3.3.3 *Releases and contributions*

All the code that builds ROS is completely open-source(BSD license), contributions to its expansion are continuously made by a large community anyone can join (see [1].

A set of versioned ROS stacks forms a ROS Distribution. These are structured as for example Linux distributions; much like them, distributions make it much easier for developers to target a consistent set of libraries to develop and test on. The released distributions up to now are: ROS Indigo Igloo (July 22nd, 2014), ROS Hydro Medusa (September 4th, 2013), ROS Groovy Galapagos (December 31, 2012), ROS Fuerte Turtle (April 23, 2012), ROS Electric Emys (August 30, 2011), ROS Diamondback (March 2, 2011), ROS C Turtle (August 2, 2010), ROS Box Turtle (March 2, 2010). The details on the distributions can be found in the project's repository documentation. In this work is used the last ROS edition (Indigo).

3.3.4 *ROS concepts*

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level [12].

3.3.4.1 *Filesystem level*

The filesystem level concepts mainly cover ROS resources that are stored on disk, such as:

- **Packages**: Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS.
- **Metapackages**: Metapackages are specialized Packages which only serve to represent a group of other related packages.
- **Package Manifests**: Manifests (package.xml) provide metadata about a package, including its name, version, description, li-

1 http://wiki.ros.org/Contributing)

cense information, dependencies, and other meta information like exported packages.

- **Repositories**: A collection of packages which share a common VCS (Version Control Systems). Packages which share a VCS share the same version and can be released together using the catkin release automation tool bloom. Repositories can also contain only one package.

- **Message types**: Message descriptions, stored in *my_package/msg/ MyMessageType.msg*, define the data structures for messages sent in ROS.

- **Service types**: Service descriptions, stored in *my_package/srv/ MyServiceType.srv*, define the request and response data structures for services in ROS.

3.3.4.2 *Computation Graph Level*

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are described in the following:

- **Nodes**: Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library, such as roscpp(for C++) or rospy(for Python).

- **Master**: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.

- **Parameter Server**: The Parameter Server allows data to be stored by key in a central location and is a part of the Master node.

- **Messages**: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).

- **Topics**: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

- **Services**: The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.

- **Bags**: they are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

The ROS Master acts as a nameservice in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run.

Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in a ROS is called TCPROS, which uses standard TCP/IP sockets.

This architecture allows for decoupled operation, where the names are the primary means by which larger and more complex systems can be built. Names have a very important role in ROS: nodes, topics, services, and parameters all have names. Every ROS client library supports command-line remapping of names, which means a compiled program can be reconfigured at runtime to operate in a different Computation Graph topology.

3.3.4.3 *Community Level*

The ROS Community Level concepts are ROS resources that enable separate communities to exchange software and knowledge. These resources include:

- **Distributions**: ROS Distributions are collections of versioned stacks that you can install. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software.
- **Repositories**: ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.
- **ROS Wiki**: The ROS community Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.
- **Mailing Lists**: The ros-users mailing list is the primary communication channel about new updates to ROS, as well as a forum to ask questions about ROS software.

3.3.4.4 *Higher level concepts*

Some higher-level concepts are provided for helping the building of larger systems on top of ROS:

- **Tf package**: provides a distributed, ROS-based framework for calculating the positions of multiple coordinate frames[2] over time.
- **Actionlib package**: provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.
- **Common msgs stack**: even if the definition of messages can be arbitrary, this stack provides a standard base message ontology for robotic systems.
- **Pluginlib package**: provides tools for writing and dynamically loading plugins using the ROS build infrastructure.
- **Filters package**: provides a C++ library for processing data using a sequence of filters.
- **Urdf package**: defines an XML format for representing a robot model and provides a C++ parser.

2 http://wiki.ros.org/geometry/CoordinateFrameConventions

The biggest part of the reported informations derive from the ROS documentation [3].

3.4 ROS IN BIOMOT

As previously described, Biomot's exoskeleton system is composed of a large number of devices and programs that need to be integrated in order to reach the final control objective. Since the number and type of these devices is not yet completely defined and may change with relation to the project evolution and with the case of application, an architecture providing high modularity is required. The software is also developed by different researchers from different countries, so a widely-known standardized architecture that allows an easy integration is necessary.For the characteristics described in the previous paragraphs, ROS stands as the ideal choice to achieve these objectives.

Figure 19 shows the diagram of a point to point communication architecture (built for example using sockets) applied to the BioMot system; it is clear how, in such a scenario, there is a very large number of connections that need to be set-up. Furthermore, each node needs to know the addresses and the available communication ports of all the programs it wants to create a connection with. Adding and removing a node would require the modification of a consistent amount of code in all the connected programs.

Figure 20 shows the diagram of a communication architecture based on a generic middleware. The number of connections is drastically reduced, the point to point connections still exist, but their creation is handled by the middleware.

Finally, figure 21 shows the diagram of a communication architecture based on ROS. To interface itself with the middleware, each element of the network needs to provide some interfacing nodes. Each of this nodes only has to know the address of the ROS-Master node. When starting, a node registers itself to the Master node, declaring which kind of resources it will produce and/or which data it is expecting to receive from the middleware (Topics, Services,...). The ROS-master node dynamically handles the point to point connections; it also can store parameters that can be accessed my any member of the middleware (parameter server).

The modularity of the ROS middleware leads to an architecture that is elegant and easy to modify: the number of devices to connect and the kinds of data to be exchanged can be easily changed. Furthermore, the node handling the high-level control can be easily moved from one device to another or, depending on the project's future objectives, be divided into more nodes implementing different activities.

3 http://wiki.ros.org/ROS/Concepts

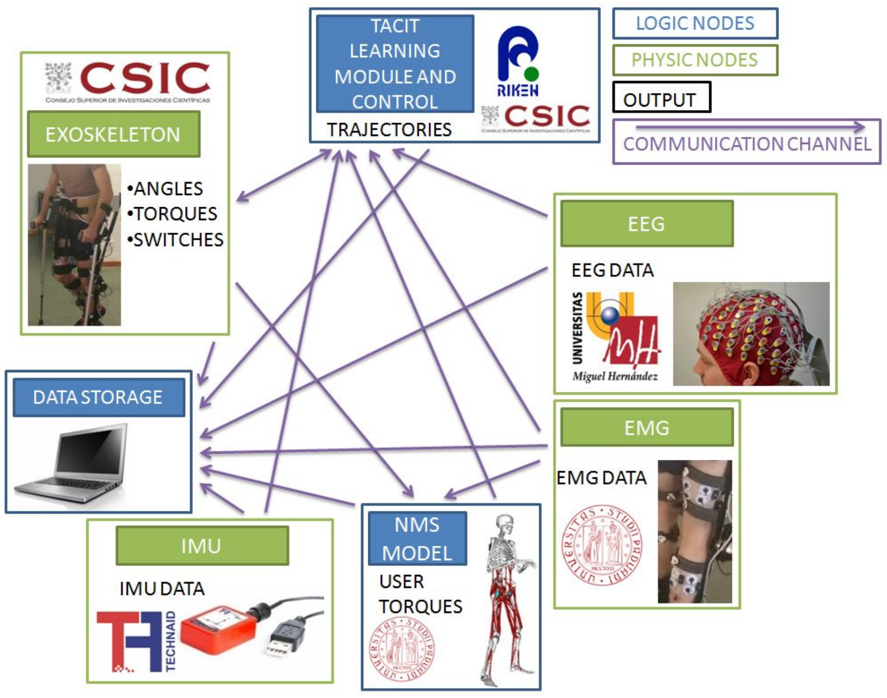


Figure 19: Point to point communication architecture

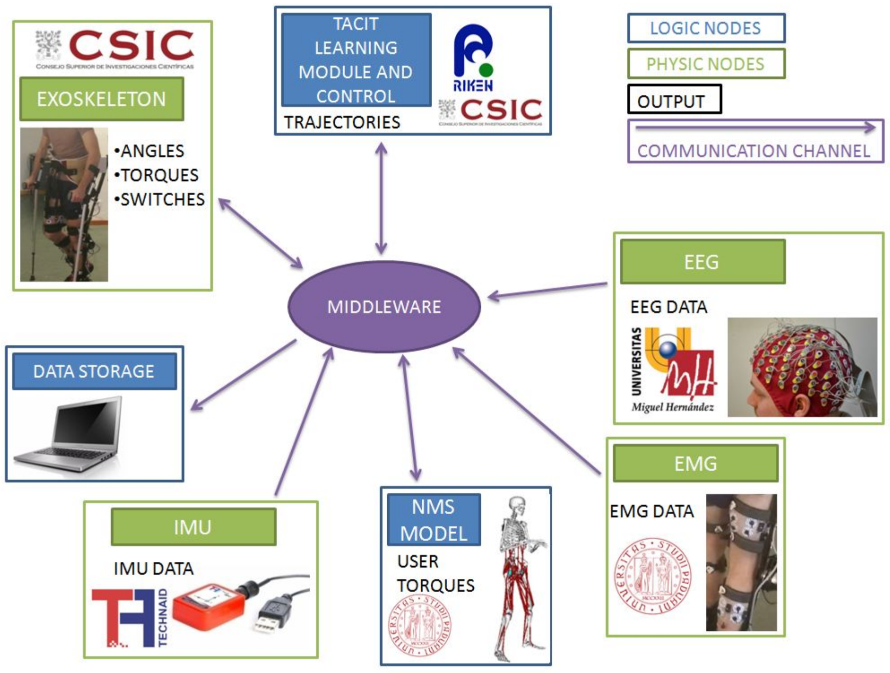


Figure 20: Middleware communication architecture

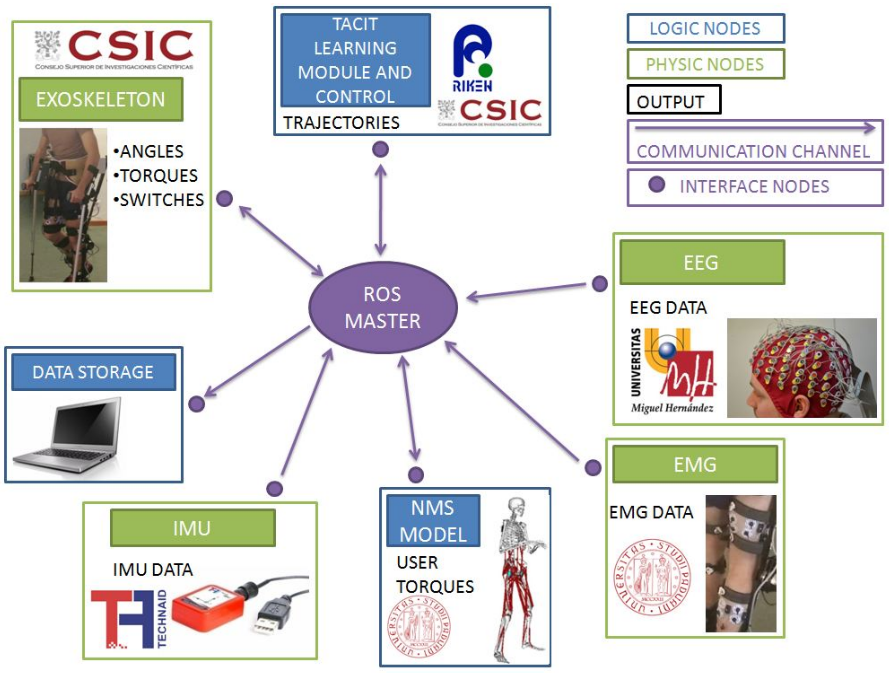


Figure 21: ROS communication architecture

In particular, the following functionalities will be required:

- The nodes located onto the exoskeleton (whose execution is set on the BBB(Beagle Bone Black) and whose implementation will be described in section 4.3 and 4.4), provide an interface between sensors, actuators and the BBB. Communicating with these nodes, the system will be able to acquire information relative to the sensors embedded in the exoskeleton (positions, interaction torques, and pressures) and to send commands as position/torque/stiffness set-points or alternatively as a selection/-modulation of pre-memorized command-patterns. The frame ROS-control will be investigated as a possible standardized way of providing this interface (section 7.3).

- EMG node will acquire data trough the EMG sensors, transmit them to a real-time *neuromusculoskeletal model* node that will then communicate the estimated user-provided torques to the high-level controller.

- IMU node will communicate data acquired from the *Inertial Measurement Unit* to the control system.

- EEG node will collect data from the *electroencephalography* system, elaborate them in order to estimate user's intentions and communicate its results to the control system.

- Tacit learning node will use the information provided by the middleware in order to optimize the activity of the high-level control. The key goal of this component is to adapt the robot motions to the patients' state through patients-robot interactions. The interactions between patient and the robot imply not only using the physical contacts (interaction torques) but also signal-based interactions through biological signals (EMG and EEG). Tacit learning is conceived to adapt the robot motions to the patients' intentions. Such controller is suitable for adapting an initial generic walking pattern performed by the exoskeleton to the changing specifics of the wearer.

- Data logging will be possible from any node connected to the ROS master: thanks to the *rosbag record* function, any data being published onto a topic can be recorded into a structure named *rosbag file*. This structure also allows for data-playback and easy data visualization. Data can still be easily recorded also in other formats (as for example is explained in 4.3.3).

The high-level control strategy and its location in the system is not yet fully defined; its development is out of the objectives of this work, but for this reason the whole system's communication architecture must be as flexible and easy-to-change as possible.

4 INTERFACING THE H2 EXOSKELETON

4.1 INTRODUCTION

As described in section 2.2.3, the HAL-board on the exoskeleton provides the lectures of the built-in exoskeleton's sensors and 3 different communication interfaces (CAN,Wireless and Bluetooth). This integrated board, by the way, is not running an operative system and is not suitable for interfacing the exoskeleton with a middleware architecture: it is ideal for handling the real-time low level control of the exoskeleton joints, but lacks the complexity and computational-power required for handling high-level interfaces. For this reason a second board, the BeagleBone Black (described in section 2.3), is added to the system. In the following sections the work made for interfacing this board with the H2 exoskeleton will be described.

4.2 READING DATA TROUGH THE CAN NETWORK

Of the three communication interfaces described in 2.2.3, only one provides the reliability required by a control messages flow: the CAN network.

As was pointed out in section 2.2.3.3, CAN communication provides message integrity-control and is ideal for sending low length messages in real-time.

In section 2.3.5, the steps required for setting-up the CAN connection on the BBB and the use of the *candump* utility for plotting the acquired messages were described. Still, in order to forward the acquired data trough the ROS middleware or to use them in other control contests, a proper CAN-reading library needed to be employed. To achieve such objective **SocketCAN**'s API were used.

4.2.1 *SocketCAN API*

"The SocketCAN package is an implementation of the CAN protocols for Linux. While there have been other CAN implementations for Linux based on character devices, SocketCAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The CAN socket API has been designed as similar as possible to the TCP-IP protocols to allow programmers, familiar with network programming, to easily learn how to use CAN sockets" [31]. A device driver for CAN controller hardware registers itself with the Linux network layer as a network device, so that CAN frames from the controller can be passed up to the network layer and on to the CAN protocol family module and also vice-versa. Also, the protocol family module provides an API for transport protocol modules to register, so that any number of transport protocols can be loaded or unloaded dynamically. This implementation allow many processes to access the same device at the same time. The previously cited *candump* and *cansend* functions were built using SocketCAN.

In order to start a communication with SocketCAN, a socket needs to be initialized using the required protocol and bound—connected to a specific CAN device. Frames of data can be sent—received using an adequate frame type (struct can_frame) and the $read()$ and $write()$ functions. In particular the raw socket protocol was chosen for the communication. All the code provided by SocketCAN's api is written using the C language.

4.2.2 *Creation of a structured communication library*

In order to get a better software-structure and to make CAN communication simpler for high-level programs, a C++ can-reading class was developed that allows an abstraction from the underlying SocketCAN implementation. To build these functions (and in general all the functions that require threading), the standard threads defined in the standard C++11 were used. This implies that, **to compile the developed code, a compiler supporting the C++11 standard is required.**

The files to reference for this implementation are (see appendix A.1):

- CANbusInterface.cpp and header;
- CANbusReader.cpp and header;
- DataFromCANbus.cpp and header.

An UML scheme showing the relations between the created classes and a flow diagram for the reading process are in figure 22.

In CANbusInterface is implemented the low level interface using the SocketCAN API. A CANbusInterface class is built, wich provides the functions: *void read(can_frame currentData)* and *void canWrite(const can_frame dataToWrite)*. The SocketCAN socket is initialized in the constructor of the class, so it doesn't need to be open for every sent/received message and only needs to be closed at the end of the communication, when the object is destroyed, reducing the number of operations needed for sending/receiving data.
In CANbusReader, a dynamic object named CANbusReader is defined. This object declares an object of type CANbusInterface, *canbusInterface* and uses it in its operator function for reading a single data from the CAN bus.

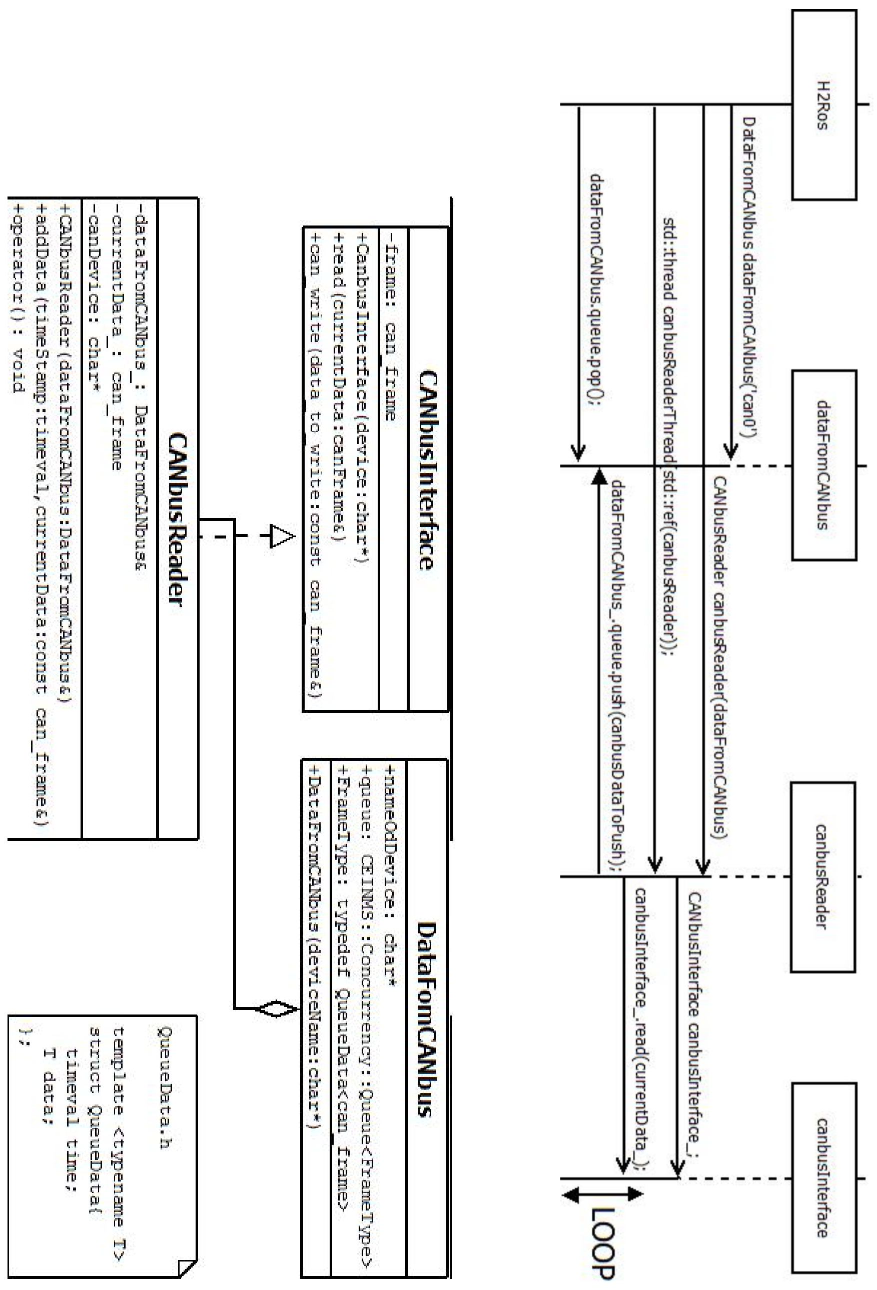

H2Ros
dataFromCANbus
canbusReader
canbusInterface
DataFromCANbus dataFromCANbus("can0")
std::thread canbusReaderThread(std::ref(canbusReader));
dataFromCANbus.queue.pop();
CANbusReader canbusReader(dataFromCANbus)
dataFromCANbus_.queue.push(canbusDataToPush);
CANbusInterface canbusInterface_;
canbusInterface_.read(currentData_);
LOOP
CANbusInterface
-frame: can_frame
+CanbusInterface(device:char*)
+read(currentData:canFrame&)
+can_write(data_to_write:const can_frame&)
DataFromCANbus
+nameOdDevice: char*
+queue: CEINMS::Concurrency::Queue<FrameType>
+FrameType: typedef QueueData<can_frame>
+DataFromCANbus(deviceName:char*)
CANbusReader
-dataFromCANbus_: DataFromCANbus&
-currentData_: can_frame
-canDevice: char*
+CANbusReader(dataFromCANbus:DataFromCANbus&)
+addData(timeStamp:timeval,currentData:const can_frame&)
+operator(): void
QueueData.h
template <typename T>
struct QueueData{
timeval time;
T data;
};

Figure 22: UML graph for the CAN library and flow diagram for the reading process

It then makes a time-stamp of the reception-time and uses a function named addData for pushing the acquired data into a communication queue. The communication queue is necessary since the thread receiving the data (this thread), might be faster than the thread reading them, if a queue is not implemented, data might be lost. Since a custom queue of communication is used instead of the queues defined in the standard library, before being published into the queue, data need to be stored in a structure named QueueData, that makes use of templates for keeping its data-type generic. This particular type of queue allows for more than one thread to read from it by subscribing to a reading list; in this particular case only one thread needs to access the read data, but its utilization makes an eventual code-expansion easier. Also, the access to the queue is automatically done in mutual exclusion without the need of declaring other mutex variables; this makes the utilization of the queue easier for a programmer.
in DataFromCANbus' object, the cited queue is declared and its type is set to be a *can frame* structure. This object only has a data-encapsulation function.

All the functions implementing the queue's structure were not written by me and hence won't be commented.

