BioMot Project

The BioMot project is an international 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 over-ground gait exoskeletons by means of mixture of bioinspired control, actuation and learning approaches" [1]. The objective is to show how the embodiment of bioinspired and architectural mechanisms can allow a user to conveniently alter the behavior of wearable robots 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.

BioMot exoskeleton will apply adaptive assistance as a function of real-time estimation of human effort provided by a detailed *neuromusculoskeletal* model that computes neuromuscular activity (sur- face electromyography, also known as 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 inferred 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 over ground 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.

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* middleware (ROS). 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 [2]. This frame is becoming more and more popular in robotic application and is already being employed by a large number of complex robots [3].

The use of a framework 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 document presents the required steps for the application of the ROS middleware to the BioMot exoskeleton.

Electronic description

The electronic part of the exoskeleton is constituted by an ARM-based board, also known as HAL (Hardware Abstraction Layer), six joint boards, one controller board (BeagleBone Black board) and two CAN BUS networks connecting all these elements.
The ARM board is in charge of collecting the data from all the joint boards and create packages with that information in order to be sent to the controller board. The small size of this board (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.

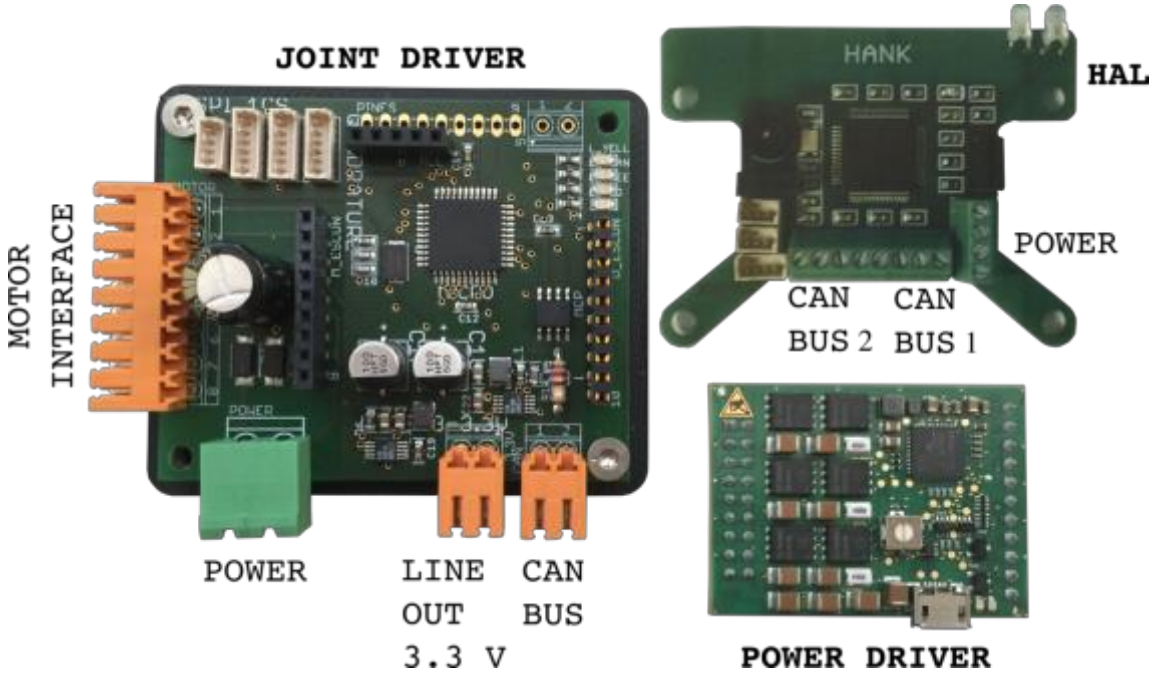


Figure 1. Electronic boards used in the exoskeleton

The HAL board can access two CAN-BUS lines; one for collecting information and sending commands to the joints and the other for sending information to the controller board. The computational power of the board relies on a STMicroelectronis ARM microcontroller STM32F405RG running at 168 MHz.
The joint boards are in charge of running control algorithms and data acquisition from the different joint's sensors: angular encoders, force resistive resistors, contact sensors, etc.

