Microcontrolled and networked based mobile robot electronic system

Microcontrolled and networked based mobile robot electronic system

Acknowledgements

Special regards own to all persons who made this project possible:

The director of the European Centre for Mechatronics: Prof. Dr.-Ing. P. Drews.

Special thanks to project supervisor Prof. Dr.-Ing. Günther Starke, who always believed on my capacity to fulfil the work.

Special thanks to project supervisor Dipl.-Ing. Christoph Dreyer, for his mentoring, guidance and kindness, in all good and difficult moments of this work.

Special thanks to the supervisor Prof. Dr. Fernando Ribeiro who always cleared doubts and uncertainties even from abroad. His advices were also essential to show a path and his availability was very kind.

Abstract

Mobility is a key issue in robotics and a challenging subject for any research and development project. It combines cognitive capabilities focused on sensorial input and human interaction with intelligent control of the drive systems and requires real mechatronics engineering solutions to enable robust and reliable operation.

This thesis describes the electronic system used on a mobile tank-robot and the network and micro-controlled based interface system to drive it.

A new control architecture is presented on this thesis. The purpose of the presented system architecture is to develop a remote control platform to replace and increase the processing capacity of an old mobile platform as well as to obtain a modular control-system inheriting flexibility and easy to upgrade platform.

A two wire bus protocol called *I2C bus* was implemented in the system to reduce the hardware upgrade issues as well as software involved, giving the system expansible capabilities.
This protocol intends to reduce system circuit complexity and makes a friendly environment where multiple devices are intended to be added and controlled.

QT libraries were widely used to develop desktop processing and to make it a platform independent system; it means that the software platform can be use in a *Windows*, *Windows CE*, *MAC*, *LINUX* or an *embedded Linux* environment.

The developed system architecture is based in a multiprocessor platform; it is composed by two computers interconnected wirelessly by a *client/server* model and by a microcontroller to perform critical tasks and to interconnect *I2C bus* communication as well as to interface the motor drivers, encoders and display.

Resulting architecture combines *I2C (Inter Integrated Circuit) bus* architecture, *Client/Server*, *Qt libraries*, *atmega16 microcontroller* technology, *MD03 drivers* and it was built from scratch (except for the mechanical structure) and all the problems that occurred during the development phase were sorted out giving a solid and confident characteristic to this robot.

A final and informal presentation was carried out at the APS – European Center of Mechatronics [W1] and as main conclusions it can be said that the robot showed high stability with both fast and smooth control. The commands sent by the client are correctly interpreted and technical problems (as loss of internet connection) passed successfully.

Key Words – Tank, Mobile Robot, Client/Server, Qt libraries, C++ desktop programming, Atmega16 microcontroller, Encoders, MD03 motor drivers, I2C (Inter Integrated Circuit) bus.

Table of contents

1. Introduction and Goals

Nowadays mobile robots are desired at several services to perform man tasks with increased safety in any kind of duties.

Mobility is a key issue in robotics and a challenging subject for any research and development project. It combines cognitive capabilities focused on sensorial input and human interaction with intelligent control of the drive systems and requires real mechatronics engineering solutions to enable robust and reliable operation.

In the European Centre for Mechatronics [W1] mobile robots have been under research for years. A mobile platform with semi-autonomous functionality has been developed years ago. It has been used as a demonstration and test platform to study mobility and cognitive control.

In the University of Minho [W3] and RWTH University [W2] that kind of research is also important and the industry support gives a solid development sign to the institutions.

As the on board vehicle control system capacity has reached its limit, a new system architecture with higher powerful controller elements was planned to replace the old one.

In this context the key objective of the project was to design and develop a new embedded system able to control the mobile platform.

To meet this goal a number of tasks were considered:

- Introduction to vehicle system architecture and drives functionality
- Introduction to microcontroller technology
- Development of a microcontroller hardware platform
- Development of a concept to control the drives and to enable driving modes
- Development of controller software
- Development of a user interface for remote control
- Integration and functionality tests
- Documentation

The practical work was carried out on the labs of the APS - European Centre for Mechatronics.

The duration of the project was 5 months.

Technical assistance was provided by the APS project engineer Christoph Dreyer.

Supervisor in Germany is Prof. Günther Starke, Head of Research at the APS.

Technical assistance and Supervisor in Portugal was provided by Prof. Dr. Fernando Ribeiro.

1.1. Thesis structure

This report is organized on the following order:

- State of the Art
- Robot presented and system block diagram implemented
- D.C. Motors theoretical introduction, PWM signals, Gear heads, Motors Drivers and Encoders, Power regulators, Microcontroller programmers and I2C protocol

- Microcontroller unit details
- The desktop programming (Server/Client) and (Server/Microcontroller)
- The schematics, Proto-board, Strip-board, PCB and hardware connections
- A chapter about software and hardware considerations
- Conclusions

2. State of the Art

Robots are more and more available for research and commercial applications. This robot is under research phase of development, as well as the control structure of the system.

The market shows some robots solutions that are either autonomous or remote-controlled being the first more common.
Remote-controlled robots can be found widely in military robots but service or tactical robots are usually autonomous.
Some chassis with only motors and without any control can also be found but comparing them to this one can lay to misunderstood since this robot has also software to the client and server and hardware like the microcontroller system and sensors.

Since this robot was developed under 5 month and is still under development, it is not fair to make a direct comparison with other already finish robots. Even through the state of the art compares this work with most relevant similar projects.

Usually, preceding the autonomous development, a solid mechanic structure and control system is designed as well as low level programming to drive the motors and to make use of sensors. In this robot system those things were tested and the *I2C* communication support was included as well of serial and *TCP/IP.* The robot is remote control and has feedback regards to temperature, current, speed and position. Further developments on this robot should consider the application of autonomous algorithms.

The proposed robot system has not a specific application. Comparisons will also be carried out with military devices, due to its powerful motor power, size and mechanical parts and like this robot military robot are not autonomous.

This work was developed in 5 months, and it included full research, testing and hardware and software built up. The products presented in the state of the art, took a few years to developed, were created by large engineering teams, reason why probably these products are often provided with more sensors and automatism controls as well as they are already on the market.

The existing robots are usually expensive and lack autonomy, reason why several approaches of robotic platforms are being devolved in several companies, according to specific requirements like size, cost, behaviours, easy user interface, etc.

Old robot platforms were designed to a specific hardware, and sensor upgrading would imply a hard work in re-programming and in physical attachments. The system proposed is applied specifically to this robot, but can easily be connected to other mobile systems.

The proposed robot uses *I2C* to avoid complexity at hardware and software level. The robot can easily grow with new hardware with only 2 wires, for communications with the controller, which decreases the complexity of adding new components. At the same time, the platform independent server computer can easily be equipped with USB components with "*plug and play"* capabilities.

2.1. iRobot Create™ Programmable Robot

Figure 1 shows the iRobot,

Figure 1 – iRobot [W24]

According to [W24], *iRobot* is, since at least 15 years, a global leader in robotics. It has platforms for technological development and its software is network based and developers can program the robot behaviour.

Like the robot system proposed in this thesis, the low-level software infrastructure is complete and in the future it is necessary to program high level behaviours. Loggers and debugging tools are also provided in both robots but the *iRobot* itself (figure 1) is a small robot with upgrade capabilities as shown in figure 2. These platforms are normally used as the base system, and then new sensors are attached and software developed so that the robot is adapted to perform the required task.

Like this proposed method, the *iRobot* Command Module also used the same family of microcontroller, the ATMega168.

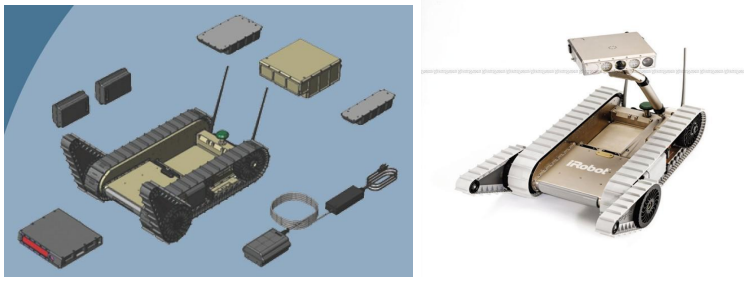


Figure 2 – iRobot pack [W24]

Figure 3 – the *iRobot Packbot* in Iraq [W25]

One of the applications is the *iRobot Packbot* used in Iraq, as shown in figure 3.

“April 24, 2007 - The remarkable success of robots in Iraq and Afghanistan is now well documented. UAVs have proven invaluable at every level and robotic ground systems, primarily *iRobot’s Packbot*, have performed tens of thousands of missions and saved countless lives from the dreaded Improvised Explosive Device (IED).” [W25]

2.2. MobileRobots's P-series

Pioneers, PeopleBots, PowerBots and PatrolBots are physically different robots, from the *MobileRobots's P-series*, but with the same standard core architecture. [W28]

“Since 1995, Mobile Robots platforms have contained all of the basic components for sensing and navigation in a real-world environment, including battery power, motors drive and wheels, position / speed encoders, and integrated sensors and accessories. They are all managed via an onboard microcontroller and mobile-robot server software.” [W28]

Differently than the proposed platform system, these robots are called “Embedding Linux in a Mobile Robot”.

Figure 4 shows the Pioneer 2-DX and block diagram, which has similarities in the block diagram even though in a primitive way.

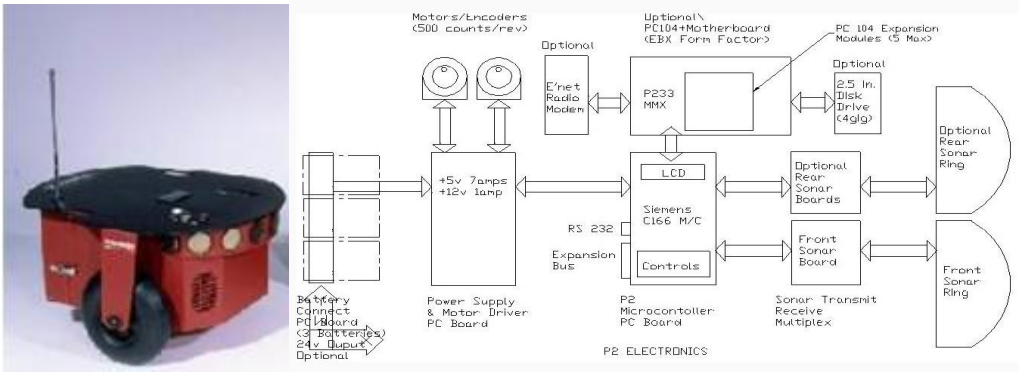


Figure 4 – *ActivMedia* Mobile Robot Pioneer 2-DX [W28]

Figure 5 shows the Pioneer 3-AT that is an evolution of the Pioneer 2-DX and it is rated at a 6,995$ cost.

Figure 5 – *ActivMedia* Mobile Robot - PIONEER 3-AT [W28]

“Pioneer 3AT is a mobile robotic skid-steer base with 4 drive wheels, microcontroller, motors, encoders, 1 battery, std. Pioneer software & no sonar, OS, ARIA, ARNetworking, MobileEyes, MobileSim, Mapper Basic & manuals; charger & ethernet or laptop connector not included.” [W28]

The bare P3-AT base with included ARIA software has the ability to:
Drive controlled by keys or joystick, paths plan with gradient navigation, display a map of its sonar and/or laser readings, localize using sonar (with optional laser upgrade), communicate sensor & control information relating sonar, motor encoder, motor controls, user I/O, and battery charge data, test activities quickly with ARIA API from C++ programs, simulate behaviors offline with the simulator that accompanies each development environment. [W28]

Like the proposed robot, this one can communicate with a client computer but the microcontroller used is an ARCOS instead of the widely known AVR *Atmega*.

Technical specifications:
“The rugged P3-AT 50cm x 49cm x 26cm aluminum body with 21.5cm diameter, drive wheels loves to run outdoors. The four motors use 38.3:1 gear ratios and contain 100-pulse encoders. This skid-steer platform is holonomic and can

rotate in place moving both wheels, or it can move wheels on one side only to form a circle of 40cm radius.” [W28]

“A small proprietary ARCOS transfers sonar readings, motor encoder information and other I/O via packets to the PC client and returns control commands. Users can run the robot from the client or design their own programs under RedHat Linux with Motif or under WIN32 using their favorite C/C++ compiler. Our robotics development environments supply library functions to handle navigation, path planning and many other robotic tasks.” [W28]

2.3. SWORDS

Figure shows the Swords, one of the tested applications of remote controlled robots.

Figure 6 – SWORDS Robot [W26]

Swords, aka Special Weapons Observation Remote Direct-Action System, is a military robot system developed to operate in a combat scene, and it was finish in January 2006.

“The diminutive remote-controlled US$230,000 SWORDS machine shares the same base as the Explosive Ordnance Disposal (EOD) Talon robots which have been deployed in Bosnia, Afghanistan and Iraq. Unlike many of it is flying robotic (UAV) brethren, the weaponised Talon is not autonomous, being under the direct control of a soldier watching from up to a mile away through an array of cameras which can include both night and thermal vision.” [W22]

It makes use of AC power or lithium batteries and is controlled by two joysticks, one for the robot platform and the other for the weapon. To provide security over the communications a 40 bit encryption is implemented.
Up to five firing systems can be dealt with this system. [W22]

2.4. Talon robot soldiers shipped to Iraq

Figure 7 shows the *Talon Robot*

Figure 7 – Talon Robot [W27]

US Army launch to Iraq and Afghanistan war one hundred of TALON robots by the end of 2004. Those robots are equipped with off-the-shelf chemical, gas, temperature, and radiation sensors.
TALON robots can be used in missions to clearing live grenades to neutralizing mines in shallow water, and can be adapted for small mobile weapon systems for military purposes. [W22]

"The TALON is a general-purpose modular robot with a versatile 64-inch pincer arm. It is controlled through RF or a fibre optic link from an attaché-sized operator control unit (OCU) or wearable OCU. On the ground the TALON can reach a vehicle speed up to 6.6 km/h and last for four-hours run time. Mounted on the TALON robot are:

- Smiths APD 2000 advanced portable chemical agent detector.
- Draeger Multiwarn II gas detector.
- Raytek Raynger MX4+ temperature sensor.
- Thermo FH40GL radiation detector." [W22]

2.5. Termibot

Termibot, as shown in figure 8, is a remote-controlled robot that makes use of thermal imaging to detect and eradicate termites.

Figure 8 – *Termibot* [W30]

Termibot was release in May 2007, to reach places where human pest controllers cannot go. [W30]

When a telltale heat or moisture signature is detected, Termicam breaks termite nests open to confirm the infestation, then pumps pest control chemicals directly into the source. It is an ingenious non-invasive pest control device - but its appeal is not limited to exterminators.

"It is basically a remote controlled robot that can fit into confined spaces," says Rice, "it carries a video camera and lights so the operator can see where it is going and steer it around obstacles. It can go over on a fairly good angle and right itself if necessary." When the thermal or moisture signature of a termite hotspot is detected on one of the device's two LCD screens, the Termibot uses a probe to break open the termite nest, exposing and video recording the insects as they scuttle to repair the breach. The operator is then able to inject pest control poisons directly into the termite colony, an effective eradication

leaving minimal toxic chemicals around the area in comparison to spraying." [W30]

"It is currently controlled via a long cable," Rice tells us, "but we'll have it fully remote once we've finished further testing. We're currently field testing it under houses, it is available to all our franchises but because we're busy expanding and franchising around the world, it won't be ready to go to market until later in the year." [W30]

"Rice says he's already had several enquiries from outside the pest control industry; the remote control thermal camera will be of interest to anyone who needs to use thermal imaging in confined spaces. Once client in Brunei is looking at having it lightly modified to act as an air conditioning duct cleaner, and Rice sees applications for the *Termibot* in sewage and water tunnel investigations, electrical equipment testing, military and bomb disposal applications, and even search and rescue to detect the heat signatures of people trapped under snow or rubble." [W30]

2.6. 914 PC-Bot

Figure 9 – 914 PC-Bot [W29]

Figure 9 shows the 914PC-Bot robot, which costs approximately $5,000.

The 914 can serve as a "networked, mobile sensor platform for RFID readers, hazmat detectors, and access management devices". The company suggests "Now you can move the sensor instead the asset”.

The 914 stands 21-inches tall, and weighs about 55 pounds (25kg). It has a two-wheel drive train with two "caster ball" wheels, each powered by a DC stepper motor. Other sensors include a camera in the head unit, and a sensor array comprising eight IR sensors is presumably used for obstacle avoidance.
The 914 is powered by twin 12-volt lead-acid batteries, and comes with a charger.

“Since the 914 is really just a standard PC trapped in a robot's body, it can run any standard PC operating system. *WhiteBox* Robotics supports Linux, as well as Microsoft's Robotics Studio. When used with Linux, the company also appears to support the open source Player/Stage robot and sensor programming library”. [W29]

3. Presentation and Implementation architecture

This chapter will present the robot and its block diagram to provide an understanding of the robot general structure system architecture and the robot components.

3.1. The robot

Figure 10 shows the robot: it is a solid multi terrain tank style robot and it can be seen the two chosen motor drivers, the DC converter, the microcontroller display, the batteries that provide power to the microcontroller system as well as to the motors. The motors and the encoders can also be seen on this picture.

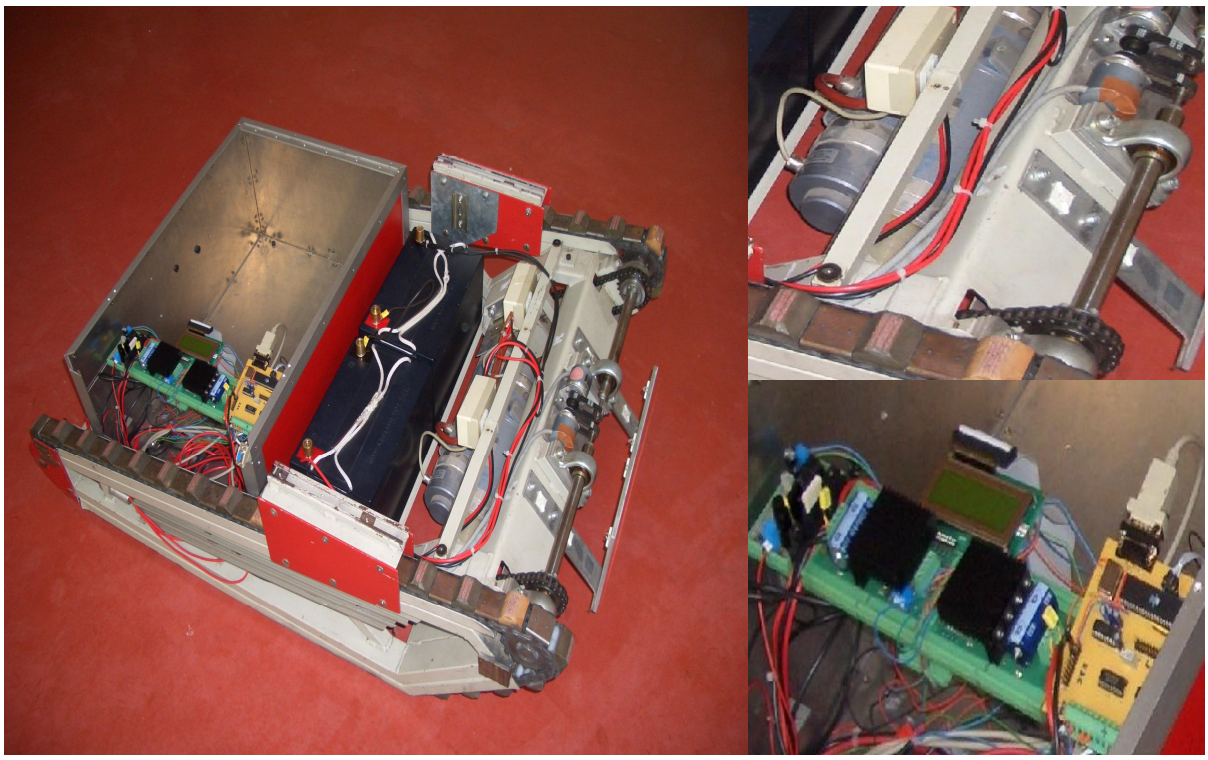


Figure 10 – The Robot

The tank is a Remotetly Operated Vehicle (ROV) and it is in the process of being built up, updating some of the hardware/software.

3.2. Block Diagram

The block diagram is shown in figure 11, and it shows the way components are connected.