In order to use this code from another thread to read messages, the DataFromCANbus.h and CANbusReader.h need to be included; a static object of type DataFromCANbus needs to be declared as global variable (so it can be shared with the created thread) and a CANbus-Reader object needs to be declared passing the DataFromCANbus' object as parameter. The reading thread then needs to be executed loading the dynamic object of type CANbusReader. The calling thread must then subscribe to the reading queue, which is part of dataFromCANbus and finally read CAN data using the *pop()* function of the queue. Ex:

```
static DataFromCANbus dataFromCANbus;
...
int main(int argc, char** argv){
...
  CANbusReader canbusReader(dataFromCANbus);
  std::thread canbusReaderThread(std::ref(canbusReader));
  dataFromCANbus.queue.subscribe();
  while(programOk){
      DataFromCANbus::FrameType newData = dataFromCANbus.queue.
          pop();
      ... /* use newData */
  }
}
```

If no data is available, the *pop()* function will block the thread, but will also make it release the use of the processor, avoiding *busy-waiting* phenomenons.

4.3 ROS NODES FOR RECEIVING DATA

4.3.1 *Structure of the package*

As described in 3.3.4, when using the ROS middleware, a program has the logical function of a node. All the nodes that need to work together to achieve an objective are grouped into the same package. Tu build these nodes, the most common type of workspace was used: the *catkin workspace*. When using this workspace, each package needs to be described by a xml manifest and a CMakeLists.txt file. The first contains general informations about the package: its name, its authors, its license, the pre-required dependences for its compilation etc. The second includes the required informations for compiling its nodes, its customized messages, services, the external packages and libraries to be included etc.

Both these files need to be included into a folder, which will also include all the developed nodes related to that package. This package-folder will then be located in the *src* folder of the catkin workspace. The source file for the package's nodes should be located in a sub-directory named *src*. In order to separate the code related to the nodes running on the BBB from that of the nodes running on another system (generally a PC), two source folders were created: *src* and *src_bbb*. The files to be included (header files) should be in a directory named *include*, the customized messages in a *msg* folder, the services in a *srv* folder and the launch files in a *launch* folder. The package including the nodes related to the h2 exoskeleton was named *h2Interface*.

4.3.2 *H2ros node*

The first node to be created inside the h2Interface package was the H2ros node. This node, running on the BBB, has the function of receiving data from the exoskeleton (from the HAL board) and to forward them to the middleware. In order to do so, it makes use of the communication library described in 4.2.2 for reading data from the CAN network. The data sent by the HAL board are the ones described in figure 9; the program can understand which kind of message is received checking its identifier, it then puts the data in their related ROS messages and publishes them on a ROS topic.

There are many possible ways for structuring data before they are published on topics:

- The most efficient way, from a computational point of view, would be to build a single customized message, including all the informations to be forwarded (angles, torques and switches), using variables as small as possible (8 bit integers can describe all the received informations). Some simple tests, made publishing faked data as fast as possible and measuring the required transmission time for a fixed number of them, confirmed this is the fastest transmission technique. At the same time this is also the less modular way for publishing data: if any of the transmitted information is not of interest for the required operation (experiment, control strategy etc.), its transmission on the middleware can't be avoided, leading to an inefficient communication. Also, being the published message of a customized type, its implementation (msg file) needs to be specified on all the receiver nodes, making the code portability complicated.

- Another way for publishing the acquired information would be defining three different customized messages: one for the angles, one for the torques and one for the foot-switches. This solution makes the transmission more modular: if no node requires a particular information (ex foot-switches values), such information is not transmitted on the middleware. On the other hand, this solution is computationally more expensive than the previous one when all the informations are being read and it still makes use of customized messages.

- A third publishing technique consists of the definition of two messages: a jointState message and a foot-Switches message. JointState messages are standard messages commonly used in ROS for describing the particular state of a joint; its implementation is as follows:

```
Header header
string[] name
float64[] position
float64[] velocity
float64[] effort
```

  Information related to the angles can be stored in the position array, torques can be put in the effort's one while no information about joints' velocity needs to be transmitted. This messages don't need to be defined by their users (receiving nodes) since they are a standard message usable including *sensor_ msgs/JointState.h*. Being a standardized message, they can be easily integrated with other ROS frameworks (they are also used by the *JointStateControllers* defined in ROS-control) and their portability is easier. On the other hand, the modularity of the communication is not as good as in the second case and, using a *float64* type instead of an *int8*, they are also less-efficient from a computational point of view.

Since switching from a publishing mode to another doesn't need a deep modification of the code and since transmission times on the ROS topics are actually not critical (each transmitted message has already got a time-stamp of the time of reception from the CAN network in its header), the third option was chosen. If publishing times should, in a second moment, become critical, one of the other two options should be reconsidered.

Two publishers and two topics are then defined for publishing the messages: jointsPublisher publishes the jointState messages onto the jointsTopic, while switchesPublisher publishes the switchesMessages onto the switchesTopic.

When a message is published to a topic, it is broadcasted to all the nodes who subscribed to that topic; the rosmaster handles the low-level connection (usually creating TCP-IP communication sockets), so the user doesn't need to specify any address.

4.3.2.1 *Available options*

The node can be launched specifying a frequency reduction factor: if for example *"-rf 2"* is passed as parameter, only one message every two will be published on the topics, getting a publishing frequency of 50 Hz instead of 100 Hz.

4.3.2.2 *Trigger signal-handling*

Another important feature provided by this node is the capability of handling a trigger signal: when the pin P9-42, whose related to the BBB's port GPIO7, detects a transition from 0V to 3,3V, an interrupt is raised and the program handles it by resetting the acquisition time to 0. This functionality is essential for data synchronization with other devices.

In order to achieve this objective, a dedicated thread, named *trigger-Waiter* was created. When this thread is created, it starts the execution of the function *resettingId()*. This function uses some functions from the Glib2 library to monitor the file *"/sys/class/gpio/gpio7/value"*. This file is connected to the pin status if the following commands were previously executed as superuser:

```
echo 7 > /sys/class/gpio/export
echo both > /sys/class/gpio/gpio7/edge
```

This can be done automatically at boot-time by creating a script as was explained in 2.3.5. In particular the file will contain the value "0" when a low tension level is applied and the value "1" when a high tension level is detected.

When the file changes the function *onTriggerEvent* is executed, it checks the value in the file and if it is "1" (meaning that the transition was positive), it updates the shared variable *startTime* to the current time of the system. Since *startTime* is made of 2 values (seconds and nanoseconds) that aren't read in a single instruction and is shared between 2 threads, its access needs to be protected by a mutex variable, named time_mutex, in order to avoid interleaving problems. Since the time of reception stored in the messages is calculated as the difference between the absolute time of reception and the *startTime*, updating *startTime* implies making it start back from 0. If a similar operation is done on the other devices recording data from the exoskeleton (ex EMG acquisition device), this time can be selected as the start-time of the experiment and data-synchronization is easily achieved.

The files related to the H2ros node are *H2ros.cpp, interruptFunctions.cpp* and *interruptFunctions.h* (appendix A.1).

4.3.3 *Receiving nodes*

In order to visualize and eventually log the data transmitted by the exoskeleton on another device connected to the middleware, other 3 nodes were created: *anglesReceiverUi, torquesReceiverUi* and *switchesReceiverUi*.

Each of these nodes provides a GUI (Graphic User Interface) to visualize the related data and eventually allow the user to log them in a specified text file. To design these interfaces, the qt libraries and the Qt-creator software were used.

Qt is a comprehensive C++ framework for developing cross-platform GUI applications using a "write once, compile anywhere" approach. Qt lets programmers use a single source tree for applications that can run on Windows, Mac OS X, Linux, Solaris and other systems [47]. Qt uses standard C++ with extensions including signals and slots that simplifies handling of events, and this helps in development of GUI. Qt-creator provides an interface that helps programmers in the development of a GUI, it can also be used as a program editor, allowing for compilation and debugging, but in order to integrate its GUIs in the ROS nodes, and build them with the *catkin_make* command, Cmake was used for compilation. This could be achieved modifying the CMakeLists files of the package as suggested in [56].

4.3.3.1 *Angles receiver*

The GUI created for receiving angles data can be seen in figure 23. To read data from a ROS topic, a dedicated thread is created in order to don't block the execution of the GUI running in the main program. A reader subscribes to the *jointsTopic* and invokes the function *chatterCallback* every time a message is read. This function updates the values of a structure shared with the GUI thread with the newly read values; if the log is enabled trough the GUI, it saves the acquired data in a text-file whose location is also specified by the GUI. The values

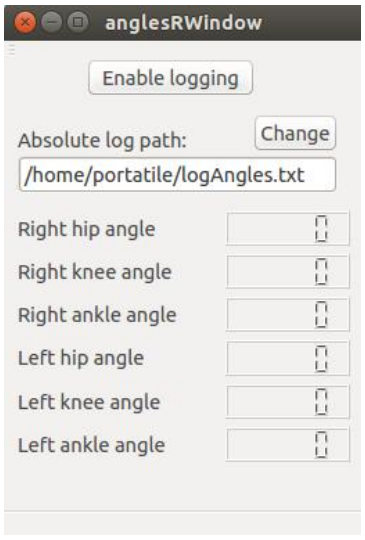


Figure 23: GUI provided by anglesReceiverUi

and their related time-stamps are separated by a white space. In order to don't open and close the file each time arrives, a log-object is created that closes the file only when it's destructor is invoked. This form of data-storage is an alternative to the already described bag-files, which is not as powerful but easier to use.

The GUI thread periodically updates the values of the LCD numbers with the values of the shared structure.

The files related to this node are *anglesReceiverUi.cpp*, *anglesrwindow.cpp*, *anglesrwindow.h* and *anglesrwindow.ui* (appendix A.1).

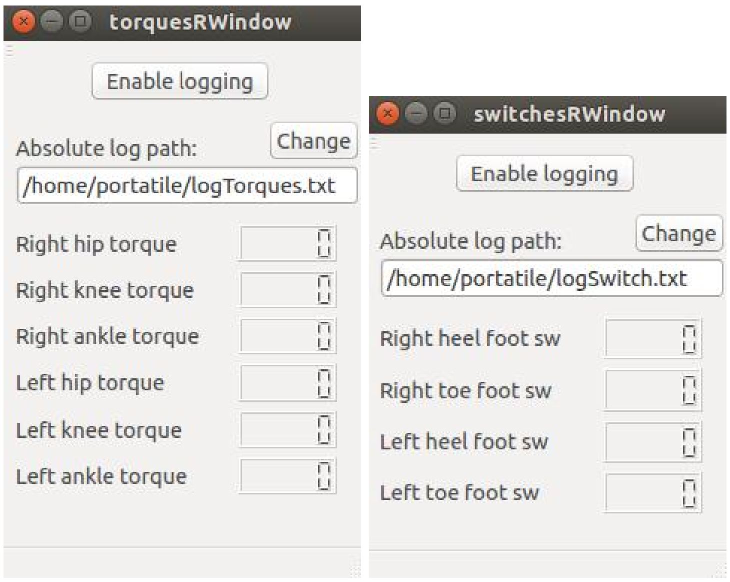
4.3.3.2 *Torques and switches receivers*

The GUIs created for receiving torques and foot-switches data can be seen in figure 24.

These nodes almost provide the same functions as *anglesReceiverUi*, with the only difference that *switchesReceiverUi* reads the data from *switchesTopic*.

The files related to these nodes are: *torquesReceiverUi.cpp*, *torquesrwindow.cpp*, *torquesrwindow.h*, *torquesrwindow.ui*, *switchesReceiverUi.cpp*, *switchesrwindow.cpp*, *switchesrwindow.h*, *switchesrwindow.ui* (appendix A.1).

A graph describing the relation between the nodes, obtained using the *rqt_graph* function, can be seen in figure 25.


torquesRWindow

Enable logging

Absolute log path: Change

/home/portatile/logTorques.txt

Right hip torque 0

Right knee torque 0

Right ankle torque 0

Left hip torque 0

Left knee torque 0

Left ankle torque 0

switchesRWindow

Enable logging

Absolute log path: Change

/home/portatile/logSwitch.txt

Right heel foot sw 0

Right toe foot sw 0

Left heel foot sw 0

Left toe foot sw 0

Figure 24: GUIs provided by torquesReceiverUi and switchesReceiverUi

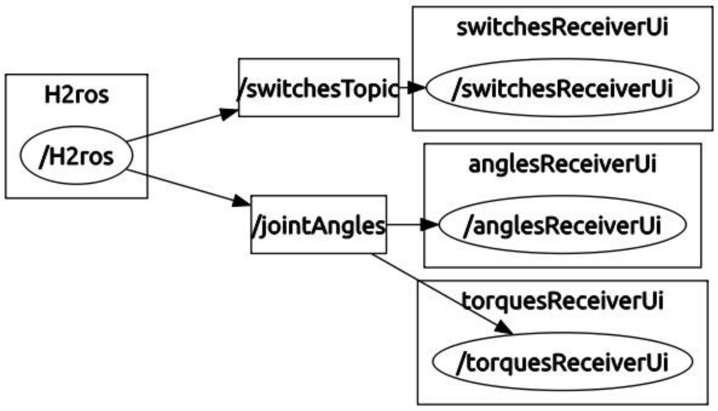


Figure 25: Relations between the acquisition nodes

4.4 NODES FOR SENDING DATA TO THE EXOSKELETON

4.4.1 *Command sender UI*

In order to test the possibility of sending single commands to the exoskeletons, a ROS node named *commandSenderUi* was developed. This node allows the user to easily send all the messages accepted by the exoskeleton (see figure 8 and the following explications) trough a GUI. This GUI too is built using the Qt libraries as explained in the previous section. A generic CAN-frame message is properly filled with the settings provided by the user and when it's ready, the message is published on the *senderTopic* topic. In order to allow all the possible options, more than a window is defined for this program. The related code files are: *commandSenderUi.cpp* , *senderwindow.cpp* with header and ui, *controltype.cpp* with header and ui, *chooseMotor.cpp* with header and ui, *selectStifness.cpp* with header and ui ,*readSetPoint.cpp* with header and ui, *anglesLimits.cpp* with header and ui. The windows forming the GUI are in figure 26.

4.4.2 *H2ros sender*

The H2rosSender node, running on the BBB, creates a listener for the *senderTopic* topic and handles the reception of a *genericFrame* message forwarding its content on the CAN network (to the exoskeleton). To transmit the frame it makes use of the *send* function of the library described in section 4.2.2. All the code related to this node is in *H2rosSender.cpp*.

A graph describing the relation between these nodes can be seen in figure 27.

4.4.3 *Pattern sender UI and H2 pattern sender*

To test the possibility of continuously sending position set-points from the BBB, instead of using the pre-memorized pattern located on the H2-HAL board, the nodes *H2patternSender*, running on the BBB and *patternSenderUi*, running on a remote PC, were developed. The *patternSenderUi* node provides a user interface that allows the definition of the joints to be actuated, of the pattern's velocity and joints' disabling. The velocity can be changed dynamically, while to change the joints to be actuated, message sending needs to be stopped. The communication between the nodes is made with a customized message, named *patternCommand*, that is published on a *patternTopic* topic. This message doesn't include the position set-points for the joints since timings in publishing and reading on standard ROS topics are not precise enough for a control task (a solution for timing precision might be achieved using real-time ROS topics, but the physical com-

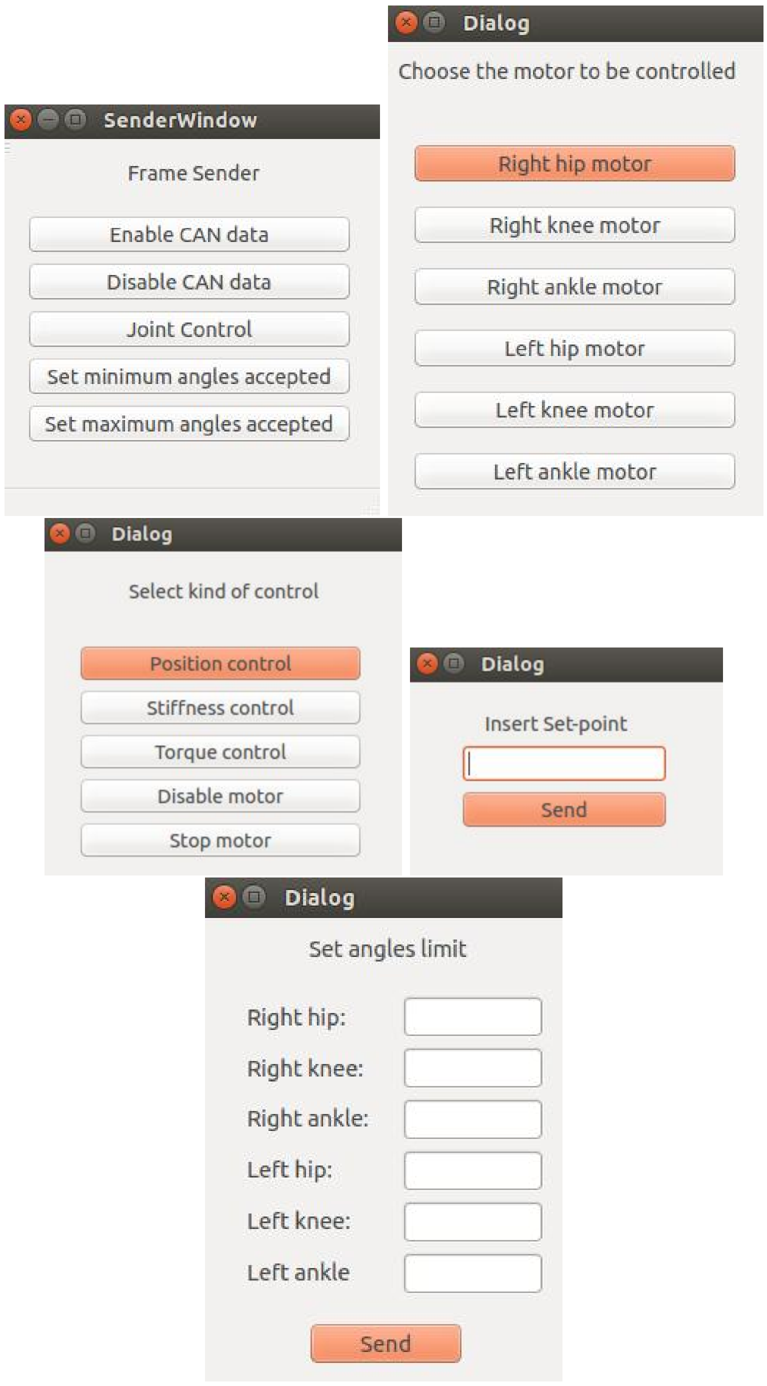


Figure 26: Windows created in commandSenderUi

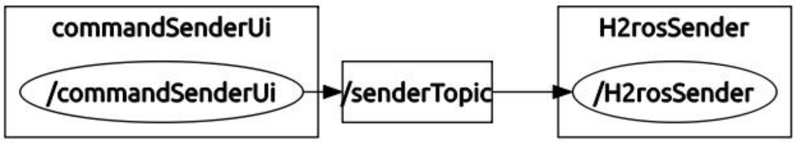


Figure 27: Relation between commandSenderUi and H2rosSender

munication with the BBB trough wi-fi would still be unreliable). The *patternCommand* message includes an array specifying the joints to be actuated (a value of 1 means that message-sending for the corresponding joint is activated, 0 means deactivated), an array specifying the joints to be disabled (a value of 1 means that the corresponding node needs to be disabled), a velocity value (that can vary between 1 and 10) and a reset value, that is set to 1 when message sending is enabled or re-enabled, to let *H2patternSender* node know that it needs to restart sending the pattern from the initial set-point.

The position set-point values are stored in a bi-dimensional array (*pattern*) defined in *H2patternSender*. For each joint, 51 angular positions are used to define a single walking step. *H2patternSender* periodically loads one of these values for each of the activated joints and sends them to the exoskeleton trough the CAN network. The period between a message and the other depends on the velocity setted. No command is sent when all the joints are disabled.

When a *patternCommand* message is received, a dedicated thread (*commandThread*) handles the reception. This thread shares with the main thread the *counter* value, that specifies which value in the set-point's array has to be loaded, a *selectedJoints* array, specifying to which joints message-sending is enabled, a *timer*, defining the period for message-sending and a mutex variable (*send_mutex*) that needs to be locked when the shared values are modified. The *commandThread* checks if the reset value in the message is set to 1 and if so resets the *counter* variable to 0. It then checks if any of the joints needs to be disabled and eventually sends the proper can-disabling message to the exoskeleton. Finally, it switches the *velocity* value setting the *timer* value accordingly.

The GUI provided by *patternSenderUi* is showed in figure 28.

Even if *H2patternSender* is not running in real-time, starting it with a high priority has demonstrated to allow for a correct timing in message-publishing. In particular, at the highest velocity, a message needs to be published every 20ms (corresponding to about one step per-second). In order to understand if publishing timings are correct, a reception program (*H2patternReceiver*) was created. This program, running on a second BBB connected to the publishing one trough a CAN network, records the times of message-reception and calculates the time differences between one set-point and the other. A maximum deviation of 0,4ms from the scheduled time was recorded during the tests. Since the dynamics of the process are relatively slow, such worse-case deviation has no influence on the device's behaviour.

If a real-time underlying kernel is available, (see section 7.4 for examples), the process can be easily modified to run in real-time mode, reducing time-deviations.

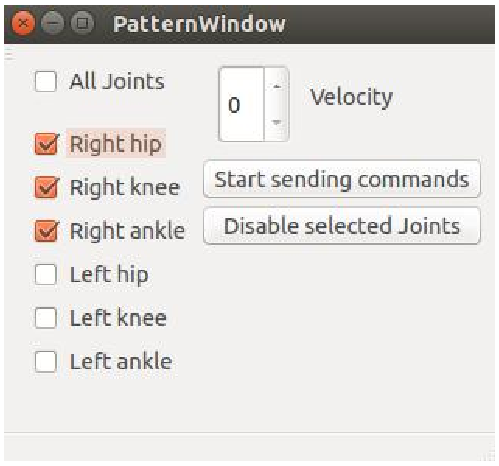


Figure 28: GUI provided by patternSenderUi

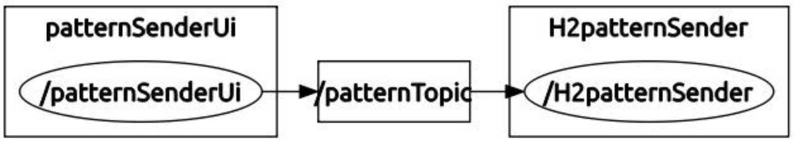


Figure 29: Relation between patternSenderUI and H2patternSender

These nodes are not yet providing a standard interface for sending data to the exoskeleton from a high-level controller, but are created to test the feasibility of correctly sending set-points from an external device as the BBB.

A graph describing the relation between these nodes can be seen in figure 29.

4.5 COMPILING THE NODES

Thanks to the *catkin workspace* structure, all the nodes can be compiled by simply running the command *catkin_make* in the *catkin_ws* folder.

To compile the GUI nodes running on the remote PC, the qt libraries (freeware) must be set-up on the system.