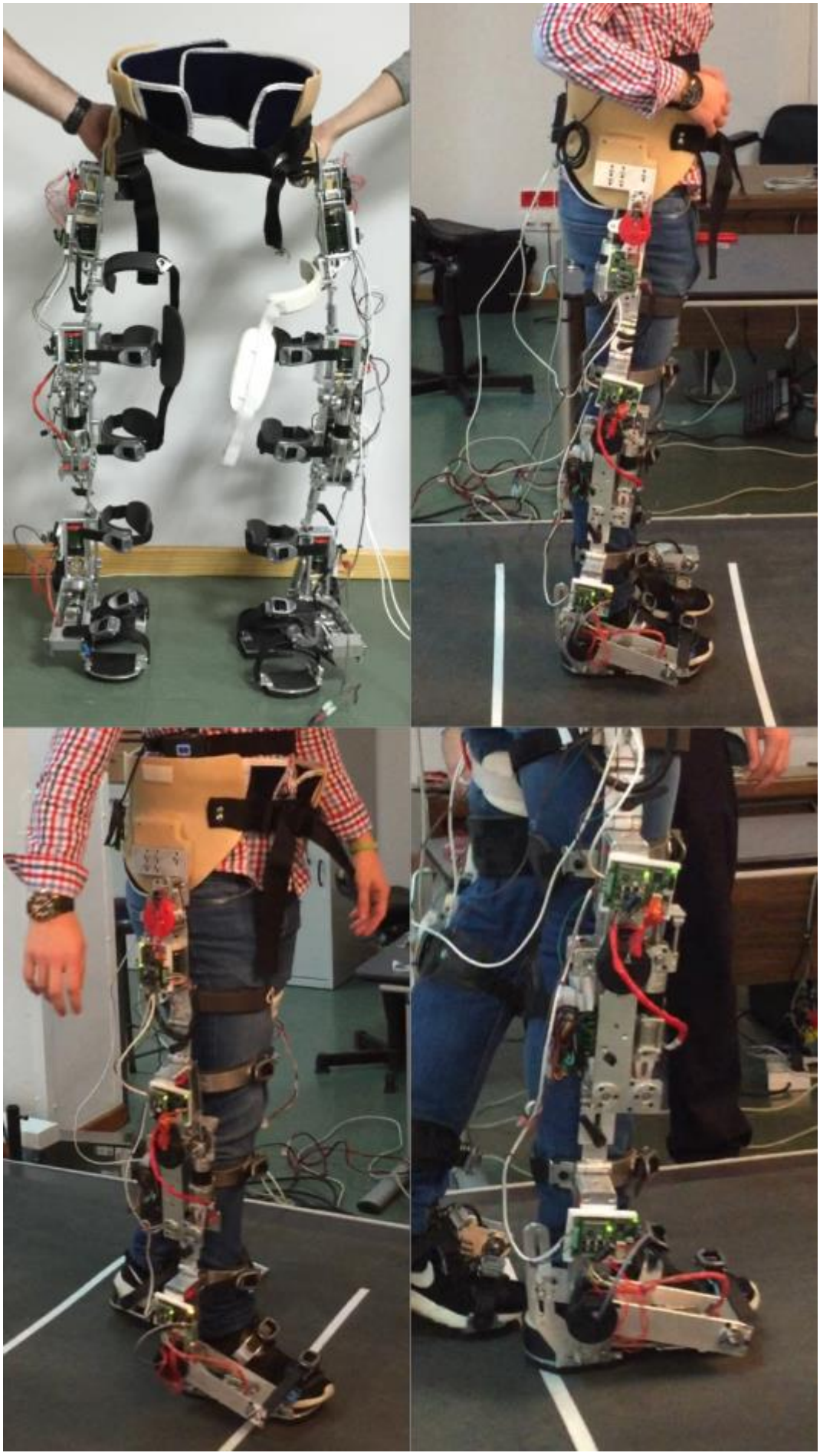


Figure 2. BioMot exoskeleton

A small data packet of six bytes aggregates the sensor's information on each joint and these data are sent to HAL board at 100 Hz. The boards also include a specifically designed driver for the brushless motors. Each joint is controlled independently by the associated joint board. Different types of control are possible for each joint; the most basic ones are position and torque control. The commands are received by CAN.

Figure 3. CAN messages exchanged between the joints drivers and the BeagleBone Black in both directions.

CAN communication

Since CAN bus is the chosen communication interface between all the electronic boards used in the exoskeleton a brief description of its working principles is given.

CAN bus (Controller Area Network) is a vehicle bus standard designed to allow microcontrollers 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. 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 which 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 message 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** that converts the data stream from CAN bus 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 CAN bus levels.

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, au- tomatically avoids data collision and corrects errors regarding to data packets transmission.

CAN buses in BioMot project are configured at 1Mbps. The CAN bus among the HAL board and the joint boards uses standard packets of 6 bytes. The CAN bus between the HAL board and the controller board (BeagleBone Black board) uses extended packets of 8 bytes.

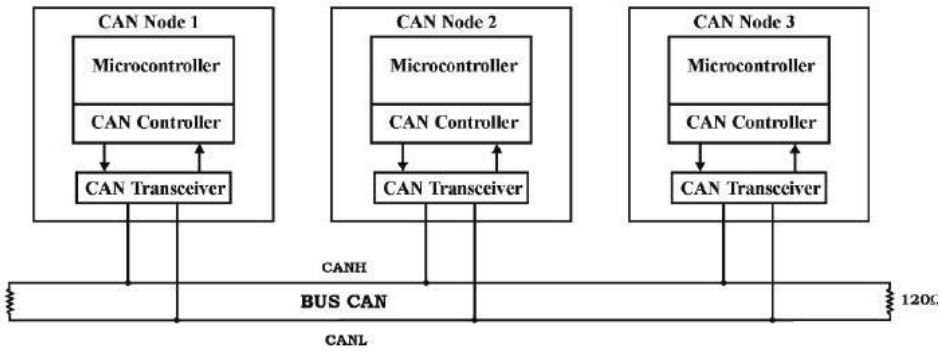


Figure 4. Components in a CAN network.

The BeagleBone Black board

BeagleBone Black board is a low-cost, 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.

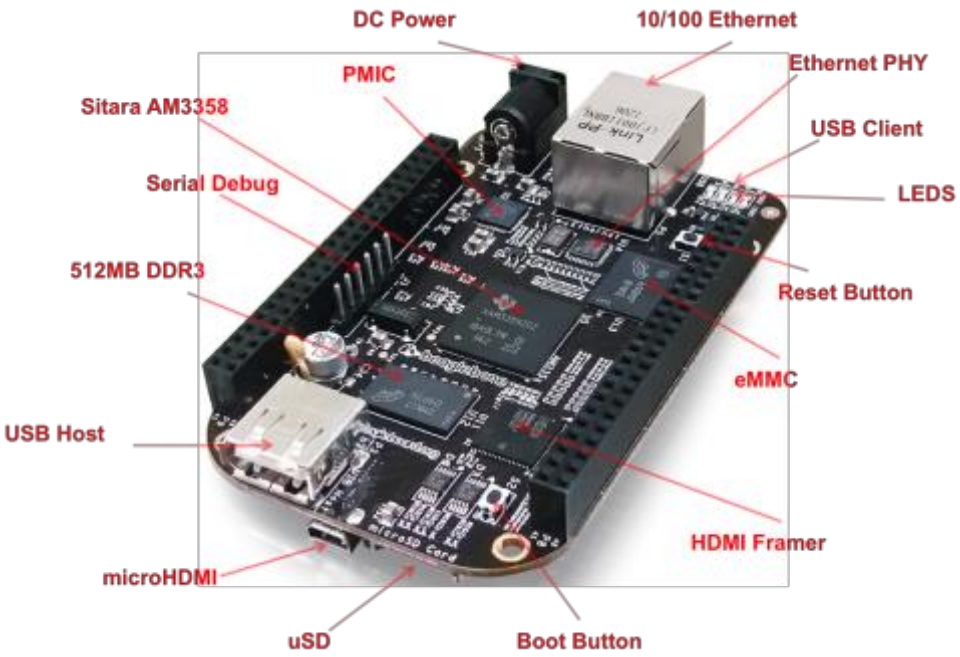


Figure 5. BeagleBone Black board and its different parts.

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 resume of the board's features is in figure 6.

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 the electronic boards controlling the joints for real-time communication.
Another important requirement is the support for ROS (Robot Operating 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 6. Features of the BeagleBone Black board.

	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 7. Comparison between several embedded boards available nowadays.

