Fachhochschule
Dortmund
University of Applied Sciences and Arts

Fachhochschule Dortmund

Master's Thesis

Development of a real-time software architecture for AMiRo robot based on the operator controller-module

---

Author:
Hector Gerardo Munoz Hernandez

Supervisors:
Prof. Dr. Carsten Wolff
Uwe Jahn

A thesis submitted in fulfilment of the requirements for the degree of Master of Engineering (M.Eng.) in Embedded Systems for Mechatronics

September 16, 2018

Contents

List of Figures

List of Tables

Abstract

The objective of this project is to develop and implement a software architecture that fulfills the real-time system requirements. The development and implementation are done for the Autonomous Mini Robot or AMiRo. This thesis starts discussing the hardware structure of the robot, and the review and analysis of the AMiRo OS that was done in the University of Bielfield. The AMiRo OS was replaced by a software solution that uses the *Operator Controller Module* approach. This software solution is called the DA_AMiRo project and it discussed over the next few chapters as it is the main part of the thesis. The purpose of this software architecture is to build a real-time scheduler for the values of the sensors and actuators that AMiRo has and are communicated via the Controller Area Network protocol. There are also some sample applications and demos that show the functionality of the DA_AMiRo. The thesis ends with a comparison between the AMiRo OS and the DA_AMiRo project. The entire DA_AMiRo project can be found in the gitlab repository [18]. As for the AMiRo OS project, the reader can also get access to the complete codes in the AMiRo Wiki [26].

Chapter 1

Introduction

1.1 Real-time Operating Systems

The necessity to meet deadlines in today's projects is growing constantly. Most applications consist of different tasks that can be executed in parallel, have different priority, have different repetition period, and frequently use shared resources. Using a multi-threading approach, however, can create a challenge in terms of performance, stability and time of implementation.

An operating system is a software component of a computer system that is responsible for the management and coordination of activities and the sharing of the resources of the computer [8]. "A real-time operating system perform these tasks, but is designed to run applications with a very precise timing and a high degree of reliability. This can be especially important in measurement and automation systems where downtime is costly or a program delay could cause a safety hazard" [11].

In other words, a real-time operating system is an operational system that provides certain capability according to preset timing constraints [2]. This is achieved through a scheduler that is designed to provide a deterministic execution pattern. And so, the RTOS are used in systems with critical timing requirements, when each task should be executed within a restricted time interval [2].

When it can be guaranteed that a task will never exceed a maximum amount of time in being completed, it can be said that the operative system is "hard real-time". If it can be guaranteed that a task will most of the time not exceed a maximum amount of time in being completed, it can be said

that the operative system is "soft real-time". An airbag system is an example of a "hard real-time" system, where the task has to be guaranteed to execute always within a time limit. For a "soft real-time" example, a video streaming can be considered, where an occasional loss of data can be accepted because it does not compromise the whole functionality [11].

In the real-time operating systems the managing of the tasks is crucial. To this end, programmers have to decide which task is running at certain time and how other tasks can preempt the processing resources if their priorities are higher. In other words, manage the schedule of the tasks so that they meet with their deadlines.

Some of the main characteristics of a RTOS are the following:

- Determinism: An application timing can be guaranteed within a certain margin of error.
- Soft and Hard Real-Time: The validity of data after meeting a deadline. In soft real-time the deadline is not that crucial but in hard real-time the meeting of the deadline is very important.
- Jitter: Which is the amount of error in the timing of a task over subsequent iterations of a program or loop [2].

1.2 Operator Controller Module

The DA_AMiRo software architecture is based on the OCM [7] software architecture approach. In this structure there are three basic controllers: Internal Controller, Reflective Operator, and Cognitive Operator. In order to have a more precise overview of how these three controllers work, figure 1.1 is explained by a direct quote from the Wiki:

"Refering to figure 1.1, the Controllers (red) are used to connect all sensors and actuators. The Controllers need to be configured and programmed once, so the software on the controllers is mostly fixed and does not need to be flexible. The Reflective Operator(s) (yellow) control the Controllers. The Reflective Operator's software is the adaptable part of the distributed system - it changes from application to application. These Operator Controller Module(s) are located on the robot itself and let it act autonomously. The Cognitive Operator is hosted on a server and is used to optimize, program and interact with the user. The Cognitive Operator is connected via a wireless connection and can (only) interact with the Reflective Operator(s)."[13].

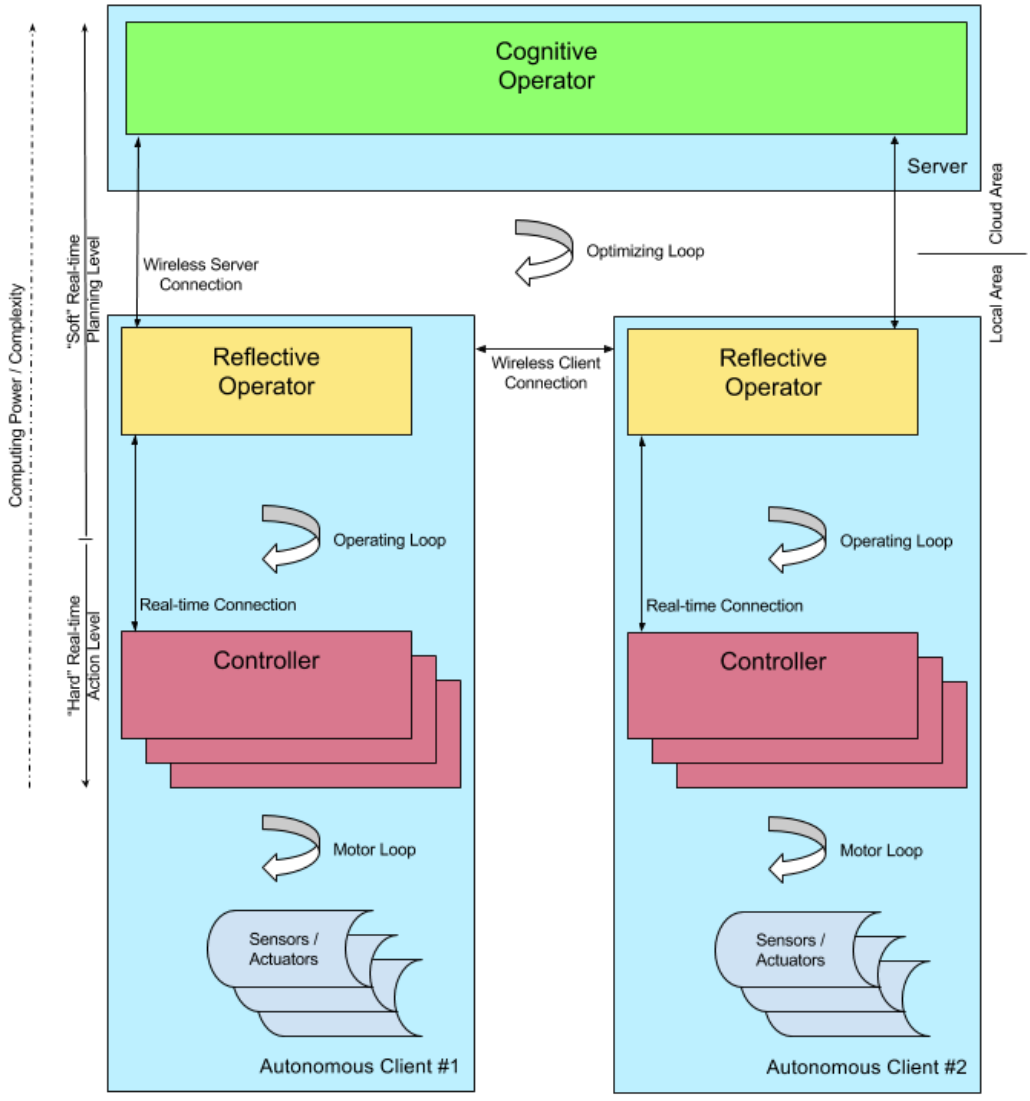


Figure 1.1: OCM software arquitecture for the DAEbot[13].

The OCM was introduced by a group of Professors in the University of Paderborn. The concept aimed to structure and design a reconfigurable controller systems [7]. As mentioned before, there are three different levels in the OCM. The first one is the lowest level of the OCM also simply known as the Controller. This level interacts directly with the plant of the system, which means that the control signal is produced and measured here. It is also necessary that the software processing in this layer is quasi-continuous so that the measured values are processed in real-time conditions [7].

The second layer is called the reflective operator. It is in this layer where the monitoring and the controlling routines are executed [7]. This layer does not have access to the actuators directly, because the Controller is already doing this. The expected result is that the Reflective Operator modifies the Controller, sometimes also changing between different pre-established Controller configurations. This level also requires a quasi-continuous operations like a continuous adaptation algorithm or a watchdog. This operator also has to operate under hard-time constraints because its relation with the Controller. In short, this layer serves as a interface between the Cognitive Operator and the Controller [7].

The third layer is the Cognitive Operator. On this layer the system gathers information about itself and its environment to improve itself. This recollection of information can be done by applying various methods such as learning and model-based optimization among others. In other words, it is in this layer where the self-improvement takes place [7].

This project compares the AMiRo OS with the OCM based architecture of the DA_AMiRo. The controller is the responsible for commanding the internal sensors of the STM32F32 boards inside DA_AMiRo in real-time. This sensor information is then retrieved by the Reflective operator board with a few demo applications that will be explained later in this document inside section 6.3.

1.3 AMiRo

The AMiRO project, which stand for Autonomous Mini Robot was started in the University of Bielefeld[1] by Stefan Herbrechtsmeier and developed by Thomas Schöpping in collaboration with few others. The motivation for this project was to use small embedded systems in order to create a small robot

[1]See: https://www.uni-bielefeld.de/

that is easy to transport, usable on a table or the floor, appealing for young people, that is created with a modular approach is able to be extendable and customizable. With all of these aspects in mind, the result should be also a powerful tool for research and education [5].