To compile the H2ros node the *glib2* library needs to be set-up on the BBB, it can be done by running:

```
sudo apt-get install libgtk2.0-dev
```

4.5.1 *Cross-Compiling for the Beagle Bone Black*

Since the BBB uses an ARM architecture, programs compiled on a PC (x86 architecture) can't be executed on-board. Since the compilation on a PC is faster (due to the use of a faster processor), it can be useful to use a cross-compile technique. "To cross-compile is to build on one platform a binary that will run on another platform. When speaking of cross-compilation, it is important to distinguish between the build platform on which the compilation is performed, and the host platform on which the resulting executable is expected to run."[59]. In this case the build platform is a generic x86 PC and the host platform is the BBB.

A description of how to cross-compile for the BBB using *Eclipse* IDE could be found in [48]. In particular the *gcc-arm-linux-gnueabi* compilation tool-chain was set-up as follows:

- **C++ compiler**: arm-linux-gnueabihf-gcc-4.8
- **C compiler**: arm-linux-gnueabihf-g++-4.8
- **C++ LINKER**: arm-linux-gnueabihf-g++-4.8
- **ASSEMBLER**: arm-linux-gnueabihf-as

The cross-compilation of single programs could be achieved easily, but a way for cross-compile a full catkin-workspace could not be found.

4.6 LAUNCHING THE NODES

To Launch multiple nodes, ROS provides the *roslaunch* function. This tool takes as input a *launchfile* (a XML configuration file) and a package in which that file is defined. As described in [14], *launchfiles* have multiple functions, in particular they can specify the nodes to be started with the relative options and parameters.

Nodes can be launched in different ways, depending on where the *rosmaster* execution is setted.

One option is to launch the rosmaster node on the BBB; in this case the following script (LRosCore.sh) is executed:

```
#!/bin/bash
source /opt/ros/indigo/setup.bash
source /home/ubuntu/catkin_ws/devel/setup.bash
export ROS_IP=BBB_IP (192.168.0.102 if connected via usb,
    192.168.0.102 via wireless)
export ROS_MASTER_URI=http://BBB_IP:11311
roslaunch h2Iterface BBBLauncher.launch
```

This script sources the required ROS files, sets the variable ROS-MASTER-URI to its IP and finally launches the nodes trough the *BBBLauncher.launch* file wich is defined as:

```
<launch>
 <node pkg="h2Interface" name="H2rosSender" type="H2rosSender"/>
 <node pkg="h2Interface" name="H2ros" type="H2ros" args="-rf 1"/>
 <node pkg="h2Interface" name="H2patternSender" type="H2
     patternSender"/>
</launch>
```

Roslaunch automatically starts the *rosmaster* if it can't find one already running in the machine. The execution of the script can be done manually trough SSH or automatically at start-up running *"crontab-e"* and adding the line:

```
@reboot /path/to/script
```

To run the nodes providing the user interfaces on a remote PC using the master launched on the BBB, the following script (LGUI.sh) can be used:

```
#!/bin/bash
source /opt/ros/indigo/setup.bash
source /home/portatile/catkin_ws/devel/setup.bash
export ROS_IP=PC_IP(192.168.7.1 if connected via USB)
export ROS_MASTER_URI=http://BBB_IP:11311
roslaunch h2Interface Gui.launch
```

The script sources ROS files and specifies the address of the ROS master node (this procedure allows ROS to run across multiple machines), it then runs the launchfile *Gui.launch*:

```
<launch>
 <node pkg="h2Interface" name="anglesReceiverUi" type="
     anglesReceiverUi"/>
 <node pkg="h2Interface" name="torquesReceiverUi" type="
     torquesReceiverUi"/>
 <node pkg="h2Interface" name="switchesReceiverUi" type="
     switchesReceiverUi"/>
 <node pkg="h2Interface" name="commandSenderUi" type="
     commandSenderUi"/>
 <node pkg="h2Interface" name="patternSenderUi" type="
     patternSenderUi"/>
patternSenderUi
</launch>
```

If the launch of some nodes is not required, they can simply be commented in the corresponding launchfile.

If the *rosmaster* node wants to be set on another machine, the nodes running on the BBB can still be launched manually trough SSH setting the correct ROS-MASTER address in the *LRosCore* script. Otherwise they can be launched trough *roslaunch* directly from the remote

PC as explained in [15]. In such case an "*environment-loader script*" needs to be available on the BBB to source the ROS files; the remote launcher will automatically connect trough SSH to the BBB, run this script and spawn the required nodes.

4.7 REPOSITORY FOR THE CODE

All the created code was uploaded to a git repository on *BitBucket*. Using a repository allowed for easy code sharing between project members, easy revision control and easy "roll-back" to previous code-versions. Informations on how to use the repository structure can be found on the Git tutorial-documentation site [23].

5 ACQUISITION EXPERIMENTS

5.1 INTRODUCTION

In order to get data useful for an application test of the RT-neuromusculo-skeletal model, some experiments involving the combined use of the exoskeleton and of the EMG sensors were realized. This chapter describes the experimental set up and reports a small sample of the acquired data.

5.2 HARDWARE INVOLVED

The experiments involved the use of a H2 exoskeleton, described in 2.2, that was connected trough a CAN network to a BBB board, described in 2.3. A Wi-fi connection between the BBB and a notebook running ROS was created as explained in 2.3.6. This PC was then connected to the Cometa's EMG acquisition system described in section 2.4.1. The application of the exoskeleton to the user allows enough space for the collocation of the EMG sensors on the legs (see figure 30). Since all the EMG sensors can communicate to the acquisition system via wi-fi, no communication cable needs to be used during the experiment. Also, being both the exoskeleton and the BBB powered by a battery, no power cable had to be used. The users were consequently able to move without any cable-interaction.

5.3 SYNCHRONIZATION OF DEVICES

Since data relative to the exoskeleton joints and data from the EMG signals are read by separated devices, a synchronization system had to be used. As described in section 4.3.2.2, the BBB is configured for receiving a trigger signal with a tension level of 3,3V on the port GPIO7 (pin P9-42); when a hi-to-low transition is detected, the timestamp value included in the acquired messages is reset to 0. At the same time, the *Cometa* acquisition device can be set to log data only when a constant 3,3V signal is provided on its trigger port.

It is then possible to use an extern voltage source to generate the trigger signal: a manual switch enables the output of the source for both the BBB (trough two simple signal cables) and the Cometa (trough a BNC connector). Alternatively, a second BBB can be used to rise the trigger signal through a GPIO pin. Once the BBB (installed on the exoskeleton) has registered the "time 0",it can be disconnected from the trigger generator and the experiment can begin.

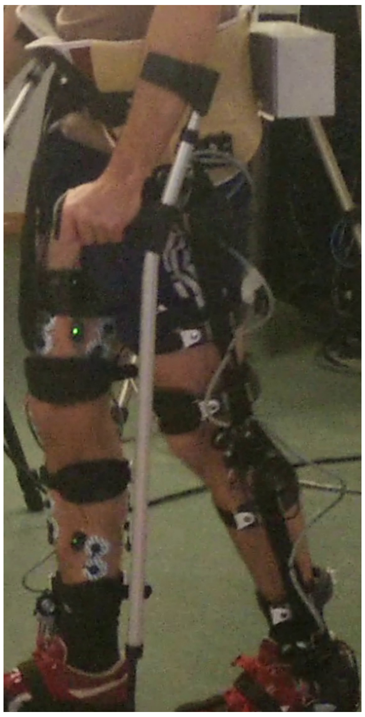


Figure 30: Combined use of the H2 exoskeleton and of the EMG sensors

Also the PC acquiring the EMG data provides an interface with the middleware. The Cometa receiving unit provides both analog and digital (USB) outputs. The latter option was preferred, as it did not require additional A/D conversion hardware; however, drivers and API are only available for Windows operating system. Since the application of ROS to such OS is still in an experimental stage, a different middleware, YARP (*Yet Another Robot Platform*), was used to interface the EMG acquisition software. Thanks to recent efforts for interoperability between the two middleware, this YARP-based Cometa EMGreader module communicates with the roscore, and publishes ROS-compatible messages to a /emgData ROS-topic.

A third notebook, running a Linux OS, hosts the ROS-master process, runs CEINMS (*Calibrated EMG-Informed Neuromusculoskeletal Modelling Toolbox*) and displays the acquired data trough the *rqt_plot* utility.

Due to the modularity of the middleware, similar experiments could be made with a different distribution of the running nodes; for example both the ROS-core and the CEINMS model could be executed on the BBB. Also different kind of EMG sensors could be used as long as they provide an appropriate interface with the middleware.

Data-log can consequently be made on any PC connected to the middleware (it can be for example the same one visualizing the data) in a *rosbag* file or in a txt format using the GUI of the acquisition nodes (only for the data from the exoskeleton).

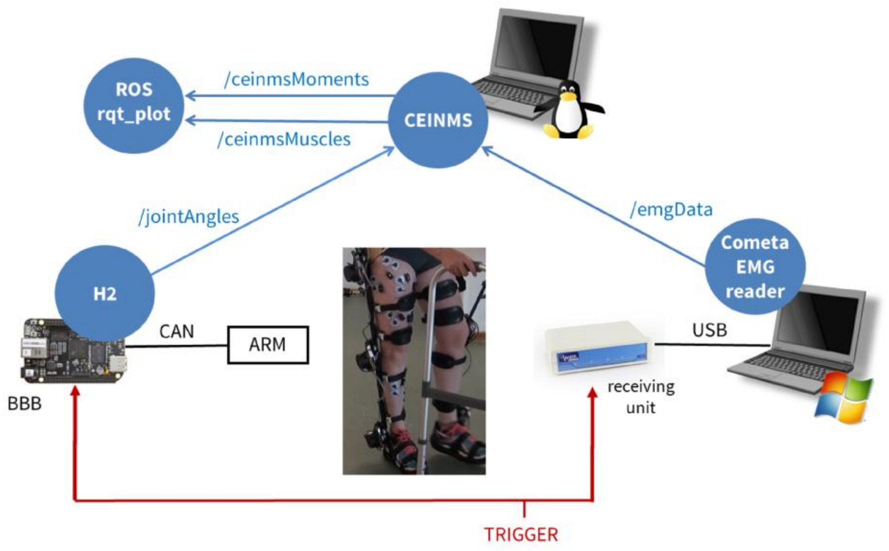
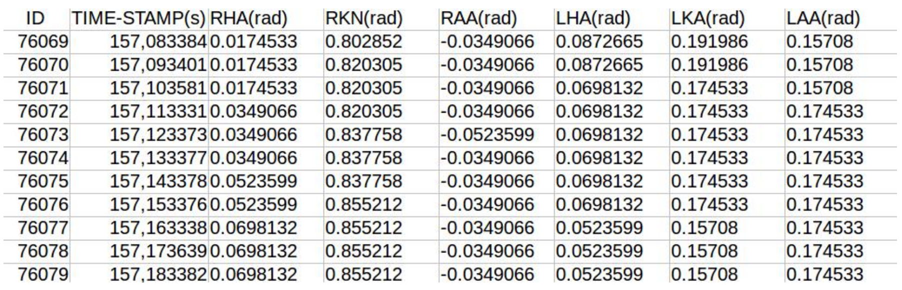


Figure 31: Scheme for device connection during the experiments



ID	TIME-STAMP(s)	RHA(rad)	RKN(rad)	RAA(rad)	LHA(rad)	LKA(rad)	LAA(rad)
76069	157,083384	0.0174533	0.802852	-0.0349066	0.0872665	0.191986	0.15708
76070	157,093401	0.0174533	0.820305	-0.0349066	0.0872665	0.191986	0.15708
76071	157,103581	0.0174533	0.820305	-0.0349066	0.0698132	0.174533	0.15708
76072	157,113331	0.0349066	0.820305	-0.0349066	0.0698132	0.174533	0.174533
76073	157,123373	0.0349066	0.837758	-0.0523599	0.0698132	0.174533	0.174533
76074	157,133377	0.0349066	0.837758	-0.0349066	0.0698132	0.174533	0.174533
76075	157,143378	0.0523599	0.837758	-0.0349066	0.0698132	0.174533	0.174533
76076	157,153376	0.0523599	0.855212	-0.0349066	0.0698132	0.174533	0.174533
76077	157,163338	0.0698132	0.855212	-0.0349066	0.0523599	0.15708	0.174533
76078	157,173639	0.0698132	0.855212	-0.0349066	0.0523599	0.15708	0.174533
76079	157,183382	0.0698132	0.855212	-0.0349066	0.0523599	0.15708	0.174533

Figure 32: Sample of the acquired angles data

The information from the devices can be easily correlated since they share the same starting-time.

A schematic structure of the implemented solution can be seen in figure 31.

5.4 ACQUIRED DATA

The experiments were realized with four healthy subjects. A sample of the data acquired trough the *anglesReceiverUi* node while the exoskeleton is walking is in figure 32.

Each message has a time distance of 10 ms from the previous, so the sending frequency of 100Hz is respected. Using a larger number of messages confirms the correctness of the frequency and that no "*holes*" are present in the can-communication.

Using bag files for data logging allows for later data-playback and easy data visualisation. Using the *rqt_plot* or the *rqt_bag* tools, static or dynamic graphs of the acquired data can be plotted.

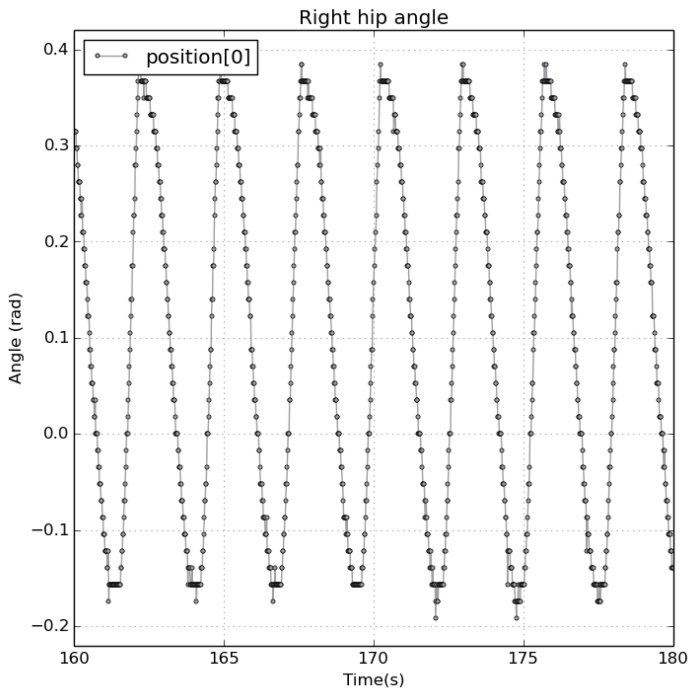


Figure 33: Sample of the acquired right hip angles

Some examples of acquired position data for a healthy user during a walking period of 20 seconds are shown: in figure 33 are the right hip angles, in figure 34 the right knee angles, in figure 35 right ankle angles, in figure 36 left hip angles, in figure 37 left knee angles and in figure 38 left ankle angles.

The figures show how acquired data are not affected by anomalies as noise or "holes".

Twelve EMG sensors were applied to twelve different muscles during the experiments, some examples of the acquired EMG data during the same period of time are in figure 39 and 40.

Analysing these data and relating them with the acquired position angles is not an objective of this work, but the periodicity of the signal can be easily observed and related with the walking period.

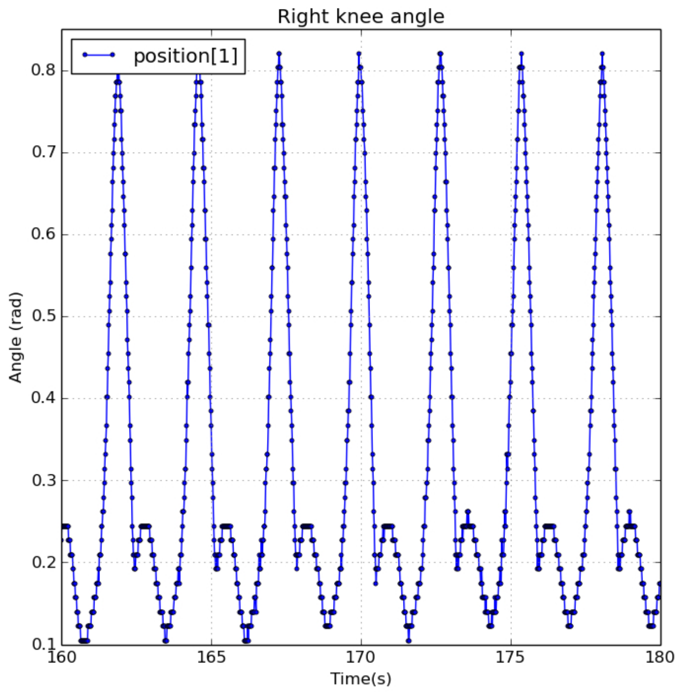


Figure 34: Sample of the acquired right knee angles

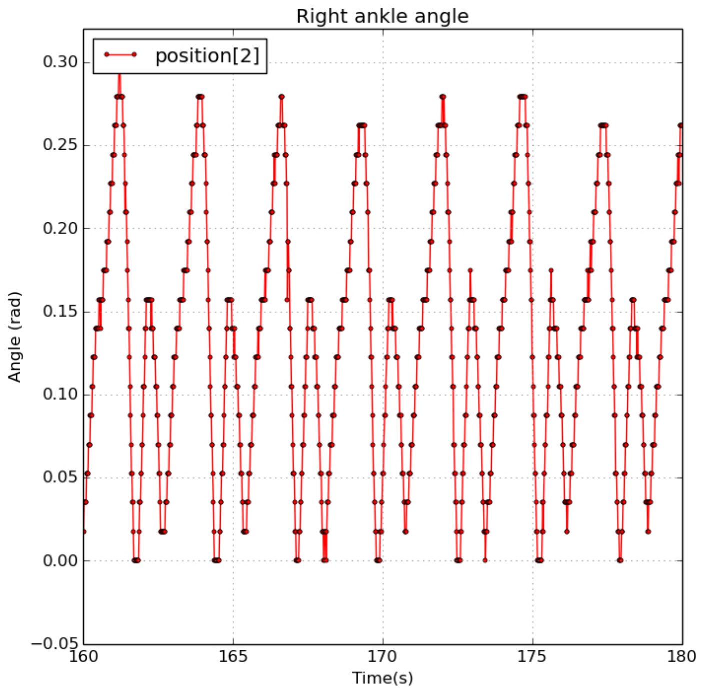


Figure 35: Sample of the acquired right ankle angles

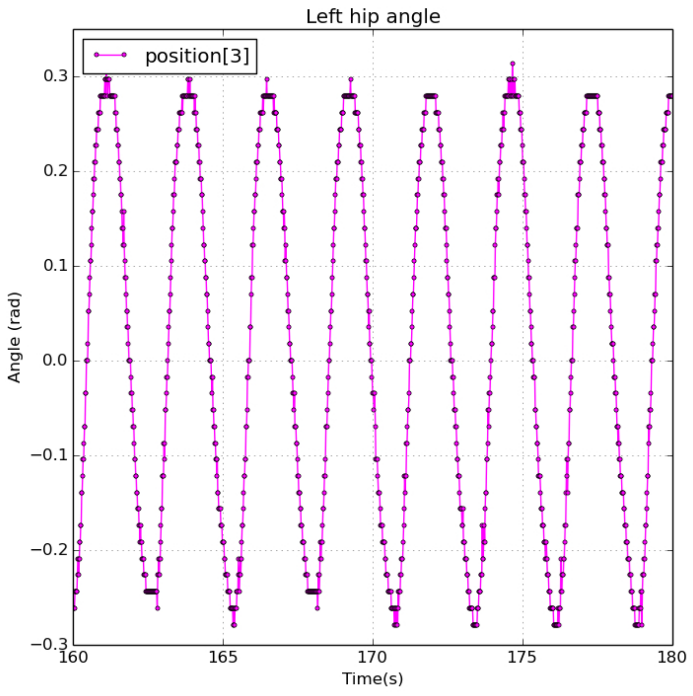


Figure 36: Sample of the acquired left hip angles

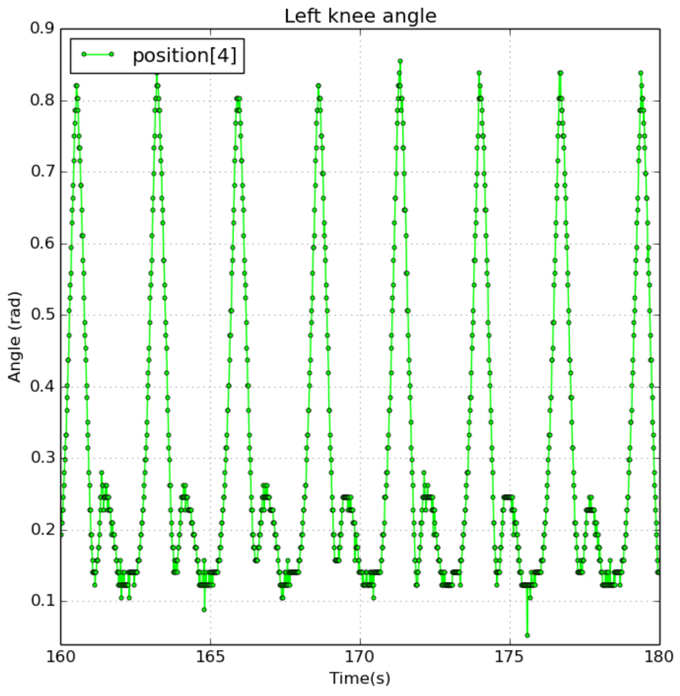


Figure 37: Sample of the acquired left knee angles

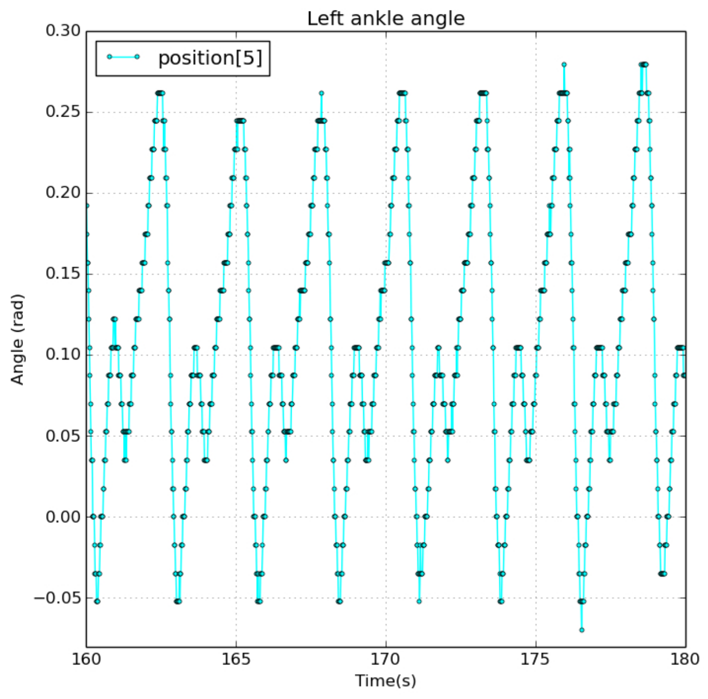


Figure 38: Sample of the acquired left ankle angles

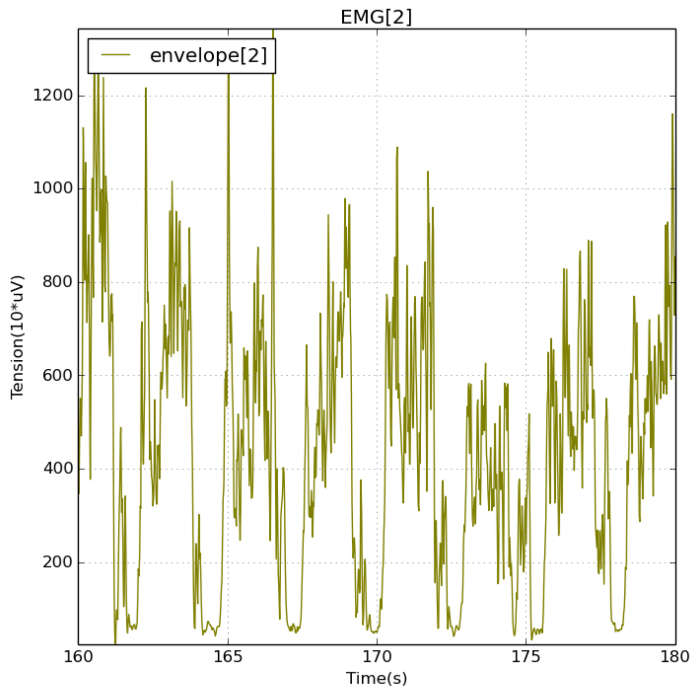


Figure 39: Sample of the acquired EMG data (Gracilis right muscle)