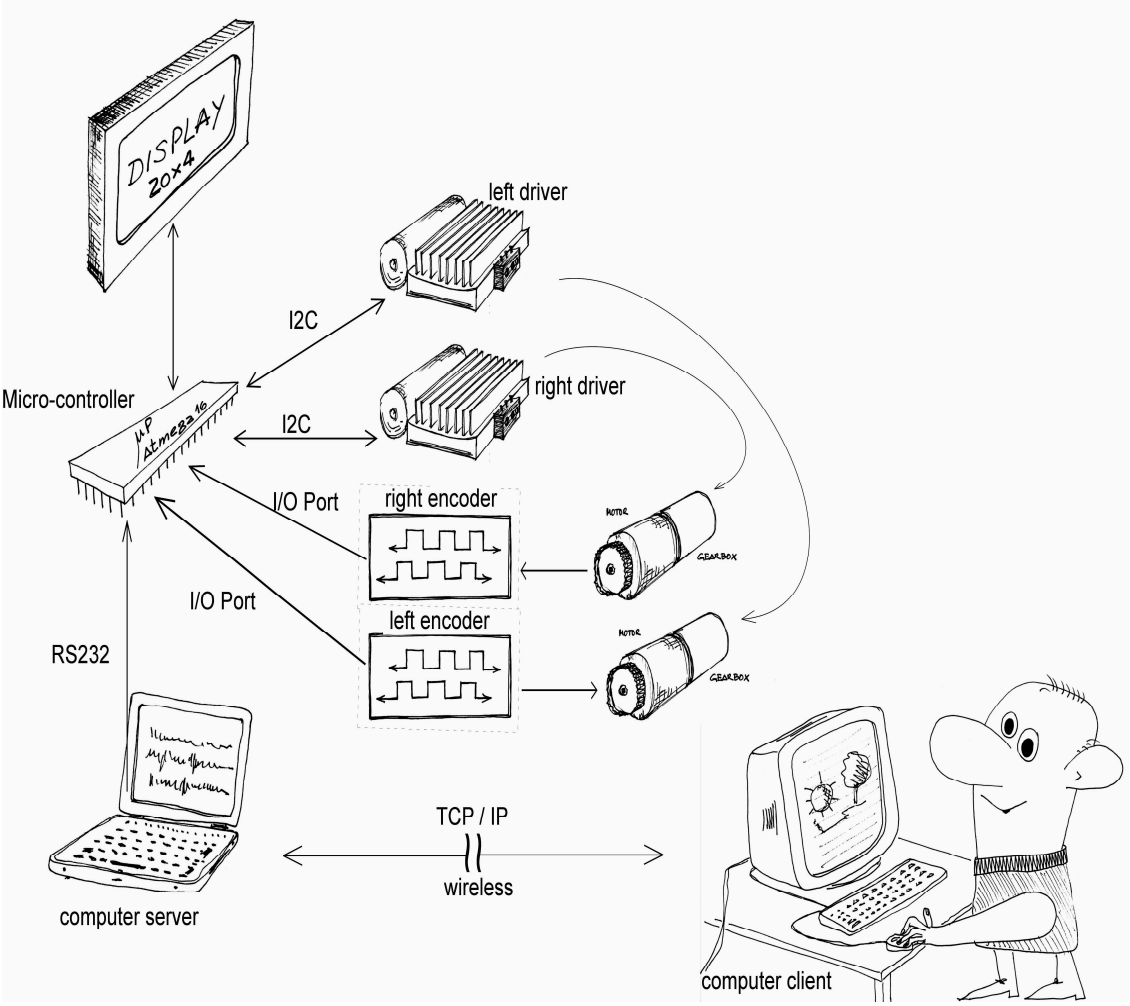


Figure 11 – Block Diagram

The system is multiprocessor based, because it uses two computers (one *Client* and one *Server)* and one microcontroller. The system is also composed by the

drivers, encoders, display. All this hardware is linked either by I2C or I/O Port, or RS232.
Client and *Server* based architecture of this robot inherits a wireless network interconnection to entitle the robot control without any wire-link to the exterior.

Tasks that are likely to be critical to performance like safety are implemented in the microcontroller. The system robustness is therefore increased comparing to robot systems based only in computers [B2].

The a*tmega16* microcontroller is the core of the robot hardware: the display connected to it gives feedback of the microcontroller state and has an important duty about diagnose, repair and robot maintenance. For example, if some hardware failure occurs like a motor drive being disconnected from the *I2C* bus, the display will show a message saying “Motor Left: Error, Motor Right: Error”.
Status monitoring of the system parameters during an operation cycle can be also achieved with the display.
The *Server* connects to the *Atmega16* microcontroller over *RS232* protocol.

Atmega16 microcontroller code was written in C language, and the Server and Client computer code was written in C++ language.

The connection between the *DC regulator*, the *Atmega16* microcontroller, the encoders and the motor driver power is not represented on the figure, to avoid confusing mesh of wires on the block diagram.

4. Detailed working method

This chapter presents the electronic and mechanical robot system components as well as the theory used to deal with them.

4.1. D.C. Motor

D.C. motors are used rather than other motor types because they are smaller and have high efficiency. Furthermore, the D.C. motor has a very high start-up torque and easily absorb sudden changes in load [W22].

DC motors are also simpler to control; even though they are heavier and less efficient than induction motors.

The use of this type of motors is also efficient in this case because the robot is powered by batteries which provide the same type of current that these motors need, and therefore power converters are avoided and consequent loss of efficiency is spare.

4.1.1 Composition of a D.C. motor

Figure 12 shows a D.C. motor insides.

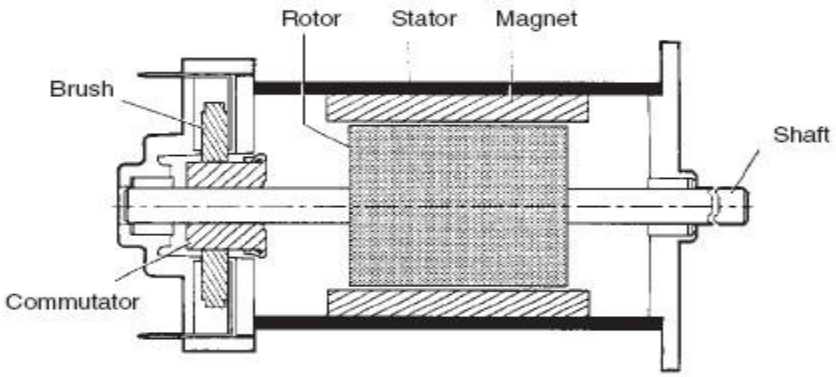


Figure 12 – D.C. motor [W22].

The stator has the motor cover and the magnets that create the stator magnetic field [W22].

The rotor is mainly formed by a metal carcass carrying coils and the commutator that selects the coil through which the electric current flows. The commutator has the duty of transforming the induced altering tension into a continuous tension [W22].

The motor used by this robot is like the one in the figure [W22] and has an additional gearbox at the shaft.

4.1.2 Principle of operation

The principle of operation of a D.C. motor is based on rules of electromagnetic attraction [W22].

The rotor is energised to act as an electromagnet with the polarity given by the current flow direction [W22].

The figure 13 shows that D.C. motors have two magnets fields, one of them is fixed (stator) and the other one is physically movable [W22].

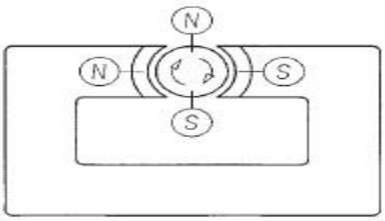


Figure 13 – Principle of operation [W22]

A torque is created to make the poles of the rotor align to the poles of the stator. This attraction and repulsion between the magnetic fields make the rotor spin, which is the movable part, and then the brushes are constantly breaking and making contact with the commutator [W22].

The maximum torque is achieved when the axis between the poles of the stator is perpendicular of the poles of the rotor [W22].

“The rotor coils are then energised and de-energised in such a way that as the rotor turns, the axis of a new pole of the rotor is always perpendicular to that of the stator. Because of the way the commutator is arranged, the rotor is in constant motion, no matter what its position. Fluctuation of the resultant torque is reduced by increasing the number of commutator segments, thereby giving smoother rotation.”

[W22]

To change the spinning direction of the motor, one of the magnetic fields must exchange, since the stator has permanent magnets, the way is to invert the rotor magnetic field. This can easily be accomplished by changing the polarity of the tension applied to the rotor coils, the direction of the current will, this way, be reversed as well as the rotation direction [W22].

4.1.3 Robot motors characteristics

The robot has two motors provided from the manufacture ENGEL, the specific series is GNM5480E and the motors are typed “Permanent Magnets, Direct

Current", they are coupled with gear-heads and main characteristics can be seen in table 1.
Full table of characteristics can be found in appendix "Motor Specifications" as well as the dimensions.

Nominal voltage	UN	24	Volt
Nominal output power	P_2	250	W
Efficiency	η max	85	%
No-load speed	no	3,267	rpm
No-load current	Io	1,435	mA
Speed constant	k_n	137	rpm/V
Nominal speed		3,000	rpm
Motor operating temperature range		–20 to 100	°C

Table 1 – Robot motor characteristics

4.2. PWM Signal

A powerful and common method to control D.C. Motors over a microcontroller is by using Pulse Width Modulation (PWM) signal [B1].

PWM signal consists of a square wave and by varying its duty cycle will provide a proportional variation mean power applied to the D.C. motor [B1].

Figure 14 shows respectively a 10%, 50% and 90% of duty cycle.

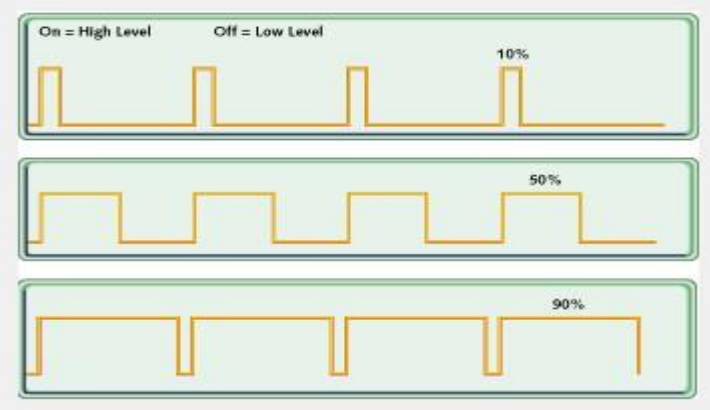


Figure 14 – PWM signals of varying duty cycles [B1].

4.3. Gearheads

When using a motor, when high speed is not as important as torque, it is usual to attach a gearhead to the motor shaft [W21].

With a gearbox the motor binary can increase/decrease and the startup effort is soften/harder according to the number of teeth (ratio).

To avoid degradation and damage of the gear/pinion several gears can be used instead of only one. This way, the forces are distributed and the material of which they are built of can be “soften”. The lubrication must be taken into account, according to the number of wheels. [W21]

To make a gear head description it is necessary to say that it has satellite gears and an annular gear. The annular gear usually forms the gear head case on the outside and has gear teeth cut in the inside diameter. Satellite gears are carrier plates with pins that fit the inside diameters of the satellite gears. Figure 15 illustrates a single-stage planetary gear head having three satellite gears. [W21]

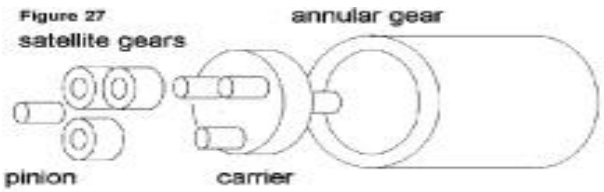


Figure 15 – Gear heads [W21]

The gear shaft is attached directly to the motor shaft and then a bearing couples the driven load.

The gearbox is selected depending on the maximum required torque and the duty cycle [W21].

The direction of rotation of this gear head output shaft is the same as the input one. One of the disadvantages of this gear head is the high noise but, besides

that, they are relatively smaller than other types found in the market to the same operation conditions. [W21]

It increases also loss of efficiency and weight to the system, which could be a problem in a platform feed by batteries.

The lengthening of the annular gear/case and multiple stages stacking can allow high gear ratios. [W21]

The gear heads series are “G6.1” and they are of planetary type; it is rated at a 16.8:1 ratio and 70% of efficiency at either clockwise or counter clockwise direction, the torque is 11Nm.

4.4. Drivers

The medium/high current motors of the robot must be able to run in both directions and in variable speed. An H-bridge should be projected, to reduce the *time to market*, or a solid driver should be chosen.

Because the actual markets provide a reasonable range of driver solutions to different applications and are price competitive, the conclusion is that the best solution is to acquire one. It saves time because that kind of project, involving high currents, from the practical point of view would increase the number of difficulties which would not provide enough time to take the project this far.

There were three possible drivers to choose from:

One of them is shown in figure 16, the RN-VNH2:

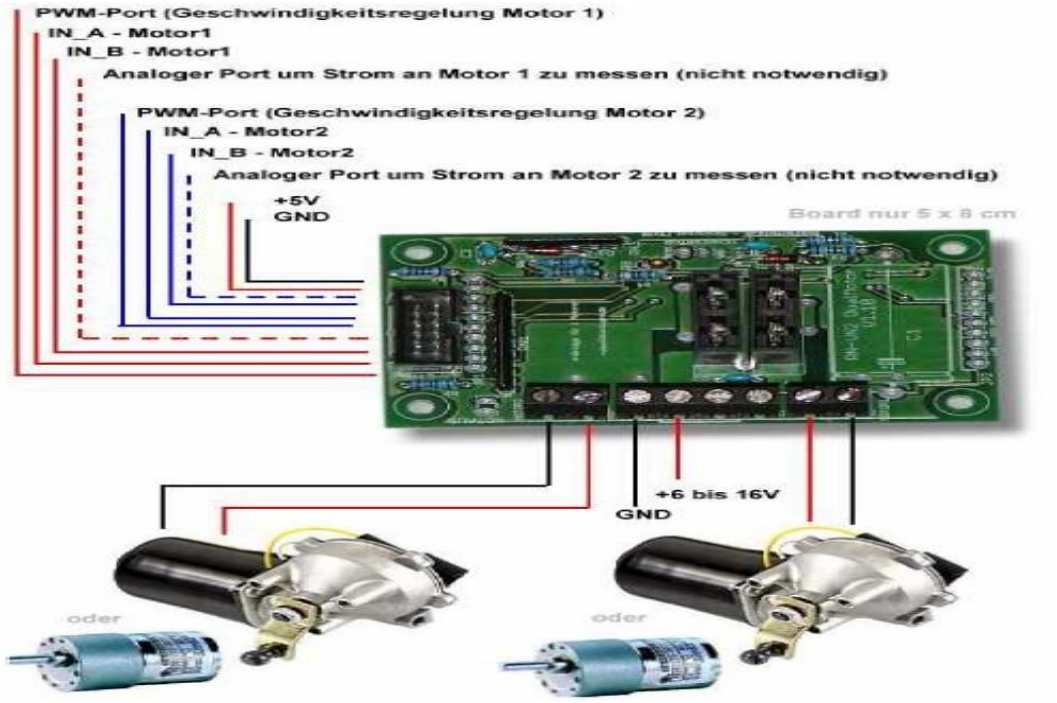


Figure 16 – RN-VNH2 Driver [*picture provided by the manufacturer datasheet]*

This driver [16] does not provide the necessary 250W, the motors require. But it was used for testing and trial, using a smaller motor.

The *atmega16* was programmed to provide the *PWM* signal [Chappter 4.2.] to the driver. The direction, speed and acceleration ramps were created and tests about controlling those values with computers were made as well, those tests involve the use of the *USART (Universal Synchronous Asynchronous Receiver Transmitter)* to perform communication between *atmega16* and one *computer*.

The work previously described was first considered within the project, but then during the research part about drivers and robot controls technologies, a protocol called *I2C* came up.
The protocol will be explained later on [chapter 4.9.], but the choices about the driver were now about two divergent drivers: one from *Sabertooth [*17*]* or the *MD03.*

The "*Sabertooth 2x10*" is a driver capable of driving the robot motors with the software that was developed and tested with RN-VNH2 Driver.

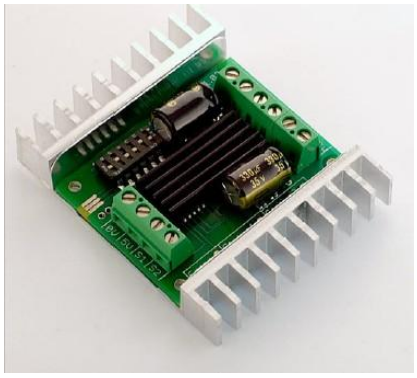


Figure 17 – *Sabertooth Driver [W12]*

The *MD03* is the driver shown at figure 18 and it is a driver capable of communicating through *I2C* and up to eight *MD03* modules can be connected (switch selectable addresses) to a system *[W23].*

The MD03 was chosen because the I2C capabilities matched the project intention of a modular system design and has the power capabilities that the system requires.

Figure 18 – *MD03 Driver [W23]*

In this robot the addresses were chosen with no specific criteria and they can be seen in table 2.

	Motor Right	Motor Left
Adresses	0xB0	0xB2

Table 2 – MD03 addresses of left and right motor

“I2C communication protocol with the *MD03* module is the same as popular *EEPROM's* such as the *24C04*.” *[W23]*

The MD03 has 8 registers numbered 0 to 7.
The reading operation of the registers follows this order:

1. send a start bit
2. send the module address with the read/write bit low
3. send the register number to be read
4. send a repeated start
5. send the module address again with the read/write bit high

[W23]

4.5. Encoder

Nowadays to make use of modern motion control techniques, values representing locations of the robot movable parts are needed. To perform that kind of task, the spin of each motor of this robot must be log.

Devices that provide knowledge of where the robot is make possible to synchronize movements of the robot and, at same time, can give feedback to the control system in order to act if some kind of behavior is not reached.

This robot has two incremental encoders that are used to precise how much is each motor running and they provide the speed of each motor after some computation performed on the microcontroller.

The working method of an incremental encoder is usually based on transmitting/reception perception:

One disk with holes is in the middle of the transmitter and the receiver.
The transmitter has a stationary light source and the receiver has two stationary light detectors.

The disk is mounted at the shaft. As the disk rotates, the holes in it make the receivers to get the light each time the hole is aligned with the light transmitter. Figure 19 can illustrate this process:

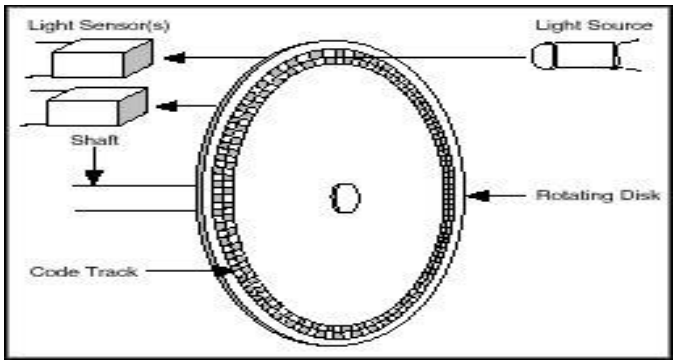


Figure 19 – Encoders signals [W20]

The outputs of this system are two square wave signals representing the number of holes that are reached between the transmitter/receptor, typically one output is called channel A and the other one is channel B and an extra channel, usually called channel Z, is often included to detect the “once per revolution index mark”.

A visual perspective of the output signals above described can be seen in figure 20.

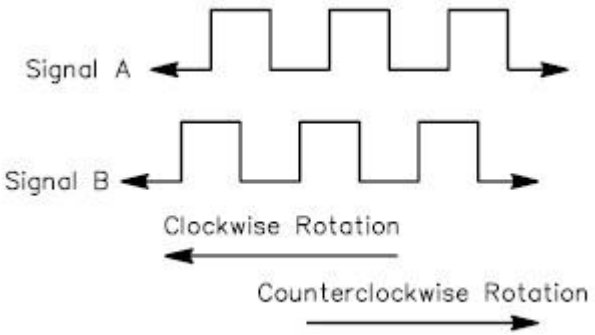


Figure 20 – Encoders signals [W19]

“The position of the two detectors is important. As one senses a change from dark to light, the other will not sense a change or transition. Because of this physical arrangement, two detectors give four transitions per division on the

disk and each transition occurs at a unique angular position on the shaft. By counting the transitions, it effectively multiplies the line count by four, hence the name quadrature (X4) multiplication.” [W18]

To sense the direction of rotation the encoders have two channels 90 electrical degrees out of phase. A rising edge of the square wave indicates one direction, and the falling edge of the square senses the other direction.

To get the direction, each encoder, as shown in figure 21, was also coupled to a D type flip-flop. With channel A as flip-flop clock input signal (*clk*) and channel B as the input data signal (*D*) is possible, using the combination of these two signals, to obtain the output of the flip-flop (*Q*) representing the direction of rotation. Output (*Q*) is connected to an input *pin* of *atmega16* so the microcontroller can correctly count pulses that are either to be increased (Output of the flip-flop is 0) either decreased (Output of the flip-flop is 1).

Combination:

Q → signal B low at signal A is rising edge

/Q → signal B high at signal A is rising edge

To make use of these sensors, a connection between channels A of each encoder was connected to an *external interrupt* of *atmega16* which is programmed to catch rising edges and make the digital count of the pulses.