The preliminary work was conducted in the University of Paderborn in the form of the BeBot mini robot [5]. The intended architecture of both robots is a modular one, which means that it has more than one board functioning as a unity. In AMiRo's case, said architecture consisted on three boards, DiWheelDrive, Power Management, and Light Ring that will be discussed in the next chapter, and two main extension boards: the Cognition board and the Image Processing board. The former will be explained in section 2.1.4, but the latter will not be discussed in this work, as it was not needed for the current reach of the project. If the reader is interested in the Image Processing board, he or she can refer to the project presentation given by Prof. Herbrechtsmeier [6].

Physically, AMiRo has a cylindrical shape with a diameter of 100mm, see figure 2.1. It is covered by a chassis that does not cover the DiWheelDrive board's bottom as this board has the proximity sensors pointing towards the ground. There are also two wheels with two separate motors directly connected to the DiWheelDrive board and two metal sliders for the robot stabilization in flat surfaces [25].

The chassis has three more openings for the following connections: USB serial that goes directly into the DiWheelDrive board, the power supply, and the charging pins. Additionally, when the Cognition board is attached, there are three more possibilities to physically interact with the robot: USB serial, USB dongle for wireless connection with the Linux OS, and a micro USB cable for connecting also with the Linux OS which is used in this board. The Cognition board will be later discussed in section 2.1.4.

These characteristics make AMiRo a very compact and portable robot. Every board is designed to be modifiable, stackable and exchangeable with custom elements. All boards communicate with each other with the Controller Area Network protocol [25]. This way every board can read any message from the CAN bus or write into it every time it is necessary.

Chapter 2

Architecture of AMiRo

The AMiRo project was started in 2015 and ever since, the repository supporting the AMiRo project has had several updates [26]. For future references, the version used as the basis for the present work was the number 1.0 stable. The AMiRo's repository[1] consists of three basic folders: AMiRo bootloader, AMiRo Operating System, and ChibiOS.

Although this software architecture will be explained with more detail in section 2.2, it is worth mentioning that AMiRo's bootloader was used as it is in version 1.0 from the original repository in order to configure the computer for programming the AMiRo. The ChibiOS was also used as it is in version 1.0 from the repository [26]. The AMiRo Operating System was hugely modified to meet the requirements of the present work, which can be found under the DA_AMiRo git repository[2] [18].

In the left picture of figure 2.1 the AMiRo can be seen with the three basic boards and the chassis on, with a human hand for visualizing the scale. In the right picture however, the robot has its three basic boards and one extension board can be seen without chassis. In order from bottom to top, the displayed boards are: *DiWheelDrive*, *PowerManagement*, *Cognition board*, and *LightRing*. It is also possible to see the wheels, the motors, and the batteries attached to their corresponding boards. These structure will be explained in the next section.

[1] See: https://openresearch.cit-ec.de/projects/amiro-os/wiki
[2] See: https://gitlab.idial.institute/DAdevs/DA_AMiRo.git

Figure 2.1: AMiRo with and without chassis [25].

2.1 Hardware Architecture of AMiRo

As mentioned in the previous chapter, the AMiRo is equipped with three basic board modules and two extension boards. In this section each board will be briefly discussed with a special emphasis in the functionalities that are of interest for this thesis. For a more complete discussion about the boards or the AMiRo in general, the reader is invited to read the documentation referenced in this work, specially [23], [25], [24], [5], [6] whic can be found on the CITEC website or on the repository of this work on gitlab [18].

The functionality of the sensors used in the project will be briefly explained in this chapter. For the explanation on how to read the output of the sensors that are sent via the Controller Area Network protocol, the reader is invited to jump directly into section 4.1 for reviewing the CAN convention used in the entire project and section 4.2 for understanding each sensor's output.

In figure 2.2 a schematic of the AMiRo can be seen, with the three basic boards *DiWheelDrive*, *PowerManagement*, and *LightRing* but also the exten-

sion boards *ImageSensor*, *Cognition*, and *Image Processing*. The electrical interfaces are also displayed. For the current work, the most important thing to note in this image is that every board is able to talk to the CAN bus.

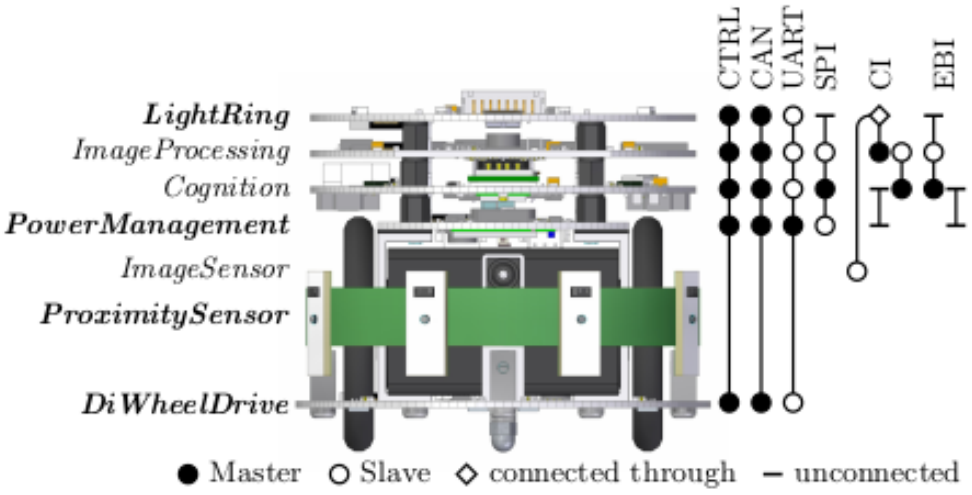


Figure 2.2: Inner parts of AMiRo with all extension boards, and overview of the electrical interfaces [25].

2.1.1 DiWheelDrive

The DiWheelDrive is the bottom board of the AMiRo. It is equipped with an ARM Cortex-M3 based STM32F103 MCU from STMicroelectronics [25]. It is through the DiWheelDrive's UART interface that the AMiRo's three basic modules get programmed. Each board gets programmed at a time with help of the AMiRo bootloader which will be discussed in section 2.2.1.

This board has two motors of 1W DC. The purpose of having two motors is to be able to have a differential kinematic to allow flexible movements [25]. Among other sensors and actuators, the ones that are in this board and are being used in this thesis are: three axis gyroscope, accelerometer, and magnetometer, floor proximity sensors, the two motors, and optical motor encoders. The functionality and some specifications of the above mentioned sensors and actuators will be discussed in the next subsections.

As mentioned before, the DiWheelDrive board has a Cortex-M3 MCU for calculating such things as motion control, odometry, and dead reckoning algorithms at high sampling rates [25]. This board can be accessed by the other boards via the Controller Area Network protocol or via USB UART interface directly with the user. It is important to mention that every board has also a USB UART interface to communicate directly with the user but all of them are covered by the chassis except for the DiWheelDrive board [25].

The *DiWheelDrive* board ready to be mounted into the robot can be seen in figure 2.3. In this figure the board can be seen from both, top and bottom perspectives.

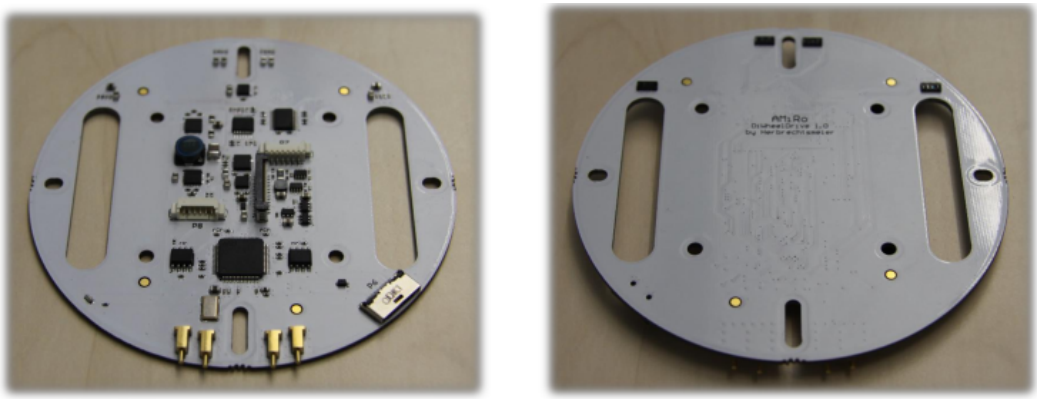


Figure 2.3: Top and bottom view respectively of the DiWheelDrive board [5].

Gyroscope

The gyroscope is a sensor that measures rotational motion. The units of the resulting measurement are revolutions per second or degrees per second. As a quick example, in a balancing robot a gyroscope can be used to measure how much the robot has rotated from the desired position and this information can be sent to an actuator to make corrections in order to reach the desired angle [30]. A gyroscope normally consists of a mass that moves with constant angular momentum, so when the gyroscope is tilted the axis of rotation of said mass reacts to the applied rotational movements making the axis of rotation tilt. This tilting leads to a displacement of the capacitance fingers

inside the sensor which changes the capacitance. Finally when compared to a reference capacitance level, the angular velocity can be determined [5].

The gyroscope used in the DiWheelDrive board is the l3g4200d from STMicroelectronics. This sensor communicates via SPI with the processor. This sensor is configured to have a resolution of $+/-500dps$ in accordance with the user's manual [16] and it is possible to obtain the output in degrees per second, micro degrees per second, revolutions per second, and micro revolutions per second with the current code. The actual units are degrees per second but the user can easily change the output selecting a different function from *updateSensorVal()* function inside DiWheelDrive.cpp. For a more clear explanation on the code structure, see section 2.2.3. The sensor communicates with the processor via SPI communication.