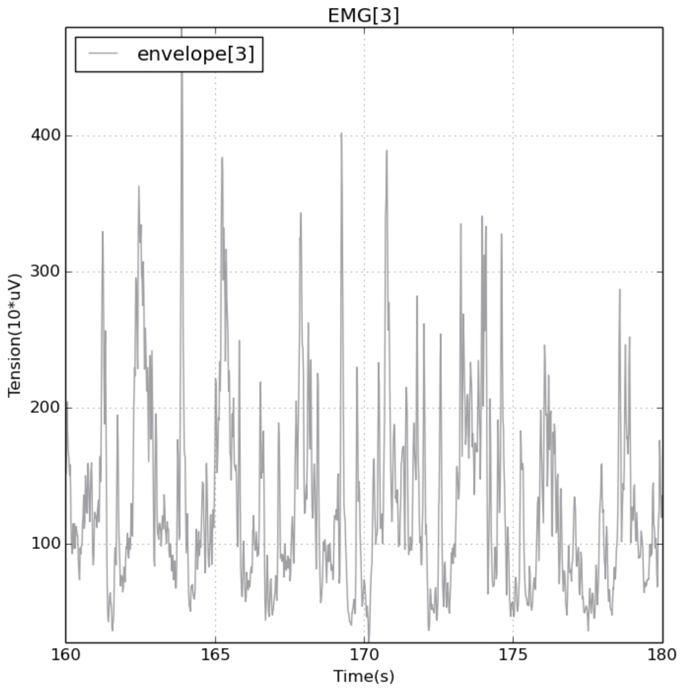


Figure 40: Sample of the acquired EMG data (Lateral Gastrocnemius right muscle)

6 INTERFACING COMPLIANT ACTUATORS

6.1 INTRODUCTION

As was pointed out in section 2.2, the H2 exoskeleton is only a testing device used by Biomot to gather information on the gait process and about the human-robot interaction.

A new exoskeleton, that should provide an improvement in both energy efficiency and in the human-robot safe interaction, is being developed by the consortium. One of the most important differences between the H2 exoskeleton and the new developed one, that allows for the achievement of such objectives, is the use of *compliant actuators*.

Following is a brief description of compliant actuators and of the MACCEPA actuators used my BioMot. It is then presented the implementation of the CAN-based communication protocol and of the ROS nodes (with graphic user interfaces) used to communicate with the actuators.

6.2 COMPLIANT ACTUATORS

"In classical robotic applications, actuators are preferred to be as stiff as possible to make precise position movements or trajectory tracking control easier (faster systems with high bandwidth). The biological counterpart is the muscle that has superior functional performance and a neuro-mechanical control system that is much more advanced at adapting and tuning its parameters. The superior power-to-weight ratio, force-to-weight ratio, compliance, and control of muscle, when compared with traditional robotic actuators, are the main barriers for the development of machines that can match the motion, safety, and energy efficiency of human or other animals. One of the key differences of these systems is the compliance or springlike behaviour found in biological systems.

In the growing fields of wearable robotics, rehabilitation robotics, prosthetics, and walking robots, variable stiffness actuators (VSAs) or adjustable compliant actuators are being designed and implemented because of their ability to minimize large forces due to shocks, to safely interact with the user, and their ability to store and release energy in passive elastic elements. This new type of actuator is not preferred for classical position-controlled applications such as pick and place operations but is preferred in novel robots where safe human-robot interaction is required"[44].

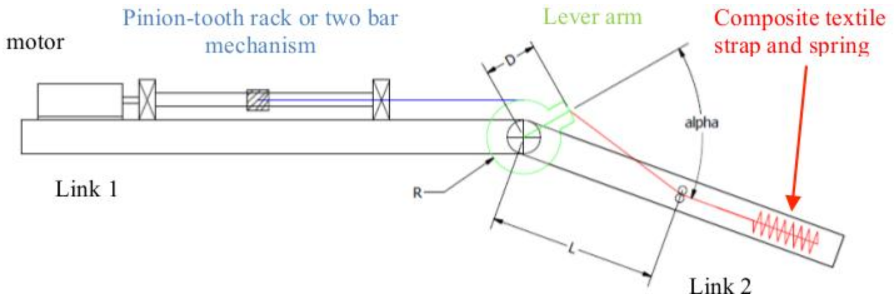

motor
Pinion-tooth rack or two bar mechanism
Lever arm
Composite textile strap and spring
D
alpha
R
L
Link 1
Link 2

Figure 41: flat MACCEPA concept scheme

6.3 BIOMOT'S ACTUATORS

"Two novel modular compliant actuators will be used for actuating the ankle and the knee joint of the exoskeleton. The first actuator is a flat version of the MACCEPA. The second actuator consists of an actively controlled additional spring working in parallel with the flat MACCEPA. Novelty of the proposed MACCEPA designs, when compared to both previous and current designs, is the introduction of a ball screw that drives the lever arm of the actuator. Introduction of the spindle drive leads to better inertia distribution of the actuator, which improves the mechanical and control efficiency of the actuator and brings more comfort and safety for a wearable device" [54]. The flat-MACCEPA concept scheme can be seen in figure 41.

As illustrated , the actuator consists of three bodies: Link 1, Link 2 and the lever arm. Its working principle is based on force component, exerted by the spring, acting on the Link 2 (consider Link 1 to be grounded), and thus trying to align Link 2 with the lever arm. If these two are aligned, no force, and thus no torque, is exerted and actuator is in its equilibrium position. Stiffness of the actuator can be changed by changing pre-tension/compression of the spring (depending on whether tension or compression spring is used), thus effectively altering MACCEPA's characteristic.

Both the actuators were designed to give a peak torque of 25Nm ,chosen to guarantee about a 50% assistance to muscle weakness.

During this work only the ankle actuators were available for testing (figure 42). The mass and inertia of this prototype actuators still needs to be optimized.

In these actuators the angular positions of the lever-arm and of the second link can be read through respectively a magnetic and an optic encoder. In a second moment, a further sensor reading the length of the spring or the force that it exerts might be added to the system.

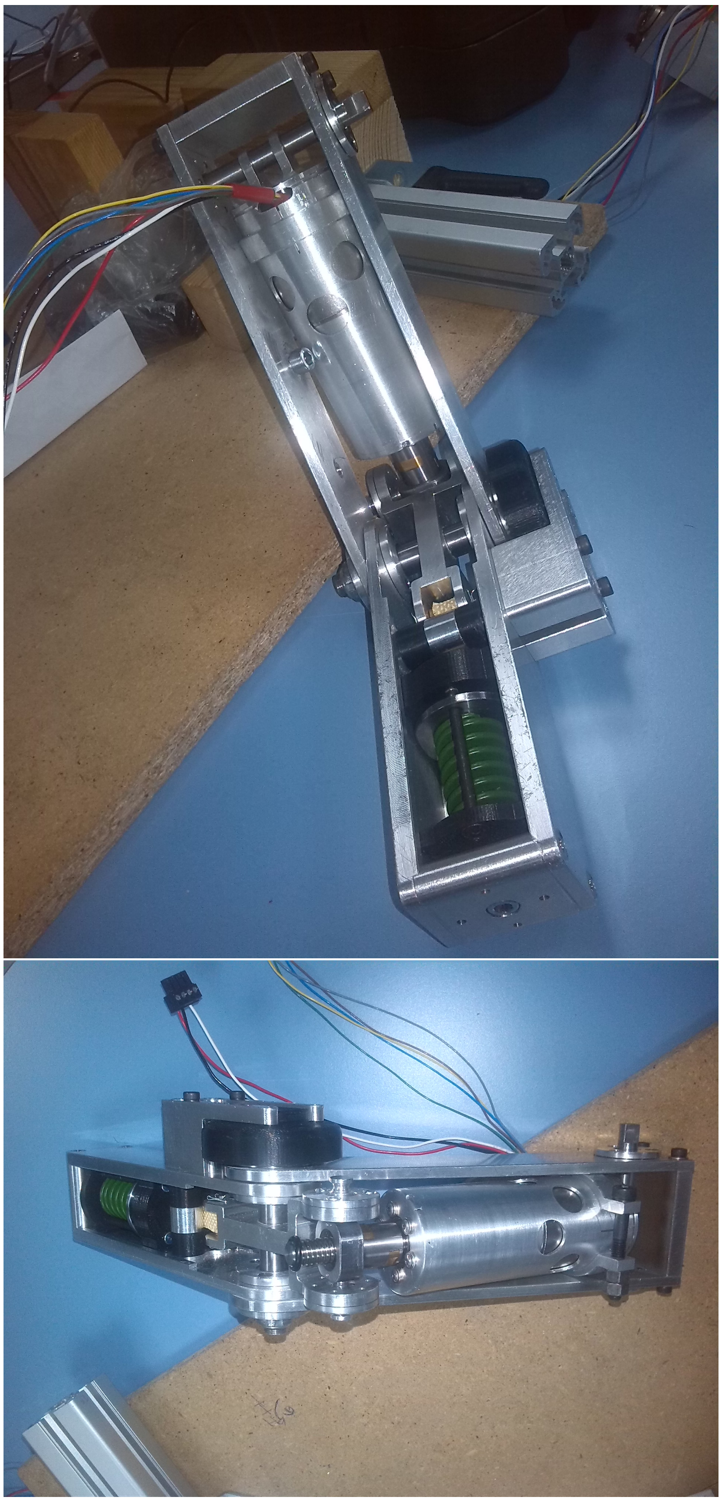


Figure 42: Picture of the prototype for ankle's actuation

6.3.1 *Variable compliance*

An option for the Biomot group, is to improve the capabilities of the actuators making their physical stiffness dynamically variable (*Variable compliance actuators*). "Variable compliance actuators are actuators which are capable of passively regulating their physical compliance. Obvious advantages that a variable stiffness implementation offers when compared with the fixed passive compliance units are the ability to regulate both stiffness and position and the wide range of stiffness and energy storage capability. The advantages gained by this capability are clearly shown in mammals: muscles and tendons change their stiffness as a function of the motion/task they have to perform. Arm muscles assume a stiff configuration when the arm has to perform an accurate task, while they are compliant when they are performing the "loading" phase of a throw. "[49] To achieve this objective, a second motor for the regulation of the compression of the actuator's spring needs to be added to the system (forming in this way a proper *MACCEPA actuator*). This modification leads to an improvement in the system's complexity, weight and cost, so it's application is not still certain and needs to be carefully evaluated. Also since a lot of force is required for stiffness changes on the spring, such modifications can't be instantaneous and might require more than a step-cycle.

6.4 INTERFACING THE ACTUATORS

In order to gather information about the actuator's behaviour and to develop a low-level torque controller, an interface allowing the user to easily visualize data coming from the sensors and to set the control parameters had to be developed. The same solution had do be provided for the MACCEPA actuators used in the H2R (*Integrative approach for the emergence of Human-like locomotion*) project [42] to actuate the lower limbs of a human-like robot. To save time and resources, a generic interface able to handle both the solutions was designed.

The MACCEPA actuators interact with 2 embedded boards, one providing power-handling and the other providing signal-handling and control trough a PIC microprocessor. On these boards the encoders outputs are read and the low-level controller is run. The control board can communicate with other devices trough a CAN network. A picture of the used control board is in figure.43

6.4.1 *Data structurization*

The firmware running on the boards was programmed to send data as shown in figure 44. The data were structured in order to minimize the number of CAN-messages to be sent and consequently reduce

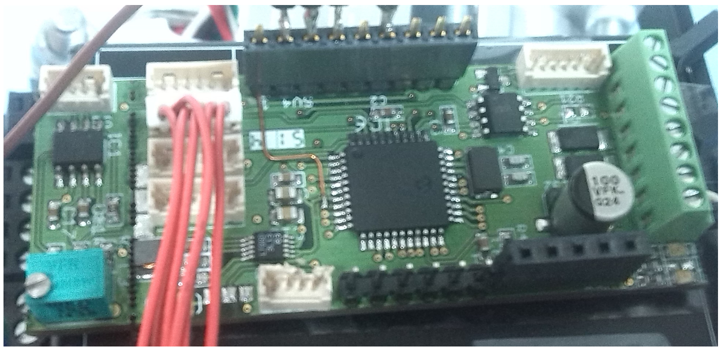


Figure 43: Embedded control board for the MACCEPA actuators

the risk of saturation on the CAN-bus once all the actuators will be connected.

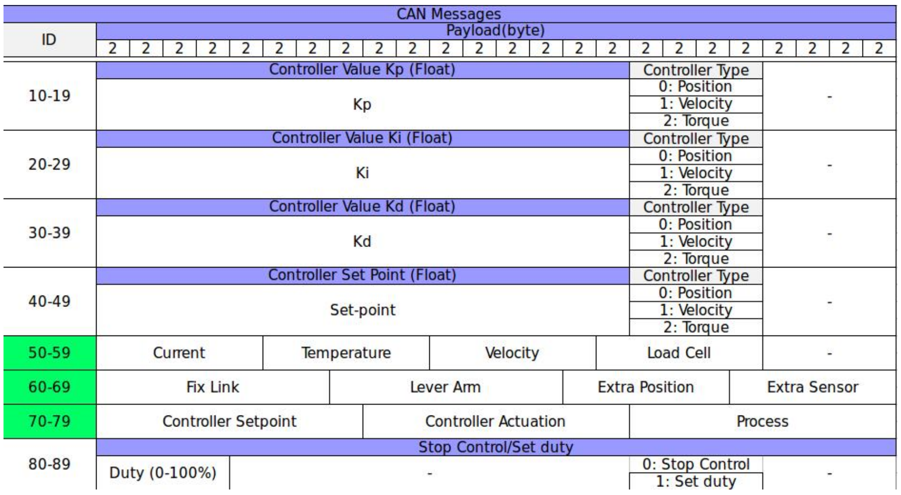
The sent data are:

- Current absorbed by the motor (mA);
- Temperature of the board (°C);
- Velocity of the motor(rpm);
- Force sensed by the load-cell (N);
- Position of the fixed link (deg);
- Position of the lever arm (deg);
- Controller set-point (depending on the chosen controller);
- Controller actuation;
- Controller process variable.

While the data that can be sent to the embedded board are

- Kp control parameter (and type of control selection);
- Ki control parameter;
- Kd control parameter;
- Set-point;
- Duty cycle set-point;
- Control disable message.

Some extra-space is reserved for future utilization.



CAN Messages

ID	Payload(byte)				
	2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2			2 2 2 2	2 2 2 2
10-19	Controller Value Kp (Float): Kp			Controller Type: 0: Position, 1: Velocity, 2: Torque	-
20-29	Controller Value Ki (Float): Ki			Controller Type: 0: Position, 1: Velocity, 2: Torque	-
30-39	Controller Value Kd (Float): Kd			Controller Type: 0: Position, 1: Velocity, 2: Torque	-
40-49	Controller Set Point (Float): Set-point			Controller Type: 0: Position, 1: Velocity, 2: Torque	-
🟩 50-59	Current	Temperature	Velocity	Load Cell	-
🟩 60-69	Fix Link	Lever Arm		Extra Position	Extra Sensor
🟩 70-79	Controller Setpoint	Controller Actuation		Process	
80-89	Stop Control/Set duty: Duty (0-100%)	-		0: Stop Control, 1: Set duty	-

Figure 44: Can messages sent (green) and received by the actuators

6.5 ROS NODES AND GUI

To interface the embedded boards with the ROS middleware and the user, a new catkin workspace was created, named *h2r*. As was done for the H2 exoskeleton, some of the nodes are dedicated to the reception of messages from the actuators and some for message-sending.

6.5.1 *Nodes for receiving data*

6.5.1.1 *H2Rros*

The node reading incoming CAN data from the actuators is *H2Rros*. As the name suggests, this node is very similar to the previously developed *H2ros* node (4.3.2). The differences reside in the kinds of data that are transmitted and consequently in the topics/messages to be created. Also, parsing the CAN data is more complicated since a single message is generally split on more than a byte as described in figure 44.

The CAN data to be parsed are those with ID 50-59,60-69 and 70-79. They are read trough the same CAN library previously developed, described in section 4.2.2. Once each of these messages is properly parsed, its values are stored in a corresponding ROS message and sent on a ROS topic:

- Data with identifier **50-59** are stored in a *mSensor.msg* message defined as:

```
Header header

int16 velocity
int16 current
int16 loadCell
```

```
int16 temperature
```

this message is published on the *sensorsTopic* topic.

- Data with identifier **60-69** are stored in a *mPositions.msg* message defined as:

```
Header header

int16 external
int16 position2
int16 extraPosition
int16 extraSensor
```

this message is published on the *positionsTopic* topic.

- Data with identifier **70-79** are stored in a *mPID.msg* message defined as:

```
Header header

int16 error
int16 actuation
int16 process
```

this message is published on the *PIDTopic* topic.

Even if trigger-handling on the reception node running on the BBB is not yet necessary, its implementation is still present since it will probably become useful in the future.

6.5.1.2 *Data receiving nodes*

The published messages can be received by 4 nodes: *positionsReceiverUi, sensorsReceiverUi, PIDReceiverUi* and *positionsGraph*. The first 3 nodes are very similar to the *AnglesReceiverUi, TorquesReceiverUi* and *SwitchesReceiverUi* nodes previously described in 4.3.3.

A dedicated thread creates a ROS listener that waits for incoming messages on the corresponding topic and handles the reception updating a structure shared with the main thread, where the graphic interface runs. If the corresponding option is enabled, data are saved in a txt file whose pattern is specified by the user. The GUIs are built using the qt libraries and periodically update the output of the LCD numbers (setting a timer) with the values in the shared structure.

The developed GUIs are displayed in figure 45.

6.5.1.3 *Drawing received data*

During the setting of the PID controller running on the embedded board, a graph displaying both the angular position of the *link2* and

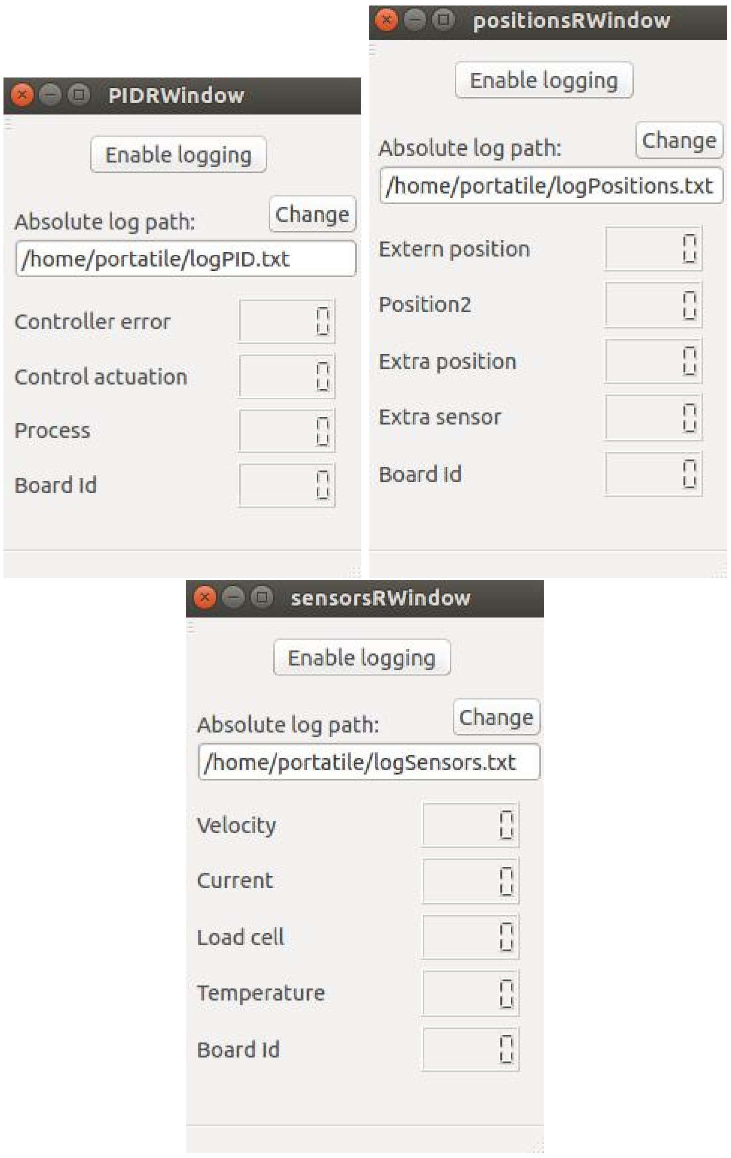


Figure 45: GUIs for receiving data from the actuators

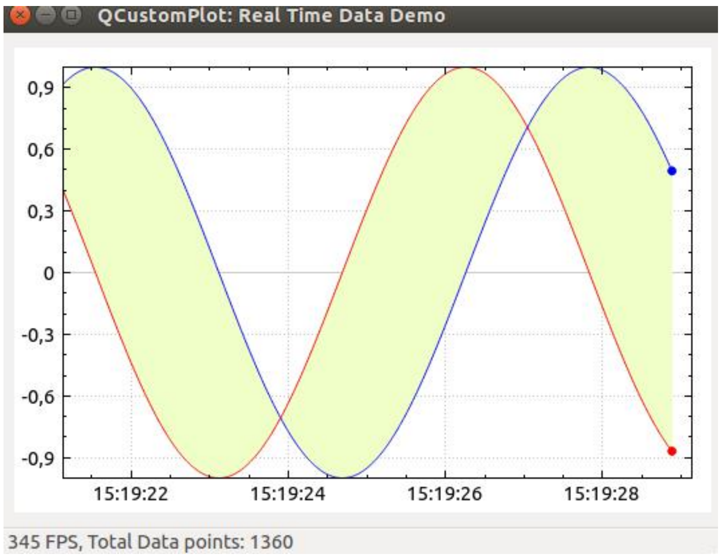


Figure 46: Example for real time positions plot

of the *lever-arm* is useful. To plot these data in "real-time", an expansion library for qt was used, named *Qcustomplot*. This plotting library focuses on making good looking, publication quality 2D plots, graphs and charts, as well as offering high performance for real-time visualization applications [1].