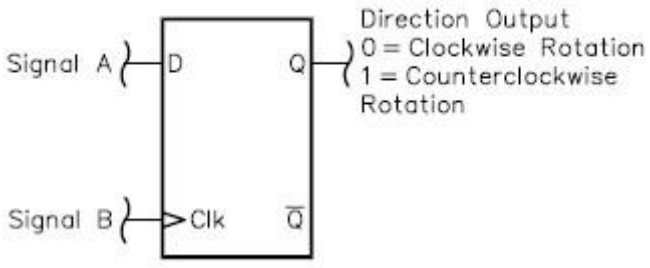


Figure 21 – Encoder Flip-Flop [W19]

Both encoders of the robot can be seen on figure 22.

Figure 22 – Robot encoders

4.6. Batteries

The batteries used in this robot are Lead Acid batteries. They are popular because they are easily available, rechargeable and inexpensive.

The problem is the heavy weight and large size but, in this robot that is not a huge problem because the tank is powerful and big enough to carry them. However, that can be a problem should the robot go on the market or suffer further improvements.

Another problem is the loose of charge even if they are not in use and high discharge rates will be translated in a short time battery life.

There exist three main types of lead acid batteries: Wet Cell, Gel Cell, and Absorbed Glass Mat (AGM). They are mainly distinguished by the price, degrade, and deep cycle needs.

The Gel Cell batteries were chosen to this project and they are best used in very deep cycle applications, even so the AGM batteries provide a greater life cycle. They do not need maintenance and do not flow out acid.

“80% of all battery failure is related to sulfating build-up. This build-up occurs when the sulfur molecules in the electrolyte (battery acid) become so deeply discharged that they begin to coat the battery's lead plates. The buildup will become so bad that the battery will die.” [W13]

It is important to know and have in mind some things about lead acid batteries this way preventing battery failure:

- The first point to remember is not to make a partial recharge of the batteries, and all charges should be integral accomplished to its full potential. [W13]
- A second point, and also a very important one, is to recharge them often because without being used for a long time these batteries will slowly discharge internally. [W13]

These robot batteries are serial connected to make a 24V power supply, that connection can be seen on Figure 23.

Battery type is *Exide* and they are rated as gel cells which are a maintenance-free motive power batteries technology as well as they are robust, safe and reliable Low self discharge is also achieved by those.

Figure 23 – Serial Connection between two *Exide* batteries

4.7. Step-Down Switching Regulator

A regulator was needed to convert the 24V from the batteries to a regulated 5V to feed the microcontroller, encoders and the other components.

The regulator component is a LT1076 that is rated at 2A. It is a monolithic bipolar switching regulator and requires only a few external parts for normal operation. It has built-in power switch, oscillator, control circuitry, and all current limit components.

The classic positive “buck” configuration was adopted and the switch output is specified to swing 40V below ground which is perfect to the 24V of the robot because it is in the middle of the rated range.

The schematic of the regulator can be seen on figure 24 and the board on figure 25.

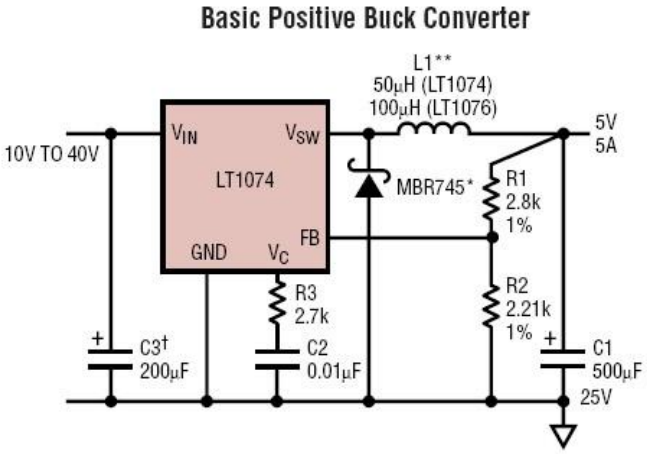


Figure 24 – Regulator Schematic [LT1074 datasheet]

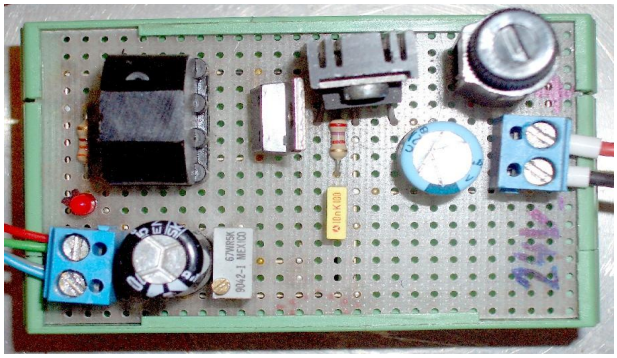


Figure 25 – Regulator Board

4.8. AVR Programmer

To flash the microcontroller with applications, a programmer is needed.
The microcontroller chosen for the project is part of the AVR family.

Although the microcontroller itself is described in chapter 5. , the programmer is here described.

The circuit presented on figure 26 and *PonyProg* flash program [W6] was chosen because of its low price and because it can be easily built by anyone.
The problem of this circuit is that it needs to be used together with *PonyProg* [W6] to enable flashing the microcontroller using *RS232,* although "*USB to RS232"* adapters often do not work or they are very slow (more than 10 minutes to program). To avoid *USB* adapter, the solution to a laptop could be a *PCMCIA* or a *PCI* adapter that natively emulate a *COM* port but a *PCMCIA* card was tested and even so it was very slow.

So, to flash the microcontroller with this programmer, a desktop computer with native *COM* port was used. This approach will not allow robot microcontroller remote programming and beside that JTAG and AVR Studio integrations are not possible.

In the future, to provide a fast remote microcontrollers programming the "*Atmel AVRISP MK2"* or "AVR Dragon" programmer would provide better results as

well as other advantages as the AVR Studio Integration, USB Serial In-System Programming and the JTAG Support (“Hardware debug” with “AVR Studio” in real time, which means that the instructions carried out with AVR Studio debugger can automatically be seen on the hardware).

The board can be seen on figure 27.

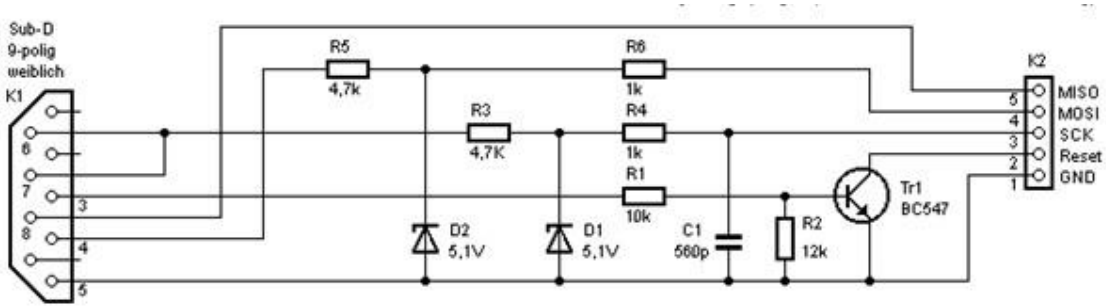


Figure 26 – “*SI-Prog”* Programmer Schematic

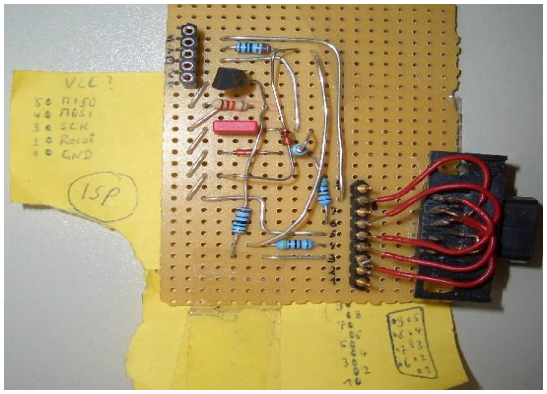


Figure 27 – Programmer Board

4.9. *I2C - TWI*

One of the robot problems is that, to provide them with more “intelligence” more and more sensors are added and that implies more cabling.

To minimize that, *I2C (Inter Integrated Circuit)* also known as *TWI (Two Wire Interface) [W15]* is the communication protocol chosen because it can easily link multiple devices together with only two wires each [W17] in a bus style.

I2C devices have a built-in addressing scheme to be distinguishable and avoid the need for chip select or arbitration logic which increases system simplicity as well as reduces budget in extra hardware such as multiplexers and logic chips.

Standard *I2C* devices operate up to 100Kbps but fast-mode devices can operate up to 3.4Mbps with the version 2.0 high speed mode.
Almost all available *I2C* devices can operate at speeds up to 400Kbps.

I2C provides good support for communication with various devices. On-board peripheral devices can be accessed intermittently; it is a simple, low-bandwidth, short-distance communication protocol.

Philips originally developed *I2C* for communication and due to patent concerns *Atmel* uses the name *TWI* (Two Wire Interface).

Several *I2C*-compatible devices are manufactured by several companies and can be found in embedded systems. Some example are *EEPROMS*, thermal sensors, and real-time clocks, video decoders and encoders, audio processors, displays, motor driver, etc.

Figure 28 shows a typical *I2C* interconnection system:

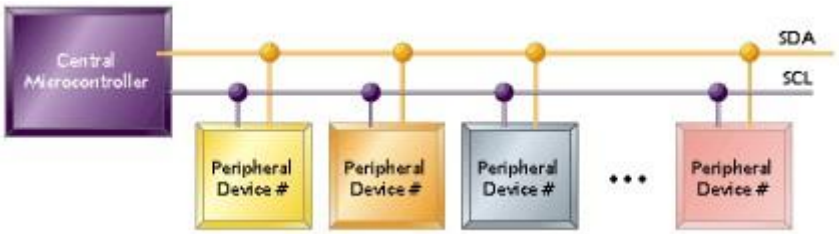


Figure 28 – I2C typical interconnection system [W14]

Figure 29 shows the specific interconnections with the I2C bus: the microcontroller is the master of the I2C Bus and both drivers are I2C slave devices.

The two resistors are called “pull-up resistors”; they need to be present on the clock line (SCL) and on the data line (SDA). They are used to do the interface between different types of logic devices and they ensure that the circuit assumes the default value when no other component forces the line input state.
Since the chips design are often open-collector, the chip can only pull the lines low and they float to VDD otherwise; this way, the master can sense if a simultaneous transmission occurs, letting the pin float and sensing the line, if the line is still at VDD, probably, no transmission is being accomplished from any other device [W17].

Programming of *atmega16* master software is described at chapter 5.4.8

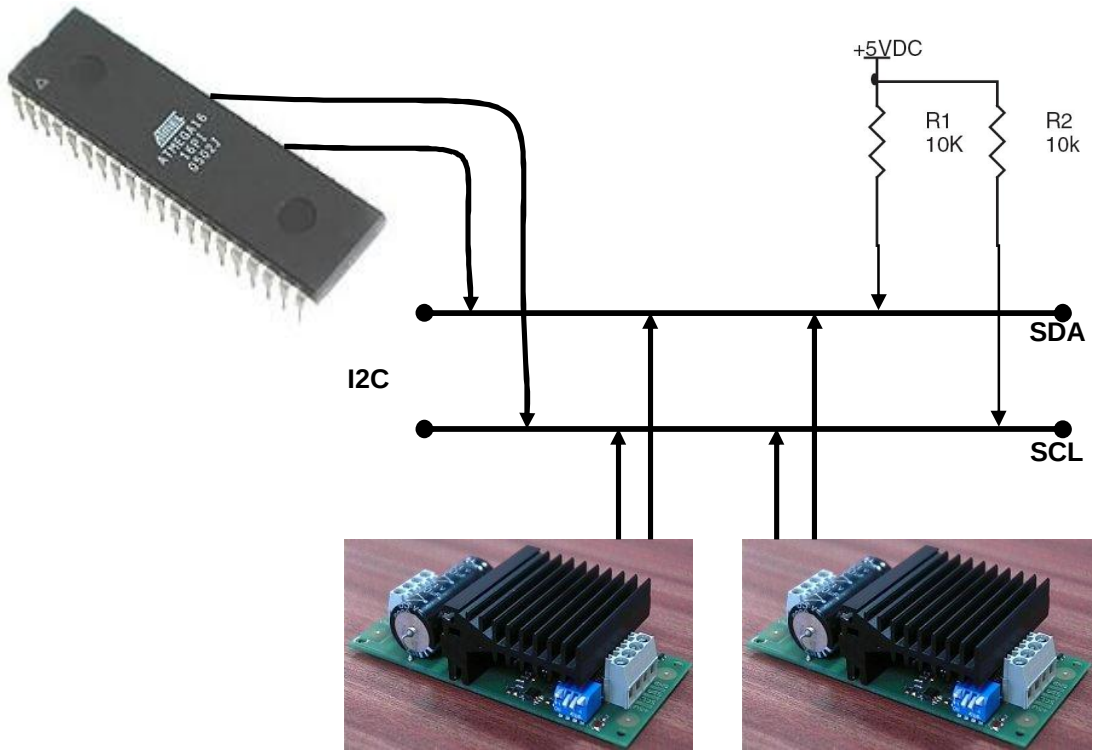


Figure 29 – Robot I2C interconnection system

The two *I2C* signals are *Serial Data* (*SDA*) and *Serial Clock* (*SCL*).
I2C matches *Master/Slave* topology.
The *I2C Master* is the device that can start and stop communications and has usually the duty of controlling the clock.

An *I2C Slave* is a device that is addressed by the master. When the master asks a slave for data, the slave has the possibility to hold off the master in the middle of a transaction using “clock stretching” [W17] (the slave keeps SCL pulled low until it is ready to continue). This makes synchronism of slow slave devices possible, but most I2C slave devices do not use this feature.
It is duty of every *I2C Slave* to monitor the bus and to respond only to its own address.

I2C protocol supports multiple masters and multiple slaves.

Transmitting protocol inherits that data sending of each byte, starts with the MSB (Most Significant Byte).

Figure 30 shows a typical communication between a master issuing the start condition (*S*) followed by a 7-bit slave device address to start a communication with a slave.

The eighth bit after the start (*read/not-write*) is used to signal the slave if the master sends more instructions (slave will receive more data) or if the master is ready to receive data (slave can transmit data).

After each byte sent by the master, the slave must reply with an *ACK bit* to signal the reception of the previous byte.

This 9-bit pattern is repeated if more bytes need to be transmitted.

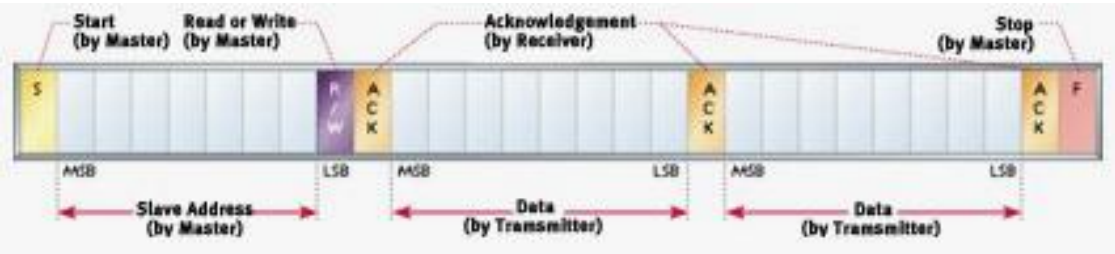


Figure 30 – I2C Packages [W14]

The issue of the stop condition (*P*) is accomplished instead of the *ACK* at the end of a master reading transaction (slave transmitting).

If a master write transaction (slave receiving) is being performed, the master issue the stop condition (*P*) when it receives the last *ACK* of the data sent.

This chapter presented some of the robot system electronics, some mechanical components were also presented, as well of drivers and I2C characteristics.

5. Microcontroller Unit

In this chapter the microcontroller software is presented in order to provide understanding about the microcontroller unit and its duties.
Communications protocol used from the server to the microcontroller and from the microcontroller to the server will also be presented.

5.1. Introduction

There is a large variety of microcontrollers on the market. *Atmega16* belongs to AVR family and was the microcontroller chosen to make a new embeddable system capable to control each *I2C motor driver* system, to read encoders and to give local feedback, through a display, and to perform communications with the server.

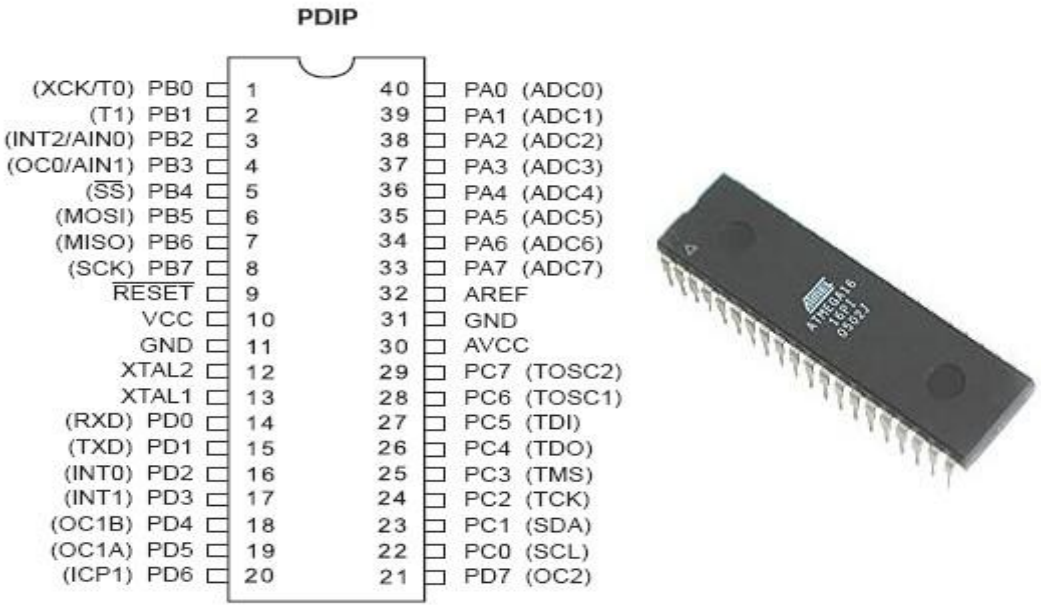
Other microcontroller family could be chosen to the robot system but the cost of the device programming and compiler should not get high and preferably must be freely available. 8051, Microchip PIC®, and Atmel AVR® were possibilities that matches the criteria.

Traditional 8051 have a simple architecture and it is familiar to most embedded engineers. The amount and quality of tools and sample source code available is large but it is common that each manufacturer provides proprietary features and migration from one variant to another usually implies a new programmer circuit. The typical architecture of some models are standard for several manufacturers but those do not have engrossing stuff like A/D and D/A converters, I2C, In-circuit programming, etc. [B2] That lack of standardization and upgrading problems do not meet this project objectives, so it was placed apart.

A PIC microcontroller was considered an expensive solution, much more than the Atmel AVR (The PIC official programmer (PICstart Plus) cost 3 times more than the AVR one (STK500) [B2]).

AVR microcontroller is manufactured by Atmel [W4] and its family is largely used worldwide so it is easy to get access to libraries or fragmented source codes all over the internet [W10], its versatility possibly to make use of several different features and to perform simple future migration of the source code within the same microcontrollers family; it is also possible to use different compilers and different programming languages.

Atmega16 has a number of features which make it desirable to this project. It has 3 Timers, 4 PWM channels, I2C also known as TWI (Two Wire Interface), 8 ADCs (Analogic/Digital Converter), USART *(Universal Synchronous Asynchronous Receiver Transmitter)*, SPI (Serial Port Interface) and 32 I/O



ports [W9]. Atmega16 pinout can be seen on figure 31.

Figure 31 – *Atmega16* pinout [W9]

“AVR Studio” was the editor and debugger used; it is freeware and has a very good and powerful debug mode and simulator [W11] .

The language used is C; the medium level rate of this language makes a good power/control ratio which makes the robot programming flexible.

The way the program was made intended to have modularity and an easy to use structure for any future programmer who will improve the robot.

Modularity was achieved by using many files, each one giving its own main functions; even so they can depend on others, for example: *USART* functions uses *MOTOR* functions after decoding a command sent by the *Server* computer.

Each file can be compiled separately and then linked together. This provides a saving of time since it is not necessary to recompile the complete application when making a single change but only the file that contains it.

A systematic way of writing the program was chosen to provide an easy reading of it.

All *include files* that some C file need is specified at the header file (.h)
Main and external variables are also included at the header file (.h)
Only the local variables are defined on the respective ".Cpp" file

This chapter will present each one of the several ".Cpp" files and flowcharts of some functions.

5.2. Header Files Structure

The organization makes possible a fast access of functions, variables, etc.

So each header file has the same template layout which consist of defining, at first include files, followed by constant definitions and variables used, at last, functions prototypes are declared.