Using a well-developed and open-source operating 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 into the exoskeleton, low power consumption is essential for achieving a long time duration of system's battery.

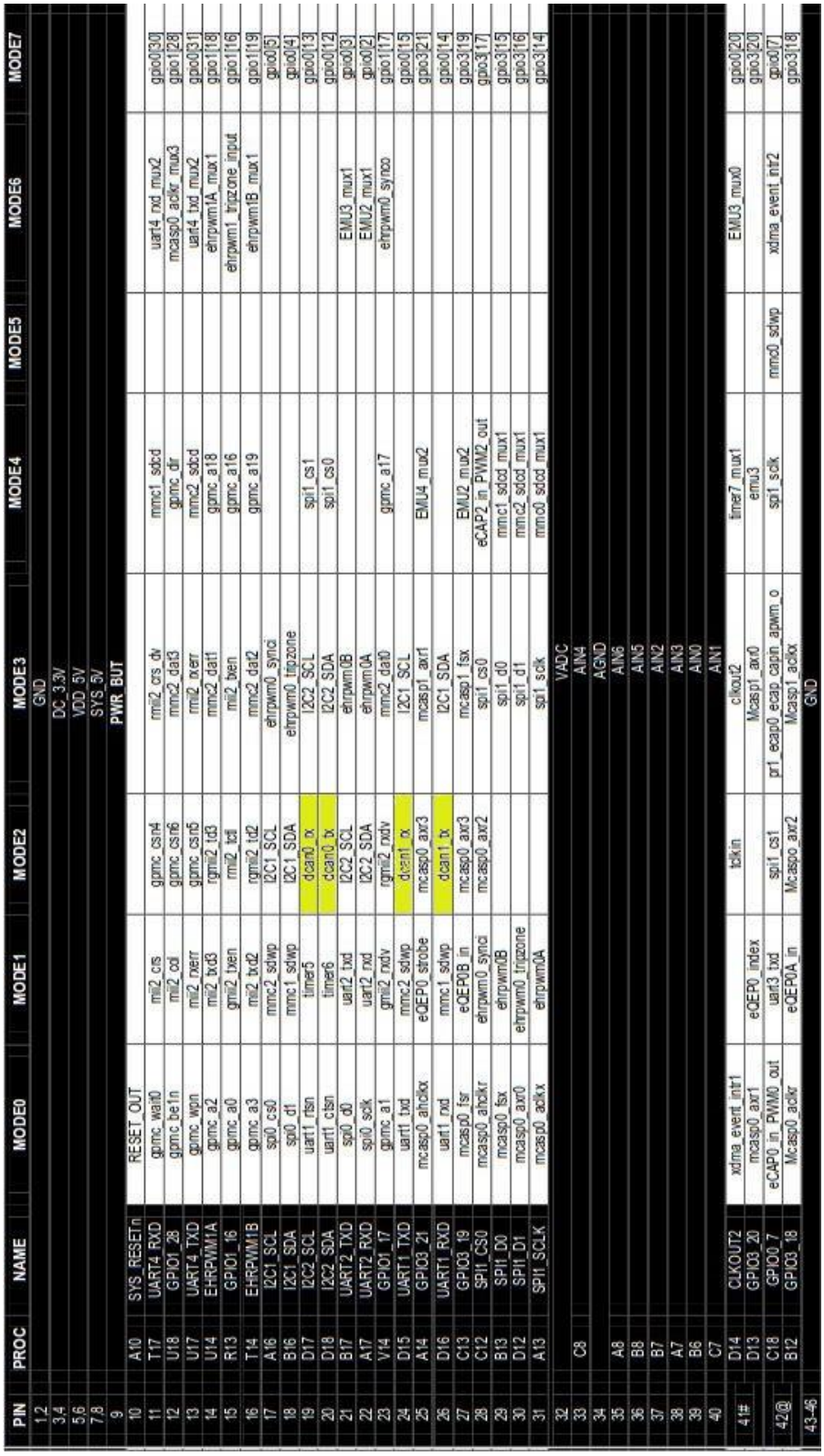
Finally, the possibility of finding the board on the market is, of course, crucial. Figure 7 shows a comparison between BBB and other similar boards that were available on the market at the moment of the choice by the BioMot group (the newly released Raspberry Pi 2 could not be taken into consideration).

Set-up of the operating system

The BeagleBone Black board is shipped with a Linux-Debian OS (Operating 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 BeagleBone Black board can be found on-line, for example in the official distribution page [4]. An Ubuntu OS image could be found on the *elinux* site [5]. 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 BeagleBone Black board 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.

Set-up of ROS middleware

As will be described later on, 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. 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 can be found in the ROS distribution web page [6]. The download of the ROS middleware, of course, requires an internet connection that can be easily gained connecting the board via the Ethernet port to an internet-provided network.



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_rxer	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_td	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 8. Expansion header P9 pinout.

Set-up of the CAN interface

Even if the BeagleBone Black board is natively equipped with a CAN controller integrated in the microprocessor (as described in [7]), 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 BeagleBone Black board expansion headers, it is possible to use an expansion cape in order to provide the missing elements. As previously said, the BeagleBone Black board 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 8) 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. 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 to work.

As shown in figure 8, 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 [8]. 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 [8].
The required expansion cape was built using a VP230 transceiver from TI (Texas Instruments), as described in [9]. The BeagleBone Black board with the CAN expansion cape can be seen in figure 9.

Figure 9. ¬Beagle Bone Black with a customized CAN-cape expansion.

In order to understand if the device was working properly, a software named *can-utils* was set-up on the system [10]. This utilities, based on the socket-can libraries (described in detail later), 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 BeagleBone Black board needs to be powered with an extern power supply since the power/tension level provided by an USB cable is not high enough. With such configuration the BeagleBone Black board was able to send/receive data from other devices. Here is an example of data from one of the joint's 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
```

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/rc<runlevel>.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.

Set-up of the WiFi interface

The 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 very long Ethernet cable indeed. 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 WiFi interface, but it allows for the installation of a WiFi adapter through the USB host. Since the BeagleBone Black board's OS uses a quite old kernel, many WiFi adapters are not supported, so the device was chosen from a list of tested devices (See reference [10]). 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 process, 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 its DHCP service.

Since the ground and power planes of the BeagleBone Black board's HDMI port are right below the USB port, they can decrease the power of 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 through the device tree overlay. To do this, the file */boot/uEnv.txt* on the BeagleBone Black board was modified adding the following line:

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

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"
```

THE ROS MIDDLEWARE

Introduction

An exoskeleton system is usually composed of a large number of sensors, devices, and consequently, many different programs that need constantly interact with each other for achieving the final control. This is true also for the BioMot system, which 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 benefit the 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 robotics.
A possible way to achieve these 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.