To use the library, the files *qcustomplot.cpp* and *qcustomplot.h* need to be added to the project; in particular *qcustomplot.h* needs to be included in the UserInterface files that use its functions.

The node providing the graph interface is named *positionsGraph*; as *positionsRwindow*, it subscribes to the *positionsTopic* and uses the information from the received messages to update the structure *dataSt*. The GUI window periodically checks the values in the structure and prints the position values on a graph in which the x axis displays the time currently running on the machine. An example plotting two sinusoids with a difference of phase is in figure 46.

An alternative to the use of this node, is the use of the *rqt_plot* function. This function allows to plot on a graph data that is being published on a specified ROS topic. This function requires the set-up of other ROS components (graphic packages) and is less customizable than the provided qt solution.

A graph showing the relation between the receiving nodes can be seen in figure 47.

The code of the developed software is in the files *H2Rros.cpp*, *positionsReceiverUi.cpp*, *PIDreceiverUi.cpp*, *sensorsReceiver.cpp*, *positionsGraph.cpp*, *positonsrwindow.cpp* with hearder an UI file, *sensorsrwindow* with header

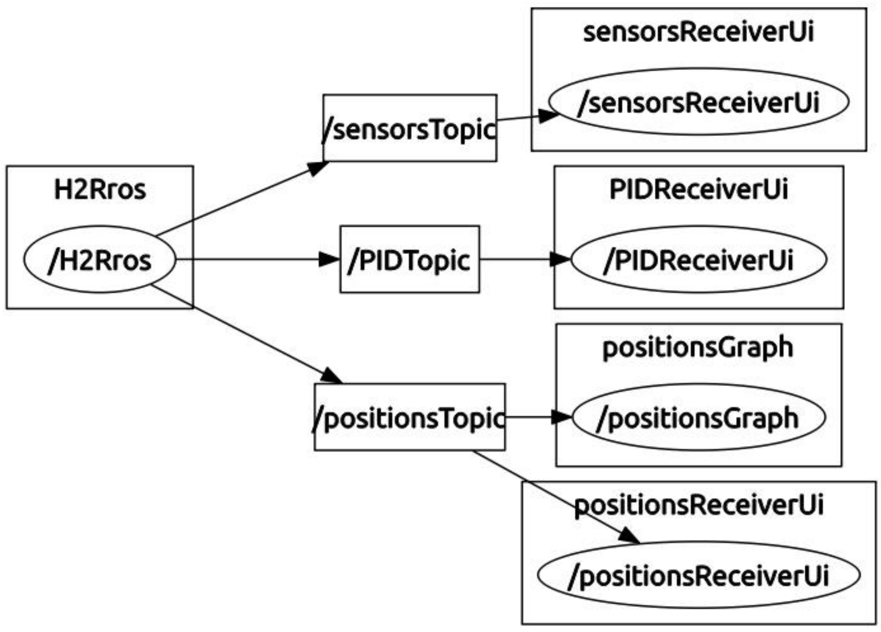


Figure 47: Receiving nodes relations

and UI file, *PIDrwindow*, with header and UI file, *graphWindow.cpp* with header and UI file (appendix A.2).

6.5.2 *Nodes for sending data*

In order to easily set the parameters of the embedded controllers and to make tests during the set-up of the actuators used both in Biomot and in the H2R project, an interface for sending commands in the form of can-messages was developed.

The GUI is provided by the *commandSenderUi* node.
Each of the control board running on the exoskeleton or on the H2R robot will be identified by an integer ID. This ID won't be included in the message-information as was previously done with H2 when sending a control message, but will be added to the CAN-identifier. For example, to set the control set-point of the board which identifier is 2, a can message with ID 42 needs to be sent (see figure 44).

To choose the board to be controlled, the first window asks the user to specify its ID and if it's in the correct range of values (0-9), it opens the second window (*senderWindow*). Through the *senderWindow* the user can specify which kind of control will be used, change the parameters of the embedded PID controller and its-set point. Also a fixed duty-cycle can be specified to run preliminary tests. The created windows can be seen in figure 48.

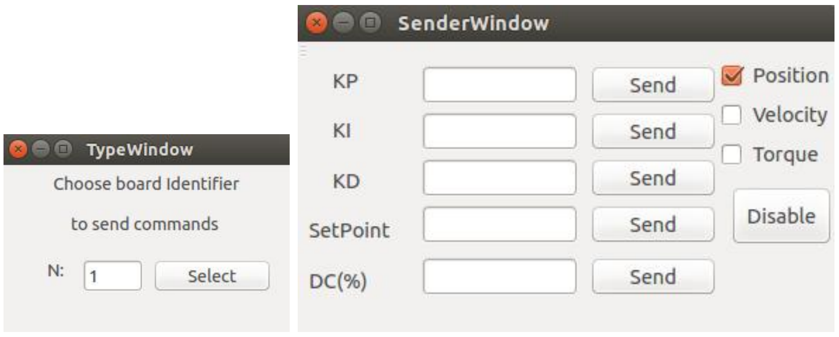


Figure 48: Windows created by commandSenderUi

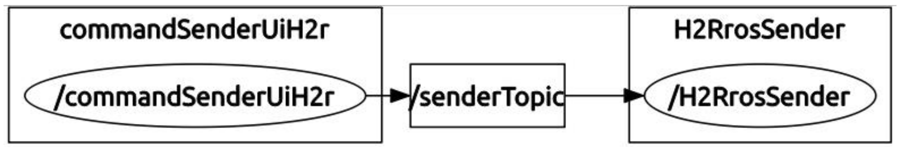


Figure 49: Sending nodes relation

The user-provided informations are transmitted from the *command-SenderUi* node to the *H2RrosSender* node running on the BBB through the *senderTopic* topic that makes use of *genericFrame* messages defined as:

```
int32 id
int32 type
float32 data
```

The *H2RrosSender* node handles the reception of these messages on a dedicated thread, it then converts the data in order to store them as showed in figure 44 and sends them through the CAN network to the embedded board using the library described in 4.2.2.

A graph describing the relation between the sending nodes can be seen in figure 49.

The code of the developed software is in the files *H2RrosSender.cpp*, *commandSenderUiH2r.cpp*, *senderwindow.cpp* with hearder and UI file, *typewindow.cpp* with header and UI file (appendix A.2).

The nodes can be compiled through a *CMakeLists.txt* file as was explained in section 4.5.

They than can be launched both remotely or automatically as was described in section 4.6 for the H2 nodes.

7 CONTROL INTERFACE

7.1 INTRODUCTION

Exoskeletons are complex systems that involve a large number of sensors and actuators. Studying the possible ways to control their motion, allowing for multiple functions and a comfortable, safe human-robot interaction is one of the main goals of the Biomot project. Both the high-level and low-level control techniques are under development, but their direct implementation is not a problem faced in this work. This chapter instead presents the problem of creating a flexible control architecture that will support the creation of such controllers. In the following, the possible ways of interfacing the high level and low level controllers of the system are described, pointing out the advantages and disadvantages of each solution. The possibility of applying an already defined high-level control structure (ROS-Control) to the exoskeletons is then investigated. Finally, several ways to give real-time capabilities to the Beagle Bone Black board are described; the execution of the controllers in real-time mode is necessary to achieve a deterministic behaviour of the system.

7.2 POSSIBLE CONTROL-ARCHITECTURES

Both the H2 exoskeleton and the newly developed one make use of six actuated joints to support the user movements. Each of these joints requires a dedicated low-level controller, usually in the form of a PID controller, that makes the actuators follow a desired position or torque set-point using a fast feed-back loop. The high level controllers elaborate the set-points for the low level controllers using all the informations coming from the middleware system. Three different options for physically interfacing the high and low level controllers of the exoskeleton were individuated and described in sections 7.2.1, 7.2.2 and 7.2.3.

7.2.1 *Using only a central controller*

In the first scenario, the whole control is administrated by a central unit (for example the BBB or a similar board). This controller would administrate on the same unit both the high-level and the low-level control using the feedback provided by the joint-boards. Being the high-level control also related to the data provided by other devices using the middleware, the ROS interface needs to be set-up on the

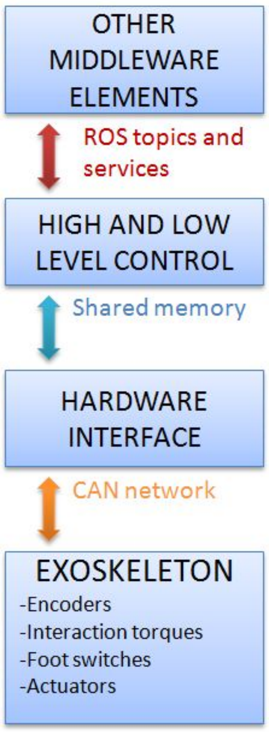


Figure 50: Unified control architecture

central unit too. A schematic representing this solution is in figure 50.

The positive aspects of this solution are:

- Using only one device allows for power and system-complexity reduction;
- Running on the same controller, the high-level control and the low-level control can communicate quickly (using shared memory) without the need of a communication channel.

The negative sides are:

- To provide the low-level control functionalities with correct timings, the controller needs to run in a hard real time mode. As is pointed out in section 7.4, handling a hard real-time operative system is not a simple task and a lot of efforts need to be done to balance the system's performance and get adequate timing latencies.
- Putting high-level and low-level control on the same device reduces system's reliability: it's not possible to define *safety-exit* functions that would handle the exoskeleton's behaviour in the case of a system-crash on the central controller.

7.2.2 *Send only set-points from the central controller*

In this scenario the low level controller and the high level controller run on different platforms:

- The low-level control runs in hard real-time and administrates feedback signals coming from the exoskeleton's sensors with reduced delays. This level doesn't have a direct connection with the middleware and shouldn't require a great amount of computational power. It can be centralized on a single controller as was done for the H2 exoskeleton or decentralized on multiple joint controllers.

- The high-level control runs in soft real-time and is interfaced with both the middleware and the low-level control. It can provide position, torque, stiffness set-points for the low level control basing its choices on both the data coming from the exoskeleton and the output of other devices in the system. To achieve this objective it will run multiple algorithms that at the moment of writing aren't still fully developed (neuromusculoskeletal model, tacit learning algorithms etc) but that in general require a relatively long execution, an higher computational power and also might be executed on other remote machines.

A schematic representing this solution is in figure 51.

In section 7.4 it is pointed out that reaching very low response times on an embedded-linux system like the BBB is a problematic task. On the other hand, the dynamic of the exoskeletons are very slow and consequently set-point sending is done at a relatively low frequency; for example on the H2 exoskeleton running at the fastest velocity, position set-points are currently sent with a time period of about 20ms. As was described in section 7.4.2 and tested with the ROS nodes of section 4.4.3, the BBB can follow this task with a maximum deviation of about 400 µs that doesn't affect the system's behaviour. If needed, better performances can probably be achieved using the PREEMPT-RT patch, Xenomai or changing the platform.

The positive aspects of this solution are:

- Decoupling high-level and low-level control allows for the definition of low-level functions that can handle the system behaviour also in the case of a crash of the high-level control.

- Balancing system's performances is easier since the time of execution of the high-level algorithms doesn't affect the response time of the low level controllers.

- The programmers and control developers working in the group are already familiar with the development of controllers running on embedded platforms like the H2-arm due to the expe-

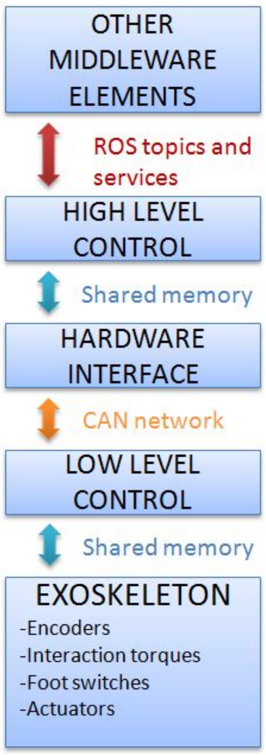


Figure 51: Decoupled control architecture

rience acquired with the H2 exoskeleton; maintaining a similar structure for the low-level control allows for faster development.

The negative aspects of this solution are:

- Using more platforms leads to a bigger system's cost, complexity and power consumption (the last aspect is very limited if compared with the power absorbed for the motors actuation).
- Being the high-level control and low-level control logically and physically separated, a communication channel between the two needs to be created (trough the CAN network) leading to (limited) delays in their communication.

7.2.3 *Select and modulate pre-memorized patterns*

As in the previous scenario, the high-level control and the low-level control are physically separated. The difference resides in the information exchanged between the two: the high-level control doesn't provide position or torque set points, but these are previously memorized on multiple patterns resident in the low-level controllers. The high-level control only sends information regarding the required gait velocity, the modulation of the torque provided by each joint (from very hi to very low assistance), the selection of different patterns with relation of the estimated user intentions and eventually the regulation of the actuators' stiffness. The schematic representation of the solution is still the one in figure 51.

The positive aspects of this solution are:

- All the aspects of the previous solution.
- Using only a set of pre-memorized patterns makes the control of the device easier; also testing if its behaviour is safe for the user is simpler.
- The high-level controller can work and communicate with hi delays since the modulation or the change of the loaded patterns is not a time-critical operation.
- A system-failure in the high-level controller always leaves the user in a safe condition and can be easily handled.

The negative aspects of this solution are:

- All the aspects of the previous solution.
- Using only pre-memorized patterns reduces the system's flexibility since unforeseen user's actions can't be handled. This leads to a lower quality human-robot interaction.

7.2.4 *Conclusions*

Among the exposed solutions, the second one is probably the more promising since it combines the advantages deriving from the decoupling of the high-level and low-level control with a large system's flexibility. On the other hand, being the third option simpler to apply, it might be a good solution for the initial tests on the system. For example it could be used to verify the capability of the tacit learning module to regulate the gait velocity or the possibility of modulating the torque provided by each joint on the base of the torques provided by the user, estimated through the neuromusculoskeletal model and/or interaction sensors.

7.3 ROS-CONTROL INTERFACE

ROS-Control is a set of ROS packages that include controller interfaces, a controller manager, transmissions, hardware interfaces and a control toolbox to provide a standard control infrastructure for robots using ROS. These packages are a generalization of the *pr2_mechanism*'s packages, which form a set of packages for controlling a *pr2* robot[1].

1 http://wiki.ros.org/Robots/PR2

7.3.1 *ROS-Control in Biomot*

The ROS-Control workflow is an optional structure for the organization of the high-level control used with the newly developed exoskeleton. Using such solution would lead to a standardized interface, similar to the ones used in other robots, that might allow for the reuse of already developed and tested modules.

As was previously pointed out, the high-level control strategy of the exoskeleton isn't, at the moment of writing, already developed. In general it will make use of the information coming from external modules and from the exoskeleton to elaborate the correct input for the lower control, hence the standardized controllers provided with the ROS-Control module won't probably be sufficient and customized ones will be needed. Also, being the exoskeleton usable in different scenarios and with different auxiliary module-inputs, the possibility of easily switching between different controllers needs to be implemented.

At the moment of writing, the available documentation about ROS-control is fragmented; getting a clear idea about its working principles and about the correct techniques to solve the presented problems, is a task that usually requires a lot of effort from the programmers. In the following sections a generic review of its working mechanisms and an example of application for the Biomot exoskeleton will be provided. This documentation will eventually be a start-point for the creation of a full, ROS-Control based, control-architecture.

7.3.2 *ROS-Control workflow*

The generic organization of a ROS-Control infrastructure can be seen in figure 52 from [19].

At the lowest level, data are read from generic robot sensors (encoders, load cells,...) and their information is transmitted to an "*Hardware interface*". This interface stores the information regarding the status of the robot and allow their access from multiple controllers. Different controllers can be loaded by a *Control manager*, depending on the provided hardware-interface; their output goes back trough the hardware interface to the low level controllers of the robot. The hardware interface can optionally provide joint safety limitations and data conversion (transmission elements).

If standardized interfaces are provided by the hardware interface, a set of already exiting controllers can be used. The kind of controllers that can be loaded depend on the interfaces registered into the robot class; for example, if the input required by the low-level controller is a torque set-point, a standard effort interface can be initialized that would allow the usage of a *effort_controllers/ JointPositionController*.

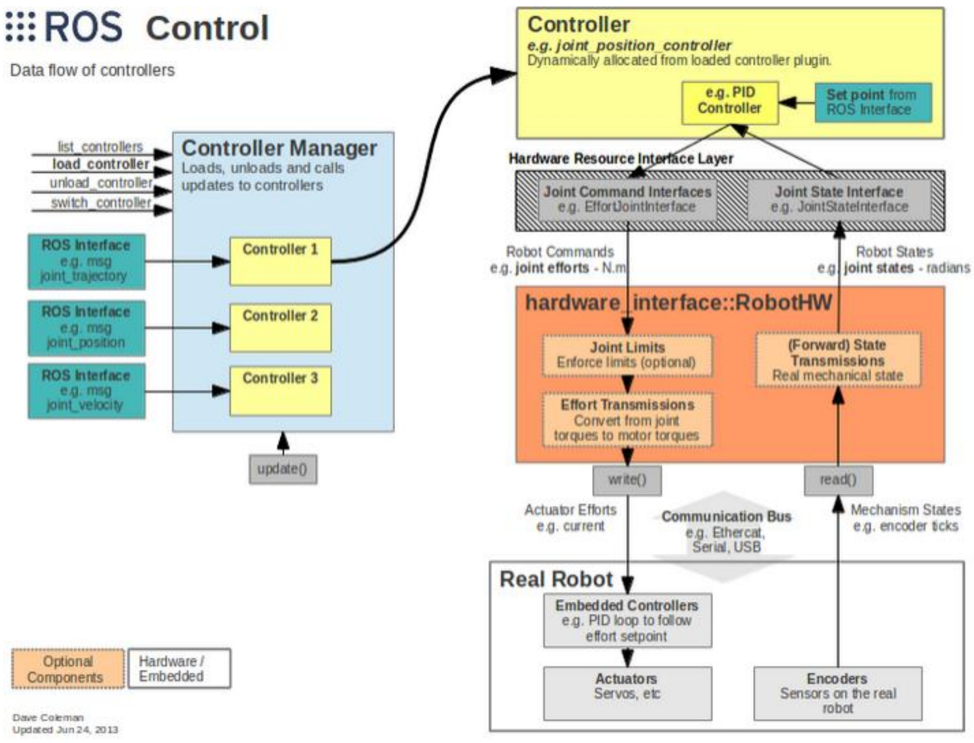

ROS Control
Data flow of controllers
list_controllers
load_controller
unload_controller
switch_controller
Controller Manager
Loads, unloads and calls updates to controllers
ROS Interface e.g. msg joint_trajectory
ROS Interface e.g. msg joint_position
ROS Interface e.g. msg joint_velocity
Controller 1
Controller 2
Controller 3
update()
Controller
e.g. joint_position_controller
Dynamically allocated from loaded controller plugin.
e.g. PID Controller
Set point from ROS Interface
Hardware Resource Interface Layer
Joint Command Interfaces e.g. EffortJointInterface
Joint State Interface e.g. JointStateInterface
Robot Commands e.g. joint efforts - N.m
Robot States e.g. joint states - radians
hardware_interface::RobotHW
Joint Limits Enforce limits (optional)
Effort Transmissions Convert from joint torques to motor torques
(Forward) State Transmissions Real mechanical state
write()
read()
Actuator Efforts e.g. current
Communication Bus e.g. Ethercat, Serial, USB
Mechanism States e.g. encoder ticks
Real Robot
Embedded Controllers e.g. PID loop to follow effort setpoint
Actuators Servos, etc
Encoders Sensors on the real robot
Optional Components
Hardware / Embedded
Dave Coleman
Updated Jun 24, 2013

Figure 52: ROS-Control general scheme

The ROS-control work-frame can be used in conjunction with the *Gazebo* simulation environment. A generic robot can be described trough a *URDF*(*Uniersal Robot Description Format*) format, that specifies the geometrical relationships between the joints/links and optionally their physical characteristics. Thanks to a dedicated plug-in, these models can be used in conjunction with ROS-Control and Gazebo to run simulations and visualize the predicted robot-behaviour. A complete example of integration, with detailed set-up instructions, can be found in [22].

7.3.2.1 *The case of the PR2 robot*

As was previously pointed out, the ros-control package is a generalization of the code written for the PR2 robot; thanks to the modularity of the code's structure and to its generic form, a part of the solutions provided for this robot can be easily extended to a wide set of robots including exoskeletons. Following is a description of how, the set of packages forming the pr2 robot's code, can interact to provide a customized controller. A similar structure will later be used to organize the control-code for the exoskeleton.

The first package forming the PR2'control code is the pr2_mechanism_model package. This package includes classes that allow to access the joint's information, to write joint's commands and also contain a physical description of the robot in an URDF format. When a controller gets

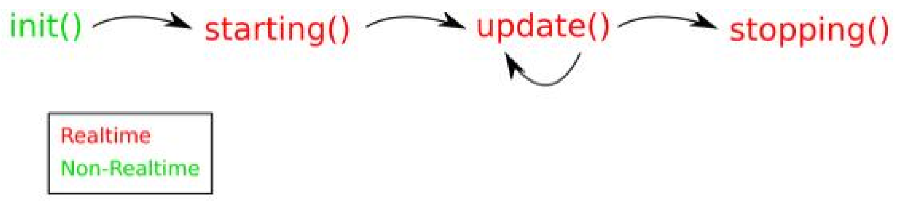


Figure 53: Flow chart for controller interafce's methods

initialized, the controller manager passes the controller a pointer to the *RobotState* class. The RobotState describes both the kinematic/dynamic model of the robot and its current state, providing an hardware interface. The state of the robot is defined by the position/velocity/-effort of the joints in the robot and can be updated trough a *read* function. Additionally, the *RobotState* provides access to the 'controller time', the time at which a controller cycle is started, and to its output, that is sent to the robot trough a *write* function. In contrast to the system time, the controller time is not affected by the time other controllers consume in their update loop. Moreover, the controller time is the best measure of when the communication with the hardware actually occurs.