Accelerometer

The accelerometer is a sensor that measures the acceleration of an object. The units of the resulting measurement are meters per squared second or in g forces, where g as a unit equals $9.81m/s^2$ [31]. It is typically a system composed by a mass, a spring, and a damper. The mass moves according to the applied acceleration and changes the resulting capacitance that can be sent as an output for further reading and processing [5].

The accelerometer used in the DiWheelDrive board is the lis331dlh from STMicroelectronics. This sensor is configured be scaled with a $+/-8g$ factor in accordance with the user's manual [17]. The output is in g units and the sensor is being communicated via SPI with the processor.

Magnetometer

The magnetometer is a sensor that measures the magnetic field for all three physical axes [34]. This sensor works thanks to three principles. The first one is called the *Anisotropic magneto resistance effect* discovered in 1857 by William Thomson, in which an external magnetic field can make the sensor change the electrical resistance of a ferromagnetic material. The electrical resistance will be at a maximum value when the direction of the electrical current is parallel to said magnetic field [5].

The second principle is called *Fluxgate* and it is a way of calculate the vectorial measurement of a magnetic field invented by Friedrich Förster in 1937. The method starts with two cores wrapped by two coils of wire. There

should be an electrical current driving the core through cycles of magnetic saturation. The resulting electrical current of the second coil will depend on the external magnetic field [5].

The last principle is called the *Hall effect* discovered in 1897 by Edwin Hall. In this principle can be visible when an electric current flows through a conductor in a magnetic field, the magnetic field will exert a transverse force on the moving charge carriers which tends to push them to one side of the conductor. The result is a voltage difference between both sides of the conductor [10].

The magnetometer used in the DiWheelDrive board is the hmc5883l from STMicroelectronics. This sensor is configured to have a field resolution of $+/-5Gauss$ according to the user's manual [9] and the current output is set to be in micro Gauss units. The sensor is being communicated via I2C protocol with the processor.

Floor Proximity Sensor

The floor proximity sensor used is the "Fully Integrated Proximity and Ambient Light Sensor With Infrared Emitter", part vcnl4020 of Vishay [32]. Four of these sensors are in the bottom of the AMiRo, programmed as light sensors, while other eight of these sensors are being used as proximity sensors and ambient light detectors coordinated by the Power Management board, see section 2.1.2 for more information.

The ambient light sensor measures the intensity of light and its is measured in luminance. In order to do this the sensor usually consists of a photoresistor or a photo-sensitive material, outputting a resistance that can be measured and compared to a default value to estimate the luminance [15].

The goal of providing AMiRo with these sensors attached to its bottom is to detect changes in the color of the floor. An example application can be for the AMiRo to follow a line or path painted on the ground, by making the robot recognize when it is over one color and when it is not.

Motors

As previously described in section 2.1.1, the AMiRo has two DC motors of 1W each to offer a differential kinematic and to allow agile movements [25]. These motors are connected to the DiWheelDrive board which means that

this board controls the velocity of the motors and also knows the actual velocity of the motors.

The velocity of the motors can be set anytime from the Cognition board and the actual velocity of the AMiRo is one of the sensor values that can be asked from AMiRo's internal controller. The user can set and view the velocity in $\mu m/s$ or in $\mu rad/s$.

Encoder

AMiRo has optical encoders as well, that allow the user to know how much the motors have turned from a certain checkpoint. This data can be used to know how far the AMiRo has moved from a certain point. This sensor returns the x coordinate in μm, the y coordinate in μm, and the orientation of the wheels in μrad.

2.1.2 Power Management

The Power Management board is the central basic module of AMiRo. This board is responsible for the power supply and elementary behavior of the system. This is why it has the most powerful microcontroller. This board has an ARM Cortex-M4 based STM32F405 from STMicroelectronics. The characteristics from this microcontroller were needed to ensure a fast reaction times on critical system events and still provide enough headroom to additionally perform basic behaviors like obstacle avoidance, or homing [25].

There are two lithium-ion batteries connected to the Power Management board that power the whole system, and there are also some sensors on this board. Among the sensors that this board has, the ones that are being used by this project are: eight proximity sensors that act as proximity sensors and also as ambient light sensors, sensors for knowing if the charging cable is connected, the remaining battery charge or the time for a complete battery charge, the time for this both events to be completed, and the actual current consumption [25].

The *PowerManagement* board ready to be mounted into the robot can be seen in figure 2.4. In this figure the board can be seen from both, top and bottom perspectives.

Figure 2.4: Top and bottom view respectively of the Power Management board [5].

Power Status

For the tracking of the power status, AMiRo is equipped with a bq27500 Impedance Track from Texas Instruments that allows the robot to obtain information such as remaining battery capacity, state-of-charge, run-time to empty, battery voltage, and temperature [12].

Proximity Ring

As seen in section 2.1.1, the eight vcnl4020 sensors in connected to the Power Management board are functioning as ambient light sensors and also as proximity sensors. For this last functionality, the sensor has a built-in infrared emitter and photo-pin-diode, having a maximum relying measurement of 200mm [32]. These sensors are connected in a ring-fashion way so that the AMiRo has eight sensors throughout its entire circumference, making it able to detect obstacles in all horizontal directions. The units of the output are also in mm.

The *Proximity Ring* ready to be mounted into the robot can be seen in figure 2.5. The proximity ring gets attached to the *Power Management* board as mentioned before.

Figure 2.5: Proximity Ring view [5].

2.1.3 Light Ring

The Light Ring board is the top-most board in the AMiRo. This board has a STMicroelectronics ARM Cortex-M3 based STM32F103 MCU. It contains eight RGB LEDs arranged in a circular way corresponding to the Proximity Ring discussed in last section. The LEDs can be set from the Cognition board, setting each color of the RGB channels [25].

The *LightRing* board ready to be mounted into the robot can be seen in figure 2.6. In this figure the board can be seen from both, top and bottom perspectives. As it can be seen in the image, the LEDs are positioned equidistant in a circular fashion. This board is the upmost board which is covered by a translucent material, allowing the user to see the colors of the LEDs.

2.1.4 Cognition Board

This board was thought as an extension board which is no longer from the basic set of boards that make the AMiRo. The *Cognition board* is being used as the Reflective Operator of the DA_AMiRo project according to the OCM model. It provides the high-level functionality, gathering and coordinating the information via CAN that is generated in the three basic boards also known as the internal controller [25].

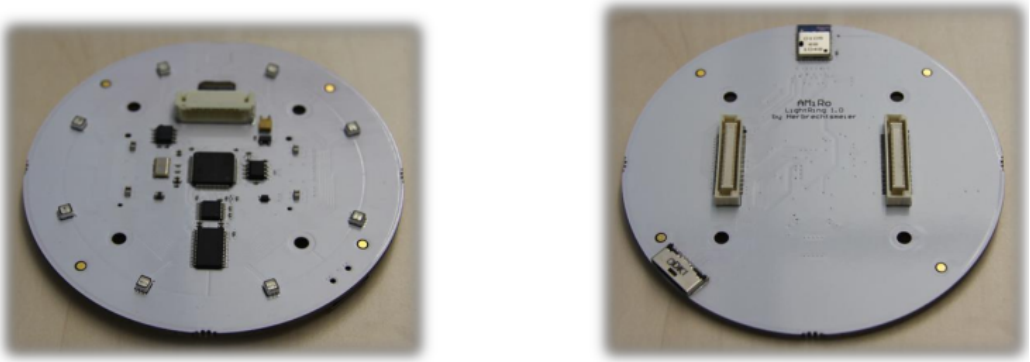


Figure 2.6: Top and bottom view respectively of the Light Ring board [5].

The most important feature of this board is the attached Gumstix Overo Computer-on-Module. In the current setup, the Overo TidalSTORM COM is being used, which combines an ARM Cortex-A8 based DaVinci DM3730 SoC from Texas Instruments [25]. This board has the same serial programming port as on the three basic boards. Additionally, the board has also a Type A, a micro-AB, and an internal pin header USB interfaces [25].

The operating system running on the Overo COM is based on the Linux Yocto reference distribution Poky [3], and the way it communicates with the three basic boards is through CAN, using a simple SocketCAN software-wrapper that is offered by this distribution [25].

The *Cognition* board ready to be mounted into the robot can be seen in figure 2.7. In this figure the board can be seen from both, the bottom and the top perspectives.

Figure 2.7: Bottom and top view respectively of the Cognition board [6].

2.2 Software Architecture of AMiRo OS

As it has been discussed in previous sections, the AMiRo project was started back in 2015 and it is still being maintained. The version 1.0 of the project is the one from which this thesis is based on [18].

2.2.1 AMiRo bootloader

The open-source bootloader for MCUs is the bootstrap for the three basic boards of the AMiRo. The AMiRo's bootloader is the responsible of configuring and setting up the computer for working with the AMiRo. The set-up that is provided in the bootloader is more than enough to successfully start working on the Operating System of AMiRo and programming it [33].

As said before, the DiWheelDrive is the only module of which the programming port can be accessed by the user. This is why new software is applied to the two other basic boards by remote flashing via OpenBLT. Using CAN communication the new binary is forwarded by the DiWheelDrive board so that the target board receives the data and stores it in its flash memory [25].

2.2.2 AMiRo real-time operating system kernel

The real-time operating system that is being used is the open-source ChibiOS. This project aims at creating a portable, light weight, and fast real-time operating system for embedded applications [27]. ChibiOS not only provides low-level drivers for all interfaces of the most common microcontrollers, but it also gives an unified hardware abstraction layer, priority based preemptive scheduling, and high-performance event and message passing systems. ChibiOS can be configured statically and dynamically to exactly fit the application in order to maximize efficiency and performance [25].