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 10. 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.

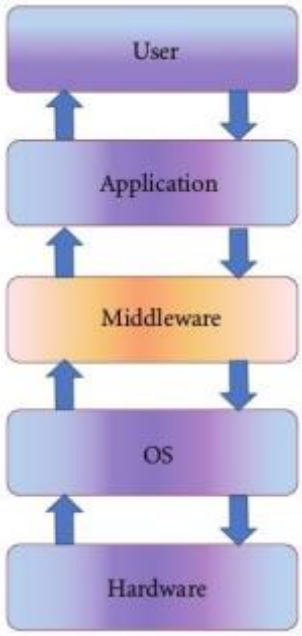


Figure 10. Middleware layers.

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 *robotics 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 software 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 costs.

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 they want to modify and improve their component, they only need 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.

A list of common robotics middleware frameworks is: Player, CLARAty, ORCA, MIRO, UPNP, RT-Middleware, ASEBA, MARIE, RSCA, OPRoS, ROS, MRDS, OROCOS, SmartSoft, ERSP, Skilligent, Webots, Irobotaware, Pyro, CARMEN, RoboFrame, etc. In particular ROS (Robot Operating System) is becoming one of the most utilized robotic middleware nowadays due to the open source community supporting the project.

The Robot Operating System (ROS) framework

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.
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 STAIR project (STanford AI Robot). 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.

Figure 11. ROS logo.

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, 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.
- Cross-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.

Releases and contributions

All the code that ROS builds is completely open-source (BSD license), contributions to its expansion are continuously made by a large community anyone can join.
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 Box Turtle (March 2, 2010).
- ROS C Turtle (August 2, 2010).
- ROS Diamondback (March 2, 2011).
- ROS Electric Emys (August 30, 2011).
- ROS Fuerte Turtle (April 23, 2012).
- ROS Groovy Galapagos (December 31, 2012).
- ROS Hydro Medusa (September 4th, 2013).
- ROS Indigo Igloo (July 22nd, 2014).

The details on the distributions can be found in the project's repository documentation. In this work is used the last ROS edition (Indigo).

ROS concepts

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

- 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 (an xml files called package.xml) provide metadata about a package, including its name, version, description, license 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.

- 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 wouldn't 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 sub- scribe to multiple topics. In general, publishers and subscribers are not aware of each other's 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.
  When a message is published associated to a topic, it is broadcasted to all the nodes who are subscribed to that topic; the ROS-Master handles the low-level connection (usually creating TCP-IP communication sockets), so the user doesn't need to specify any address.
- **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.

- 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.

- Higher level concept.

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 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.

ROS in BioMot

BioMot exoskeleton is composed of a large number of devices and programs that need to be integrated in order to reach the final control objective so 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 12 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 have 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 can store also 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 control can be easily moved from one device to another or, depending on the project's objectives, be divided into more nodes implementing different activities.

In particular, the following functionalities will be required:

- The nodes located into the BeagleBone Black board provide an interface between sensors, actuators and the board. Communicating with these nodes, the system will be able to acquire information relative to the sensors embedded in the exoskeleton (positions and interaction torques), other useful information published by the joints boards and to send commands as position/torque set-points.
- EMG node will acquire data through 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. 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).
- 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.

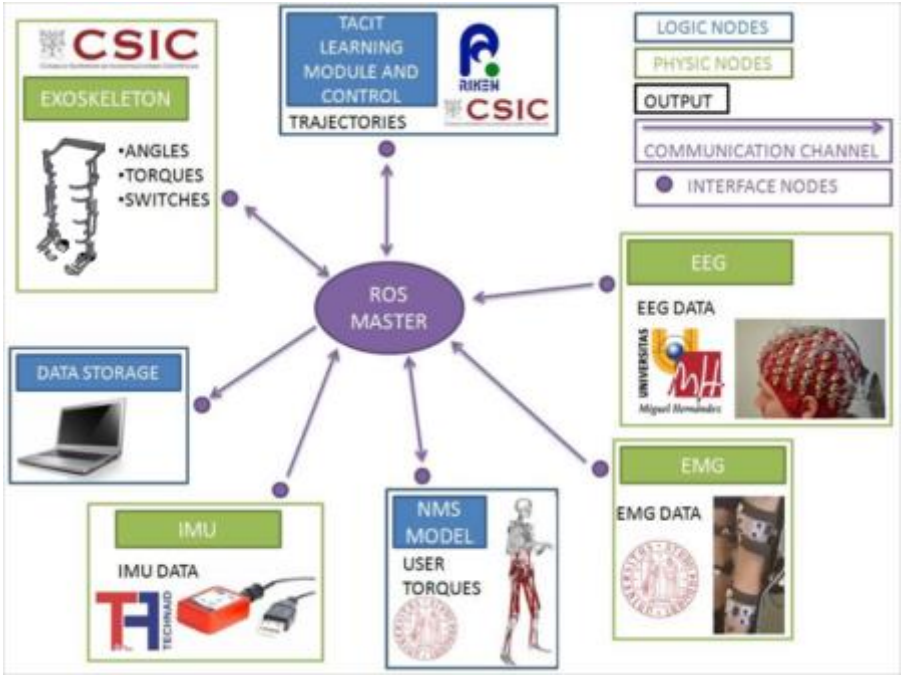


Figure 12. Communication architecture using ROS.

Reading data through the can network (SocketCAN API and proprietary library)

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. 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 allows 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 to a specific CAN device. Frames of data can be sent/received using an adequate frame type (struct *canframe*) 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.

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.

ROS nodes

When using the ROS middleware an executable is called node. All the nodes that need to work together to achieve an objective are grouped into the same package. To 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 package.xml manifest and in a CMakeLists.txt file. The first contains general information about the package (its name, its authors, its license, the pre-required dependences for its compilation etc). The second includes the required information 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 BeagleBone Black board from the code of the nodes that will be 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.

RX node

This node, running on the BeagleBone Black board, has the function of receiving data from the exoskeleton (from the HAL board) and to forward them to the ROS network in form of messages. In order to do so, it uses the communication CAN libraries described before for reading data from the CAN network. This node can understand which kind of CAN message is received checking the CAN message identifier, and then it puts the data in their related ROS messages and publishes them on a specific ROS topic.