The class member *urdf::Joint::safety* also contains various parameters used to configure joint safety controller position and velocity limits (see [27]).

To create a controller compatible with the *pr2's controller manager* the **pr2_controller_interface** package is used. The package basically contains the C++ controller base class that all controllers need to inherit from. To implement a real time controller, it needs to be inherited from the *pr2_controller_interface::Controller* base class. The base class contains:

- four methods that need to be implemented: init, starting, update and stopping;
- one method that can be called: getController.

Figure 53, from [24], shows the order of call for these methods .

The *init* method, executed in non real-time, is called when a controller is loaded, to initialize it. Note that initializing a controller is independent of starting it. The init method takes two arguments:

- a *pr2_mechanism_model::RobotState* that, as previously said, describes the robot model and state;
- a ros::NodeHandle, which is the "namespace" of the controller. In the namespace of this node handle, the controller can read configuration from the parameter server, advertise topics, etc.

An example of initialization is:

```
virtual bool init(pr2_mechanism_model::RobotState *robot, ros::
    NodeHandle &n);
```

The *init* method returns if the initialization was successful or not. If the initialization fails, the controller will get unloaded by the *pr2 controller manager*.

The *starting* method, executed in real-time, is called once every time a controller is started, by the controller manager. Starting is executed in the same cycle as the first update call, right before this update call.

The *update* method, executed in real-time, is called periodically by the *pr2_controller_manager* at a standard frequency of 1000 Hz (that in a generic case can be changed). This means that the execution time of all controllers combined cannot take more than 1 milli-second. In the update loop, the control algorithms are run.

The *stopping* method, executed in real-time, is called once every time a controller is stopped. Stopping is executed in the same cycle as the last update call, right after this update call. The stopping method does not return anything, it is not allowed to fail.

The *getController* method allows a controller to get a pointer to another controller. This can be used to create a "chain" of controllers, where each controller sends its output to the next controller, in real-time.

More detailed information about the controller interface for the pr2 robot can be found in [24].

In [28], an explanation of how to build a real-time joint controller using the pr2_controller_interface, the pr2_mechanism_model and the pr2_controller_manager is given. To write the controller, a package depending on the pr2_controller_interface, the pr2_mechanism_model and *pluginlib* needs to be created. The pluginlib package allows to add the customized controller as a plugin into the controller manager.

A controller class is then created in the package as a son of the pr2_controller_interface::Controller class and the *init, starting, updating* and *stopping* methods are properly overloaded. The code can then be compiled as a library using the rosbuild_add_library command in the project's CmakeLists.txt file.

In order to make the process run inside the real-time process, it needs to be earmarked as destined for it, to achieve this it needs to be registered as a plugin. Then the pr2_controller_manager can use the pluginlib to administer the linking, loading, and starting of the controllers. In [28] is a full description of how to export the controller as a loadable class, specify the required dependencies and create a plugin description file accordingly to the pluginlib documentation. In section 7.3.4 a more generalized example will be given.

In [26] is then described how to run the created controller into the Gazebo simulation environment. In general, before being started, a controller needs to be initialized with its parameters. This can be done with commands in the form:

```
rosparam set my_controller_name/parameter_name parameter_value;
```

The controller can then be loaded running:

```
rosrun pr2_controller_manager pr2_controller_manager load my_
    controller_name
```

and started with:

```
rosrun pr2_controller_manager pr2_controller_manager start my_
    controller_name
```

To make configuration and start-up faster, all the parameters can be included in a *YAML* file that is loaded on the *param server* via a roslaunch file (an example valid for a generic robot will be provided in the next section).

Communication with the created controller can then be achieved creating dedicated ROS services that dynamically update the controller's parameters or set-points. The customized controllers can also easily include commonly used elements from the *control_toolbox* package, for example a PID controller or a white-noise generator.

7.3.3 *Developing a standard hardware interface and running standard controllers for Biomot*

To make controllers communicate with the actuators and receive information coming from the exoskeleton's sensor through the low-level controller, an hardware interface needs to be provided for the middleware. To achieve this objective with ROSControl, a robot class, similar to the *RobotState* class described for the pr2 robot, needs to be created.

Since not all the robot-features are already defined at the moment of writing, a standard interface, similar to the one described in [29], is initially written for the Biomot exoskeleton. This class can later be easily expanded and personalized including robot-specific features in the interfaces (see next section).

A UML file showing the relations between the developed classes is in appendix A.3.

As for the pr2 robot, the robot class, named *BiomotHardwareInterface*, is derived from the *hardware_interface::RobotHW* class; this will later allow to use it in conjunction with the *controller_manager*, which is a generalized implementation of the *pr2_controller_manager*. In its private members, the class stores the robot's state information: position, velocity, effort and the output elaborated by its controllers. In the case of the Biomot exoskeleton, velocity information won't probably be needed, but their inclusion allows for a more general structure.

These variables are used to initialize a set of standard interfaces, all derived from the *hardware_interface* class:

- **JointStateInterface**: to initialize this interface, the three robot-state variables need to be passed as arguments; the controllers loaded by the controller manager can in this way gain access to the robot's information.
- **PositionJointInterface**: to initialize this interface, an output variable needs to be passed as argument; the compatible controllers loaded by the control manager can later use this variable to write their output as a position set-point.
- **EffortJointInterface**: to initialize this interface, an output variable needs to be passed as argument; the compatible controllers loaded by the control manager can later use this variable to write their output as an effort set-point.

The initialization of the updating frequency, of the joints and of the controllers to be loaded with the relative parameters, is made trough a *YAML* configuration file. Such file is structured for example as follows:

```
hardware_interface:
   loop_hz: 1 # hz
   joints:
      - rHip
      - rKnee
      - rAnkle
      - lHip
      - lKnee
      - lAnkle

# Publish all joint states -----------------------------------
joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 50

# Position Controllers ---------------------------------------
rHip_position_controller:
  type: effort_controllers/JointPositionController
  joint: rHip
  pid: {p: 100.0, i: 0.01, d: 10.0} #random example values

  ...
```

The parameters contained in this file are loaded on the *param-server* when the YAML is launched. The hardware interface will look for them when it's initialized and will build as many interfaces as the joints found on the server, thanks to the usage of *std::vectors* that can be resized. This makes the interface flexible and potentially allows it to be applied with other robots that use a different number of joints (for example the H2R robot) simply changing the *YAML* file.

Once the interfaces are built, the *controller_manager* is created inside the robot-object itself. Another option would be to start the *controller_manager* outside the object, as is proposed in [29], but the way of initialization used in this example seems to be the most frequent.

The *loop_hz* variable is used to set a *ros::Duration* variable that is then used to initialize a timer that handles the periodic execution of an update function. The update function calls a *read* function that updates the robot-state variables reading their values from the CAN network. The controller's update function is then called trough the controller_manager, passing the current time and the time elapsed since the last execution as parameters. The outputs of the controllers are finally read by the *write* function that sends their values to the robot trough the CAN network. All the functions called by the *update* function (read, update and write) can potentially be executed in real-time, but their implementation would in that case be platform-specific. The created example creates a not real-time loop that can be easily converted to a real time loop once a specific real time capable kernel is provided.

The created class is compiled as a library in a dedicated catkin package named *biomot_control*. To create an instance of it and consequently create a control loop, an executable is also created implementing a ROS node named *controlLoop*. This node also creates a parallel thread to handle service callbacks from the controller manager; if this thread was not created, every service call (ex to load a controller) would block the execution of the control loop.

A generalized example, that was used as a starting point to build this class and also allows for robot-visualization of a *rrbot* robot, can be found in [8].

To run certain kinds of controllers, for example effort based position controllers or the *JointState* controller, a robot model must be loaded on the parameter server with the name *robot_description*. A simple, schematic, exoskeleton description can be built using the *URDF* (Universal Robot Description Format) format. A simple text file is used to describe the relationships between the six joints forming the exoskeleton. It includes the robot name, the list of the links forming the robot and the list of the nodes. For each node is specified the parent link, the child link, the origin position and the axis of rotation. Optionally, other parameters like links' inertia can be specified, allowing for the implementation of simulations in the Gazebo environment. Creating simulations is out of the scope of this work, consequently only a schematic description of the exoskeleton is given. The created URDF file for Biomot is described in figure 54. In this scheme, rectangles represent the links of the robots, ellipsis the joints, the (x, y, z) coordinates represent the position of each system with respect to the

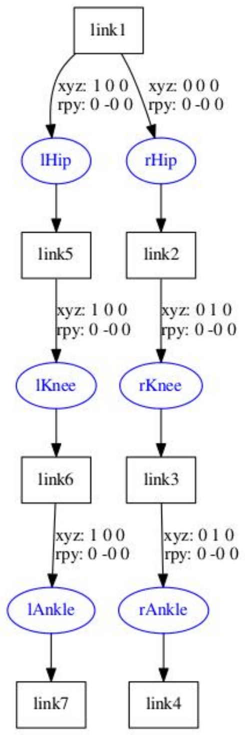


Figure 54: Schematic of the Biomot's URDF file

previous and the (r, p ,y) values show the degrees of rotation around respectively the axes x, y and z of each reference system.

To load the exoskeleton model, the hardware interface, the *YAML* file and the desired controllers with a single command, a launch file was created; its implementation is as follows:

```
<launch>

  <!-- Load biomot URDF -->
  <param name="robot_description" textfile="$(find biomot_control
      )/urdf/biomot.urdf"/>

   <group ns="biomot">

    <!-- Load hardware interface -->
    <node name="biomot_HInterface" pkg="biomot_control" type="
        controlLoop"
          output="screen" />

    <!-- Load controller settings -->
    <rosparam file="$(find biomot_control)/config/biomot_config_
        eff.yaml" command="load"/>

    <!-- Load controller manager -->
    <node name="ros_control_controller_manager" pkg="controller_
        manager" type="controller_manager" respawn="false"
```

```
          output="screen" args="spawn rHip_position_controller
              rKnee_position_controller rAnkle_position_
              controller lHip_position_controller lKnee_position_
              controller lAnkle_position_controller" />

  </group>

</launch>
```

In its beginning, the *URDF* file containing the description of exoskeleton is loaded from the related package folder. The control loop node is then started with the name *biomot_HInterface* and consequently the hardware interface is built. The configuration *YAML* file is then loaded on the parameter server and the controller manager is started passing the controllers to be spawned as arguments. The spawn option allows the controllers to be loaded and started with a single command.

Different launch files can be implemented loading different yaml files and consequently starting a different set of controllers. It is important to note that the *controller_manager* won't allow more than a controller to access a resource at the same time; for example every single joint can be controlled by only one controller at a time.

Once the *controller_manager* and the controllers are started, a set of services are provided to the middleware users. A list can be visualized with the "*rosservice list*" command. For example, if a set of effort based PID position controllers are loaded, the command returns:

```
/biomot/biomot_HInterface/get_loggers
/biomot/biomot_HInterface/set_logger_level
/biomot/controller_manager/list_controller_types
/biomot/controller_manager/list_controllers
/biomot/controller_manager/load_controller
/biomot/controller_manager/reload_controller_libraries
/biomot/controller_manager/switch_controller
/biomot/controller_manager/unload_controller
/biomot/lAnkle_position_controller/pid/set_parameters
/biomot/lHip_position_controller/pid/set_parameters
/biomot/lKnee_position_controller/pid/set_parameters
/biomot/rAnkle_position_controller/pid/set_parameters
/biomot/rHip_position_controller/pid/set_parameters
/biomot/rKnee_position_controller/pid/set_parameters
/rosout/get_loggers
/rosout/set_logger_level
```

These services can be accessed by other nodes in the middleware or by the user through command line (*rosservice call SERVICE PARAMETERS*). For example the *list_controllers* service returns information about the running nodes:

```
controller:
  -
```

```
    name: joint_state_controller
    state: running
    type: joint_state_controller/JointStateController
    hardware_interface: hardware_interface::JointStateInterface
    resources: []
  -
    name: rHip_position_controller
    state: running
    type: effort_controllers/JointPositionController
    hardware_interface: hardware_interface::EffortJointInterface
    resources: ['rHip']
  -
    name: rKnee_position_controller
    state: running
    type: effort_controllers/JointPositionController
    hardware_interface: hardware_interface::EffortJointInterface
    resources: ['rKnee']
  -
    name: rAnkle_position_controller
    state: running
    type: effort_controllers/JointPositionController
    hardware_interface: hardware_interface::EffortJointInterface
    resources: ['rAnkle']
  -
    name: lHip_position_controller
    state: running
    type: effort_controllers/JointPositionController
    hardware_interface: hardware_interface::EffortJointInterface
    resources: ['lHip']
  -
    name: lKnee_position_controller
    state: running
    type: effort_controllers/JointPositionController
    hardware_interface: hardware_interface::EffortJointInterface
    resources: ['lKnee']
  -
    name: lAnkle_position_controller
    state: running
    type: effort_controllers/JointPositionController
    hardware_interface: hardware_interface::EffortJointInterface
    resources: ['lAnkle']
```

The *list_controller_types* service returns a list of known controllers that can be loaded using the registered hardware interfaces:

```
types: ['controller_manager_tests/EffortTestController', '
    controller_manager_tests/ExampleController', 'diff_drive_
    controller/DiffDriveController', 'effort_controllers/
    GripperActionController', 'effort_controllers/
    JointEffortController', 'effort_controllers/
    JointGroupEffortController', 'effort_controllers/
    JointPositionController', 'effort_controllers/
    JointTrajectoryController', 'effort_controllers/
    JointVelocityController', 'force_torque_sensor_controller/
    ForceTorqueSensorController', 'imu_sensor_controller/
    ImuSensorController', 'joint_state_controller/
    JointStateController', 'position_controllers/
    GripperActionController', 'position_controllers/
    JointGroupPositionController', 'position_controllers/
    JointPositionController', 'position_controllers/
    JointTrajectoryController', 'velocity_controllers/
    JointGroupVelocityController', 'velocity_controllers/
    JointPositionController', 'velocity_controllers/
    JointTrajectoryController', 'velocity_controllers/
    JointVelocityController']
```

The other services allow for unloading controllers, loading new ones, make dynamic changes in the controllers' parameters and other functions.

When the controllers are running, they can also create a set of ROS-topics that can be used for getting and sending information. A complete list can be visualized running "*rostopic list*" from command line. For example, the topics created by the *rHip_position_controller* of type "*effort_controllers/JointPositionController*" are:

```
/biomot/rHip_position_controller/command
/biomot/rHip_position_controller/pid/parameter_descriptions
/biomot/rHip_position_controller/pid/parameter_updates
/biomot/rHip_position_controller/state
```

A position set-point for this controller can for example be sent executing manually:

```
rostopic pub /biomot/rKnee_position_controller/command std_msgs/
    Float64 "data: SET_POINT"
```

or a dedicated node can publish it at fixed frequency.

The *joint_state_controller* is a read-only controller that publishes the state of the robot on the topic "*/biomot/ joint_states*". The state, described by position, velocity and effort arrays, is stored in "*sensor_msgs/ JointState*" messages that can be read by any node in the middleware.

7.3.4 *Writing customized interfaces and controllers*

The class described in the previous section registers some standard interfaces for the exoskeleton. Using these interfaces allow for the usage of already written controllers, making the development of the control architecture easier. These controllers by the way, might be insufficient for the exoskeleton functionalities and consequently new ones might be needed. Customized controllers can still make use of the standard interfaces and be consequently robot-agnostic. If particular robot-features need to be accessed by the way, also customized interfaces must be provided since standard interfaces only allow the controllers to access information regarding the joint's position, velocity and provided torques.

A basic example explaining how to build a customized interface can be found in [29].

The simplest way to build it, consists in declaring the robot-specific variables and functions inside the robot class and then call the function *registerInterface* passing the class itself as a parameter. Ex:

```
class BiomotHardwareInterface: public hardware_interface::RobotHW
    , public hardware_interface::EffortJointInterface
{
public:
  BiomotHardwareInterface()
  {
    // Gets parameters and joint names from the param-server
    ...
    // Registers standard interfaces (usable by standard
        controllers)
    ...
    //Registers the whole class itself as an interface (usable by
         custom controllers)
    registerInterface(this);
    ...
  }
  //Registers robot specific functions
  double get_estimated_torque(std::string jointName);
};
```

To make this possible, the custom hardware interface class needs to inherit from the standard HardwareInterface class; such derivation by the way, doesn't include functions that allow the class for resource handling as for example *getHandle*. To make them available, the class inherits instead from the *hardware_interface::EffortJointInterface* class.

The robot-specific functions can for example allow the controllers to access some specific private variable relative to the output of an exoskeleton's module. In the created example, a further vector, named *estimated_torque_* is added to the already existing ones. The controllers based on the Custom interfaces and standard interfaces can coexist; different controllers can work in parallel on different resources as schematized in figure 55.

A simple example of how a customized controller can be created is in [30]; the generic implementation is similar to the one described in section 7.3.2.1 for a PR2 robot. To write a controller, a new class, deriving from the *controller_interface* class needs to be created. To provide a general example, a customized controller class, named *CustomController*, was created. This class implements (at least) four functions: *init*, *update*, *starting*, *stopping*, similar to the ones described for the PR2 robot.

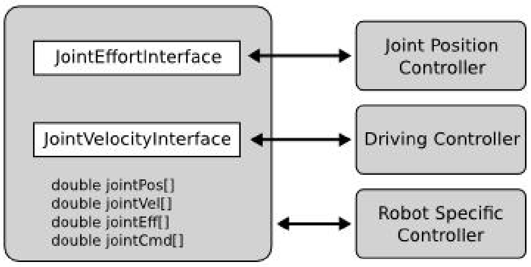


Figure 55: Schematic of the relations between controllers and interfaces

The *init* function requires two arguments: a pointer to the related hardware interface and a pointer to the node-handler variable. It uses the node-handler to load the joints names and their required parameters from the param-server. The function then claims the joints' handle, letting the controller manager know that the resources are allocated and can't be used by other controllers.

If the controller is based on the customized *BiomotHardwareInterface* interface, the function looks like:

```
bool init(biomotNS::BiomotHardwareInterface* hw, ros::NodeHandle
    &n)
  {
    // Get joint name from the parameter server
    if (!n.getParam("joint", my_joint)){
      ROS_ERROR("Could not find joint name");
      return false;
    }
    // Get the joint object to use in the loop and reserves the
        relate resource
    joint_ = hw->getHandle(my_joint);
    //Initializes optional services for custom controllers
    srv_ = n.advertiseService("setCommand",&CustomController::
        setCommand, this);
    //Initializes some example parameters
    gain_=2;
    setpoint_=0;
    //Creates a copy of the pointer to the hardware interface to
        access robot-specific functions
    hwInterface=hw;
    return true;
  }
```

The customized controller can later get access to standard resources (position, velocity and torque) through the Joint handler (*joint_* in the example) and to robot-specific resources through a pointer to the hardware interface (*hwInterface* in the example). The *update* function takes as arguments the current time and the time elapsed since it's last execution. In this function the robot's state variables are accessed trough the provided interface (e.g. through a *joint.getPosition()* function or trough hwInterface -> get_estimated_torque_ (my_joint) for our example of robot-specific function), the control algorithms are executed and their output is stored back in the hardware-interface class (ex with *joint.setCommand(command)*). In the provided example, a simple proportional position controller that writes an effort command is implemented.

In the *starting* and *stopping* functions, generic code can be executed when the controller respectively starts or stops.

The so created class (as the one describing the hardware interface) is compiled as a library trough the *CMakeLists.txt* file. The package where the controller is included needs to depend from the *pluginlib* class, the *controller_interface* and the *hardware_interface*. To make the controller visible to the controller_manager, it also needs to be declared into *PluginLib*. PluginLib is a library for loading and unloading plugins from within a ROS package. Plugins are dynamically loadable classes that are loaded from a runtime library. Pluginlib can open a library containing exported classes at any point without the application having any prior awareness of the library or the header file containing the class definition.

To export the library into PluginLib, a line like this needs to be added in the class definition:

```
PLUGINLIB_DECLARE_CLASS(controller_pkg,ControllerPlugin,
                         controller_ns::MyControllerClass,
                         controller_interface::Controller)
```

That in the case of our example becomes:

```
PLUGINLIB_DECLARE_CLASS(biomot_control, CustomController,
    biomotControllerNS::CustomController, controller_interface::
    ControllerBase);
```

Also a Plugin description file needs to be created; it needs to be declared in the export section of the *package.xml* file as follows:

```
<export>
    <controller_interface plugin="${prefix}/controller_plugins.
        xml" />
</export>
```

and then defined as:

```
<library path="lib/libbiomot_control">
  <class name="biomot_control/CustomController"
         type="biomotControllerNS::CustomController"
         base_class_type="controller_interface::ControllerBase"
             />
</library>
```

More than one controller can be declared into the same xml description file.

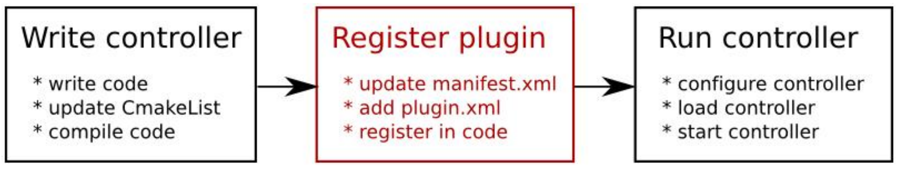


Figure 56: Required steps to write a customized controller

The so created controllers can then be started and configured through YAML configuration files as was explained in the previous section, otherwise they can be loaded/unloaded dynamically through the controller_manager services. Running "*rosservice call /biomot/controller_manager/ list_controller_types"* now also displays "*biomot_control/CustomController*" in the list of available controllers. A diagram resuming the required steps to set-up a new customized controller is in figure 56.

7.3.5 *Adding services to communicate with the custom controllers*

As the standard controllers, the created custom controllers can load a set of parameters from the *param-server* at start time. To let them communicate with other nodes or with the user by the way, a set of ROS-services needs to be added to the controller class. As described in [17], ROS-services allow to define a request / reply mechanism between multiple nodes. The information passed between the nodes is described by service files (.srv); in the first section of these files, is described the kind of data sent by the node that requests the service, while in the second is described the kind of answer provided by the server.