5.3. PIN List

List of all currently used PIN's as well as a description is presented at table 3, even so, some might be described during this chapter.

PIN Variable Name	Description	PIN	DDR	PORT
error_led1_PIN	LED signal of error nr. 1.	7	DDRA	PORTA
error_led2_PIN	LED signal of error nr. 2.	7	DDRD	PORTD
encoder0_direction_PIN	Encoder right Direction Signal. Set this pin high means running forward and set this pin low means running backward.	4	DDRD	PORTD
encoder1_direction_PIN	Encoder left Direction Signal. Set this pin high means running forward and set this pin low means running backward.	5	DDRD	PORTD
INT0	Encoder right Channel Signal, Used to count pulses from the encoder.	2	DDRD	PORTD
INT1	Encoder left Channel Signal. Used to count pulses from the encoder.	3	DDRD	PORTD
LCD_DATA0_PIN	Pin for 4bit data bit 0 (Least Significant Data Bit).	0	DDRA	PORTA
LCD_DATA1_PIN	Pin for 4bit data bit 1.	1	DDRA	PORTA

LCD_DATA2_PIN	Pin for 4bit data bit 2.	2	DDRA	PORTA
LCD_DATA3_PIN	Pin for 4bit data bit 3 (Most Significant Data Bit)	3	DDRA	PORTA
LCD_RS_PIN	Pin for RS (Register Select) line This pin determines whether the data you're about to write is a command or a data byte.	4	DDRA	PORTA
LCD_RW_PIN	Pin for RW (Read/Write) line. Set this pin high to read from the display. Set this pin low to write to it.	5	DDRA	PORTA
LCD_E_PIN	Pin for Enable line. This line works to clock in data and commands.	6	DDRA	PORTA
SDA	I2C/TWI Data line.	1	DDRC	PORTC
SCL	I2C/TWI Clock line. It is used to synchronize all data transfers over the I2C bus.	0	DDRC	PORTC

Table 3 – All Pin List

5.4. Files

The files that compose *atmega16* applications will be presented below:

5.4.1 atmega16.c

Atmega.c is the main file of whole microcontroller application.

Initializations of the modules (error support, *usart*, lcd, encoder, i2c, timers, interrupts and motor drivers) are accomplished at the main file (*atmega16.c*), reason why *atmega16* header file connects all needed modules, as shown in figure 32.

After all initializations, the program will run in an infinite loop waiting for any external interrupt, commands sent over the serial port and waiting for timer interruptions that will perform some computation as velocity calculations, Virtual Heart Beat commands and to share data from the sensor to the *server*.

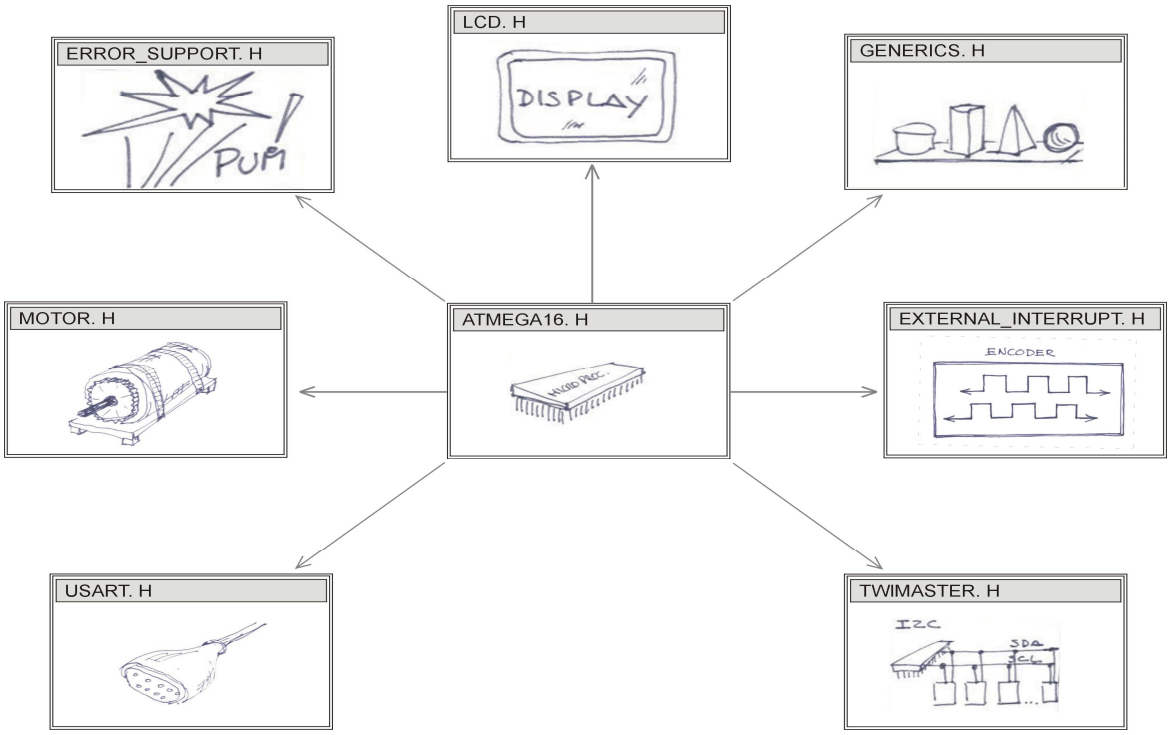


Figure 32 – *atmega16.h* interconnections

5.4.2 error_support.c

This module is used to help the programmer at the debug stage.
Features provided are the basic turn on and turn off of *LEDS*.
At the moment two LED's are defined:

LED Number 1: PORTA.7

LED Number 2: PORTD.7

It is very easy to include this file in any other and give them this kind of debug capability and to increase the number of *LEDS* or change its port connections!

The functions provide are:

void error_support_init(void);

Initialization of *LEDS ports and pins* (output).

void error_on(int led_number);

Turn a LED On.

void error_off(int led_number);

Turn a LED Off.

5.4.3 external_interrupt.c

This module is used by the encoders.

Robot has two quadruped encoders, each one is connected to an *External Interrupt* and, each time a transition is made by any encoder the microcontroller will count it.

Derived from the asynchronous and unpredicted pulse occurrence, an external interrupt was configured, this way this “time-critical” operation is separated from the main program execution [B1].

Generically, two main types of external interrupts could be implemented:
Figure 33 plots those types of signals.

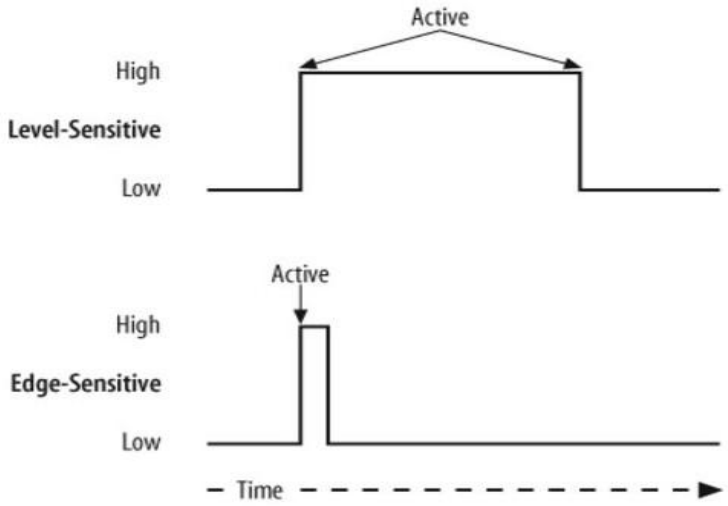


Figure 33 – Level- and edge-sensitive interrupt signals [B1]

Level-sensitive interrupts are attended at either a low or high level [B1].
Edge-sensitive interrupts are attended at a transition that can be defined to be rising edge or falling edge sensitive interrupt [B1].

An edge-sensitive approach was chosen because even if, in theory, a pulse count can be skipped when a subsequent interrupt occurs [B1] (in practice, test with both robot wheels running at same speed shown that the processor catch all pulses); an approach with a level-sensitive interrupt would certainty provide worth results because that type of interruption suspends other processing during all level time [B1] an then pulses would be missed if both wheels were running at the same speed.

Another PIN is defined, for each encoder, at the *header file,* with purpose of know whenever the encoder is running forward or backwards and then the microcontroller knows if it has pulses to be increased or decreased respectively. The way those PINs gets its state was detailed described at chapter 4.5.

5.4.4 generics.c

This module provides two functions:

void delay_ms(unsigned short ms);

Used to make a variable delay.

void wait_until_key_pressed(void);

Used to read a switch, actually is define to read PIND.2.

5.4.5 lcd.c

This module implements a free to use *HD44780U LCD library*; the author is *Peter Fleury [W16]*, after changes of *PINs* options, adjustments to use a 4 *PIN* data transfer and after prepare it to a 20x4 LCD, it looks just perfect to communicate with the LCD.

The main provided functions are:

void lcd_init(uint8_t dispAttr);

Initialize the display and selects the type of cursor.

void lcd_clrscr(void);

Clear the display and set cursor to home position.

void lcd_gotoxy(uint8_t x, uint8_t y);

Set cursor to specified position.

lcd_puts(const char *s);

Display string without auto linefeed.

A function to display an integer number was also added:

void lcd_puti(int int_value);

Display *int* value.

An example of application can be written with the following commands:

```
lcd_gotoxy(0,2);
lcd_puts("MD03 Right = OK!!");
```

This example is used, after the LCD initialization, at the start of the main program, inside a function to test the communication to the motor drivers (*motor_drivers_init_test();*) the result is writing on the display "MD03 Right = OK!!" at the 1st column and 2nd line, providing that feedback to the robot user.

5.4.6 motor.c

This module functions control the motor drivers, main functions are below described and interconnections can be seen in figure 34:

void motor_drivers_init_test(void)

It is used to make initial tests to the motor drivers by testing *I2C* communication with both drivers; it also updates the variable *motor_driver_error* with the result of the test.

That variable is also useful to avoid sending commands to the driver when it is not connected, that way the microcontroller does not halt trying to send commands to a disconnected motor driver and so tests with others sensors and the *server* computer can be made without these drivers.

Feedback is also achieved through the *LCD*.

void break_motor(void)

It is used to provide a simple and fast way to break the motors.

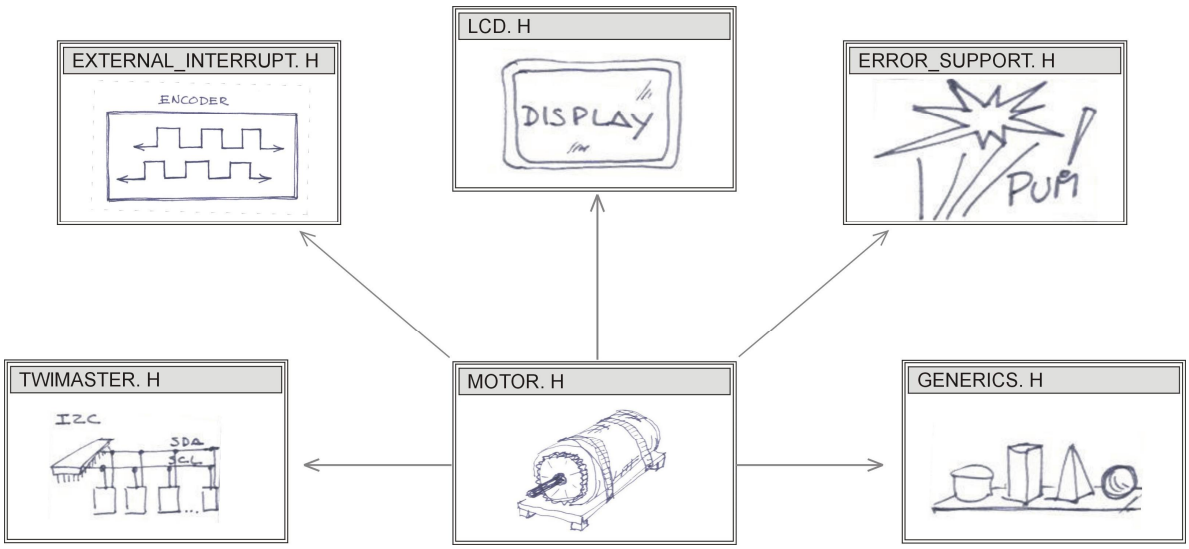


Figure 34 – *motor.h* interconnections

5.4.7 timer.c

Functions related to the timers are implemented in *timer.c* / *timer.h* files.

As usual at embedded systems [B1], when the timer is activated, the program will change its flow to the respective interruption function, at atmega16 that function is ***SIGNAL (SIG_OVERFLOW1)*** and when it is completed it returns to the place where it was before.
(SIG_OVERFLOW1 is the address of the interruption vector respect to the timer/counter 1 overflow)

At figure 35, interconnections with this module can be seen.
Timer purpose is to:

- Calculate speed of each robot motor.
- Deal with “Virtual Heart Beat” a.k.a. “Emergency Ping”.
- Automatically send sensor data to the *Server.*

void starttimer1(void);
Used to start number 1 timer.

void stoptimer1(void);

Used to stop number 1 timer.

A “Virtual Heart Beat”, is identified at the code as an “emergency ping”, one was created between *Server*/*Client* and other between *Server*/*Microcontroller*. The purpose is to avoid a robot control loss.

What concerns the microcontroller, it can be described as a command that is sent to the *server* every 2 seconds. When the *server* receives it, it has the duty to resend that command. If, after four seconds, no acknowledge of the previous *ping* is received then the microcontroller sends a command to both drivers to stop the motors. This way the robot stops and damages caused by an uncontrolled robot are avoided.

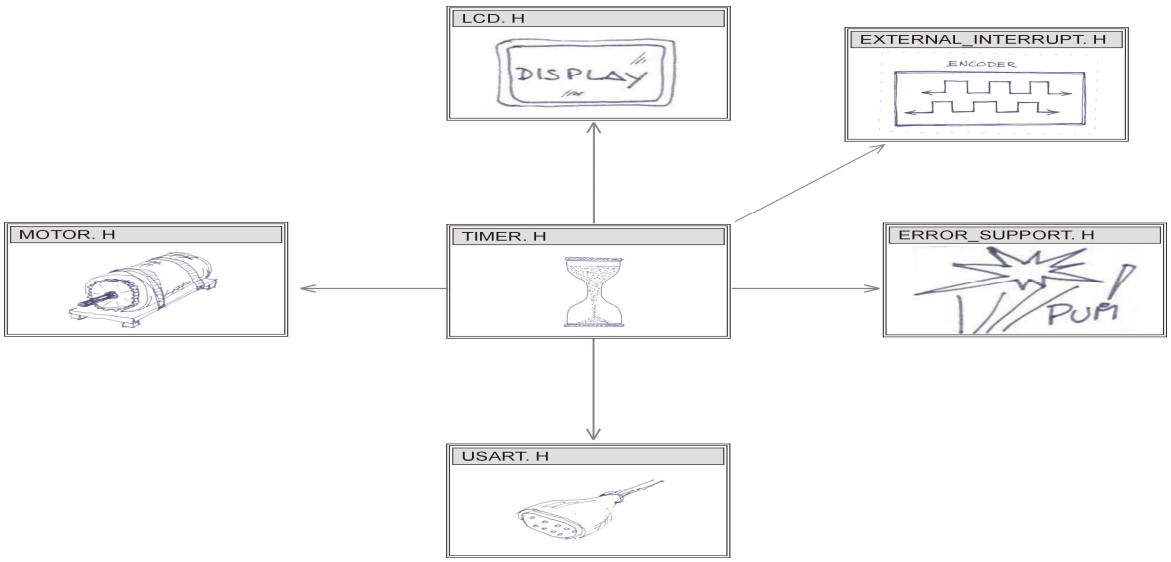


Figure 35 – *timer.h* interconnections

5.4.8 twimaster.c

This module implements a free to use *I2C library; a*uthor is *Peter Fleury [W16]*.

It is used to provide functions to operate the I2C bus and to communicate with I2C devices.

The main functions provided in this library are:

void i2c_init(void);

I2C master interface initialization.

Need to be called only once for each device.

void i2c_stop(void);

Terminates the Data transfer and releases the I2C bus.

unsigned char i2c_start(unsigned char addr);

Issues a start condition and sends slave address and transfer direction; returns 0 if the device is accessible or 1 if failed to access device.

unsigned char i2c_rep_start(unsigned char addr);

Issues a repeated start condition and sends slave address and transfer direction;

Returns 0 if the device is accessible or 1 if failed to access device.

void i2c_start_wait(unsigned char addr);

Issues a start condition and sends slave address and transfer direction.

unsigned char i2c_write(unsigned char data);

Sends one byte to I2C device;

Returns 0 for a successful writing or 1 if the writing process fails.

unsigned char i2c_readAck(void);

Reads one byte from the I2C slave device, requests more data from device and returns read byte from I2C device.

unsigned char i2c_readNak(void);

Reads one byte from the I2C slave device; Reading operation is followed by a stop condition.

Returns read byte from I2C device.

As an application example, the following commands:

```
i2c_start_wait (MD03_I+I2C_WRITE);    // set device addr. & write mode
i2c_write(addr_direction);            // write address
i2c_write(0);                         // ret=0 -> Ok, ret=1 -> no ACK
i2c_stop();
```

Writes to the left motor driver a clockwise direction (0).

5.4.9 usart.c

This module is used to interact with the USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter).

To connect RS-232 devices to the atmega16 USART a line drive is need [Max232 Datasheet].
Connection schematic is show in figure 36.

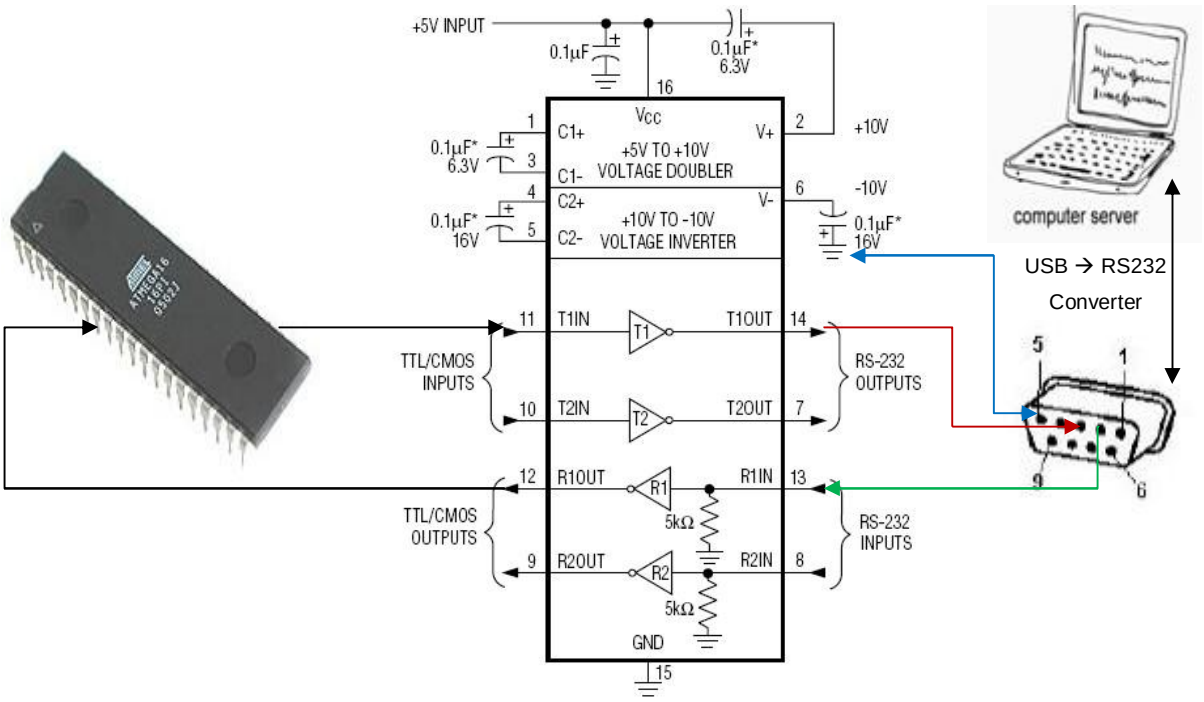


Figure 36 – *Atmega16* and RS-232 connection

Table 4 specifies connections made:

Atmega16	**Max232**
Pin 14 (PD0/RX)	Pin 12 (R1Out)
Pin 15 (PD1/TX)	Pin 11 (T1IN)
Pin 31 (GND)	Pin 15 (GND)

Max232	**USB-RS232 converter**
Pin 13 (R1IN)	Pin 3 (TX)
Pin 14 (T1OUT)	Pin 2 (RX)
Pin 15 (GND)	Pin 5 (GND)

Table 4 – Pin connections between server and atmega16

This module provides a communication system between *atmega16* and the *server* program.
It also does interpretation, followed by correspondent actions, of any commands provided from the *server* and is able to send commands provided from other modules of atmega16 microcontroller (e.g. timers [chapter 5.4.7]) to the server.