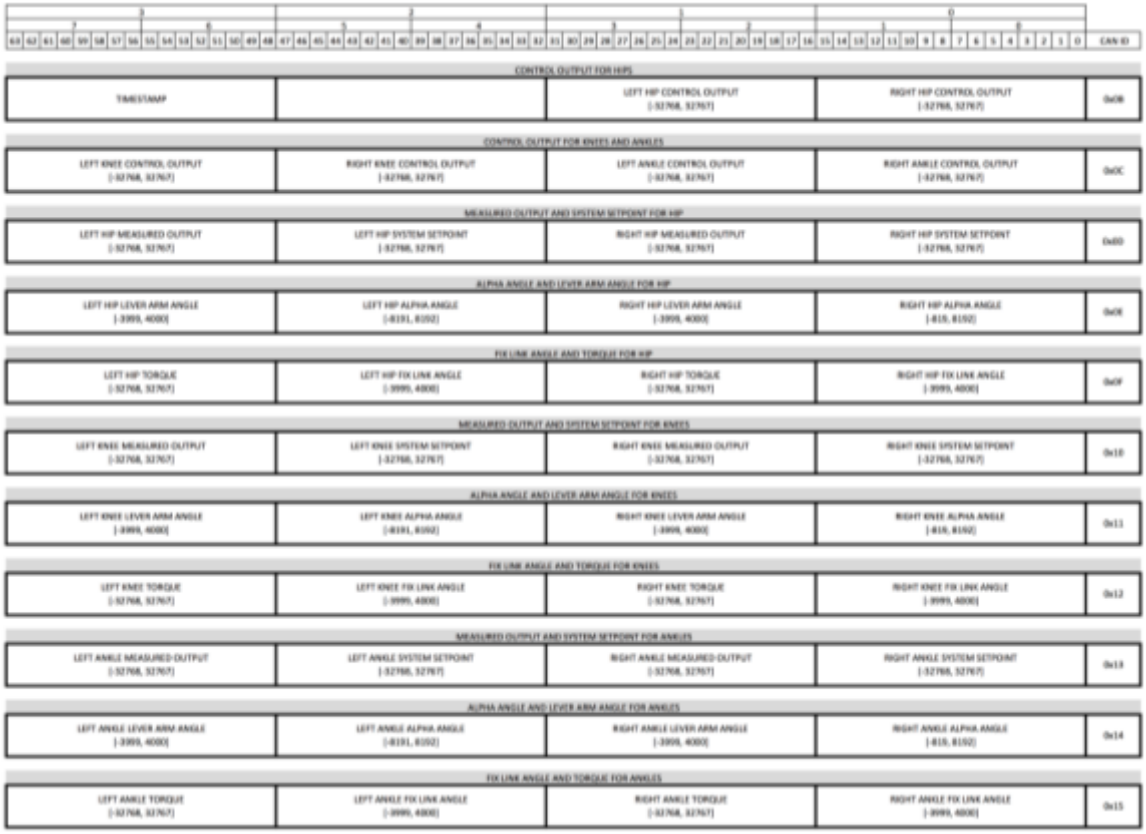


Figure 13. CAN messages exchanged between the ARM-HAL board and the BeagleBlack Bone board.

Customized message formats have been defined in order to transmit data within the ROS network. Only one default message type has been used in order to broadcast the information of torque (in Nm) and angle (in radians) of the same joint as a closed set in the same message. This message is the *JoinsState* message, publish under the topic *JointState*.

The other message formats created are:
- JointAnglesMessage.msg: This message format is used for transmitting in the ROS network all the angles gathered from all the joints of the exoskeleton in a certain time instant.

```
Header header
float32[] fix_link_angle    # 6 cells for 6 joint angles
float32[] alpha_angle
float32[] lever_arm_angle
```

The messages of this format are published under the topic *joint_angles_topic*.

- PIDMessage.msg: This message format is used for transmitting in the ROS network all the information related to PID controllers running in each joint of the exoskeleton.

```
Header header
float32[] setpoint
float32[] control_output
float32[] measured_output
```

The messages of this format are published under the topic *pid_topic*.

- ParallelSpringMessage.msg: This message format is used for transmitting in the ROS network the information related with the parallel spring mechanism present in each knee of the exoskeleton.

```
Header header
float32 right_knee_fix_link_angle
float32 left_knee_fix_link_angle
int32   right_number_revolutions # Position of the motor attached to the parallel
int32   left_number_revolutions  # spring in the right and left knee
int32   right_limit              # Flag to indicate that the parallel spring is
int32   left_limit               # fully elongated in each knee
```

The messages of this format are published under the topic *parallel_spring_topic*.

The signature *int32* and *float32* used by ROS correspond to the datatypes *int* (4 bytes in complement 2 format) and *float* (4 bytes in IEEE754 format) of the C++ language. This is also true for Python, because its compiled default libraries are written in C/C++.

Since data relative to the exoskeleton joints and data from the *EMG* signals are read by separated devices, a synchronization system has to be used. The *RX* node is also responsible of setting the reference time on the system, in order to have all the devices synchronized. This functionality is essential for data synchronization with other devices.
The elapse time with respect to that registered reference time is set within the field header, of type Header, in each new message, with resolution of nanoseconds.
The reference time is registered by the BeagleBone Black board whenever a hi-to-low transition of 3.3V trigger signal from an external resource occurs into its *GPIO7* (P9-42).
In order to achieve this goal, the *RX* node runs a second thread which is in charge of detecting a transition from 0V to 3.3V in the *GPIO7*. When this event occurs the RX node resets the reference time to 0 seconds. At the same time, the *Cometa EMG* acquisition device can be set to log data.
Also the Windows PC acquiring the *EMG* data has to use a different middleware to publish *EMG* data since the ROS framework is still in an experimental stage on Windows OS. The middleware used by this Windows PC is *YARP* (*Yet Another Robot Platform)*. Thank to recent efforts for getting interoperability between the two middlewares, this YARP-based *Cometa EMG* logger module communicates with the ROS-Master and publishes ROS-compatible messages associated to the *emgData* ROS-topic.
Another notebook, running a Linux OS, hosts the node CEINMS (*Calibrated EMG-Informed Neuromusculoskeletal Modelling Toolbox*) and displays the acquired data through the *rqt_plot* utility.