In this example only one simple service is provided to modify the set-point of the controller; the related srv file is:

```
#setCommand.srv:
float64 command
---
float64 command
```

In general, the set-point sent by the service server can be different from the received one, due to set-point limitations. To use the described service, the controller-package needs to depend from the *message_generation* class. If customized services are used, they need to be declared in the related section of the *CMakeLists.txt* file in order to generate the related header file when the package is compiled; the server class then includes the headers and declares the related service-handling functions during its initialization.

In the created example, this is done as:

```
//In the class description's header file
    ----------------------------------
```

```
#include "biomot_control/setCommand.h"
ros::ServiceServer srv_;
//In the init function
    ---------------------------------------------------
srv_ = n.advertiseService("setCommand",&CustomController::
    setCommand, this);
//The setCommand function to handle the service call
    -----------------------
bool CustomController::setCommand(biomot_control::setCommand::
    Request& req, biomot_control::setCommand::Response& resp){
  setpoint_=req.command; //TODO: eventually check setpoint limits
       here
  resp.command=setpoint_;
  return true;
}
```

Other services can be added to dynamically change the controller's parameters or to provide other set-points.

The provided example now describes the biggest part of the instruments required when building an high-level control architecture and can be easily expanded to provide multiple functionalities.

For more details about the code forming this example see appendix A.3.

7.4 REAL-TIME IMPLEMENTATION ON THE BBB

The *Beagle Bone Black* board was introduced in the exoskeleton system as an interfacing platform between the low-level hardware and the ROS middleware. The whole set of functionality that will be required from this board is not yet fully defined in the project, but in general it might execute algorithms that will affect the system's high-level control in order to improve particular aspects of the human-robot interaction (for example the ROS-Control framework described in section 7.3). A part of the required tasks might include the execution of operations with strict time-constraints (*real-time* functionalities). In general, a *real-time* software executes functions that are subject to "real-time constraints", for example operational deadlines from event to system response, ensuring correct response-times.

Natively, the board runs a Linux OS distribution that, being a user-oriented operative system, is not designed to execute *real-time* tasks. Still, the analysis of the problem pointed out three different ways to run real-time tasks on the board: the usage of the *Programmable Real-time Units* (PRU), described in section 7.4.1, the application of the *Preempt-RT patch*, described in section 7.4.2, or the *Xenomai* software, described in section 7.4.3.

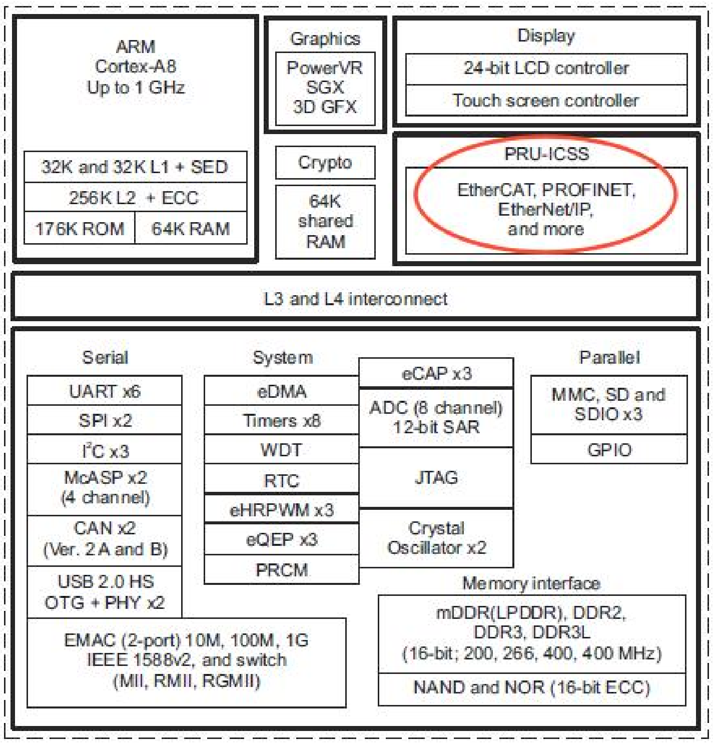


Figure 57: Scheme of the BeagleBone Black's TI chip

7.4.1 *Programmable Real-time Units*

The BeagleBone Black's TI chip (XAM3359AZCZ revision 2) contains the main processor (ARM) along with a number of other modules (see figure 57).

The TI chip provides, next to the ARM Cortex-A8 processor, two additional CPUs (known as PRU-ICSS or PRUSSv2) on the same silicon. Separate software can be run on this processor, offloading hardware interfacing and processing of low-level protocols. These CPUs run at a frequency of 200Mhz and are free from the execution of any software related to the operative system; they are consequently ideal for the execution of fast, low-level tasks that involve real-time constraints. Their software can be downloaded from the main processor and also a shared memory exists between the two, allowing for software communication (see [11] for more information).

There are, however, two main problems related with the utilization of this solution:

- **Programming language**: code for the PRUs is written in assembler currently, some experimental C compiler has been developed (see [50]), but their utilization is in general not straightforward for a programmer, requiring a certain learning curve.

- **Communication with the CAN network**: as previously described, to communicate with the low-level hardware of the exoskeleton, the BBB makes use of a CAN network. Providing CAN communication in real-time would be consequently very useful, but no way for using the PRUs with such interface could be found.

7.4.2 *Preempt-RT patch*

The *Preempt-RT* is a patch that modifies the whole Linux kernel in order to make it fully preentable and gain hard real-time capabilities. Without preemption support, when higher priority tasks wake up, they are delayed until current task exits from a syscall or yield explicitly. Also, user created processes always have lower priority than processes created in kernel-space. These things together lead to high time-latencies and poor time-resolution for processes running in user-space. In a preentable system these problems are avoided, leading to an improvement in response latency of high priority tasks.

The *Preempt-RT* patch is developed by a small group of kernel-programmers and can be freely downloaded and applied to many versions of the Linux kernel. More information about its working principle can be found in [10].

To apply it on the BBB, the kernel needs to be cross-compiled on another machine and downloaded on the micro-sd card or directly on the EMMC memory trough ssh. Its application to the BBB should be possible, and a set-up procedure can be found in [46]. With old kernel versions, like the one that was available when this work began (3.8.13), the modified kernel couldn't boot the board. To make it possible, a newer kernel was set-up on the board; in particular, a pre-built kernel image for the BeagleBone black could be set-up using *apt-get* (*linux-image-3.14.35-ti-rt-r55*); this kernel, by the way, only supports the *PREEMPT_RTB* (basic) option. Enabling full preemption still leads to the system's instability. Also, the tested kernel doesn't support the *device tree overlays*, that, as explained in section 2.3.5, are used to enable the CAN network; consequently no communication test could be done. Newer/other kernel editions might solve this problems and also allow for running performance tests like *cyclictest*.

A comparison between the performances that can be achieved with and without the application of a preemption patch on the BBB could be found in [45]. This comparison uses three types of kernel: standard 3.14.0, 3.14.0 with a preent patch and a 3.14.0-rt1 kernel (with the full preempt-RT patch).

In figure 58 the performances are measured as the *Interrupt response time*, which is the time period between the expiry of a hardware timer and the scheduling of a userspace task; measurement was done through the *cyclictest* test program. The graph shows as the applica-

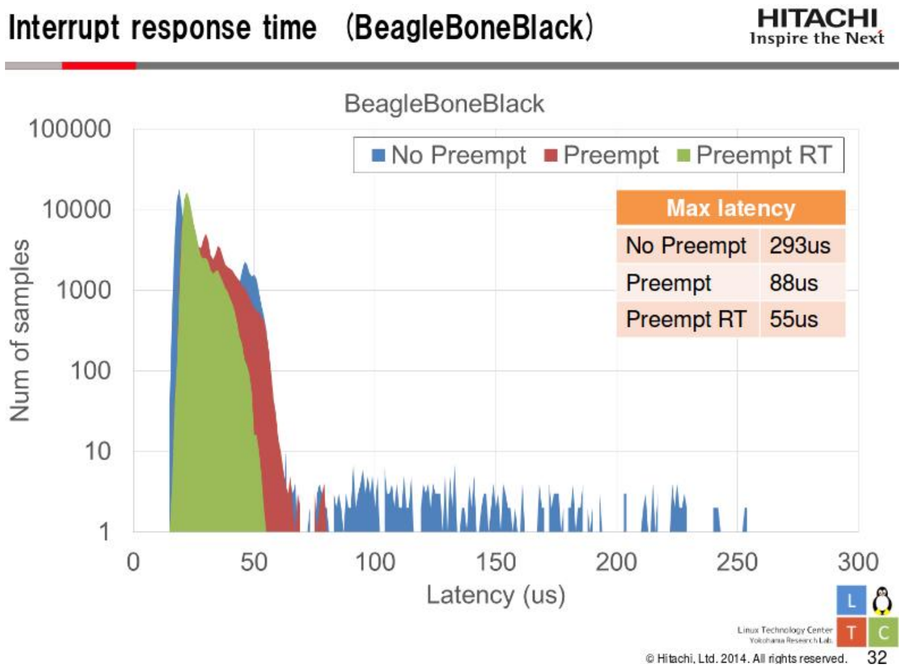


Figure 58: Interrupt response times for the BBB

tion of the patches leads to a consistent reduction of the response time.

In figure 59 the performances are measured as the *Task switching time*, which is particularly important when the running real-time application is formed by a large number of threads/processes. Also in this case the graph shows as the application of the patches leads to a consistent reduction of the switching time.

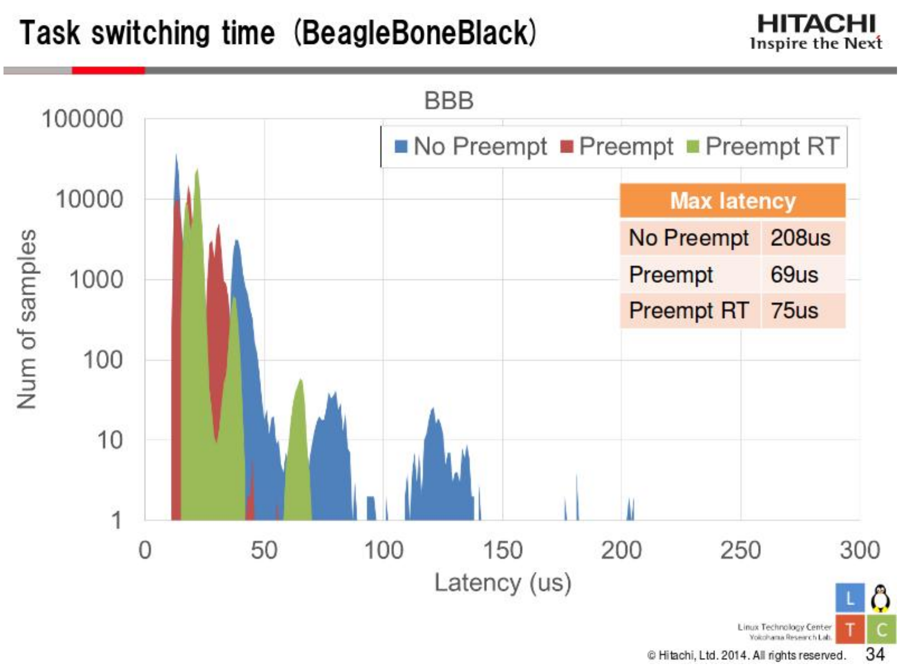
Some real-time frameworks that can be based on this patch and run on the BBB to execute control functions already exists, like for example *OpenRTDynamics* [6].

More information about the ways this patch can be used to run programs in real-time can be found in [51]; basic real-time programs could be compiled and executed on the BBB.

In general, the use of the real-time functionalities of the modified kernel should be done with caution since unbalanced scheduling times can lead to the instability of the whole system. Also, due to economic problems, the development of the PREEMPT-rt patches by the community is paused at the moment of writing.

7.4.3 *Xenomai*

Another possible solution is to modify the kernel using the *Xenomai* framework. Xenomai's software builds a parallel kernel running in parallel with the standard Linux kernel. Once the kernel is modified with the relative Xenomai's patch, applications can use Xenomai's



Max latency	
No Preempt	208us
Preempt	69us
Preempt RT	75us

Figure 59: Task switching times for the BBB

libraries to create particular pThreads that can run in kernel-space and access to real-time functionalities. More detailed information on it's working principles can be found in [2].

This kind of solution provides acceptable time latencies for many real-time applications; a detailed comparison with the PREEMPT-RT patch, made on an Beagle Board (not a BBB) is in [32].

The set-up of the patched kernel on the BBB can be achieved using pre-patched kernel editions or patching a chosen kernel "manually". A set-up instructions set can be found in [37]. In particular the *3.8.13-xenomai-r70* kernel from *Robert C. Nelson* could be successfully set-up on the board and some real-time example program could be executed.

An experimental ROS package, named *rosrt*, provides classes for interfacing with ROS from within realtime systems, such as realtime-safe Publisher and Subscriber classes and is intended to be used with Xenomai (see [25]).

Also some real-time functions for sending data via-CAN network can be found between the Xenomai examples. These functions make use of a real-time version of the socket-CAN API previously described in section 4.2.1. Unfortunately, in order to use such functions, a real-time CAN driver, named *rtcan*, needs to be loaded on the system. Using the driver provided with the Xenomai distribution didn't work with the BBB; the problem might be solved porting the *c_can* linux kernel drivers in *rtcan*.

Another problem related to the use of the Xenomai framework comes from the fact that, to use its real-time functionalities, programs must be modified in order to use its API. This leads to a quite long learning curve for the programmers.

Other solutions that provide real-time functionalities to Linux are available, like for example *RTAI* (RealTime Application Interface for Linux) or similar. Also full real-time operating systems could be set-up on the board, for example *QNX* is compatible with the BBB (see [52]). These ones are not described since no good documentation about their usage with the BBB could be found, but might still be a valid resource.

CONCLUSIONS

During this work, a multiple set of objectives were reached and described.

The chosen interfacing platform for the ROS middleware (the *Beagle Bone Black* board) was successfully set-up. Thanks to the created CAN communication library, and to the set-up of the necessary software and scripts, the board was able to communicate with the exoskeleton and read its internal data. It was then possible to set-up the ROS software and forward the received data to a middleware architecture. Some experiments were executed that showed the possibility to log the acquired data with a correct time stamp and to synchronize the exoskeleton with other external modules, in particular with an EMG acquisition system and the related neuromusculoskeletal model.

The possibility of using the Beagle Bone Black platform to send set-points to the H2 exoskeleton's low level controller was also investigated, giving positive results.

A set of graphical user interfaces were built to provide data visualization and support the activity of other programmers and control-developers in the group.

Finally, a generic architecture for supporting the construction of multiple, high level, joint controllers, was created using the ROS-Control framework. Using this architecture, multiple controllers can be created and dynamically loaded and setted. Different controllers can access different resources from the middleware system, depending on the employed modules and on the case of application. A set of already existing control-tools can also be employed to easy the work of the control-developers and make the code more standardized.

Thanks to the modularity of the employed solutions, all the developed code can be easily changed to support new modules and functionalities. The modularity and the use of standard interfaces also brings to an easy portability of the programs to other robots, allowing for cooperation between programmers working on distinct projects.

From a performance point of view, it is not possible to assert that the provided solutions are the optimal ones. Building optimized, application specific, software, might lead to better system performances, reducing delays in the communication between the devices forming the exoskeleton system. On the other hand, the modularity of the architecture would be lost, the code development would be delayed, modifications in the system structure and interfacing the programs built by programmers working in distinct groups would become more difficult tasks. In a complex, international project as is BioMot, using a widely known middleware architecture as ROS, can facilitate such operations and at the same time allow for an easier knowledge diffusion. All the developed code was based upon freeware software, allowing for reduction of the project's cost.

FURTHER EXPANSION

This work only formed the starting point of the implementation of a middleware architecture for the exoskeletons used and developed by the BioMot project. Being the BioMot exoskeleton and all the related modules still under development, it wasn't possible to create and integrate all the necessary middleware interfaces. For the same reason, it was not possible to insert in the ROS-Control framework the real structure of the hi-level controllers to be implemented.

Also the real-time framework to be used in conjunction with the hi-level controllers was not chosen, and it was not possible to evaluate the final performances of the communication system.

Once the exoskeleton and its modules will be completed, or partially completed, the programmers of the group will need to evaluate the possible control solutions and the related control time-constraints. The code provided by this work can then be easily expanded to achieve real-time capabilities and properly integrate the informations coming from the middleware system into the hardware interface.

If the interfacing platform won't be able to provide all the computational power required by the new algorithms, the code can be easily ported to more powerful linux-based platforms. The threaded structure of the programs eventually allows for distribution of the tasks on different processors if a multi-core platform is used.

Also newer interfaces, allowing for easy access to the control parameters by programmers or by users/therapists will be developed.

The modularity of the code and the features provided by the ROS middleware should make these tasks easier.

APPENDIX

A DEVELOPED PACKAGES

Following are some tree schematics representing the distribution of the files in the packages and some UML schematics representing the relation between the classes.

A.1 THE H2INTERFACE PACKAGE

The h2Interface package contains the developed code for communication with the h2 exoskeleton. As all the ROS packages, the code is compiled through a CmakeLists.txt file. The description of the provided solutions can be found in section 4.2.2 for the CAN communication library and in the sections 4.3 and 4.4 for the ROS packages. In section 4.5 is described how to compile the code and in section 4.6 how to launch it. A tree diagram describing the distribution of the files is in figure 60.

The developed code can be found on the REPOSITORY: `https://bitbucket.org/RehabEngGroup/h2`

A.2 THE H2R PACKAGE

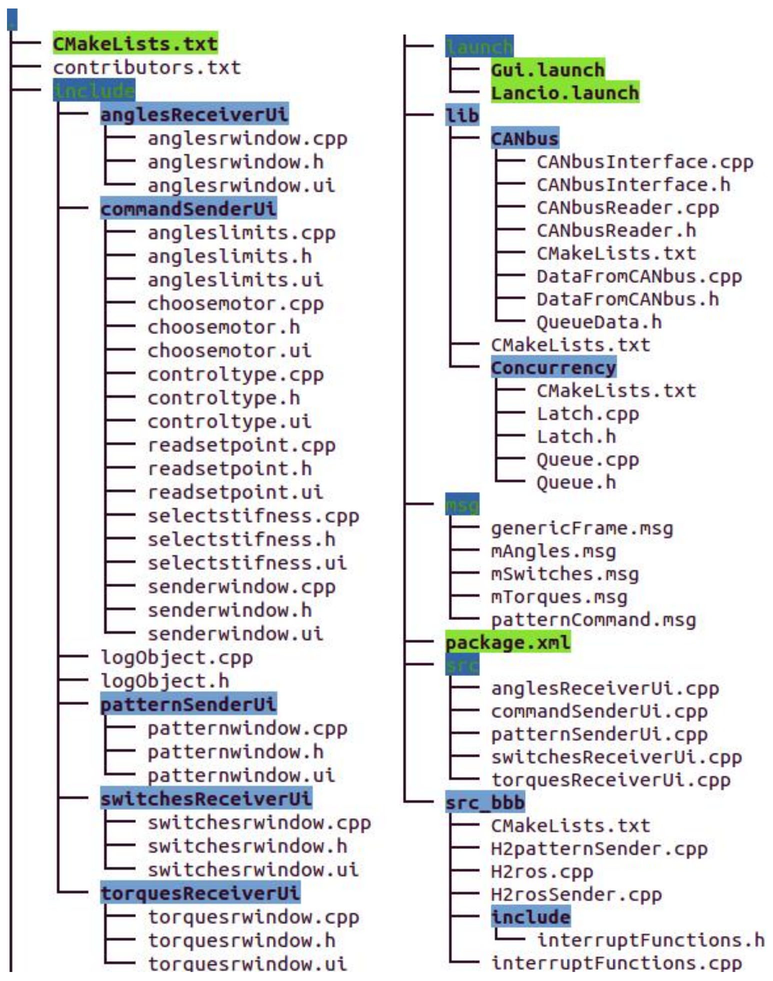
The h2r package contains the developed code for communication with the compliant actuators used in both the Biomot and the h2r project. The description of the provided solutions can be found in sections 6.5.1 and 6.5.2 . A tree diagram describing the distribution of the files is in figure 62.

The developed code can be found on the REPOSITORY: `https://bitbucket.org/beagleboneblack/h2r_ros`

A.3 THE BIOMOT_CONTROL PACKAGE

The biomot_control package contains the developed code for interfacing the newly developed Biomot exoskeleton with the ROS-Control framework. The description of the provided solutions can be found in section 7.3. A tree diagram describing the distribution of the files is in figure 63.