ChibiOS

Some of the main characteristics from ChibiOS are:

- System time: ChibiOS has a 16 or 32 bits system time counter.
- Real tick-less mode: There is no periodic system tick for optimal power management.
- IRQ Management: ISRs abstraction.
- Preemption: Fully preemptive scheduling.
- Round robin scheduling: Round robin scheduling for threads at equal priority.
- Messages: Inter-thread synchronous messages.
- Mailboxes: Message queues.
- Counter semaphores: Semaphores with boolean state.
- Binary semaphores: Semaphores with boolean state.
- Mutexes: Mutexes implementing the priority inheritance algorithm.
- Events: Events, event flags, event sources.
- Dynamic services: Dynamic threading.

Note: All the above mentioned characteristics can be found and further explained in the ChibiOS homepage. [27]

A project containing ChibiOS consists on the Operating System source files, few configuration files and a HAL file for the peripherals handling.

2.2.3 AMiRo Operating System structure

AMiRo's Operating System or the board specific abstraction layer from the original repository [26] includes two basic applications, the first one is a rudimentary command shell and the second one a hardware test. The first one is already integrated in ChibiOS and it is extended by module specific commands [25].

The structure of the AMiRo Operating System, including the files that are being used with the current project can be visualized in figure 2.8. The entire project is written in C++ language, with some inclusions of C code. In this figure, it can be seen that inside *AMiRo_OS* there is the *components* folder which contains the shared codes that the three boards are using. The other part of the structure worth mentioning now is the *devices* folder in which the three boards have their own set of files, always maintaining the same structure among them.

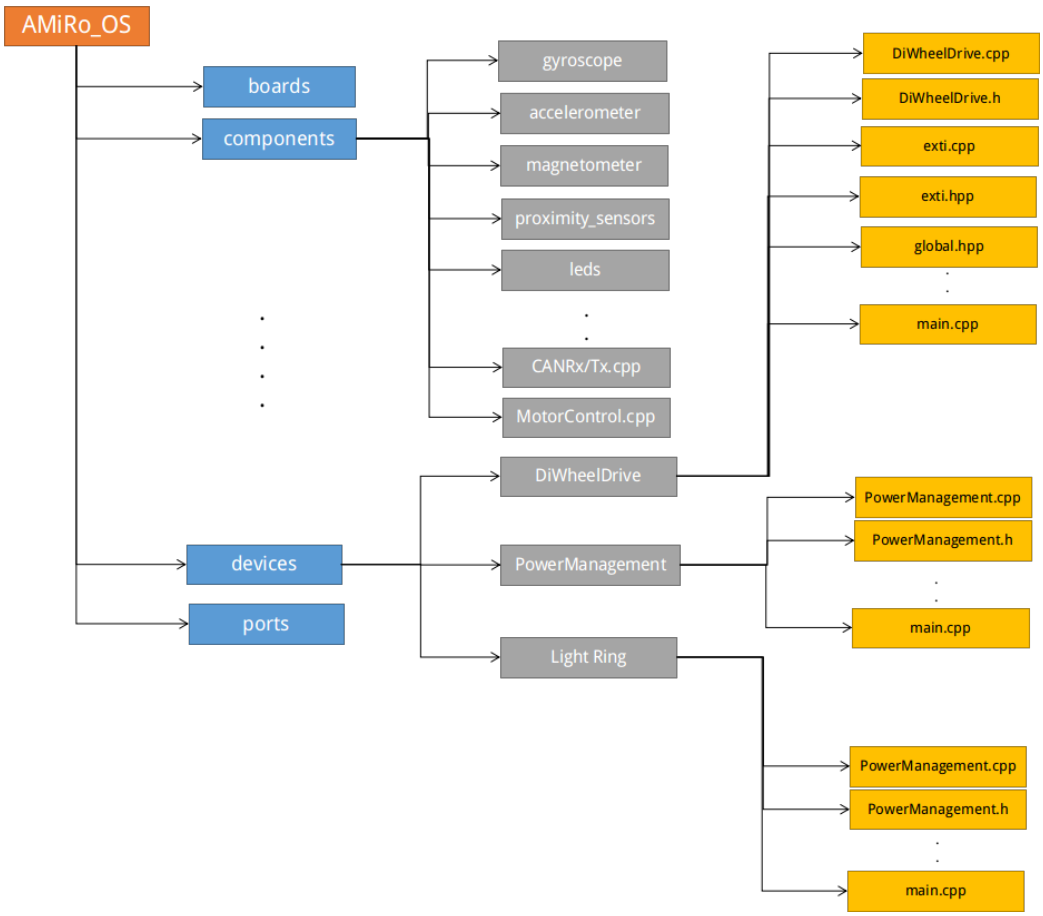


Figure 2.8: AMiRo_OS breakdown structure

2.2.4 AMiRo Operating System functionality

As mentioned before, the AMiRo OS has a command shell application which has a set of pre-stablished commands corresponding to commands that can set the lights of the AMiRo on, the motors, calibrate sensors, set a desired velocity or a distance for AMiRo to travel, get some sensor values, among other things.

There are two parent classes for all of the three basic boards: ControllerAreaNetworkTx.cpp and ControllerAreaNetworkRx.cpp. The ControllerAreaNetworkTx.cpp has the following functions:

- setLightBrightness(int brightness)
- setLightColor(int index, Color color)
- setOdometry(types::position robotPosition)
- setTargetSpeed(int32_t leftURpm, int32_t rightURpm)
- setTargetSpeed(kinematic &targetSpeed)
- setTargetPosition(types::position &targetPosition, uint32_t targetTime)
- setKinematicConstants(float Ed, float Eb)
- broadcastShutdown()
- txQueryShell(uint8_t toBoardId, char *textdata, uint16_t size)
- txReplyShell(uint8_t toBoardId, char *textdata, uint16_t size)
- main(void)
- updateSensorVal()
- encodeBoardId(CANTxFrame *frame, int board)
- encodeDeviceId(CANTxFrame *frame, int device)
- encodeIndexId(CANTxFrame *frame, int index)
- transmitMessage(CANTxFrame *frame)

Among these functions, it can be seen that the ControllerAreaNetworkTx.cpp is the responsible for updating the sensor values and for broadcasting them. Another feature of this class is to manage the outgoing CAN frames, coding the device id, the board id and also the id of the index from the commands.

The ControllerAreaNetworkRx.cpp is the responsible of managing the incoming CAN frames and the functions of this class are:

- getProximityRingValue(int index)
- getProximityFloorValue(int index)
- getActualSpeed(types::kinematic &targetSpeed)
- getOdometry()
- getPowerStatus()
- getRobotID()
- getMagnetometerValue(int axis)
- getGyroscopeValue(int axis)
- calibrateProximityRingValues()
- calibrateProximityFloorValues()
- rxCmdShell(CANRxFrame *frame)
- main(void)
- decodeBoardId(CANRxFrame *frame)
- decodeDeviceId(CANRxFrame *frame)
- decodeIndexId(CANRxFrame *frame)
- receiveSensorVal(CANRxFrame *frame)

From the shown functions in ControllerAreaNetworkRx.cpp it is clear that this class receives the CAN frames, decodes the board, index, and device information. Once the purpose of the incoming CAN frame has been found out, the ControllerAreaNetworkRx.cpp proceeds to make use of the CAN

frame according to its topic, if for example the CAN frame wanted to set the robot's position, the CANRx.cpp will set the values of robotPosition.x, robotPosition.y, and so on with the content of said frame. From the names of the functions, the reader can have a broad idea from what each of them does with the received CAN frame.

Now that the functionality of the CANRx/Tx.cpp has been shown, each one of the three boards that inherit from them will be briefly discussed.

DiWheelDrive project

Lets have a look at *main.cpp* from *DiWheelDrive* board. The shell which was already mentioned is being managed by this board. Every sensor that will be used is waken up and the communications protocols for each one is started. It is also in this code where all the ChibiOS threads are being started. The threads are:

- Power status
- Magnetometer
- Gyroscope
- Accelerometer
- Proximity floor sensors
- Motor control
- Odometry
- User thread

Each one of these threads is inside a class with the same name. The instances of these classes are created in the *global.hpp* so that the encapsulated values of each object can be accessed from the outside. In global.h all the sensors corresponding to the *DiWheelDrive* project are initialized and configured for its use. As an example, the low level registers for using the magnetometer or *hmc5883l* are configured to set the number of samples averaged, select the data output rate bits, set the measurement configuration bits, set the gain configuration bits, among other things. For more precise information about the registers and how to configure the sensors, please refer

to the specific sensor's data sheet that are listed in the bibliography, in this case [9].

The *global.cpp* code is also a class and an instance of it is created in *main.cpp* which is called *robot*, meaning that from *main.cpp* the user can access the elements of *robot* among which are the instances of the sensors listed above. For example, in order to access the magnetometer, the user has to type something like this: *robot.hmc5883l....*

Regarding the threads listed above, the Power status in this board only makes a self test and initiates a power monitor. The magnetometer, gyroscope, accelerometer, power status, odometry, and the proximity floor classes, have functions for updating their sensor's data, but also functions for getting the actual sensor's value in a raw format or with a specific unit. For the available units for every sensor, see section 4.2.1.

The threads of the magnetometer, gyroscope, accelerometer, power status, odometry, and proximity floor sensors are constantly updating their values so that they are ready whenever they are required. As a side note, the magnetometer thread, the gyroscope thread, the accelerometer thread, and the proximity floor sensors thread are constantly broadcasting to the CAN bus at a given frequency which is set to $16Hz$.