Due to the modularity of these middlewares, a different set-up could be made with a different distribution of the running nodes; for example, both the ROS-Master and the *CEINMS* model could be executed on the BeagleBone Black board. 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 ROS-network (it can be for example the same one visualizing the data) in a *rosbag* file or in a txt format.

Graphical User Interfce nodes

By GUI nodes we mean ROS-Nodes that are attached to Graphical User Interfaces that can be used for displaying data or inserting data for commanding the exoskeleton. To build these interfaces, the Qt library, the “real-time” *QCustomPlot* library (compatible with the Qt library) and the Qt-Creator software were used.

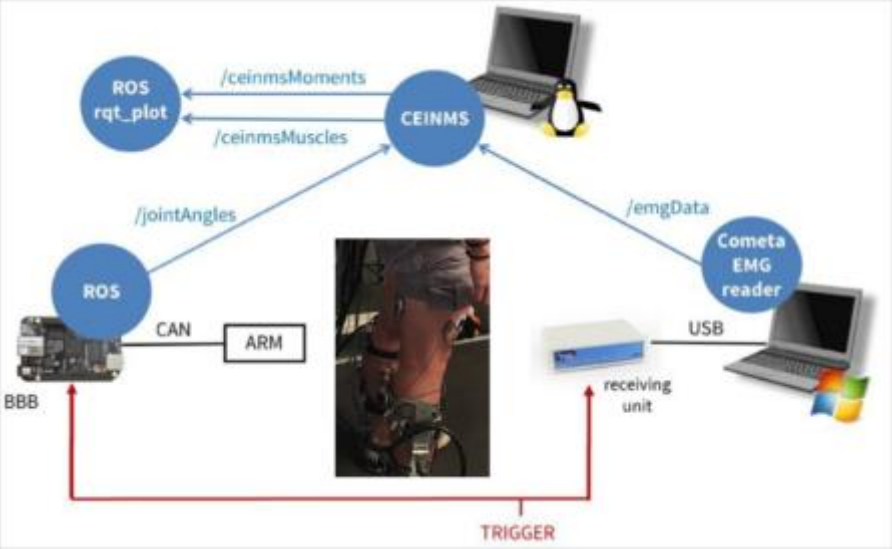


Figure 14. . Communication schema among devices.

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. *Qt* uses standard C++ with extensions including signals and slots that simplifies handling of events, and this helps in development of GUI.

QCustomPlot is a plotting library focused on making good looking 2D plots, graphs and charts, as well as offering high performance for real-time visualization applications (it provides publication quality).

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 Cmake was used for compilation.

An alternative to the use of these graphical libraries is the use of the *rqt_plot* tool. This tool allows to plot graph data that is being published on a specified ROS topic. This tool requires the set-up of other ROS components (graphic packages) and is less customizable than the provided programmed solution.

In figure 15 are shown the GUI's used for displaying data to the user.
The *joint_angles_gui_node* is subscribed to the *joint_angles_topic* and associated with the *Realtime Joint Angles GUI.* In this GUI the user can view the evolution of the fix link angle, lever arm angle and alpha angle of each joint.
The *joint_information_gui_node* is subscribed to the topics *JointState* and *joint_angles_topic* and associated with the *Joint Information GUI.* In this GUI the user can view numerically the value of the fix link angle, lever arm angle, alpha angle and torque of each joint.
The *parallel_spring_information_gui_node* is subscribed to the topic *parallel_spring_topic* and associated with the *Parallel Spring Information GUI.* In this GUI the user can view numerically the value of the fix link angle of each knee, the number of revolutions of each motor associated with the parallel spring and a red led that lights whenever the parallel spring is fully elongated.
The *pid_gui_node* is subscribed to the topic *pid_topic* and associated with the *Realtime PID GUI.* In this GUI the user can view the evolution of the setpoint, measured output and control output of each actived PID controller in each joint.