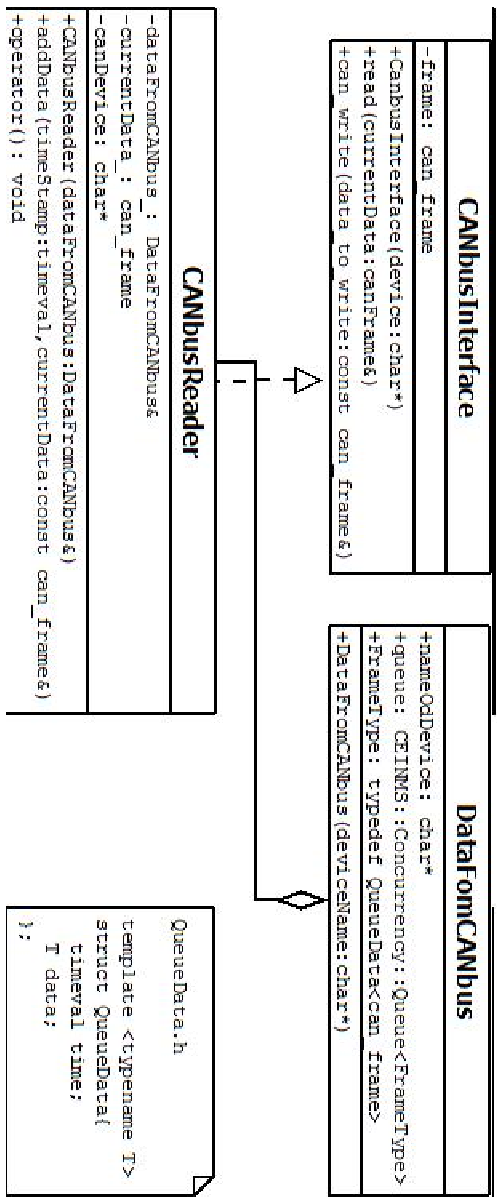
The developed code can be found on the REPOSITORY: `https://bitbucket.org/RehabEngGroup/h2`



```
.
├── CMakeLists.txt
├── contributors.txt
├── include
│   ├── anglesReceiverUi
│   │   ├── anglesrwindow.cpp
│   │   ├── anglesrwindow.h
│   │   └── anglesrwindow.ui
│   ├── commandSenderUi
│   │   ├── angleslimits.cpp
│   │   ├── angleslimits.h
│   │   ├── angleslimits.ui
│   │   ├── choosemotor.cpp
│   │   ├── choosemotor.h
│   │   ├── choosemotor.ui
│   │   ├── controltype.cpp
│   │   ├── controltype.h
│   │   ├── controltype.ui
│   │   ├── readsetpoint.cpp
│   │   ├── readsetpoint.h
│   │   ├── readsetpoint.ui
│   │   ├── selectstifness.cpp
│   │   ├── selectstifness.h
│   │   ├── selectstifness.ui
│   │   ├── senderwindow.cpp
│   │   ├── senderwindow.h
│   │   └── senderwindow.ui
│   ├── logObject.cpp
│   ├── logObject.h
│   ├── patternSenderUi
│   │   ├── patternwindow.cpp
│   │   ├── patternwindow.h
│   │   └── patternwindow.ui
│   ├── switchesReceiverUi
│   │   ├── switchesrwindow.cpp
│   │   ├── switchesrwindow.h
│   │   └── switchesrwindow.ui
│   └── torquesReceiverUi
│       ├── torquesrwindow.cpp
│       ├── torquesrwindow.h
│       └── torquesrwindow.ui

├── launch
│   ├── Gui.launch
│   └── Lancio.launch
├── lib
│   ├── CANbus
│   │   ├── CANbusInterface.cpp
│   │   ├── CANbusInterface.h
│   │   ├── CANbusReader.cpp
│   │   ├── CANbusReader.h
│   │   ├── CMakeLists.txt
│   │   ├── DataFromCANbus.cpp
│   │   ├── DataFromCANbus.h
│   │   └── QueueData.h
│   ├── CMakeLists.txt
│   └── Concurrency
│       ├── CMakeLists.txt
│       ├── Latch.cpp
│       ├── Latch.h
│       ├── Queue.cpp
│       └── Queue.h
├── msg
│   ├── genericFrame.msg
│   ├── mAngles.msg
│   ├── mSwitches.msg
│   ├── mTorques.msg
│   └── patternCommand.msg
├── package.xml
├── src
│   ├── anglesReceiverUi.cpp
│   ├── commandSenderUi.cpp
│   ├── patternSenderUi.cpp
│   ├── switchesReceiverUi.cpp
│   └── torquesReceiverUi.cpp
└── src_bbb
    ├── CMakeLists.txt
    ├── H2patternSender.cpp
    ├── H2ros.cpp
    ├── H2rosSender.cpp
    ├── include
    │   └── interruptFunctions.h
    └── interruptFunctions.cpp
```

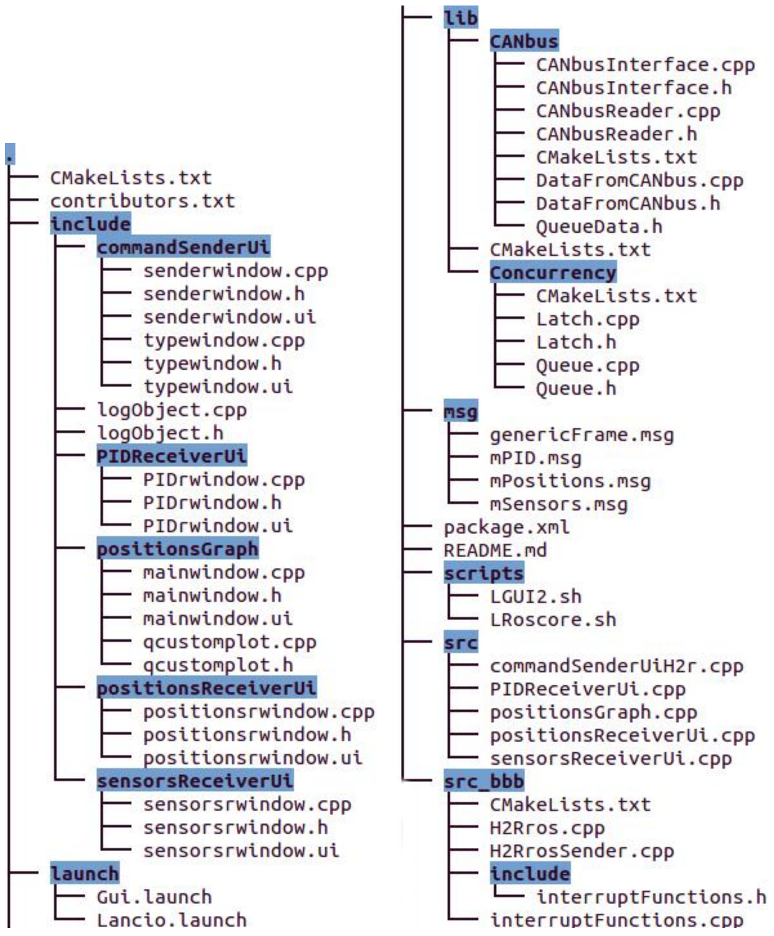
Figure 60: Tree diagrams for the h2 package


CANbusInterface
-frame: can_frame
+CanbusInterface(device:char*)
+read(currentData:canFrame&)
+can_write(data_to_write:const can_frame&)
DataFomCANbus
+nameOdDevice: char*
+queue: CEINMS::Concurrency::Queue<FrameType>
+FrameType: typedef QueueData<can_frame>
+DataFromCANbus(deviceName:char*)
CANbusReader
-dataFromCANbus_: DataFromCANbus&
-currentData_: can_frame
-canDevice: char*
+CANbusReader(dataFromCANbus:DataFromCANbus&)
+addData(timeStamp:timeval,currentData:const can_frame&)
+operator(): void

```
QueueData.h

template <typename T>
struct QueueData{
   timeval time;
   T data;
};
```

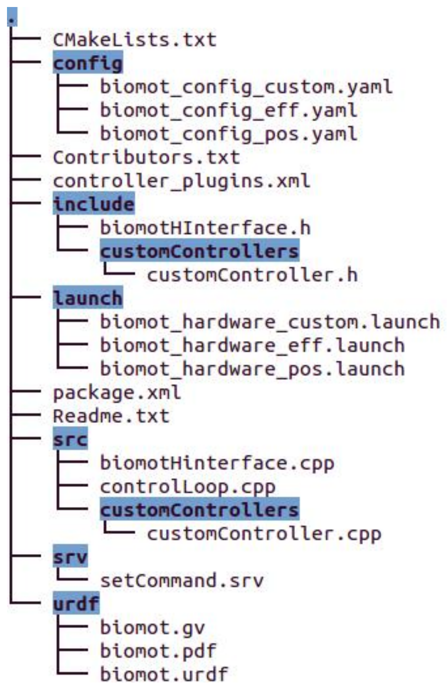
Figure 61: UML graph for the CAN library



```
.
├── CMakeLists.txt
├── contributors.txt
├── include
│   ├── commandSenderUi
│   │   ├── senderwindow.cpp
│   │   ├── senderwindow.h
│   │   ├── senderwindow.ui
│   │   ├── typewindow.cpp
│   │   ├── typewindow.h
│   │   └── typewindow.ui
│   ├── logObject.cpp
│   ├── logObject.h
│   ├── PIDReceiverUi
│   │   ├── PIDrwindow.cpp
│   │   ├── PIDrwindow.h
│   │   └── PIDrwindow.ui
│   ├── positionsGraph
│   │   ├── mainwindow.cpp
│   │   ├── mainwindow.h
│   │   ├── mainwindow.ui
│   │   ├── qcustomplot.cpp
│   │   └── qcustomplot.h
│   ├── positionsReceiverUi
│   │   ├── positionsrwindow.cpp
│   │   ├── positionsrwindow.h
│   │   └── positionsrwindow.ui
│   └── sensorsReceiverUi
│       ├── sensorsrwindow.cpp
│       ├── sensorsrwindow.h
│       └── sensorsrwindow.ui
├── launch
│   ├── Gui.launch
│   └── Lancio.launch
```

```
├── lib
│   ├── CANbus
│   │   ├── CANbusInterface.cpp
│   │   ├── CANbusInterface.h
│   │   ├── CANbusReader.cpp
│   │   ├── CANbusReader.h
│   │   ├── CMakeLists.txt
│   │   ├── DataFromCANbus.cpp
│   │   ├── DataFromCANbus.h
│   │   └── QueueData.h
│   ├── CMakeLists.txt
│   └── Concurrency
│       ├── CMakeLists.txt
│       ├── Latch.cpp
│       ├── Latch.h
│       ├── Queue.cpp
│       └── Queue.h
├── msg
│   ├── genericFrame.msg
│   ├── mPID.msg
│   ├── mPositions.msg
│   └── mSensors.msg
├── package.xml
├── README.md
├── scripts
│   ├── LGUI2.sh
│   └── LRoscore.sh
├── src
│   ├── commandSenderUiH2r.cpp
│   ├── PIDReceiverUi.cpp
│   ├── positionsGraph.cpp
│   ├── positionsReceiverUi.cpp
│   └── sensorsReceiverUi.cpp
└── src_bbb
    ├── CMakeLists.txt
    ├── H2Rros.cpp
    ├── H2RrosSender.cpp
    ├── include
    │   └── interruptFunctions.h
    └── interruptFunctions.cpp
```

Figure 62: Tree diagrams for the h2r package



```
.
├── CMakeLists.txt
├── config
│   ├── biomot_config_custom.yaml
│   ├── biomot_config_eff.yaml
│   └── biomot_config_pos.yaml
├── Contributors.txt
├── controller_plugins.xml
├── include
│   ├── biomotHInterface.h
│   └── customControllers
│       └── customController.h
├── launch
│   ├── biomot_hardware_custom.launch
│   ├── biomot_hardware_eff.launch
│   └── biomot_hardware_pos.launch
├── package.xml
├── Readme.txt
├── src
│   ├── biomotHinterface.cpp
│   ├── controlLoop.cpp
│   └── customControllers
│       └── customController.cpp
├── srv
│   └── setCommand.srv
└── urdf
    ├── biomot.gv
    ├── biomot.pdf
    └── biomot.urdf
```

Figure 63: Tree diagrams for the biomot_control package

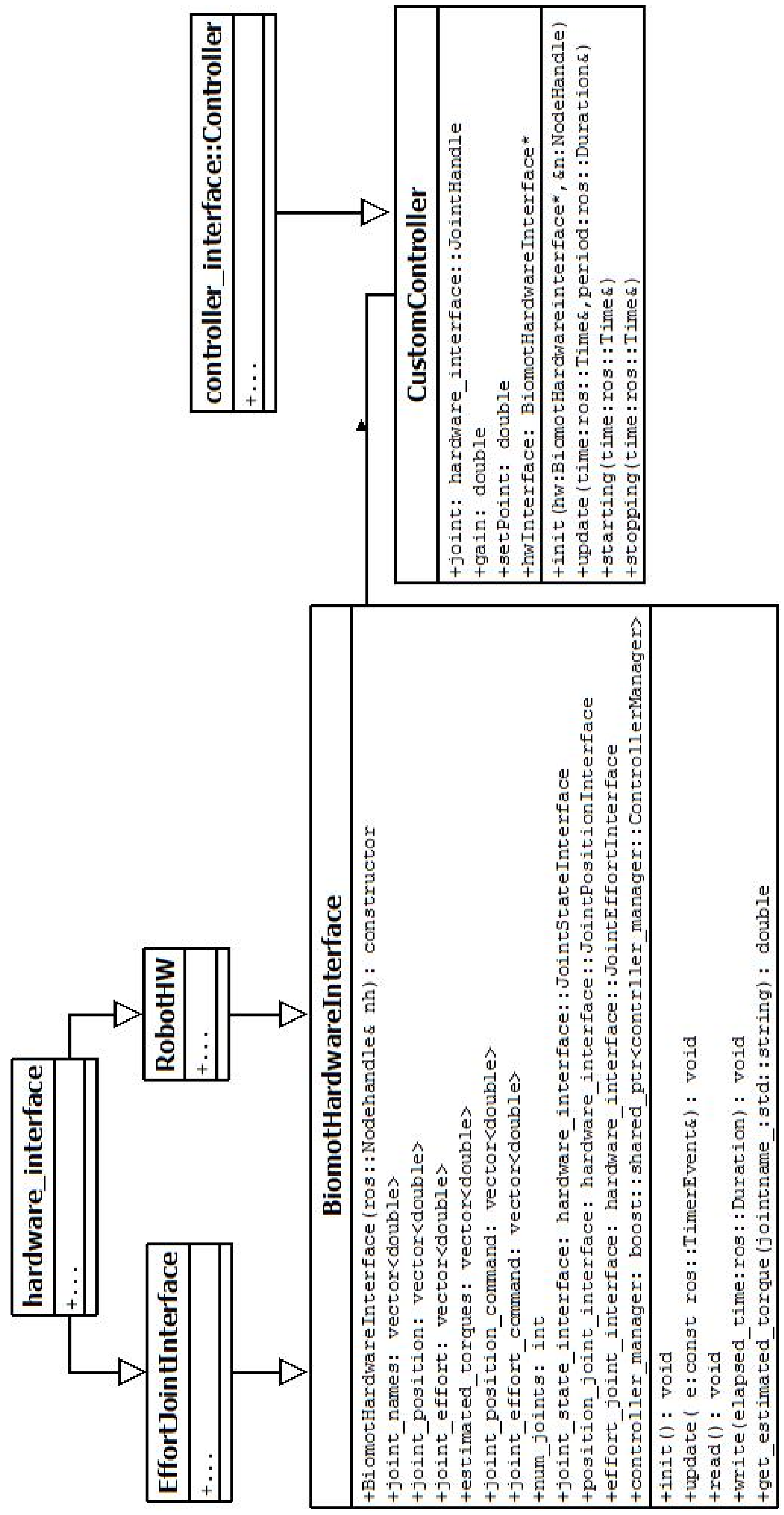

controller_interface::Controller
+...
CustomController
+joint: hardware_interface::JointHandle
+gain: double
+setPoint: double
+hwInterface: BiomotHardwareInterface*
+init(hw:BiomotHardwareInterface*,&n:NodeHandle)
+update(time:ros::Time&,period:ros::Duration&)
+starting(time:ros::Time&)
+stopping(time:ros::Time&)
hardware_interface
+...
RobotHW
+...
EffortJointInterface
+...
BiomotHardwareInterface
+BiomotHardwareInterface(ros::Nodehandle& nh): constructor
+joint_names: vector<double>
+joint_position: vector<double>
+joint_effort: vector<double>
+estimated_torques: vector<double>
+joint_position_command: vector<double>
+joint_effort_command: vector<double>
+num_joints: int
+joint_state_interface: hardware_interface::JointStateInterface
+position_joint_interface: hardware_interface::JointPositionInterface
+effort_joint_interface: hardware_interface::JointEffortInterface
+controller_manager: boost::shared_ptr<contrller_manager::ControllerManager>
+init(): void
+update(e:const ros::TimerEvent&): void
+read(): void
+write(elapsed_time:ros::Duration): void
+get_estimated_torque(jointname_:std::string): double

Figure 64: UML graph for the application of ROS Control to BioMot

BIBLIOGRAPHY

[1] *Introduction to Qcustomplot*. URL http://www.qcustomplot.com/.

[2] *Xenomai official web site*. URL https://xenomai.org/start-here/.

[3] Robot suit offers glimmer of hope to the paralysed. *Times of Malta*, 2011.

[4] *AM335x Sitara Processors Technical Reference Manual*. Texas instruments, 2014. pg 4657.

[5] Tarek Sobh Ayssam Elkady. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012(959013), 2012.

[6] beagleboard.org. *OpenRTDynamics: a framework for signal processing and real-time control. Port to Beagleboard-bone*. URL http://beagleboard.org/project/OpenRTDynamics/.

[7] Beagle bone community. *Official OS releases for the Beagle Bone Black board*. URL http://beagleboard.org/latest-images.

[8] Dave Coleman. *the ros-control-boilerplate package*. URL https://github.com/davetcoleman/ros_control_boilerplate.

[9] Gerald Coley. *BeagleBone Black System Reference Manual*. 2013.

[10] CONFIG PREEMPT RT community. *Real time Linux wiki*. . URL https://rt.wiki.kernel.org/index.php/Main_Page.

[11] Element 14 community. *Using the BBB's Programmable Real-time Units*. . URL http://www.element14.com/community/community/designcenter/single-board-computers/next-gen_beaglebone/blog/2013/05/22/bbb--working-with-the-pru-icssprussv2.

[12] ROS community. *Main concepts of ROS*. . URL http://wiki.ros.org/ROS/Concepts.

[13] ROS community. *Introduction to ROS*. . URL http://wiki.ros.org/ROS/Introduction.

[14] ROS community. *ROSLaunch documentation*. . URL http://wiki.ros.org/roslaunch.

[15] ROS community. *ROSLaunch for remote nodes trough SSH*. . URL http://wiki.ros.org/roslaunch/XML/machine.

[16] ROS community. *Robots using ROS*. . URL http://wiki.ros.org/Robots.

[17] ROS community. *ROS Services description*. . URL http://wiki.ros.org/Services.

[18] ROS community. *ROS distribution page*. . URL http://wiki.ros.org/indigo/Installation/UbuntuARM.

[19] ROS community. *ROS-Control package description*. . URL http://wiki.ros.org/ros_control.

[20] Cometa company. *Brouchure of the Cometa Wave Wireless EMG device*. URL www.cometasystems.com/cometasystems/it/products/wave-wireless-emg.

[21] Steve Corrigan. *Introduction to the Controller Area Network(CAN)*. Texas instruments, 2008.

[22] Gazebo developers. *ROS-Control tutorial with Gazebo*. . URL http://gazebosim.org/tutorials/?tut=ros_control.

[23] Git developers. *A tutorial introduction to Git*. . URL http://git-scm.com/docs/gittutorial.

[24] ROS developers. *The controller interface class*. . URL http://wiki.ros.org/pr2_controller_interface.

[25] ROS developers. *rosrt package description*. . URL http://wiki.ros.org/rosrt.

[26] ROS developers. *Running a realtime joint controller for the PR2 robot*. . URL http://wiki.ros.org/pr2_mechanism/Tutorials/Running%20a%20realtime%20joint%20controller.

[27] ROS developers. *Joint Safety Limits Explained*. . URL http://wiki.ros.org/pr2_controller_manager/safety_limits.

[28] ROS developers. *Writing a realtime joint controller*. . URL http://wiki.ros.org/pr2_mechanism/Tutorials/Writingarealtimejointcontroller.

[29] ROS-Control developers. *Writing a generic hardware-interface*. . URL https://github.com/ros-controls/ros_control/wiki/hardware_interface.

[30] ROS-Control developers. *Writing a customized controller*. . URL https://github.com/ros-controls/ros_control/wiki/controller_interface.

[31] Linux documentation. *CAN Socket documentation*. URL https://www.kernel.org/doc/Documentation/networking/can.txt.

[32] Brad Martin Dr. Jeremy H. Brown. *How fast is fast enough? Choosing between Xenomai and Linux for real-time applications*. URL https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf.

[33] Niedermeyer E. and da Silva F.L. *Electroencephalography: Basic Principles, Clinical Applications, and Related Fields*. 2005.

[34] elinux.org. *Beagle Bone Black OS images setup*. . URL http://elinux.org/BeagleBoardUbuntu.

[35] elinux.org. *Can-utils description*. . URL http://elinux.org/Can-utils.

[36] elinux.org. *Supported wi-fi adapters for Beagle Bone Black*. . URL www.elinux.org/Beagleboard:BeagleBoneBlack_WIFI_Adapters.

[37] elinux.org. *EBC Xenomai*. . URL http://elinux.org/EBC_Xenomai.

[38] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012.

[39] Biomot group. *Biomot description in the Neural Rehabilitation Group website*. URL http://neuralrehabilitation.org/projects/bioMot/research.html.

[40] gTec. *Description of the g.GAMMA EMG system*. URL http://www.gtec.at/Products/Electrodes-and-Sensors/g.GAMMAsys-Specs-Features.

[41] Juan C. Moreno Guillermo Asin. *Physically based simulations with partial demonstrators I*. October 2014.

[42] H2R. *H2R project web site*. URL http://www.h2rproject.eu/.

[43] S. Hadim and N. Mohamed. Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, 7(3), 2009.

[44] Brussels BelgiumSugar T.G. ; Vanderborght B. ; Hollander K.W. ; Lefeber D. Ham, R. Vrije Univ. Brussel. Compliant actuator designs. *Robotics and Automation Magazine, IEEE*, 16(3), 2009.

[45] Hitachi Hiraku Toyooka. *Evaluation of Real-time Property in Embedded Linux*. 2014. URL http://events.linuxfoundation.org/sites/events/files/slides/toyooka_LCJ2014_v10.pdf.

[46] http://dev.ardupilot.com. *Building ArduPilot for BeagleBone Black on Linux*. URL http://dev.ardupilot.com/wiki/building-the-code/building-for-beaglebone-black-on-linux/. Section 5.

[47] Mark Summerfield Jasmin Blanchette. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. Preface.

[48] Michael H. Leonard. *Cross compilation for the BBB within Eclipse*. URL http://michaelhleonard.com/cross-compile-for-beaglebone-black/.

[49] Alexander Sproewitz Dongming Gan Nikos G. Tsagarakis Matteo Laffranchi, Hide SUMIOKA. *Compliant Actuators*. Adaptive Modular Architectures for Rich Motor Skills, March 2011.

[50] Fabien Le Mentec. *Using the C language to program the am335x PRU*. URL http://www.embeddedrelated.com/showarticle/603.php.

[51] PREEMPT patch developers. *How to use the preemption patches for linux*.

[52] qnx.com. *Texas Instruments AM335x Beaglebone/Beaglebone Black Board Support Package*. URL http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/TiAm335Beaglebone.

[53] Hema Priya B Rajam AP. *CAN interface testin on Beagle Bone*. Embedded Division, Accel Frontline Ltd, 2013.

[54] Tomislav Bacek Karen Junius Heidi Cuypers Ramazan Unal, Marta Moltedo. *Conceptual design of modular compliant actuators for use in the knee and the ankle joint of a lower limb exoskeleton*. October 2014.

[55] Technaid. *Inertial Measurements Units technical specifications*. URL http://www.technaid.com/en/products/techimu.

[56] Johan Thelin. *Using CMake to Build Qt Projects*. URL https://qt-project.org/quarterly/view/using_cmake_to_build_qt_projects.

[57] Juan C. Moreno Weiguang Huo, Samer Mohammed and Yacine Amirat. Lower limb wearable robots for assistance and rehabilitation: A state of the art. *IEEE SYSTEMS JOURNAL*, PP, 2014.

[58] www.embedded things.com. *Enable Can-bus on the Beagle Bone Black*. URL http://www.embedded-things.com/bbb/enable-canbus-on-the-beaglebone-black/.

[59] www.gnu.org. *Cross-Compilation*. URL http://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Cross_002dCompilation.html.