Baud Rate constant is calculated at the header file, it is needed for the oscillator frequency (*F_OSC*) and the desired baud rate (*UART_BAUD_RATE*), so, it is easy to change the crystal oscillator because no extra math is needed to be carried out, only *F_OSC* constant has to be changed in that case. Similarly, changes of baud rates only need an update of the respective constant.

The actual baud-rate, data bits, parity, number of stop bits and flux control type can be seen at table 5 and are specified at the *server* and at *atmega16* microcontroller.

Baudrate	38400 bps
Databits	8 bits
Parity	None
Stopbits	1bit
Fluxcontrol	none

Table 5 – Connection between server and atmega16 microcontroller through RS-232

Communication between the Serial Ports is dealt with the following functions:

void usart_putc(unsigned char c);
Send a character.

void usart_puts (char *s);
Send a string.

void USART_init(void);
Atmega16 USART initializations.

Figure 37 show interconnections within timer module.

A protocol of communication was developed to get an understanding between *atmega16* microcontroller and *server* program; two versions were experimented because problems occured with the first one.

Protocol initializations are carried out with the following function:
void USART_init_variables(void);

After a command interpreting, the following function is called to provide its execution:
void UsartExeCmd(void);

Others functions were implemented:

void Send_Sensor_data_with_Usart(void);
Sends robot sensors data to the *USART* (relatively to each motor it sends position, velocity, current and temperature)

void Send_Emergency_Ping_with_Usart(void);
Sends the *server* the “Virtual Heart Beat” command, also known as “Emergency Ping” command.

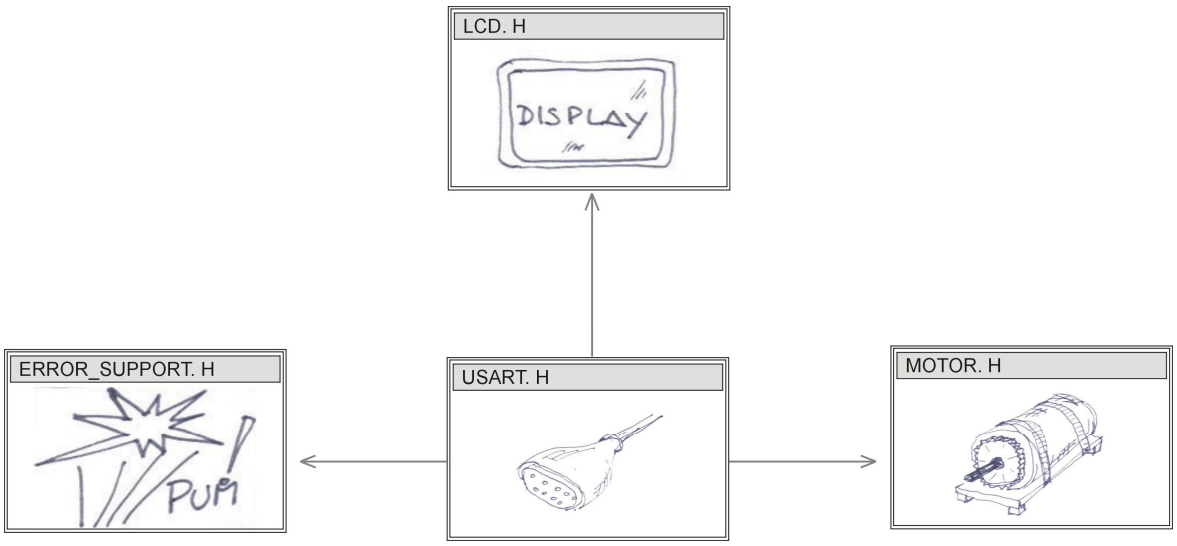


Figure 37 – *usart.h* interconnections

"1st attempt" communication protocol from the Server Computer to the Atmega16 microcontroller and problems about it

Each *Command/Value* frame received by the *atmega16 USART* must have three bytes defining a command, followed by three bytes defining a value; all of them in *ASCII* format.

Table 6 shows the word composition layout.

Word Composition nr. *1*		…….	**Word Composition nr. *N***	
Function	Value	…….	Function	Value
3bytes	3bytes	……	3bytes	3bytes

Table 6 – Word Composition (1st attempt version)

This approach worked well for some time in the practical point of view but, from time to time, a control loss occurs letting the robot drift. That happened because the microcontroller stops to interpret commands due to a synchronisation loss

which happened randomly when a transmission error occurred. Unfortunately, this error behaviour overcome could not be reached without a microcontroller reset.

That behaviour shows that commands with a fixed number of bytes, as well as a fixed number of respective values, are not a good choice due to the resynchronization problems in case of transmission errors.

Communication Protocol from the Server Computer to the Atmega16 microcontroller

Each *Command/Value* frame receive by the *atmega16 USART*, to be correctly interpreted, must have at least one alphanumeric byte followed by at least one numerical value and finally a 'Z' character will flag the end of each command/value frame.
All characters must be in *ASCII* format.

Word composition layout is shown at table 7:

Word Composition nr. *1*			…….	**Word Composition nr. *n***		
Function	Value	'Z'	…….	Function	Value	'Z'
x bytes	y bytes	1 byte	……	u bytes	v bytes	1 byte

Table 7 – Word Composition

Communication stability is improved because if a communication lack occurs, the command is misunderstood but, at least, synchronism loss is avoided.

Flexibility was also improved by this method because commands and values can have different sizes.

An example of a shared command from the *Server* to the *Atmega16* can be seen in table 8, the frame is sent over RS232, at line one, the purpose is to set the left motor acceleration to a 215 value; at line two, acceleration is set to 8.

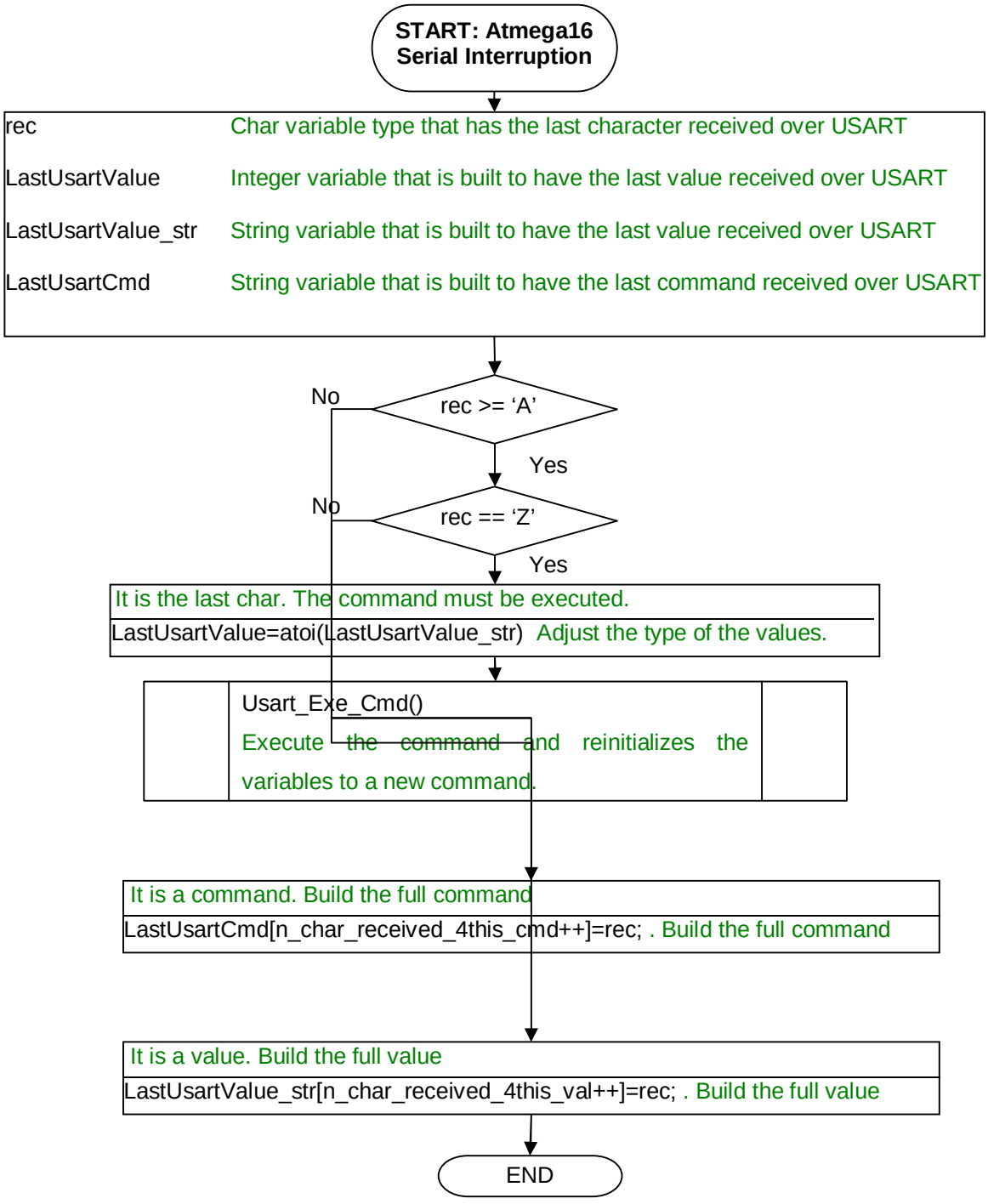
M	L	A	2	1	5	Z
M	L	A	8	Z		

Table 8 – Examples of shared commands from *Server* to *Atmega16*

A function:

QString conv_double_QString(double value_input)

It is used to convert a double integer value to a QString. It is widely used because the values are sent in *ASCII* format.



Flowchart 1 – Atmega16 Serial Interruption

Communication Protocol from the Atmega16 microcontroller to the Server Computer

Sending data protocol from the *atmega16* microcontroller to the *server* computer is different than the opposite direction.

Server computer usually requires, either all sensors data or a considerable amount of data besides only a specific value, so, instead of sending an extra character (as the 'Z' character in the communication from the *serve*r to the *atmega16*) to signal the end of a specific frame, the process is implemented to make a frame validation each time a new alphanumeric character appears.
This way, to process the last received frame, the server must receive an extra frame: that last frame is usually the "END0" and does nothing except handling the server the possibility to know that the previous value received has been completed.

"END0" frame is, in reality, an undefined *command/value* by the *server computer* and so it can be replaced by any other appearance as "E0" or any other undefined *command/value*.

After a *command/value* is identified, another function is called to execute.

Resuming, *atmega16* microcontroller can send several sets of commands and values with different sizes and, when *server* receives the *command/value* "END0" it guarantees the process of the last command/value.

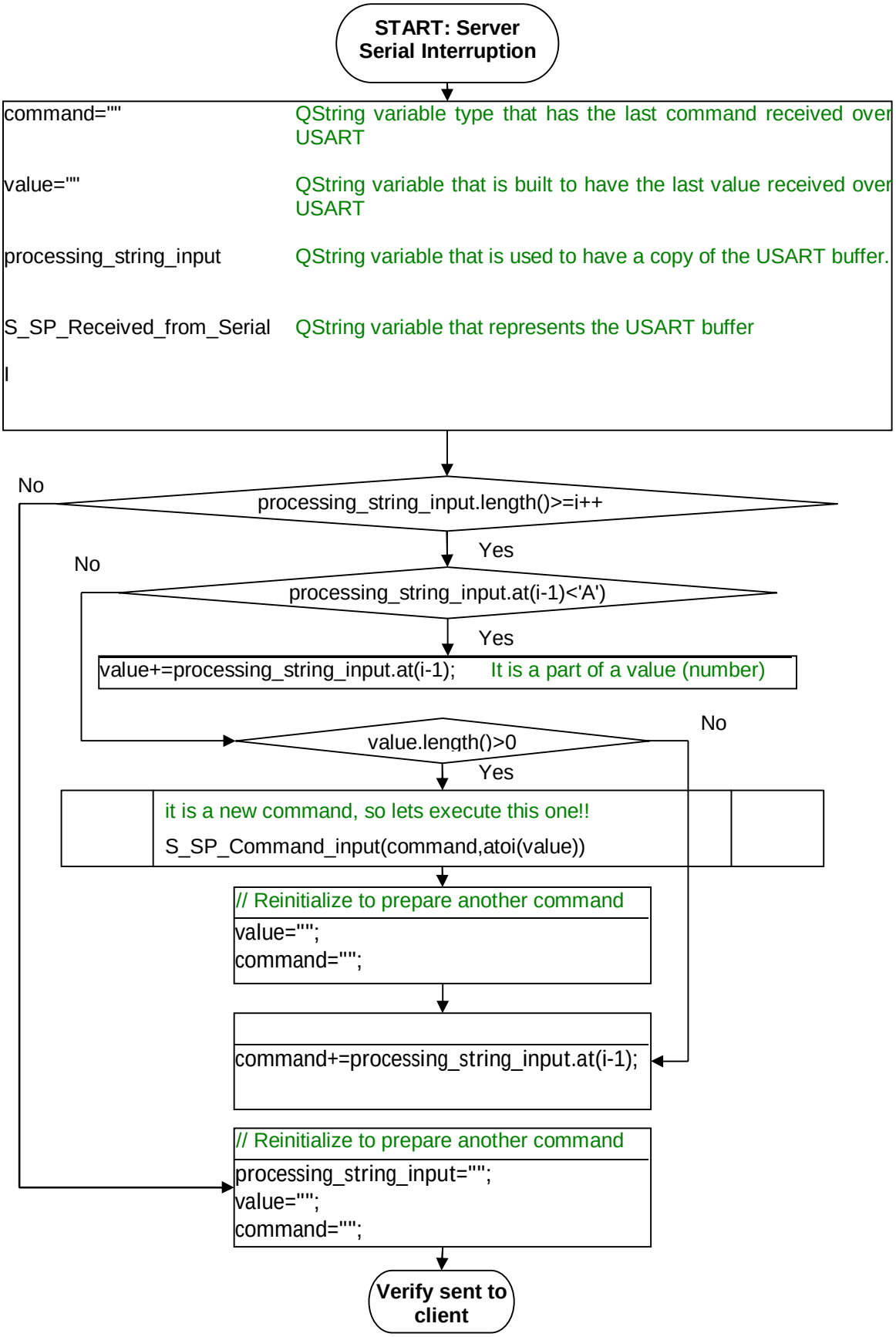
To ensure a correct explanation of the whole communication process, it needs to keep in mind that inherent the serial port process, the operating system gets a variable amount of data from the serial port buffer and the server applications get that frame which can contain several sets of *commands/value*. To deal with all amount and unpredictable data, every time the server gets data, a copy to a new variable is carried out and it is accomplished a reset of the old buffer to a null value and then the identifying process of *commands/values* is started. This

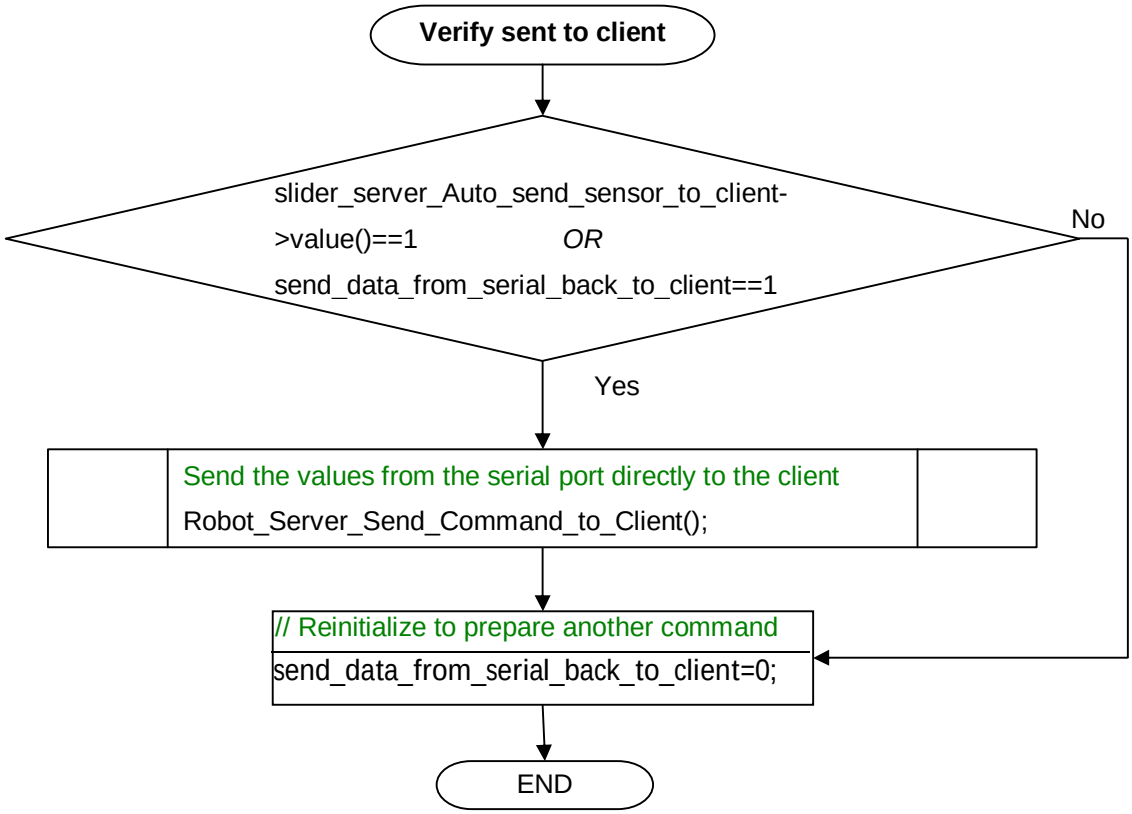
way the buffer does not get too long and the commands/values indentifying process have static data within the process (the buffer update is in other thread, reason why it has a dynamic growth).

A shared command example, from the *Atmega16* to the *Server,* can be seen at table 9, the frame is sent over RS232, at line one, the right encoder position has a 6459 value; at line two, it can be seen a frame which identifies the encoder right position to 210 value and its velocity to value 5; at line three it can be seen an usually shared command that is the “Virtual Heart Beat” aka “Emergency Ping”.

E	R	P	6	4	5	9	E	N	D	0
E	R	P	2	1	0	E	R	V	5	E	N	D	0
E	M	P	0	E	N	D	0

Table 9 – Examples of shared commands from *Atmega16 to the Server*


START: Server Serial Interruption
command="" QString variable type that has the last command received over USART
value="" QString variable that is built to have the last value received over USART
processing_string_input QString variable that is used to have a copy of the USART buffer.
S_SP_Received_from_Serial QString variable that represents the USART buffer
I
No
processing_string_input.length()>=i++
Yes
No
processing_string_input.at(i-1)<'A')
Yes
value+=processing_string_input.at(i-1); It is a part of a value (number)
No
value.length()>0
Yes
it is a new command, so lets execute this one!!
S_SP_Command_input(command,atoi(value))
// Reinitialize to prepare another command
value="";
command="";
command+=processing_string_input.at(i-1);
// Reinitialize to prepare another command
processing_string_input="";
value="";
command="";
Verify sent to client



Flowchart 2 – Server Serial Interruption

Microcontroller unit was presented as well as the developed software, used communications protocol between the *server* computer and the microcontroller were presented as well.

6. Desktop Processing

This chapter will fall upon the *server* and the *client* programming, robot user interface and system configurations.

6.1. Introduction

The proposed system makes use of two computers based on a *server-client* architecture. *Client* computer is not critical but the *server* used is, at the moment, a common laptop but later it will be replaced with an industrial and low-power consumption one.

The language used for the desktop processing is *C++* and *Qt* libraries and tools were widely used.

Since *Qt* libraries are used, this system is platform independent, which means that the software platform can be use in a *Windows, Windows CE, MAC, LINUX* or an *embedded Linux* environment.

The use of a computer in the system was made because the computer can achieve plug and play updates through USB ports, can make some parallel computation and the space of a laptop in the robot is not critical because the robot is big and a lot of space is still free for other components.

"*Qt* sets the standard for high-performance, cross-platform application development. It includes a *C++* class library and tools for cross-platform development and internationalization"

[W5]

Figure 38 shows the *Qt* framework block diagram.