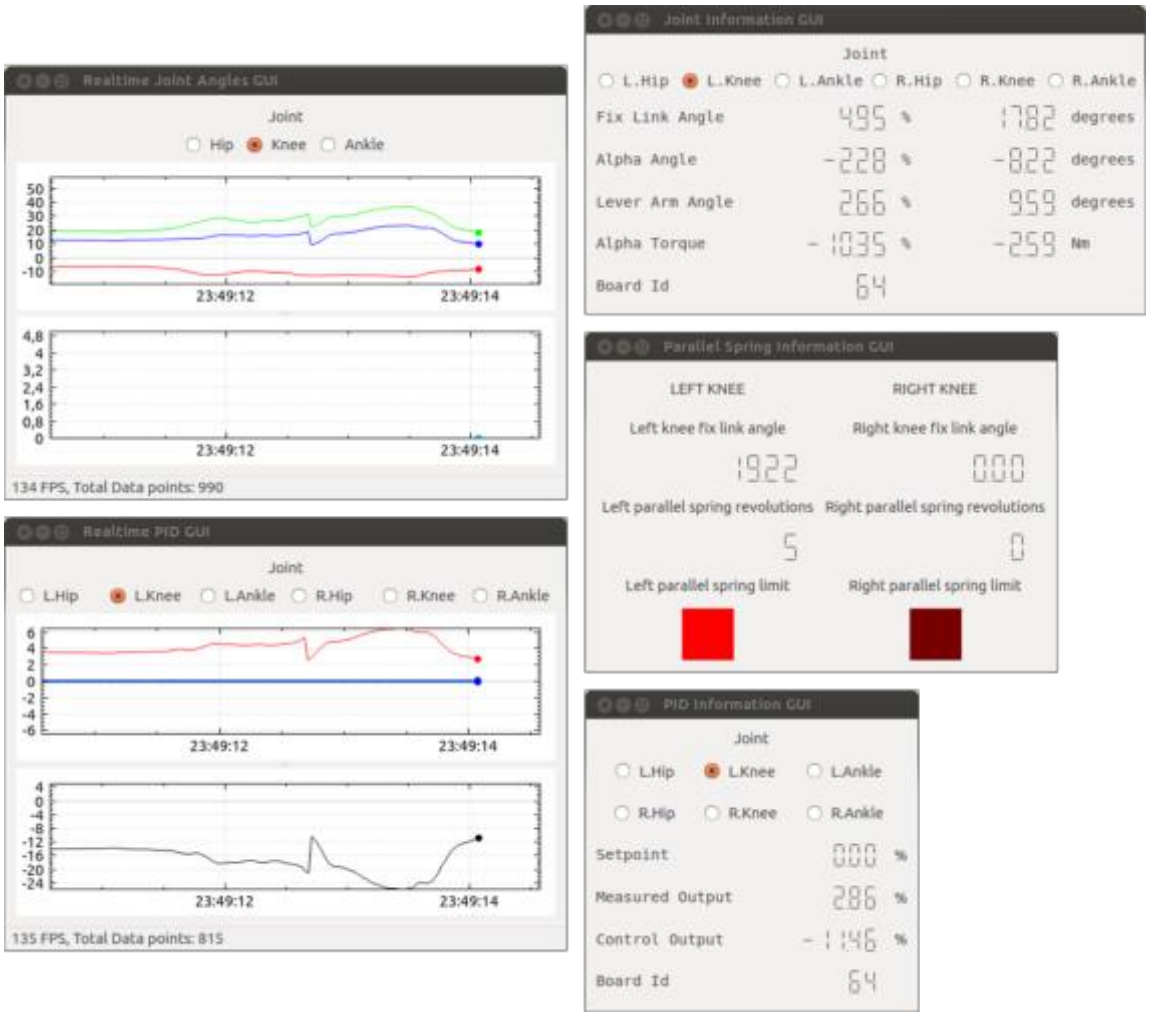


Figure 15. Graphical user interfaces used to display data or send commands to the joint drivers.

The *pid_information_gui_node* is subscribed to the *pid_topic* and associated with the *PID Information GUI*. In this GUI the user can view numerically the value of the setpoint, measured output and control output of each actived PID controller in each joint.

The *commands_gui_node* publishes command messages associated to the topics *tx_topic* and *fsrs_thresholds_topic*. In this GUI the user can introduce information to manage the behavior of all exoskeleton's joints and the thresholds for the feet's pressure sensors (FSR – Force Sensitive Resistor). This GUI offers to tabs. In the JOINT tabs can be found all the commands related with the exoskeleton's joints. From this tab the user can set the zero reference position of each position sensor (optical encoder and magnetic encoder) located in a joint. The GUI offers the opportunity of select which type of control must be used in a joint. Several kind of controls are provided, among them, the more basic ones are: position control and torque control.

It's also possible to assign values to the variables that are part of the controller running in each joint (Kp, Kd, Ki). From this tab the user can select the setpoint to be reached by the controller running in a joint. Finally, the GUI provides a security button which can disable any type of control running in a joint in case of something goes wrong and the joint must be stopped.
All these commands are packaged in individual ROS-messages with format SingleCommandMessage.msg. This format is compound by:

```
int32      number_joints  # Number of joints affected by this command message
int32[]    message_id     # CAN identificator of joints affected by this command
int32      type           # Type of message to be interpreted by the joint(s)
float32[]  data           # Data sent to the joint(s)
```

This format uses arrays of 6 cells because some commands can be applied to all joints at the same time, for example, the "calibrate all position sensor" command. This command causes calibration of the zero reference position in all position sensors in all joints.

Force sensitive resistors will vary its resistance depending on how much pressure is being applied to the sensing area. The harder the force, the lower the resistance. When no pressure is being applied to the FSR its resistance will be larger than 1MΩ.

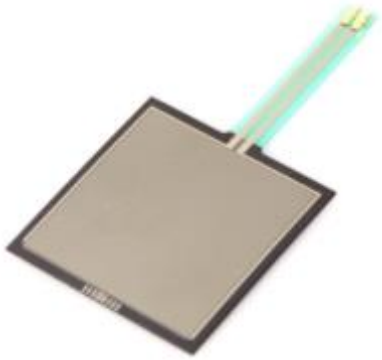


Figure 16. Force resistive resistor.

Four of these sensors are located in the exoskeleton's feet. One in the right heel and one in the forefoot. Same occurs for the left foot. Each FSR is used in circuit as the following:

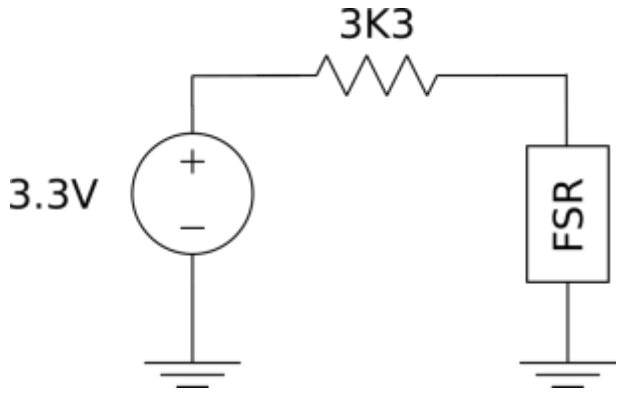


Figure 17. Electronic circuit used to sense the force applied on an FSR.

So, measuring the voltage across the FSR indicates if a force is being applied in that FSR. The harder the force applied in the FSR, the lower the resistance of that FSR, the lower the voltage measured across that FSR. This measured output is digitalized using an embedded 12bits-ADC of the BeagleBone Black board (The BeagleBone Black board has 6 12bits-ADC). If the measured value in one FSR is less than the threshold associated with that FSR, then a force is being applied to the FSR.

In the FSRS tab the GUI allows to enter individual thresholds. These thresholds are used by the node in charge of reading the FSRs in order to determine if a force is being applied over any pressure sensor. By setting individual pressure thresholds the user can configure dynamically the behavior of the algorithms to the intrinsic characteristics of the patient's gait.