The motor control thread is constantly updating the increments of velocity, the speed, calculating velocities in desired units. In addition to this, the thread updates past velocities for the derivative part of a PID controller the user has access to, and writes the calculated duty cycle of the PWM driver.

Lastly, the user thread of the *DiWheelDrive* is a complete line following application that makes AMiRo able to follow a black line on the ground that is over a white surface.

The overall structure from the *DiWheelDrive* project can be seen in figure 2.9. Also, as discussed before, the ControllerAreaNetworkRx.cpp is constantly monitoring the CAN bus for commands that would like to calibrate or select a desired end value in the encoders and the motors, but also for controlling the motors themselves.

Power Management project

The *Power Management* project follows the same structure mentioned for the *DiWheelDrive*, with a *main.cpp* and its own *global.hpp*. The instance of the global class is also called *robot* and also can be accessed from the *main.cpp*. The list of the threads of the *Power Management* is as follows:

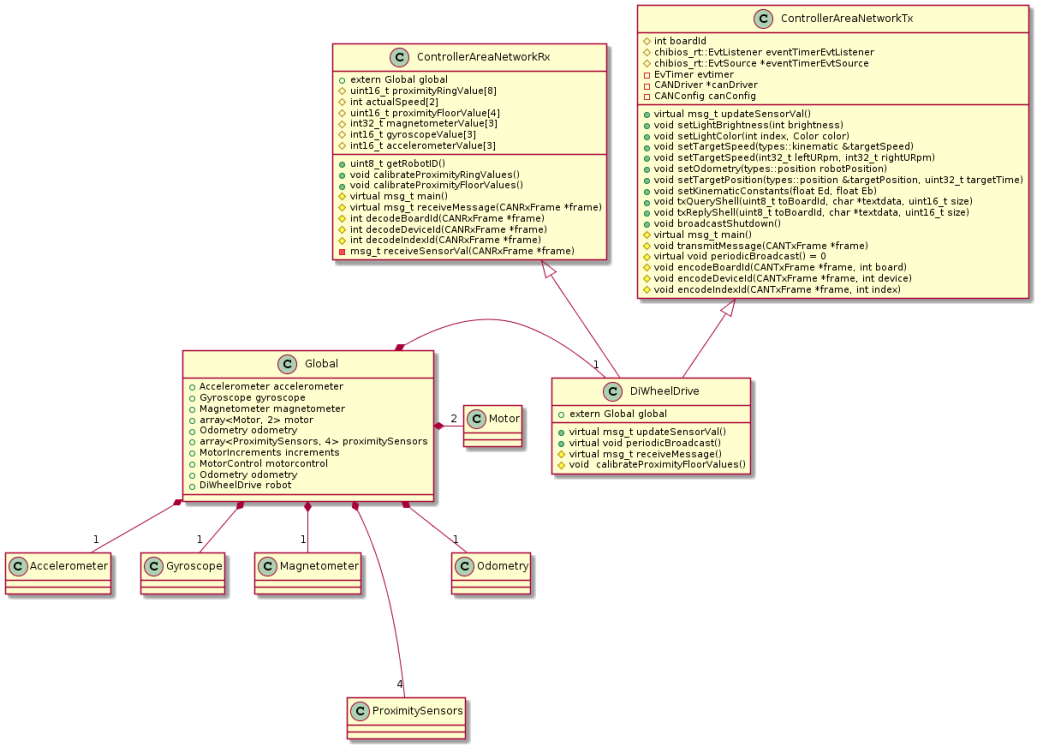


Figure 2.9: DiWheelDrive broad class diagram of AMiRo OS

- Power status
- Proximity ring sensors
- ADC
- Touch sensors
- User thread

The proximity ring and the touch sensor threads are constantly updating its sensors' values. The touch sensor *mpr121* is part of the proximity ring sensors structure and these two sensors are also broadcasting its values through the CAN bus like the ones from *DiWheelDrive*.

The power status thread receives help of the ADC thread which constantly monitoring the charge of the batteries and converting the analog levels of charge to a digital number. So the power status thread is frequently

updating values such as the current battery levels, the time of charge remaining, whether or not AMiRo is plugged in, and the current consumption.

The user thread of the *Power Management* is an obstacle avoidance application for the AMiRo which also sends commands via CAN to the *DiWheelDrive* in order to stop the AMiRo or stir it in order to stop it from crashing.

The overall structure from the *PowerManagement* project can be seen in figure 2.10 where the sensors are broadcasting their values at once to the CAN bus. Also, as discussed before, the ControllerAreaNetworkRx.cpp is constantly monitoring the CAN bus for commands that would like to calibrate the proximity ring sensors.

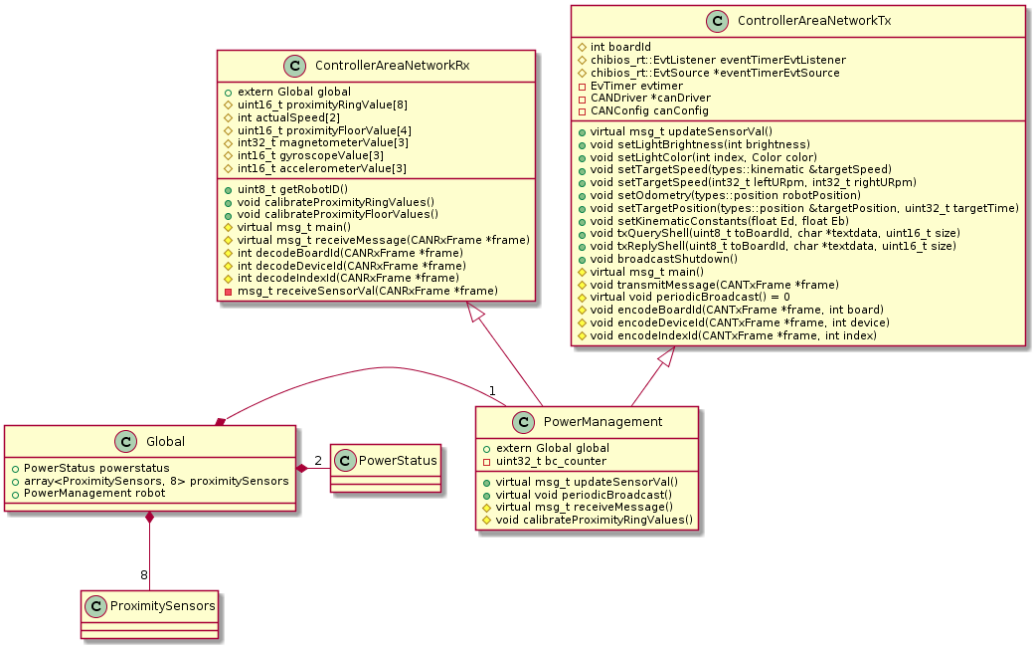


Figure 2.10: PowerManagement broad class diagram of AMiRo OS

Light Ring project

Finally the *Light Ring* project is the simplest of the three projects, it follows the same structure mentioned for the *DiWheelDrive* and *Power Management*, with a *main.cpp* and its own *global.hpp*. The instance of the global class is also called *robot* and also can be accessed from the *main.cpp*. The list of the threads of the *Light Ring* is as follows:

- Lidar
- LEDs
- User thread

The lidar class performs a configuration of the lidar and a quick check to see if it is communicating correctly with the system. The lidar thread is constantly updating its values.

The LEDs thread is constantly refreshing the LEDS with the preset brightness values and the colors assigned to each of them.

The user thread of the *Light Ring* has no implemented application and remains open for the user to create one.

The overall structure from the *LightRing* project can be seen in figure 2.11 where the sensors are broadcasting their values at once to the CAN bus. Also, as discussed before, the ControllerAreaNetworkRx.cpp is constantly monitoring the CAN bus for commands that would like to set the LEDs to a certain color and to adjust the brightness.

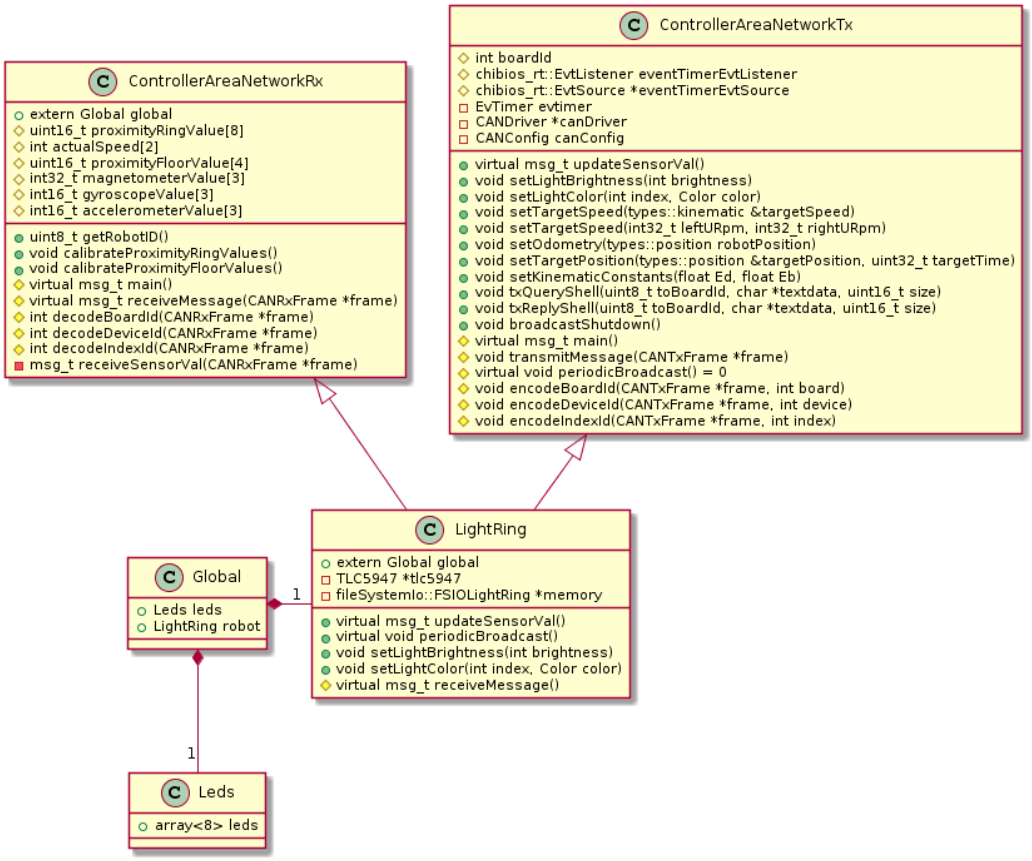


Figure 2.11: LightRing broad class diagram of AMiRo OS

Chapter 3

DA_AMiRo software architecture

From the past chapter the hardware and software structure of the AMiRo was mentioned and discussed. From this point forward the structure and functionality of the DA_AMiRo project which is based on the OCM model will be discussed as it is the central point of the master thesis. As a friendly reminder, the reader can obtain the source codes made for the DA_AMiRo project from the repository[1] [18].