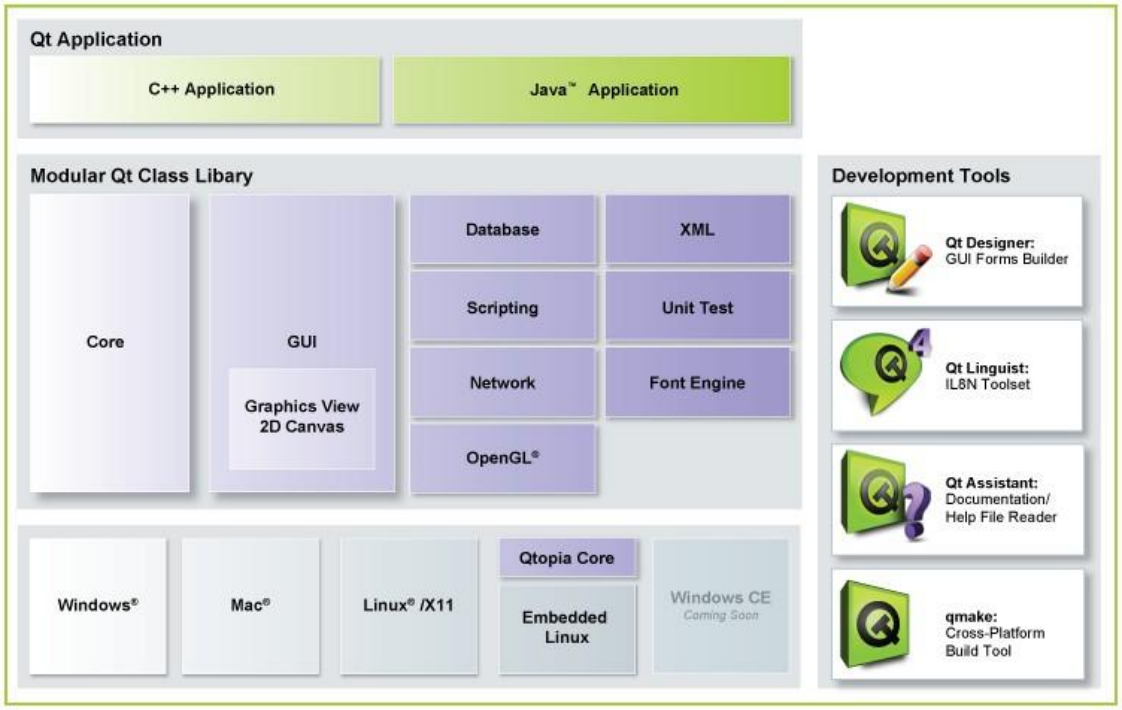


Figure 38 – Qt Block Diagram [W5]

Qt makes use of a work based philosophy in objects and uses *Signal/Slots* directives which can be connected with each others.

The idea of a desktop software application is to provide high level processing and to provide users to work remotely with the robot.

Two software applications were made:
One to the "*Server"* and another to the "*Client*" computer, communication between them is carried out through TCP/IP protocol.

A "Virtual Heart Beat" which is identified at the code as an "emergency ping", was created between the *Server*/*Client* and between the *Server*/*Microcontroller* pair to avoid a robot control loss.
What concerns the *Client*/*Server*, it can be described as a command that is sent from the *Server* to the *Client* every 2 seconds.
When the *Client* receives it, it has the duty to resend that command.

If, after four seconds, no acknowledge of the previous *ping* is received then a command to stop the motors is sent from the server to the microcontroller to avoid damages caused by an uncontrolled robot.

6.2. The Client

Client is used to transfer the user interface values directly to the server and is able to interpret incoming commands proceeding from the *server* in order to provide users to know *Temperature* and *Current* values of each motor drives and also the *Position* and *Velocity* values of each motor.
It also needs to interpret the “Virtual Heart Beat” (“emergency ping”) between the *server* and the *client* to avoid the robot control loss which can be derived, as example, from a supposedly energy lack that could turn off the wireless router or even the *client* computer.

6.3. Client Class

Client class makes possible to create a client based on an *IP* address and respective *port*:

Constructor is shown below:
Client(const QString &host, Q_UINT16 port);

The following function, **SendStrToServer(QString texto)**, makes possible to send a *QString* to the *server* computer and activates a *Signal* called ***logSentText*** providing a way to log communications data accomplished by the *client*.

Client has a socket to receive data: **readyRead()** *signal* is connected to the **socketReadyRead()** slot**;** this *slot* also provides a way to log the communications data by emitting the **logRecText()** *signal*.

An interconnection of those *signals/slots* example is shown below, the syntax is flexible but the first argument is always a *QObject*, and then the *signal/slots* are specified.

If the *signals/slots* belong to the same *QObject*, it can be specified just once, otherwise the connection needs both *QObjects.*

connect(client, SIGNAL(logSentText(const QString&)),cliente_log_sent, SLOT(append(const QString&)));

connect(socket, SIGNAL(readyRead()), SLOT(socketReadyRead()));

Others private *Slots* of each *Client* can be used, their names self-explain their behavior:

- **closeConnection();**
- **sendToServer();**
- **socketReadyRead();**
- **socketConnected();**
- **socketConnectionClosed();**
- **socketClosed();**
- **socketError(int e);**

6.4. User interface and applications

Figure 39 shows the *Client* application user interface; some description can also be seen under chapter 6.10.

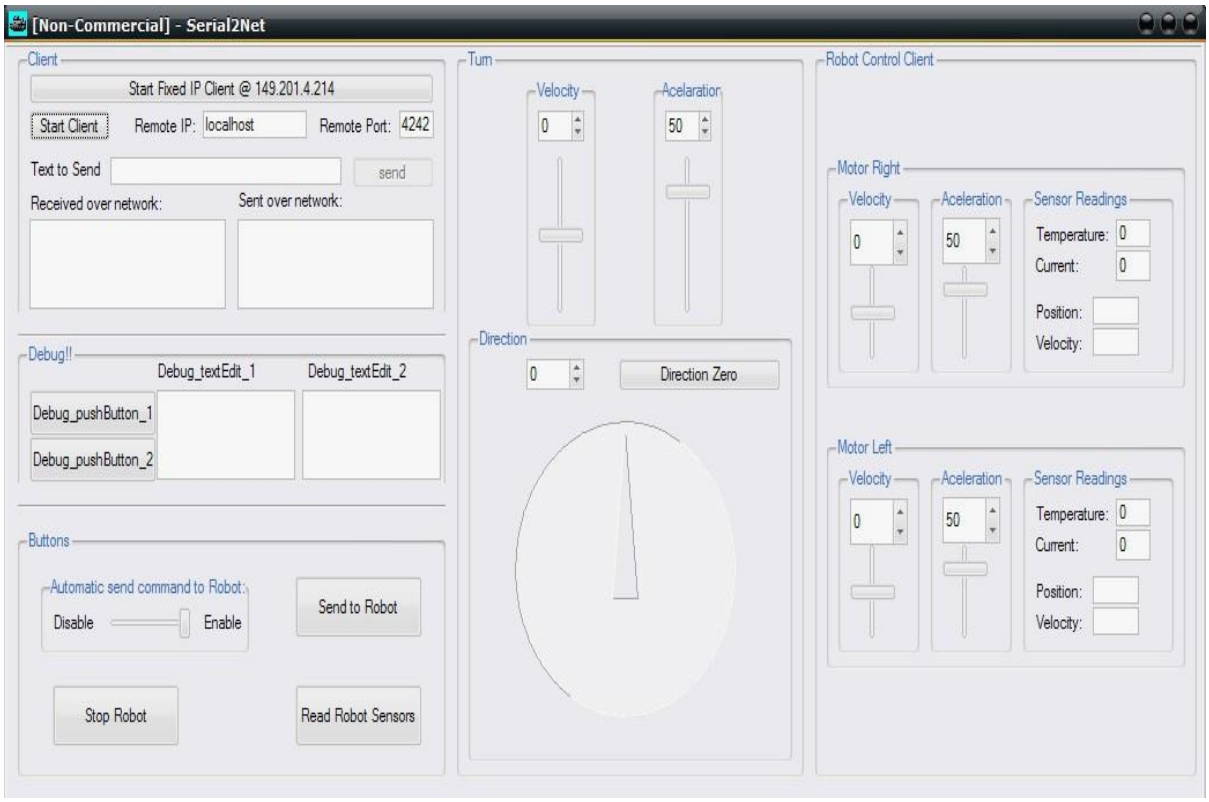


Figure 39 – *Client* User Interface

The *Client* needs to interact with the *Server* computer in order to send commands that the *user* wants the *robot* to have.

Client needs to listen data from the *Server*, that data can be provided from the robot sensors but it is the *server* that transmits it (if the *Server* options are defined to).

The user interface includes a *QgroupBox* named “Client” which has *Qbuttons* to connect to an *IP address* that can be chosen through a *QlineEdit*.

Data is sent in *ASCII* format and the adopted protocol of exchanged commands between the *server* and the *client* is the same as from *atmega16* to the *server*:

Several commands/values set with different size are being concatenated every time a character is received and when the word “END” appears the processing starts.
Client will interpret all data it received and sends, each *command/value*, one by one, to a function that will execute those commands; that function is **Client_Interpret_Command_Receive_from_Server(QString command, int value)**.

Commands that are currently being used are shown in the table 10:

TD	Turn Spin box (Turn Direction)
TV	Turn Velocity
TA	Turn Acceleration
MRA	Motor Right Acceleration
MRV	Motor Right Velocity
MLA	Motor Left Acceleration
MLV	Motor Left Velocity
RAD0	Read All Data

Table 10 – currently used commands

Examples of shared command between *Server* and *Client* can be seen on table 11: line one has a command example to set the right motor velocity to 50; line two has a command example to perform all sensors read and to set both motors velocity to 68.

M	L	V	0	5	0	E	N	D	0

T	V	0	6	8	R	A	D	0	R	A	D	0

Table 11 – Examples of shared commands between *Server* and *Client*

Client has a *QButton, as shown in f*igure 40, to signal the *Server* that the *Client* commands should automatically be sent to the robot:

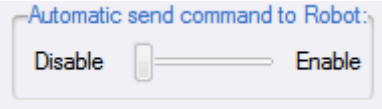


Figure 40 – Automatic send commands to robot

To do that, every time any slider is released, it will check if "automatic send option" is active, and if so, "*Send Button*" associated function is called. (**pushButton_Robot_Client_Send_Command_to_Server_clicked()**)

Independently to the "Automatic send command to Robot" option, it is possible to send the *server* data from velocity, acceleration and direction through the "Send to Robot" *QButton*, and commands to read temperature, current, velocity and acceleration through the "Read Robot Sensors" *Qbutton*; the "Stop Robot" *Qbutton* sends commands to put both motors with zero velocity.

6.5. The Server

The *Server* is used to interpret data from the *client*, to communicate with the microcontroller unit and is able to operate the robot in a standalone mode (without the *Client*).

Server has a *Client* user interface copy and once it receives any *Client* commands, it performs interpretations and adjusts the sliders values at the *Server* user interface side.

Server can send commands to provide *client* with Temperature and Current values of each motor driver and also the Position and Velocity values of each encoder.

6.6. Server Class

Server class makes possible to create a server based on a specific TCP/IP port:

Data transmission from the *Server* to the *Client* is identical as from the *Client* to the *Server*; the only difference is the behavior of the interpreted commands.

The **data_recived_over_network(const QString& str_input)** function is called each time the *server* receives data from a *client*.
When the “END” word appears in a communication frame, all commands are interpreted, one by one, by the **S_Command_received_from_client (QString command, int value)** function.

The **Robot_Server_Send_Command_to_Client()** function is used to send the robot sensors data to the *client* application.

6.7. Serial Port

With *SerialPort* class it is possible to have control of the *Server* Computer Serial Port.

Several *Slots* can be used, their names self-explain their behavior:

- **openPort();**
- **closePort();**
- **saveBaudrate(QString);**
- **sendToPort(QByteArray);**
- **end();**

As example, **saveBaudrate(QString)** slot is useful to make the *baudrate setup options* remembered between sessions.

The following *signals* are useful to get/send data from/to the serial port as well as to signal the connections results.

- **sConnected(QString);**
- **sDisconnected();**
- **sSerialError(int);**
- **sDataWritten(QByteArray);**
- **sDataRead(QByteArray);**

The **S_SP_Analise_Data_input()** function is used to interpret data from the serial port and when any *command/value* is interpreted it calls the **S_SP_Command_input (QString command, int value)** function to deal with the robot sensors data and it can follow those values to the *client* as well as it will update the *servers* user interface.

Note: Communications protocol between *atmega16* microcontroller and *server* computer can be seen at chapter 5.4.9

6.8. Setup Window and Log Window

The *Server* has a menu where the "*Setup Window*" (Figure 41) and the "*Log Window*" (Figure 42) can be reached.

Setup window allow choosing the *Com Port* definitions as well as the *Network port*. Last settings are remembered between sessions.

Log Window is very useful to show every shared data through the *Serial Port,* debug purposes and behaviors tracking may be performed by making use of this window.

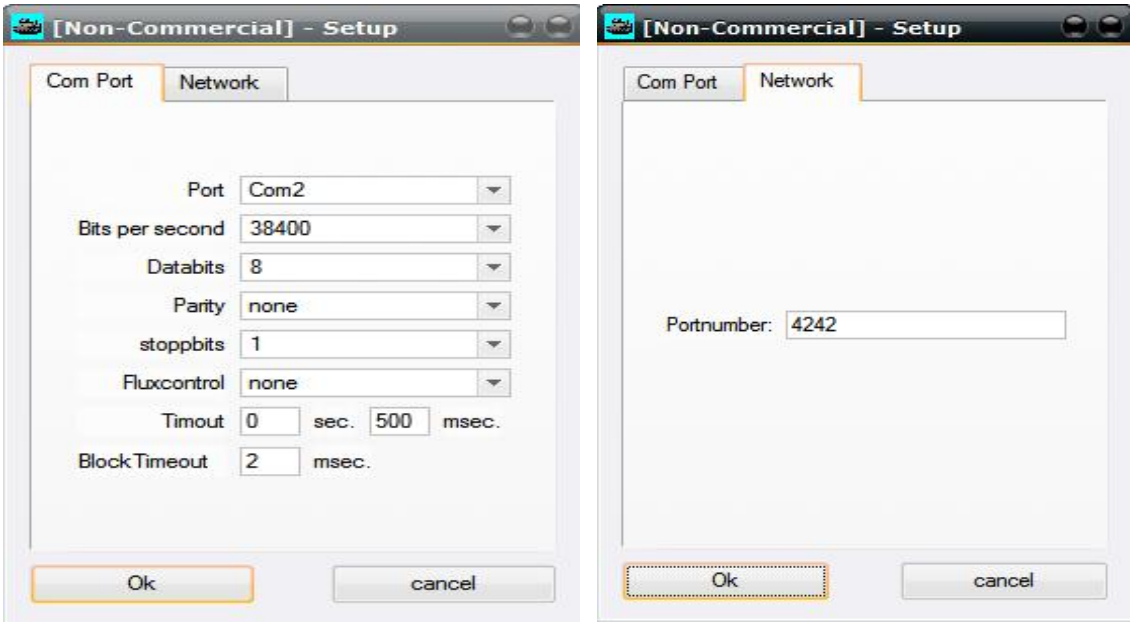


Figure 41 – Setup: Serial Com Port and Network settings

Figure 42 – Logging Serial Port

6.9. User Interface and applications

The *server* user interface is shown on figure 43, some description can also be seen under the next chapter (6.10.).

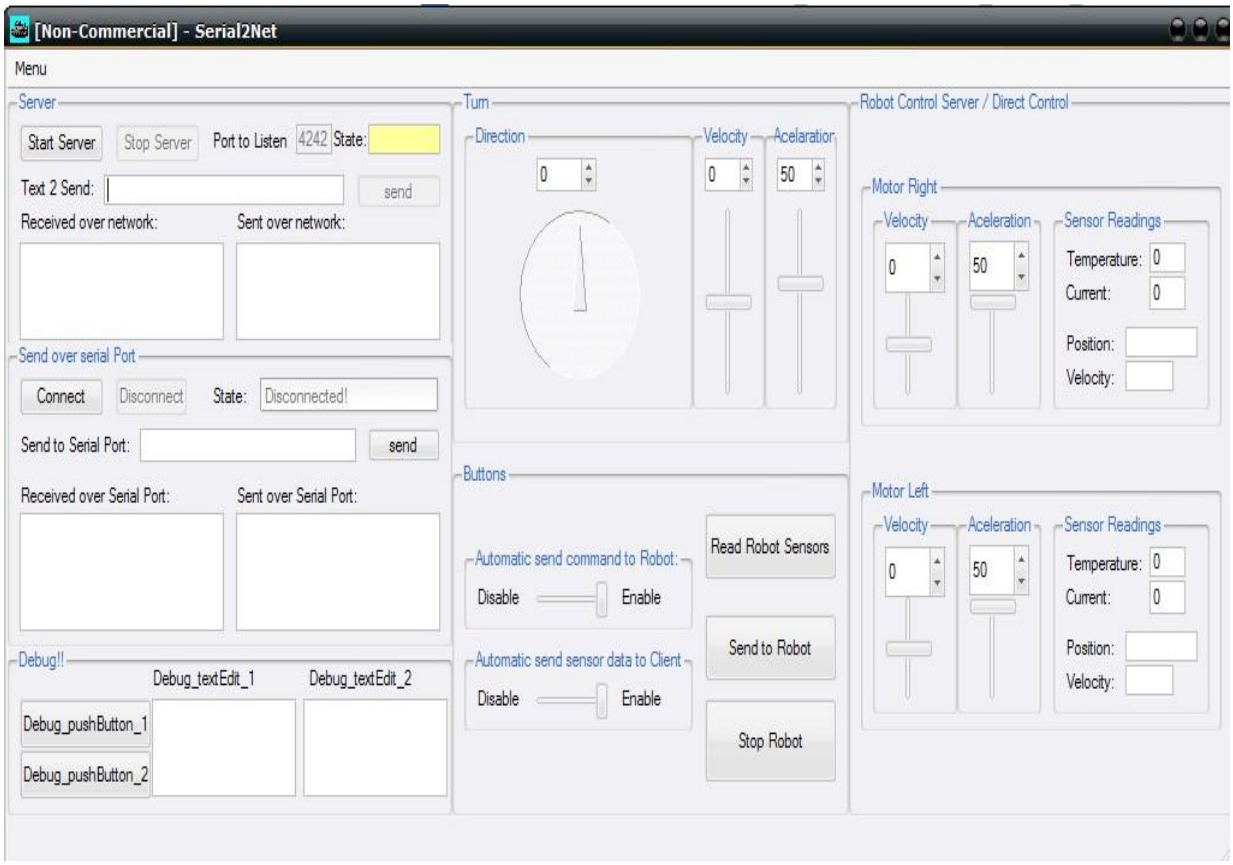


Figure 43 – Server User Interface

Server has a *QButton, as shown in figure* 44*,* to provide that the commands are automatically sent to the robot.

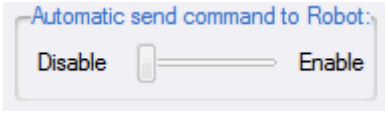


Figure 44 – Automatic send command to Robot option

The behavior is the same as described in a similar button on the *Client* side (*Figure* 40):
Each time any slider is released, it will check if automatic send option is active, and if so, the "*Send Button*" associated function is called.
(**pushButton_Robot_Server_Send_Command_To_Serial_clicked()**).

Independently of the "Automatic send command to Robot" option, it is possible at any time, to send commands to the robot (through RS232). Data from velocity, acceleration and direction can be sent through the "Send to Robot"

Qbutton, and commands to read temperature, current, velocity, acceleration through “Read Robot Sensors” *Qbutton*; the “Stop Robot” *Qbutton* sends commands to put both motors with zero velocity.

The *server* has also a *Qbutton*, as shown in figure 45, to provide data from the robot sensors to be automatic sent to the *client*.

Figure 45 – Automatic send sensor data to Client

To do that, every time the *server* interprets the sensors data from the *Serial Port*, it will forward those values to the *client* if this option is enabled.

Data transfers between the *Server* and the *Serial Port* can be seen, as shown in *figure* 46, through two *QtextEdit*regards to “Received over Serial Port” and “Sent over Serial Port”.
Qbuttons to reserve/release the *COM Port* to the server are also provided and the *Serial Port* state can be seen through a *QlineEdit*.
It is also possible to directly send any command though the “Send to Serial Port” *QlineEdit* and the “send” *Qbutton*.

Figure 46 – “Send over serial Port” - *QbuttonGroup*

6.10. Identical user interface components between *Server* and *Client*

The user interface portion shown in figure 47 is identical between the *server* and the *client,* even through the internal behavior is different because, at the *server* side, commands are interpret and sent over *RS232* to the robot but, at the *client* side, commands are sent to the *server* over TCP/IP.