The FeetFSRSThresholdMessage.msg format used for sending individual threshold to the node in charge of reading the FSRs is:

```
int32 left_toe_threshold    # No timestamp is needed, so no Header is needed here
int32 left_heel_threshold
int32 right_toe_threshold
int32 right_heel_threshold
```

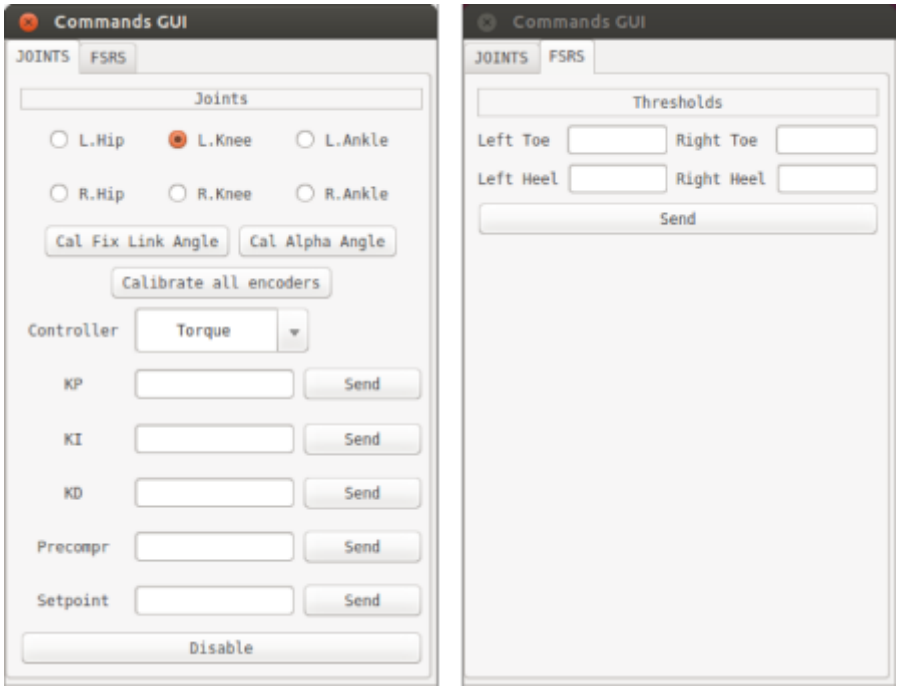


Figure 18. Commands GUI used to manage the joint driver and FSRs.

The *aux_var_gui_node* is subscribed to the *tx_topic* and associated with the *AUX VAR GUI*. In this GUI the user is provided with 4 auxiliary variables that are used for tuning different aspects of the algorithms used in the exoskeleton. These variables have been used for debugging purposes during the developing/testing processes of the algorithms implemented in the exoskeleton. Without this GUI if the developers want to change some aspect of an algorithm they have to edit the source code and compile it again. This GUI simplifies this process, getting rid of the necessity of editing the algorithm's source code.

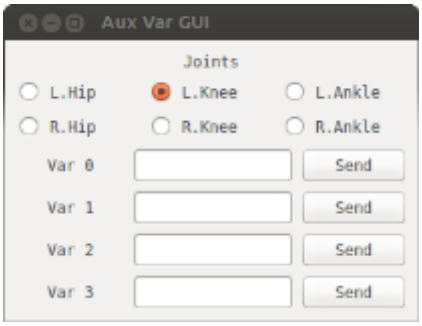


Figure 19. GUI used to configure dynamically several control algorithms.

FSRs node

This node is used for reading the FSRs located in the exoskeleton's feet. Each reading is compared with its associated threshold in order to determine if a force is being applied in any FSR. Analyzing the two FSRs of each foot this node can determine the phase of the gait in that foot.

- If there is a force applied in the heel but there isn't a force applied in the forefoot:
  Heel-strike (phase with associated number 1)
- If there is a force applied in the heel and there is a force applied in the forefoot
  Flat-foot (phase with associated number 2)
- If there isn't a force applied in the heel and there is a force applied in the forefoot
  Toe-off (phase with associated number 3)
- If there isn't a force applied in the heel and there isn't a force applied in the forefoot
  Foot on the air (phase with associated number 4)

This node publishes the readings of each FSR and the gait phase of each foot in a topic called *fsr_topic* within a ROS-Message with format FSRSMessage.msg:

```
Header  header
int32[] fsrs                    # 4 cells for 4 readings
```

```
int32    left_segmentation  # gait phase in the left foot. From 1 to 4
int32    right_segmentation # gait phase in the right foot. From 1 to 4
```

Moreover, this node publishes the same information in a second message with format *SingleCommandMessage* under the topic *tx_topic*. This is because this information should be written into the CAN BUS in order to be used by some algorithms which are out of the ROS-Network.

TX node

This node is in charge of writing data into de CAN BUS and it's located in the BeagleBone Black board. If any node wants send data through the CAN BUS it must send a ROS-Message with format *SimpleCommandMessage* under the topic *tx_topic*. The TX node is subscribed to the *tx_topic* so all the ROS-Messages associated with this topic arrive to this node. Then this node only writes this data into the CAN BUS. With this schema two main problems are solved. The first one is that nodes that are not running in the BeagleBone Black board, and need to write data in the CAN BUS, but don't have an accessible CAN interface, can do it, sending data to this intermediary node which then puts the data in the CAN BUS. An example of this situation are the GUI nodes, which typically aren't located in BeagleBone Black board but in a Linux PC. The second problem solved with this schema is that there aren't several nodes trying to access to the same CAN interface in the BeagleBone Black board at the same time, so it isn't necessary to synchronize the accesses to the CAN BUS through resource sharing techniques, which are complex to implement and error prone. This node doesn't constitute a bottleneck since the data sending rate isn't high enough to collapse the reception buffers in the TX node and its execution frequency. The CAN messages sent by this node are standard CAN messages of 6 bytes.

Finally, a graph of how ROS-nodes are connected within the ROS-Network though messages associated to ROS-topics is shown in the next figure.

Figure 20. Relation among ROS-Nodes running in the BeagleBone Black board and the display-PC.

BIBLIOGRAPHY

[1] BioMot description in the Neural Rehabilitation Group website.
http://neuralrehabilitation.org/projects/bioMot/research.html.

[2] Introduction to the Controller Area Network (CAN).
Texas instruments, 2008.

[3] Introduction to ROS.
http://wiki.ros.org/ROS/Introduction.

[4] Robots using ROS.
http://wiki.ros.org/Robots.

[5] Official OS releases for the BeagleBone Black board.
http://beagleboard.org/latest-images.

[6] BeagleBone Black board OS images setup.
http://elinux.org/BeagleBoardUbuntu.

[7] ROS distribution page.
http://wiki.ros.org/indigo/Installation/UbuntuARM.

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

[9] Enable CAN bus on the BeagleBone Black.
http://www.embedded-things.com/bbb/enable-ca nbus-on-the-beaglebone-black/.

[10] CAN interface testing on BeagleBone Black board. Embedded Division, Accel Frontline Ltd, 2013.

[11] Supported WiFi adapters for BeagleBone Black board.
ww w.elinux.org/Beagleboard:BeagleBoneBlack_WIFI_Adapters.

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