3.1 DA_AMiRo software structure

In last chapter, the AMiRo's Operating System structure was introduced. This Operating System is the one located on top of figure 3.1, inside the folder called *amiro_1.0_stable*. In order to fulfill the requirements of the DA_AMiRo project, the AMiRo OS had to be replaced for a OCM apporach, and the functionality of the resulting software structure will be discussed in section 3.2.

The configuration files for the three projects are located inside the *Devices* folder which can be visualized in figure 3.1. They configure the QT IDE for each of the three boards, so the user can easily edit the code and build the binary files to program the AMiRo's boards. It is also inside this folder where the *Reflective Operator* is located. The Reflective Operator consists of three parts: a simple C application template, a Matlab and a Simulink model that

[1] https://gitlab.idial.institute/DAdevs/DA_AMiRo.git

can be easily edited and programmed into the Reflective Operator, and four already compiled and functioning Demo Applications. The Demo Applications and the way of making custom applications from a Simulink model will be discussed in chapter 6. It is worth mentioning that the demo applications are already inside the MuRoX board, and they are ready to be used.

Inside the *Peripheral-Drivers* the user can find the ControllerAreaNetwork.h which is the responsible of defining the structures and all the definitions that are being used by both the Internal Controller corresponding to the three basic boards and the *Reflective Operator*. It is also inside this folder that the *sensorhandler.cpp* and *sensorhandler.h* are to be found. These both files manage the scheduling of the sensor data and they will be discussed in chapter 5.

Finally, the folder *Documentation* has some of the scientific papers and information that the AMiRo has had since it was created. The reader can go inside this folder to get more information about the AMiRo. In the other hand, the folder *PusleAT* was created for a future usage and it is still a work in progress.

3.2 DA_AMiRo Operating System's functionality

In this section, the functionality of the DA_AMiRo Operating System from figure 2.8 will be addressed.

The DiWheelDrive board, the DiWheelDrive class inherits from both ControllerAreaNetworkRx.cpp and from ControllerAreaNetworkTx.cpp, making it capable of sending and receiving CAN frames. This happens exactly the same in the three boards. The CAN frame structure will be further discussed in chapter 4. The DiWheelDrive class also implements the functions for updating each of the sensor's values and sending independently the data via CAN interface. As it can be seen in figures 3.2, 3.3, and 3.4, DiWheelDrive has virtual functions of every sensor including the Power Management board and also the Light Ring board. These functions however, have a '= 0' which means that they are not being implemented in this specific class. Similarly in the other two boards the same thing happens with their own set of sensors.

This means that in the DiWheelDrive class, the functions that are being implemented correspond to the gyroscope, accelerometer, magnetometer,

proximity sensors, motors, and encoders. In the Power Management class the implemented functions include proximity sensors and power status sensors, and in the Light Ring class the implemented function concerns only the LEDs for the purposes of DA_AMiRo.

The global class is the responsible for creating each object from each sensor and encapsulating them under the name of 'robot'. This makes the *robot's* attributes accessible from other classes. This means that, for example, the gyroscope's value can be updated and outputted as a part of *robot* from other parts of the code. The global object is finally initialized in the main function, as well as the threads responsible for managing the scheduler of the sensor's data but this subject will be discussed more deeply in chapter 5.