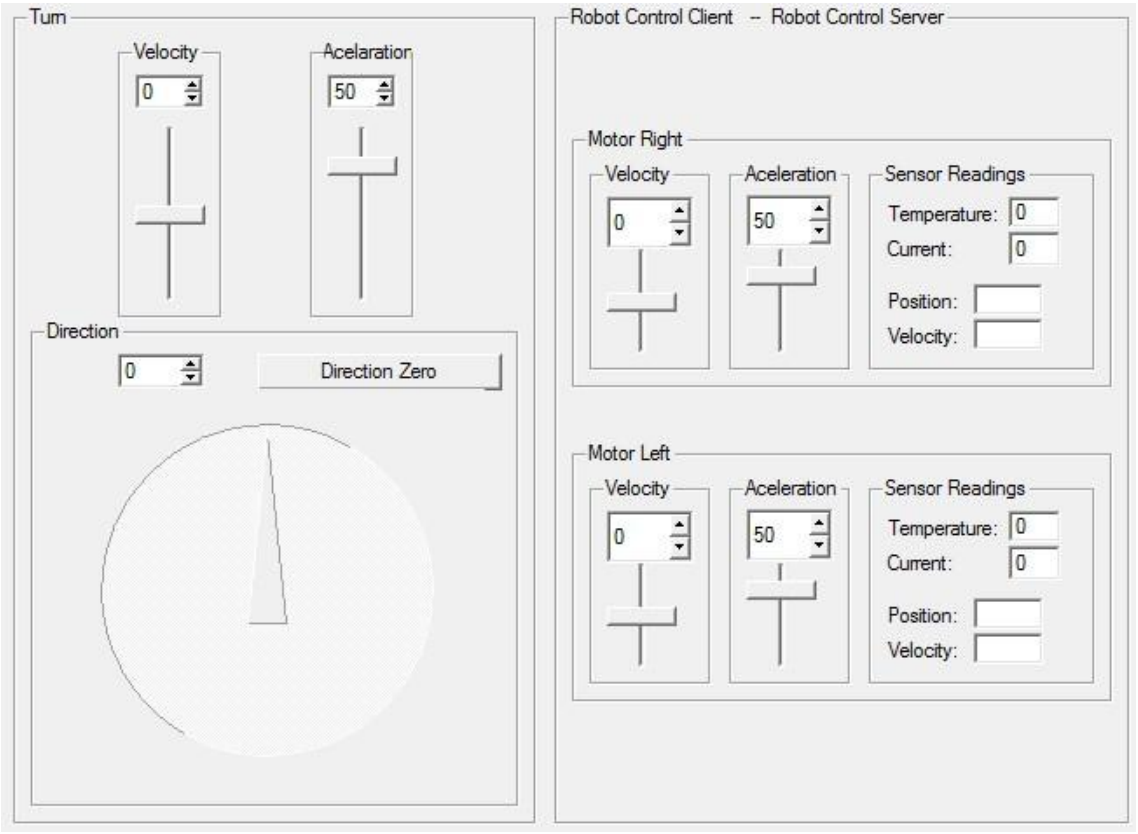


Figure 47 – Identical user interface components between *Server* and *Client*

“Motor Left” and “Motor Right” *QbuttonGroups* are used to perform the motors control individually and the “Turn” *QbuttonGroup* is used to do the control of both motors at once. Each one has *Qsliders* connected with *CspinBox* to provide a graphic interface of the desire motors velocity and acceleration.

Near each individual motor controls, the “Sensor Readings” *QbuttonGroup* is used to feedback temperature, current, position and velocity values respectively.

Typically, instead of individually control each motor, the robot is controlled by setting its velocity and acceleration and then a *Qdial* widget is used to set the robot turn direction.
“Direction Zero” *Qbutton* is used to easily put the robot running forward.

Debug facilities are achieved through two *Qbuttons* and two *QtextEdit*, as shown in *figure* 48; with simple functions presented at the code, it is possible to get feedback behaviours and it is useful to identify possible problems that might occur during the programming stage.

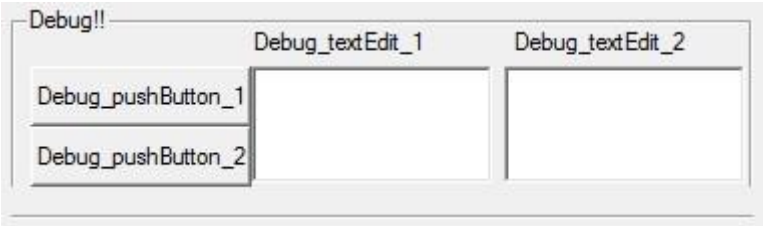


Figure 48 – Debug facilities widgets

Data transfers between *Server* and *Client* can be seen, as shown in figure 49, through two *QtextEdit* regards to “Received over the network” and “Sent over the Network”.

Figure 49 – Data transfers between Server and Client

Some developed programming examples, user interface and user options were discussed in this chapter.

7. Schematic, Proto-board, Strip-board, PCB and hardware connections

This chapter shows the physical connections and interfaces to the microcontroller as well of schematics and the control unit developed PCB's.

7.1. Control hardware schematic

The control unit schematic is shown in *figure* 50; it was designed in *Eagle* [W7].

Eagle, is a *PCB* and *schematic* design software, which is freeware making it desirable to learn and use.

Even through *Eagle* could be more "user friendly", it has an easy startup learning stage.

It is very popular software which means that it is relatively easy to get new parts.

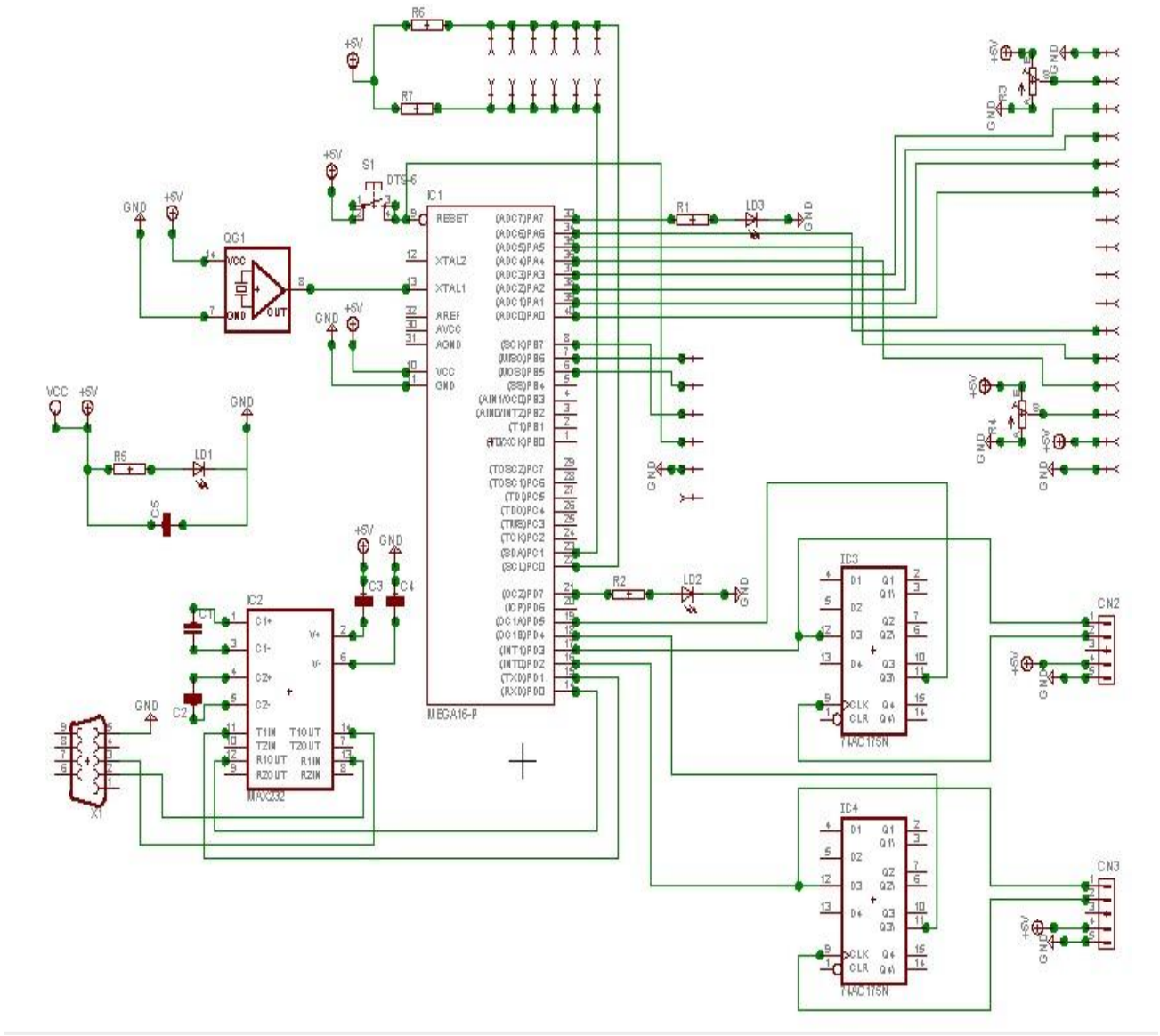


Figure 50 – Controller unit schematic

7.2. Proto-board and Strip-Board

On a first stage, all of this hardware was on a *proto-board (figure* 51), but a strip-board (*figure* 52) was also made.

The *proto-board* is still usable and it is probably desirable in future system improvements since it is easy to connect extra hardware to it and to change the components place. Even so, it is not as solid as the *strip-board*.

Proto-board organization was taken into account at the developing stage but the *strip-board* can be even more organized and compact.

Figure 51 shows the board and below is a legend to help future workers of this project to use it.

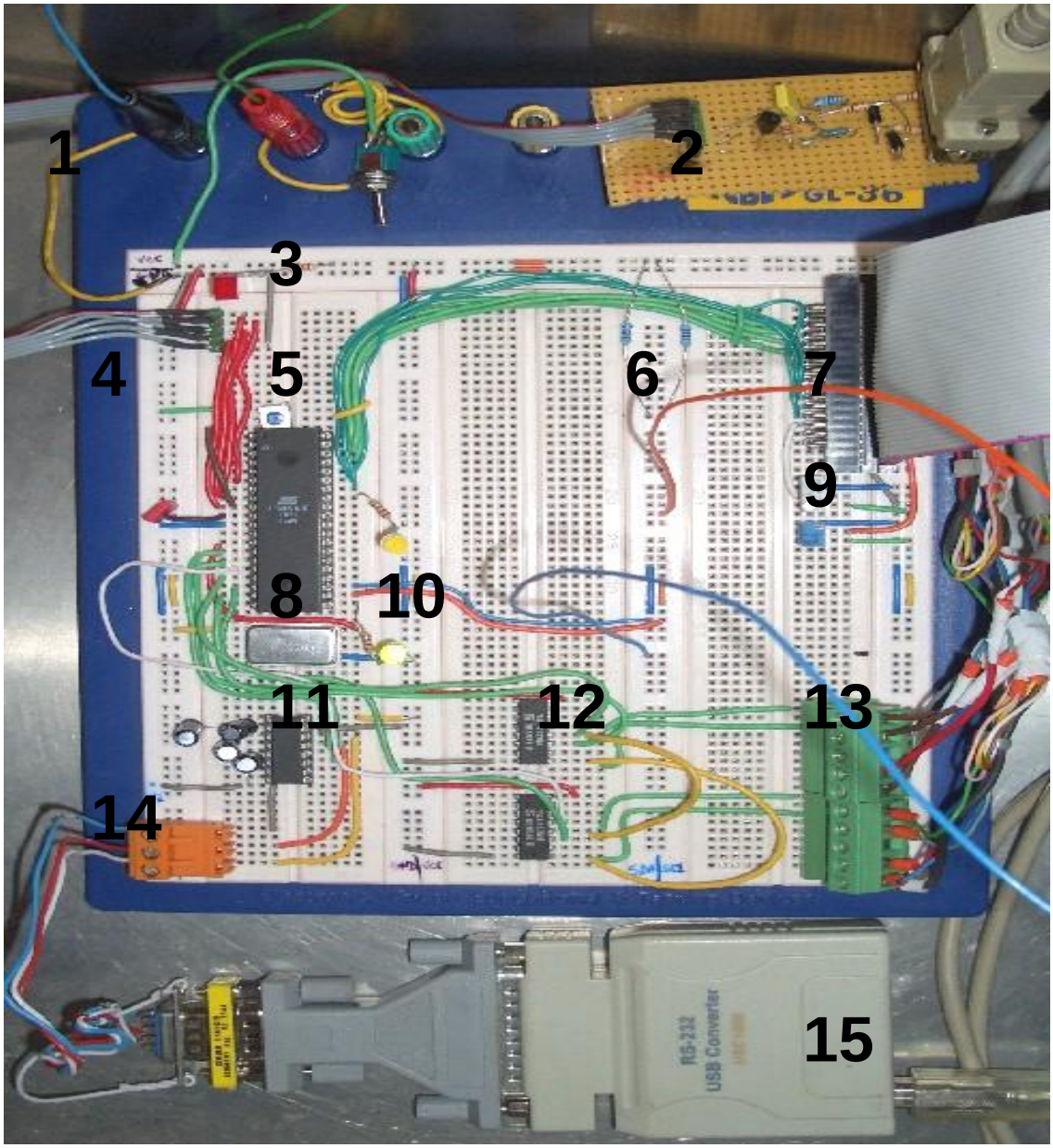


Figure 51 – Proto-Board

Figure 52 shows the *strip-board* that is more robust and much smaller than the *proto-board* but is not so flexible, even so more *I2C* devices are easily

connected to it; below it is a legend to help future workers of this project to use it.

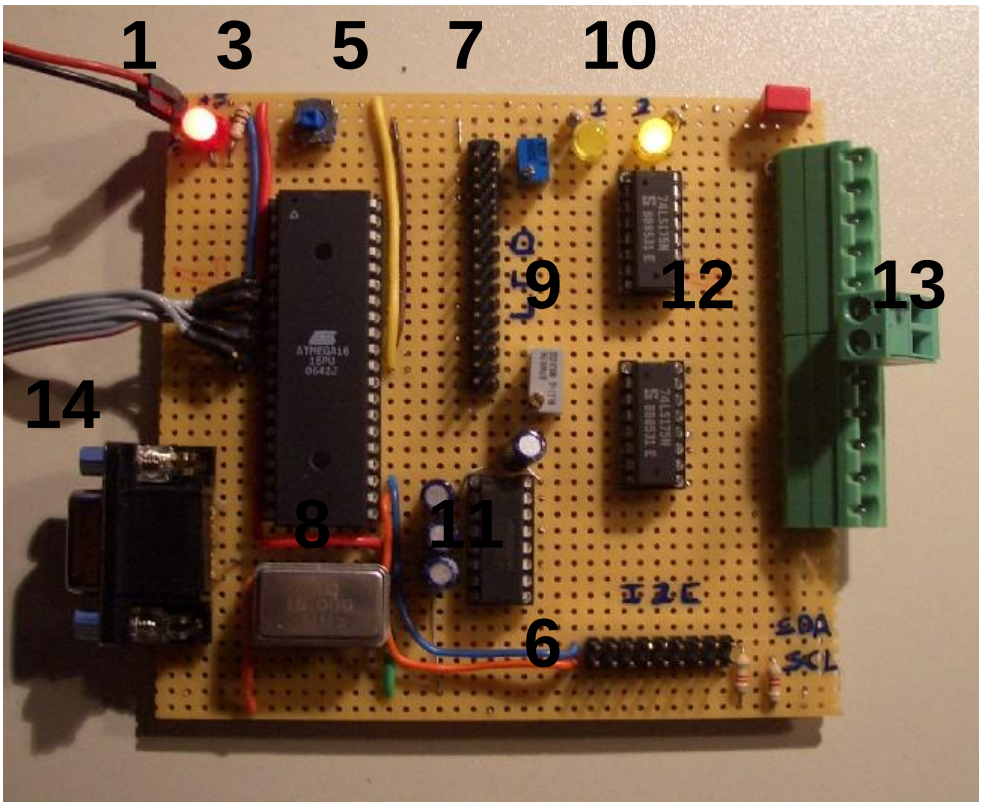


Figure 52 – Strip-Board

Figure 51 and figure 52 legends:

1) Power connectors (The red color cable corresponds to the +5V and the Black one to the ground).

2) Programmer (the red color flat cable needs to be respected).

3) Power on LED used to signal if the board is powered.

4) Proto-board connection of the Programmer (the red color flat cable needs to be respected).

5) Reset button used to reset the microcontroller.

6) I2C Bus, all I2C components are connected to this bus.
The first column/line is the SDA and has blue wires.
The second column/line is SCL and has orange wires.
(The wires of SDA and SCL from the motor driver have correspondent colors).

7) LCD Connector

8) Atmega16 Microcontroller and 16 MHz Crystal

9) Resistors to control the display backlight intensity and contrast

10) Error LEDs to debug. (Matching error (1) and error (2) at debug software) The upper one is being used when the motor driver initialization is not ok. The bottom one is used when the microcontroller looses the connections with the server. This LED is turned ON and commands to stop motor are sent to the drivers.

11) RS232-TTL converter.

12) Flip-Flop D to help the microcontroller to know the direction of the encoders.

13) Encoders connectors.

14) RS232 connector.

15) USB/RS232 converter to interconnect the server and the microcontroller.

7.3. PCB Schematic

PCB schematic was also carried out on *Eagle*, and *figure* 53 shows it.

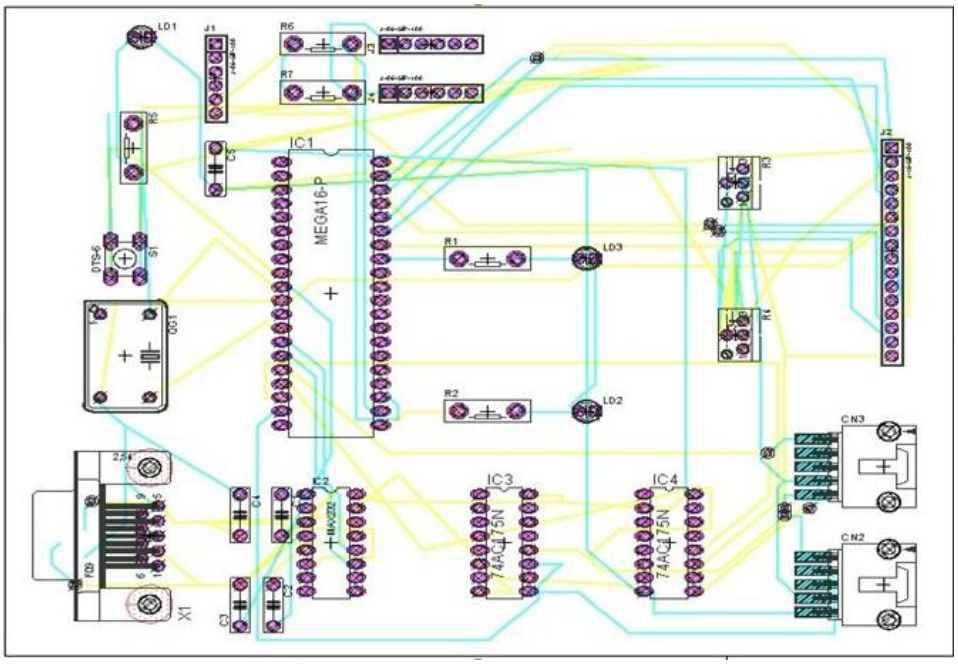


Figure 53 – PCB

7.4. 3D PCB

3D PCB, as shown in *figure* 54, was made with an *Eagle* [W8] freeware add-on help and with the *POV-Ray* that is freeware tool of 3D design.

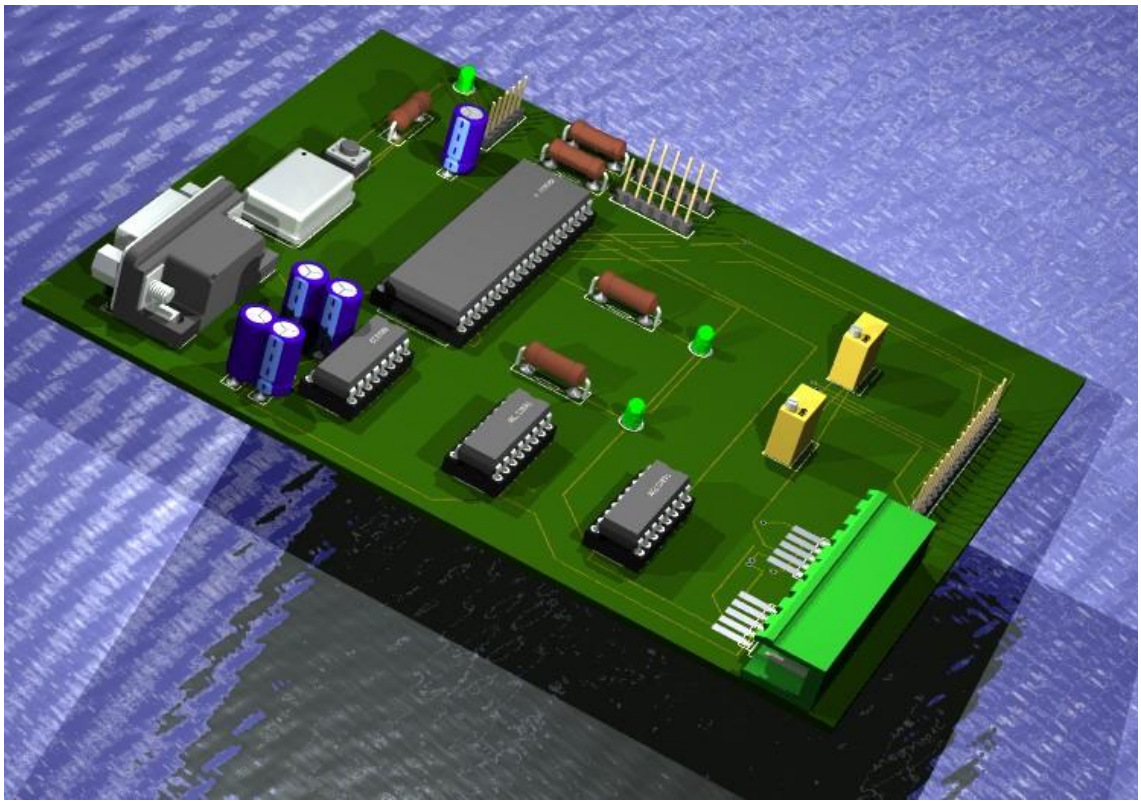


Figure 54 – 3D PCB

7.5. Hardware Connections

Figure 55 shows the D.C. regulator as well as the motor drivers, their fuses and capacitors, as well as the used robot display. Below it is a legend to help future workers of the project to use it.

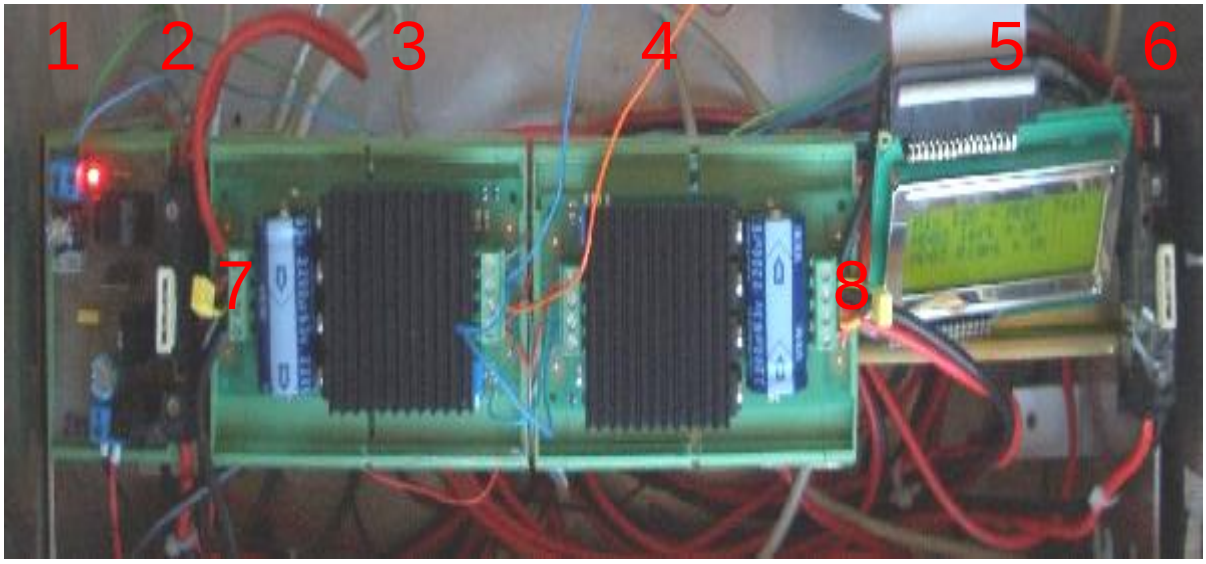


Figure 55 – Components – D.C. Regulator, Motor Drivers, Fuses and Display

Figure 55 legend:

1) 5V DC Regulator (Polarity and each side voltage is written with a pen).

2) Left driver fuse.

3) Left driver.

4) Right driver.

5) LCD display.

6) Right driver fuse.

7,8) 10n capacitors (to avoid electrical noise).

Schematics and the developed control unit PCB's are presented.

8. Software and Hardware considerations

This chapter will present the interconnections process between components, as well as the debug troubleshoots and adopted solutions to them.

One of the difficult expression points on a report is the hardware and software debug phase.

This process consists of connecting all the hardware and software components which were successfully tested individually.

What concerns hardware, tests were carried out with a small motor and with a small power supply. After that, the robot motors were tested with a 24V battery and the microcontroller was fed by the fixed power supply, now all the power is provided through the batteries.

Software debug can be split into the *Desktop* and the *Microcontroller* parts.
First the *USART* was implemented and then the *PWM* signals followed by debug *LEDs*.
Communication between the *server* and the *microcontroller* serial port were tested, followed by communication between server/client.
Everything was connected together as well as the *I2C* communications with the motor drivers.

One of these motor characteristics is the electromagnetic brake; it needs a 24V feed to release the motor. The breaks are constantly open in what this project concern, but, to achieve low power consumption, a *relay* will be added further to possibility control through the microcontroller.

Sometimes the motor driver control did crash and left the motors running. This problem was sorted out by adding a 10nF capacitor to each motor to avoid noise as well as rewriting the application with a better programming philosophy.

A “Virtual Heart Beat” was created between *Server*/*Client* and between *Server*/*Microcontroller* to avoid control loss of the robot.

This chapter presented some of the problems that occurred, as well as the adopted solutions.

9. Discussion

Some important issues are raised in this chapter.

1. **Robot has no velocity control?**

   It was planned to use *PID control* when robot showed the need for it.

   The robot shows it can go through a straight line at any speed on all practical experiments. Off-Road tests were not performed; eventually, to increase accuracy, extra control might be needed in that kind of environment.

2. **Why does the robot make use of a *Server* computer instead of only a microcontroller or a small board package like the "FOX Board" that can run a real Linux operating system with the size of only 66 x 72 mm?**

   The purpose of the system was flexibility. Investment in specific hardware was avoided because the robot has not a practical use these days. That way a laptop was used because it was cheap and flexible.

   Multi-platform capabilities were developed, so, whenever it makes sense to invest in a low-power or smaller *Server,* the software is prepared to it.

3. **Robot microcontroller makes use of *I2C* and *UART* communications. Which other choices were considered?**

   I2C and *SMBus* are popular 2-wire protocols where data transfer makes use of only two wires.

   These kinds of protocols reduce circuit complexity to a system where multiple devices are intended to be added and controlled.

   The most significant differences between *I2C* and *SMBus* are relative to timeout, minimum clock speed, voltage levels and current levels.

I2C can be more than 30 times faster and slave devices have no timeout, which means that slaves can be slower in performance. [W31]

1-WIRE bus makes use of only one wire for addressing and data transfer but achieves lower data rates and distance range, reason why it is typically used to communicate with small inexpensive devices. [W33]

These buses, and even more with the *1-WIRE* bus, increase overhead at addressing and acknowledge stages; overhead can be reduced by using more wires to address the devices.

CAN (Controller Area Network) is an advanced communication protocol, it makes use of 2 or 4 wires to data transfer and achieves a 40 meters bus instead of only 4 meter by the *I2C* but *CAN* operates only at 1Mbps instead of the 3.4Mbps of the *I2C* protocol.

SPI (Serial Peripheral Interface) is based in an 8-bit serial shift register and a programmable shift clock. It has the advantage of a better noise immunity comparing to *I2C*. Addressing *SPI* devices is made by adding an extra wire to each device, reason why this protocol increases circuit complexity as the number of devices rise in a system.
Besides that, *a*t top speed *SPI* is 3 times slower than *I2C*. [W32]

I2C is used in industry in small systems (smaller than 4 meters) and meet complexity and speed that this robot system is intended.

4. **Why was a *PCB* projected but not implemented?**

   A *PCB* was supposed to be built instead the *Proto-Board* but, due to time-limitations at the ending of this thesis it was put aside.
   Priorities/Facilities ratio had importance at this point and this project developing will be continued and improved for, probably, several years, reason why it is, at the moment, probably precocious to make it.

5. **How was the 10nF capacitor of each motor chosen?**

That was a value chosen by the motor driver manufacturer and it is specified at the *MD03* datasheet.

6. **In a remote or hard access environment reconnecting the server could be an issue. What if some communication problem happens, how is the reconnection made?**

   "*Virtual Heart Beat*" handles this situation; it stops the robot when no *ping* is received at the specified time but whenever it receives a behindhand *ping* from the microcontroller or from the *Client*, the process relaunches again.

7. **If several I2C master are connected to the system, what happens when two of them communicate at the same time?**

   At the moment only one I2C master is implemented but, if for some reason, another master exist they can interact with each other by making use of arbitration logic and the "Bus busy detection" theory. [W34]

8. **Besides the critical duty of robot stopping when the "Virtual Heart Beat" processes indicate, which other critical duties have the microcontroller?**

   It has the duty of calculating robot speed and position by reading each motor encoder with accuracy and has the duty to be the *I2C Master* which will generates the clock signal and address a slave when needed.

9. **Does the robot have some proximity sensor or any device to sense external environment?**

   Not at the moment. It will have in new improvements during further developments.

10. **Does the robot have any reference point when inserted in one environment?**

    The reference is carried out when the robot is turned on.

At start, all variables are set to zero and values relative to how much each motor run are relative to that start moment.

By adding other sensor to the system, the philosophy of setting the reference can be different than the actual.

11. Why has the *server* a graphical interface?

Could the graphic programming language be implemented only at the *Client* side?

A graphical user interface was implemented at the server because, that way it is possible to make "*Access Restrictions*".

The *Server* can have full control of the robot, but the *Client* is able only to ask data to the *Server*, and it is the *Server* who decides what the *Client* can do.

At developing stage, the *Server* can be programmed to do anything and a "*Remote Desktop Environment*" can be used to control the robot without any restrictions.

10. Conclusion / Further work

A final and informal presentation was carried out at the APS.
As main conclusions it can be said that the robot showed high stability with either fast or smooth control. The commands sent by the *client* are correctly interpreted and technical problems (as loss of internet connection) were successfully passed.

The system was built from the scratch except of the mechanical structure and all the problems that occurred during the developing phase were sorted out which gave the solid and confident characteristic of the robot.

The mobile robot was developed, with communications parts between all components.
The use of *I2C* gives a proof of efficiency, fast and expansible concept.
The software developed at desktop level, by use of Qt libraries, makes the system portable and flexible and the low level developed software at the microcontroller unit makes the system fast at duties as the robot position, speed and acceleration calculation, *I2C* communication and leave the desktop free for other duties.

Working abroad was a grateful experience, which allowed me to know people, other institutions, to meet different cultures, languages and to achieve more technical knowledge and working skills.

Bibliography and WWW References

• Bibliography

[B1] Michael Barr, Anthony Massa, ” Programming Embedded Systems”, O'Reilly, ISBN: 0-596-00983-6, cp8, cp9, cp13
Here you can have an approach of embedded systems processing.
Different types of Interruption handling on a microcontroller as showed
A PWM tutorial with and some simple examples are showed.
A brief I2C explanation is presented.

[B2] *Lewin A.R.W. Edwards*, ” Open-Source Robotics and Process Control Cookbook - *Designing and Building Robust,Dependable Real-Time Systems*”, Newnes, ISBN: 0-7506-7778-3

• References WWW

[W1] http://www.aps-mechatronik.de/
Main page of APS - European Centre for Mechatronics
Here it is possible to find information referring members of the center, investigation developing projects, etc.
(Accessed on March, 2008).

[W2] http://www.fb6.rwth-aachen.de/en/1.php
Main page of Faculty of Electrical Engineering and Information Technology of RWTH University.

Here it is possible to find information referring members of the department, teaching activities and investigation developing projects, etc.
(Accessed on March, 2008).

[W3] www.dei.uminho.pt
Main page of Industrial Electronics and Computers department of University of Minho. Here it is possible to find information referring members of the department, teaching activities and investigation developing projects, etc.
(Accessed on March, 2008).

[W4] http://www.atmel.com
Page of Atmel.
Here you can find information about microcontrollers and the datasheet about atmega16, debuggers (AVRStudio) and other Atmel product.
(Accessed on March, 2008).

[W5] http://trolltech.com/products/qt
http://trolltech.com/products/qt/features
Page of QT.
Here you can find information about QT, help documentation with libraries specifications and features.
(Accessed on March, 2008).

[W6] http://www.lancos.com/prog.html
Page of PonyProg flash programmer.
Here it is possible to find the newest version of the software as well as different solutions to the programmer hardware.
(Accessed on March, 2008).

[W7] http://www.cadsoft.de/
Page of Eagle

Here you can download the freeware version of Eagle as well as many libraries of components
(Accessed on March, 2008).

[W8] http://www.matwei.de/doku.php?id=en:eagle3d:eagle3d
Page of Eagle 3D
Here you can download the freeware and open source version of Eagle 3D add-on as well of documentations about to use it.
(Accessed on March, 2008).

[W9] http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf
Datasheet of the microcontroller atmega16 provided by the manufacturer Atmel.
Here you can find information about the microcontroller atmega16 and some example code.
(Accessed on March, 2008).

[W10] http://www.AVRfreaks.net
Here you can find a large collection of projects suitable to learn you more about the AVR.
(Accessed on March, 2008).

[W11] http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf
AVR Studio
(Accessed on March, 2008).

[W12] http://www.dimensionengineering.com/Sabertooth2X10.htm
Here it is possible to find *Sabertooth* specifications
(Accessed on March, 2008).

[W13] http://societyofrobots.com/batteries.shtml
Page of Society of Robots.
Here you can find information about robots and how they are made and a nice tutorial about the different types of batteries

(Accessed on March, 2008).

[W14] http://www.embedded.com/story/OEG20010718S0073
Page of Embedded Systems Design,
Here you can find a nice tutorial about I2C functions.
(Accessed on March, 2008).

[W15] http://www.nxp.com/acrobat_download/applicationnotes/AN102161.pdf
Phillips manual about *I2C*.
(Accessed on March, 2008).

[W16] http://jump.to/fleury
Page of Peter Fleury
Here you can find libraries to use I2C and LCD with the atmega16 microcontroller
(Accessed on March, 2008).

[W17] http://en.wikipedia.org/wiki/I2C
Page of Wikipedia.
Here you can find nice tutorial about I2C functions.
(Accessed on March, 2008).

[W18] http://www.ormec.com/mktdocs/encres.htm
Page of ORMEC's - Motion control solutions
Here you can find nice tutorial about encoders.
(Accessed on March, 2008).

[W19] http://lab.artematrix.org/papers/Homebrew_Shaft_Encoder.pdf
Encoder
(Accessed on March, 2008).

[W20] ftp://ftp.ni.com/pub/devzone/pdf/tut_4623.pdf
Page of the *national instruments* with some principles of encoders
(Accessed on March, 2008).

[W21] http://www.faulhaber-group.com/n390840/n.html
Page of the manufacturer of this robot gearheads.
This you can find a nice tutorial about choosing the gearheads and how are they composed.
(Accessed on March, 2008).

[W22] www.crouzet.com/catalogue_web/pdf/ENG/ndb12_eng.pdf
Brief Description about D.C. motors
(Accessed on March, 2008).

[W23] www.robotstorehk.com/md03tech.pdf
Motor driver MD03 Datasheet
(Accessed on March, 2008).

[W24] www.irobot.com
iRobot Homepage
(Accessed on March, 2008).

[W25] http://www.gizmag.com/go/7151/
iRobot in Iraq
(Accessed on March, 2008).

[W26] http://www.gizmag.com/go/5098/
SWORDS Robot
(Accessed on March, 2008).

[W27] http://www.gizmag.com/go/3550/
Tallon Robot
(Accessed on March, 2008).

[W28] http://www.linuxdevices.com/articles/AT3782871866.html
http://www.activrobots.com/ROBOTS/p2at.html
2008, MobileRobots Inc. (Accessed on March, 2008).

ActivMedia Mobile Robot (Pioneer and other types of robots)

[W29] http://linuxdevices.com/news/NS8152651349.html
914 PC-Bot
(Accessed on March, 2008).

[W30] http://www.gizmag.com/go/7208/
Remote-controlled robot uses thermal imaging to detect and eradicate termites.
(Accessed on March, 2008).

[W31] http://www.maxim-ic.com/appnotes.cfm/an_pk/476
I2C and SMBus comparing.
(Accessed on March, 2008).

[W32] http://www.ucpros.com/work%20samples/Microcontroller%20Communication%20Interfaces%201.htm
I2C and SPI comparing
(Accessed on March, 2008).

[W33] http://en.wikipedia.org/wiki/1-Wire
1-Wire Interface specifications
(Accessed on March, 2008).

[W34] http://www.i2c-bus.org/multimaster/
I2C Multi-Master Environment.
(Accessed on March, 2008).

Table of figures

Table of tables

Table of flowcharts

Table of abbreviations

I/O	Input / Output
TCP/IP	Transmission Control Protocol / Internet Protocol
I2C	Inter-Integrated Circuit
TWI	Two Wire Interface
DC	Direct Current
PWM	Pulse-width modulation
USART	Universal Synchronous Asynchronous Receiver Transmitter
APS	European Centre for Mechatronics
DDR	Data Direction Register
ASCII	American Standard Code for Information Interchange
A.K.A.	Also Known As
AGM	Absorbed Glass Mat

Attachments – Motor Specifications

The robot has two motors provided from the manufacture ENGEL, the series is GNM5480E and the motors are typed “Permanent Magnets, Direct Current” they are coupled with gear-heads and the characteristics can be seen at table 1 and the dimensions at figure **Error! Reference source not found.**.

Nominal voltage	UN	24	Volt
Armature resistance	R	0.106	Ω
Nominal output power	P_2	250	W
Efficiency	η max	85	%
No-load speed	no	3,267	rpm
No-load current	Io	1,435	mA
Stall torque	M_H	1,005	oz-in
Friction torque	M_R	14.16	oz-in
Speed constant	k_n	137	rpm/V
Back-EMF constant	k_E	7.30	mV/rpm
Torque constant	k_M	9.87	oz-in/A
Maximum peak current	k_I	115	Amps
Rotor inductance	L	0.33	mH
Nominal speed		3,000	rpm
Nominal torque		112.71	oz-in
Mechanical time constant	Tm	11.6	ms
Rotor inertia	J	52.4	x10-3oz-in-sec2
Thermal resistance	Rth1/ Rth 2	1.8	°C/W
Thermal time constant	τ w	40	minutes

Motor weight			lbs.
Maximum ambient temperature		40 (104)	°C (°F)
Motor operating temperature range		–20 to 100 (–4 to 212)	°C (°F)

Table 12 – Robot motor characteristics

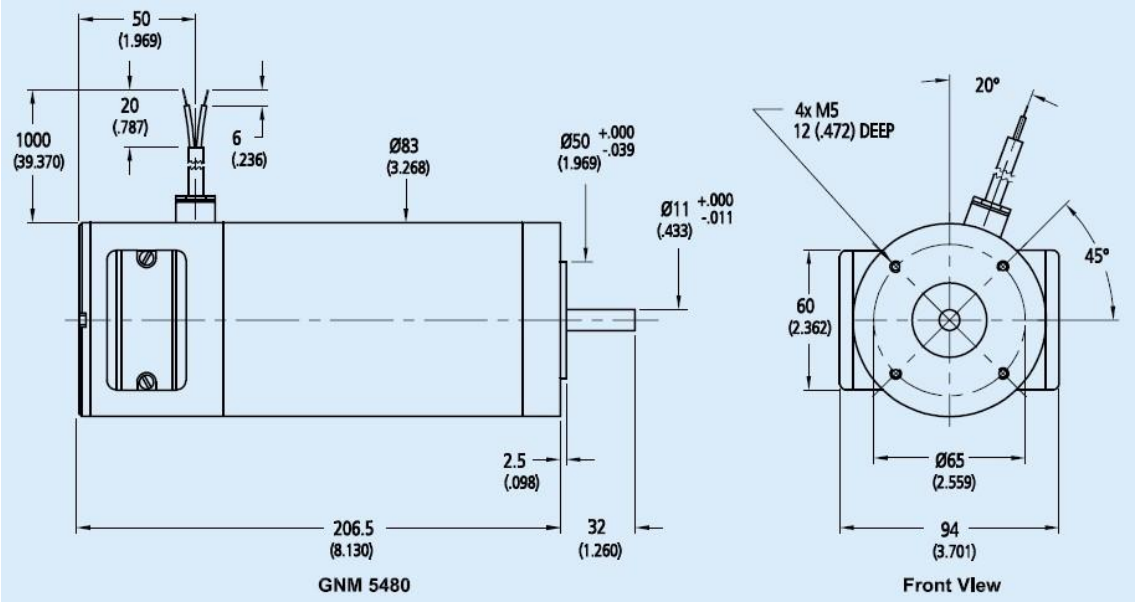


Figure 56 – Motor Dimensions