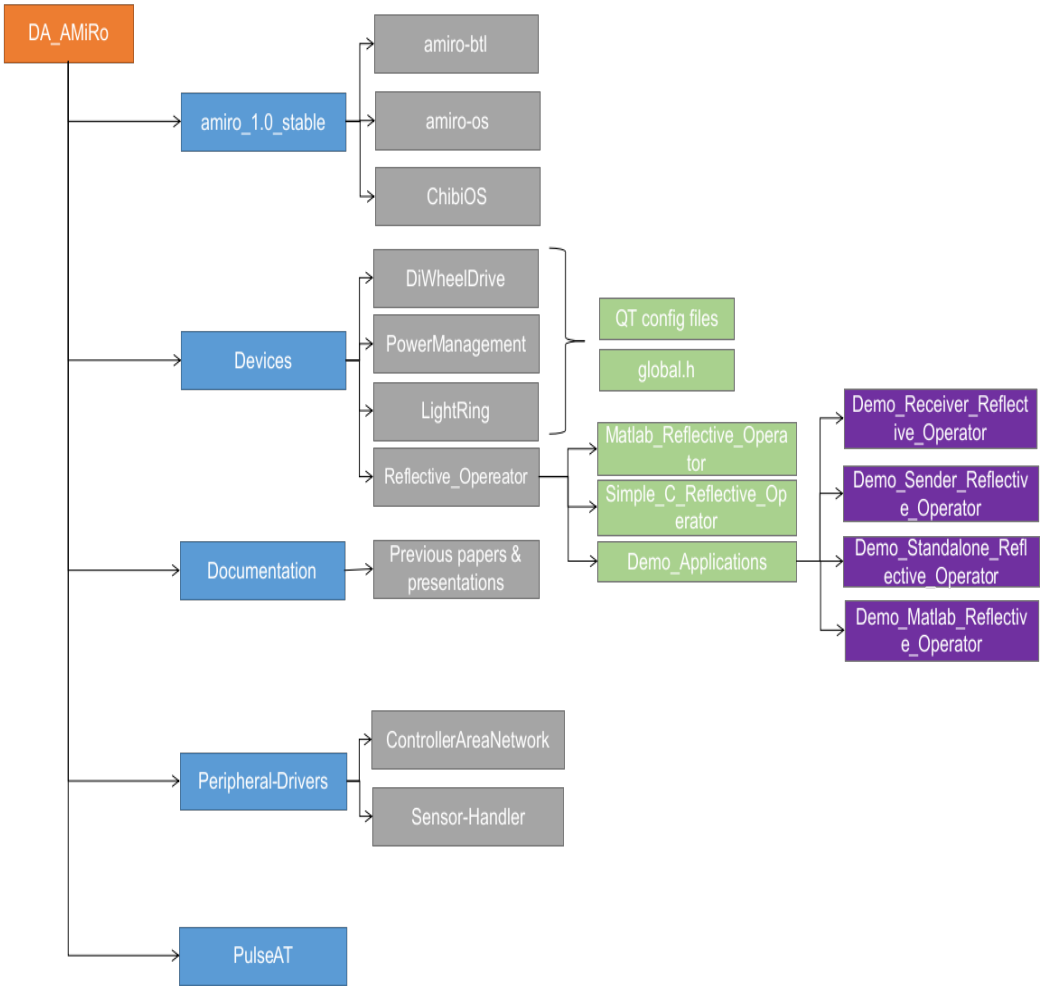


Figure 3.1: DA_AMiRo overall structure

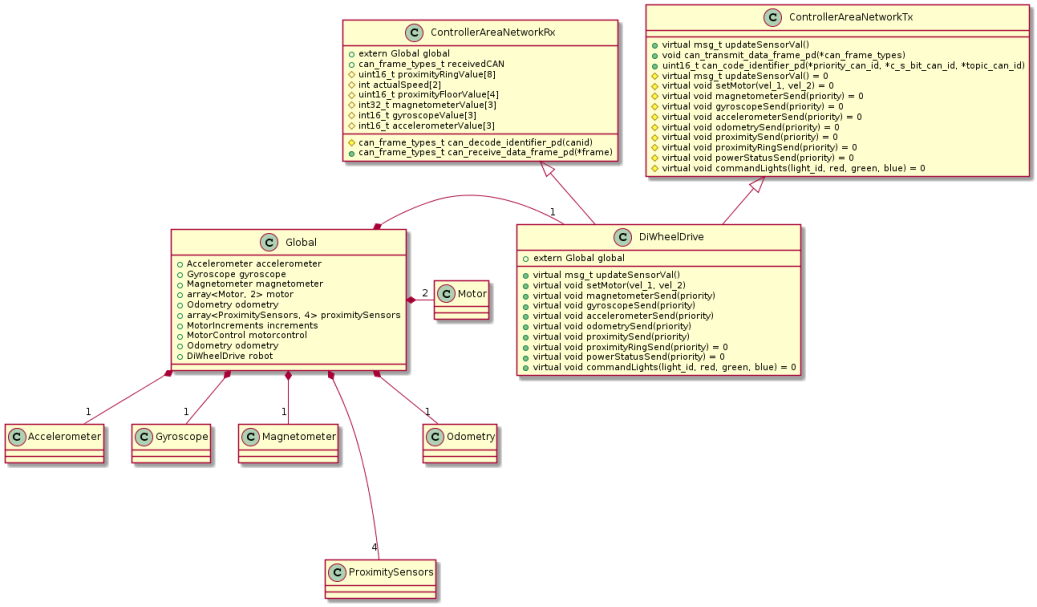


Figure 3.2: DiWheelDrive broad class diagram

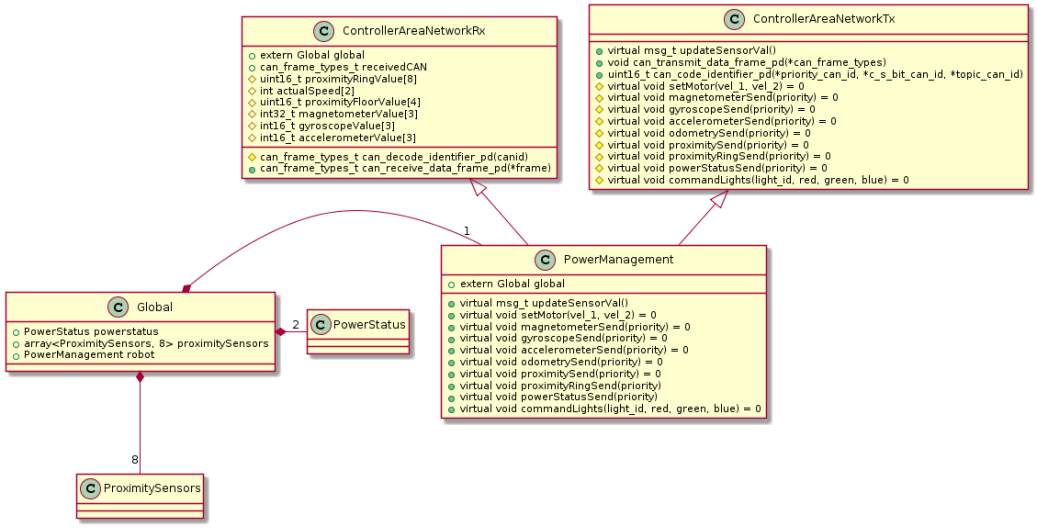


Figure 3.3: PowerManagement broad class diagram

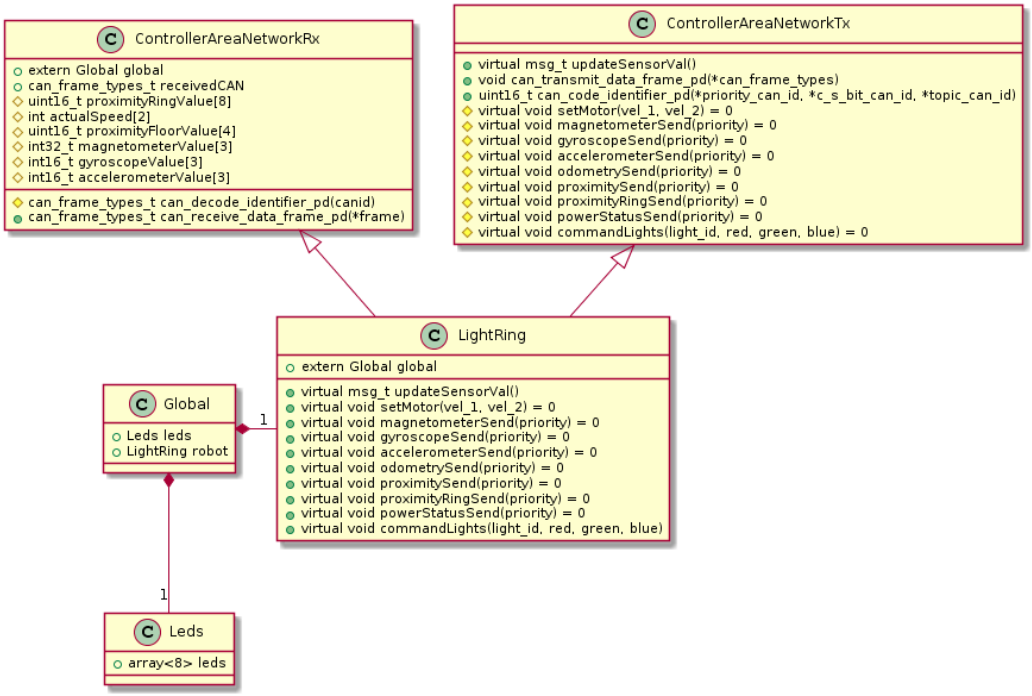


Figure 3.4: LightRing broad class diagram

Chapter 4

Controller Area Network

This chapter explains how to receive and transmit CAN frames in the AMiRo. The functionality of the CAN module had to be merged with the current ControllerAreaNetworkRx.cpp and ControllerAreaNetworkTx.cpp of the existing repository of AMiRo [26]. The ControllerAreaNetwork.h however, is the same for the internal controller and for the *Reflective Operator*. The reason behind sharing the ControllerAreaNetwork.h among projects is to be able to use a single set of definitions for names of sensors and some structures.

For starters, the baud rate requirement of the DA_AMiRo project was set to 500kHz This frequency was achieved with the following equation, and can be changed from the CANRx and CANtx.cpp files that can be located with help of figure 2.8.

$$500kHz = \frac{8MHz}{(B1 + B2 + SW) * prescaler} \tag{4.1}$$

Where the 8MHz is the clock frequency of the STM32F3, the B1 and B2 are the time quanta bit segment one and two accordingly. Finally, the SW represents the maximum time quanta. These values were configured by the previous AMiRo project which also happened to configure the CAN frequency to 500kHz. The values and the names of the variables in the actual code are:

$$B1 = CAN\_BTR\_TS1 = 13$$

$$B2 = CAN\_BTR\_TS2 = 2$$

and

$$SW = CAN\_BTR\_SJW = 1$$

$$Prescaler = CAN\_BTR\_BRP = 1$$

, which results in the following equation:

$$\frac{8MHz}{(13+2+1)*1} = 500kHz \tag{4.2}$$

4.1 Convention used for sending and receiving frames

The format of the CAN frames was done accordingly with the pre-established convention for the DAEbot project [13] by Uwe Jahn. The format of the standard id from a CAN frame can be seen in figure 4.1, where it is basically divided into three main parts. The command or sensor bit is the bit that allows devices on the CAN bus to know if the frame belongs to a command, i.e. knowing the value 'x' axis of the gyroscope, or setting the LEDs to red, or if it belongs to a sensor frame, i.e. the actual value of the 'x' axis of the gyroscope.

The second part is the priority which allows the devices on the bus know the priority of the command or sensor. If there are two frames in the bus at the same time asking for a sensor value from the same board for example, the priority will allow the system to preempt the frame of the one with higher priority.

The third and last part of the id from the frame is the topic id. The list for the topics can be seen in the ControllerAreaNetwork.h shared by all the devices or also from the ods file inside the same path *Peripheral-Drivers/ControllerAreaNetwork/* from figure 3.1.

For the transmission of a CAN frame, the id is coded according to figure 4.1 and when a CAN frame is received, the id is decoded into the same structure. The CAN frame pre-established structure can be seen in snip of code 4.1, which can be compared with the figure 4.1. In this structure the only thing that has not been yet discussed is the data itself. The dlc is the number of bytes to be transmitted and the information is stored in the data8, data16 and data32 arrays. Thanks to the *union* of these arrays, the same value can be accessed in different bit sizes.

A Base Frame Format CAN Identifier (11 Bits) will be used to identify the messages in the system.

Bit	0	1	2	3	4	5	6	7	8	9	10
Subject	C/S	Priority		Identity							
Number of Bits	1	2		8							

- The first bit, the C/S Bit is used to identify if the message is:
  - 0 = Command (/Actuator)-Data (higher priority as Sensor-Data)
  - 1 = Sensor (/Status)-Data (lower priority as Command-Data)
  - The Operator sends a command message to a Controller with the Controller's ID (take a look below)
- The Bits two and three are used to set the priority of this CAN Frame to:
  - 0 (00) = Urgent
  - 1 (01) = High
  - 2 (10) = Medium
  - 3 (11) = Low
- A unique ID is used for every topic on every component ($2^8 = 256$ possibilities). Take a look below for the specific ID's and don't forget to write them down if you create new ones.

Figure 4.1: Pre-established CAN frame format [13]

```
typedef struct can_frame_types {
    priority_can_id_t priority_id;
    c_s_bit_can_id_t c_s_bit_id;
    topic_can_id_t topic_id;
    uint8_t dlc;
    union{
        uint8_t     data8[8];
        uint16_t    data16[4];
        uint32_t    data32[2];
    };
} can_frame_types_t;
```

Listing 4.1: CAN frame

In order to send a CAN frame, can_transmit_data_frame_pd(...) can be called. When sending a CAN frame, the id is set to be in standard mode. The CAN frame that is sent consists of three main parts: id, DLC, and the data. The id is therefore set with the can_code_identifier_pd(...) function before actually calling the transmit function. This function combines the c_s bit with the priority bit and then combines that value to the topic ID. The

CAN frame is later sent through the CAN driver.

On the CAN reception side, the status of the reception is assigned when calling to the can_receive_data_frame_pd(...) function. The status of the CAN reception can be successful, timeout, or error. Then the id of the CAN frame gets decoded with the function can_decode_identifier_pd(...) and the data frames are passed into a global structure like in snip of code 4.1

Note that can_code_identifier_pd(...) and can_decode_identifier_pd(...) are functions that were already implemented beforehand, for this project it was necessary to adjust both functions to operate with the current hardware platform.

4.2 Interpreting CAN frames of the sensors

In this section it will be explained how to read the frames corresponding to each sensor. For the same reason, this section is divided into two subsections: sensors and actuators.

4.2.1 Sensors

In order to interact with a sensor, the id from the CAN frame's command has to obey the convention shown in figure 4.1 and its content, the actual data from the CAN frame has to follow the convention shown in figure 4.2 created for the DAEbot project.

There are three ways or modes in which a command can ask for data from a sensor. The first one is the mode one which asks for a sensor's data with a specific frequency. The first DLC of the CAN frame has to be set to '1' to activate the publisher mode. The second and third DLCs specify the numeric value of the frequency in format of little-endian. This means that the most significant bit should be a part of the DLC number 2 and the less significant bit should be a part of the DLC number 1. The fourth DLC specify the units of the value given in the second and third DLCs. This value can be obtained looking at figure 4.2.

For example for starting publisher of the magnetometer 'x' axis, with normal priority, the CAN id should be composed by:

- C_S bit set to '0' on binary
- Priority set to '10' on binary

	data[0]	data[1] & data [2]	data[3]
Mode 0: No data transmission, publisher off	0	-	-
Mode 1: Starts publisher*	1	sample time (little-endian) (default 50 ms)	unit (default) 0 = ms 1 = s (2 = µs)
Mode 2: Asks for one-time data transmission	2	-	-

- **Note:**
  - can be extended with sample time (for this you need to use a 4 DLC CAN Frame: data[0]=1; data[1] & data[2]=sample time (little-endian); data[3]=unit

Figure 4.2: Pre-established CAN modes for transmitting sensor data [13]

- Topic id of the gyroscope 'x' axis '0x05' on hexadecimal

So a value equal of '01000000101' in binary or '0x205' in hexadecimal. Now lets say we want the magnetometer 'x' axis value every 900μs. The value '900' in decimal is '0x0384' in hexadecimal, making the little-endian adjustment for the bytes will result in '8403'. μs is equal to '0x02'.

As a result, the complete CAN frame is: '0x205' for the id, '4' for the DLC or Data Length Code, each with the following values:

- DLC 0: 0x01
- DLC 1: 0x84
- DLC 2: 0x03
- DLC 3: 0x02

This command would trigger the DiWheelDrive to output the magnetometer 'x' axis value every 900μs, with an id that is almost the same but with the C_S bit set to '1' instead, so that the CAN bus knows that this frame belongs to a sensor data. The topic id then will be '0x605' instead of

'0x205'. The way of reading and understanding the outputted value for each sensor will be mentioned in the following subsections.

The second mode is the *One-time data transmission* and it will only return the required sensor value one time. For the example of a one time transmission of the floor proximity sensors, the id of the CAN frame should be '0x204' with DLC 0: 2.

The third mode is the mode '0' which actually stops an on-going publishing of a sensor value given the frame's id. This means that in the example used for the explanation of mode '1' the way of stopping the transmission of the magnetometer 'x' axis, the id of the CAN frame should be the same '0x205', with DLC 0: 0.

Gyroscope

The three axis of the gyroscope come together in one CAN frame with a DLC of six. Each value has a size of 16bits with a format of little-endian. Every value comes in two's complement and the units are degrees per second, and the resolution is $+/-500dps$. See table 4.1.

DLC	6
size of data	16bits
'x' axis	DLC[1] DLC[0]
'y' axis	DLC[3] DLC[2]
'z' axis	DLC[5] DLC[4]
units	dps
note	value in two's complement

Table 4.1: CAN frame format from the output of the gyroscope

Accelerometer

The three axis of the accelerometer come together in one CAN frame with a DLC of six as the gyroscope. Each value has a size of 16bits with a format of little-endian. Every value comes in two's complement and the units are $milig$, and the result is scaled $+/-8g$. See table 4.2.

DLC	6
size of data	16bits
'x' axis	DLC[1] DLC[0]
'y' axis	DLC[3] DLC[2]
'z' axis	DLC[5] DLC[4]
units	$milig$
note	value in two's complement

Table 4.2: CAN frame format from the output of the accelerometer

Magnetometer

In the case of the Magnetometer, every axis has an independent id, this is because the size of the data is 32bits in little-endian format. Every value comes in two's complement and the units are $\mu Gauss$, and the result is scaled by $+/-5Ga$. See table 4.3.

DLC	4
size of data	32bits
'x' axis	DLC[3] DLC[2] DLC[1] DLC[0]
'y' axis	DLC[3] DLC[2] DLC[1] DLC[0]
'z' axis	DLC[3] DLC[2] DLC[1] DLC[0]
units	$\mu Gauss$
note	value in two's complement

Table 4.3: CAN frame format from the output of the magnetometer

Floor proximity sensors

The four proximity sensors on the bottom of the AMiRo are together in one CAN frame with a DLC of eight. Each value has a size of 16bits with a format of little-endian. The data is in luminance with units candelas per squared meter cd/m^2. See table 4.4 and note that this sensor does not return a two's complement value.

Encoder

This sensor returns both the actual velocity and the odometry of the AMiRo. The first value will be the actual velocity followed by the odometry. For the

DLC	8
size of data	16bits
sensor 1	DLC[1] DLC[0]
sensor 2	DLC[3] DLC[2]
sensor 3	DLC[5] DLC[4]
sensor 4	DLC[7] DLC[6]
units	cd/m^2

Table 4.4: CAN frame format from the output of the floor proximity sensors

odometry, the 'x' coordinate in μm with a size of 32bits, the 'y' coordinate in μm with a size of 32bits with a format of little-endian, and the orientation of the wheels in μrad with a size of 16bits. All of these values are together in one CAN frame. See table 4.5.

DLC	8
size of data	32 bits, 32 bits, and 16 bits
'x' coordinate	DLC[2] DLC[1] DLC[0]
'y' coordinate	DLC[5] DLC[4] DLC[3]
orientation of wheels	DLC[7] DLC[6]
units	μm, μm, and μrad

Table 4.5: CAN frame format from the encoder

Actual velocity

The actual velocity will be composed of the 'x' velocity in $\mu m/s$ with a size of 32bits and the 'z' angular velocity in $\mu rad/s$ also in 32bits with a format of little-endian. These two values are together in one CAN frame. See table 4.6.

DLC	8
size of data	32bits
'x' velocity	DLC[3] DLC[2] DLC[1] DLC[0]
'z' angular velocity	DLC[7] DLC[6] DLC[5] DLC[4]
units	$\mu m/s$ and $\mu rad/s$

Table 4.6: CAN frame format from the output of the actual velocity

Power Status

When the id for the power status is asked, four values are sent within one CAN frame. The charging or no charging flag in 8bits, the state of charge in percentage in 8bits, the minutes remaining for a complete charge when the flag for charging is 'on' and the minutes remaining of battery power when the flag for charging is 'off' in 16bits and in little-endian, and the current power consumption in mW also in 16 bits and in little-endian. See table 4.7.

DLC	6	
size of data	8bits, 8bits, 16bits, and 16bits	
charging flag	DLC[0]	'0' for not charging, '1' for charging
state of charge	DLC[1]	percentage
time until charge	DLC[3] DLC[2]	
power consumption	DLC[5] DLC[4]	
units	percentage, minutes, mW	

Table 4.7: CAN frame format from the output of the power status

Proximity ring

The proximity ring will return eight CAN frames one for every sensor there is. Each value is of 16btis and it is in little-endian, and the units are mm. See table 4.8.

DLC	16
size of data	16bits
sensor 1	DLC[1] DLC[0]
sensor 2	DLC[1] DLC[0]
sensor 3	DLC[1] DLC[0]
sensor 4	DLC[1] DLC[0]
sensor 5	DLC[1] DLC[0]
sensor 6	DLC[1] DLC[0]
sensor 7	DLC[1] DLC[0]
sensor 8	DLC[1] DLC[0]
units	mm

Table 4.8: CAN frame format from the output of the proximity ring sensors

4.2.2 Actuators

There are so far two implemented actuators: the motors and the LEDs. The way of interacting with the sensors will be now described but it is important to note that they will not send anything back to the CAN bus, as they will only 'act'.

Motors

The motors can be started with a desired speed in $\mu m/s$ or in an angular speed with $\mu rad/s$ units. See table 4.9. For the id of the motor's actuator please refer to the ControllerAreaNetwork.h file found in 3.1.

DLC	8
size of data	32bits
'x' velocity	DLC[0] DLC[1] DLC[2] DLC[3]
'z' angular velocity	DLC[4] DLC[5] DLC[6] DLC[7]
units	$\mu m/s$ and $\mu rad/s$

Table 4.9: CAN frame format for setting the velocity of the motors

LEDs

In order to turn a LED on, the following data structure has to be followed:

- id: Desired LED's id obtainable from the ControllerAreaNetwork.h found in 3.1.
- DLC 0: Red value
- DLC 1: Green value
- DLC 2: Blue value

Chapter 5

Sensor data handler

The sensor data handler is the responsible for setting the publisher of sensor's data. The publisher can handle every sensor sending its value at once as they have each a dedicated task in a real-time context. This means that every sensor has its own task assigned to it and said task gets suspended or resumed with the requested frequency according to the commands of the CAN frames.

In figure 5.1 the structure of the sensor data handler can be seen as well as its connection with the sensor tasks. All of these tasks get managed by the sensor data instance.

5.1 Overall functionality

The sensor handler is a class which has only once instance and its started in the main class, along with the tasks for all the sensors inside each board. When the task for the sensor handler is started, it polls the CAN reception for a CAN frame that has the C_S bit set to a '0' which corresponds to a *Command data*, according to fig 4.1 of section 4.1.

Whenever the sensor handler detects a CAN frame corresponding to a *Command data*, the CAN frame has to be further analyzed to know if it is trying to communicate with a *sensor* or with an *actuator*. An example of this functionality can be seen in figure 5.3 which will be explained in section 5.1.3.

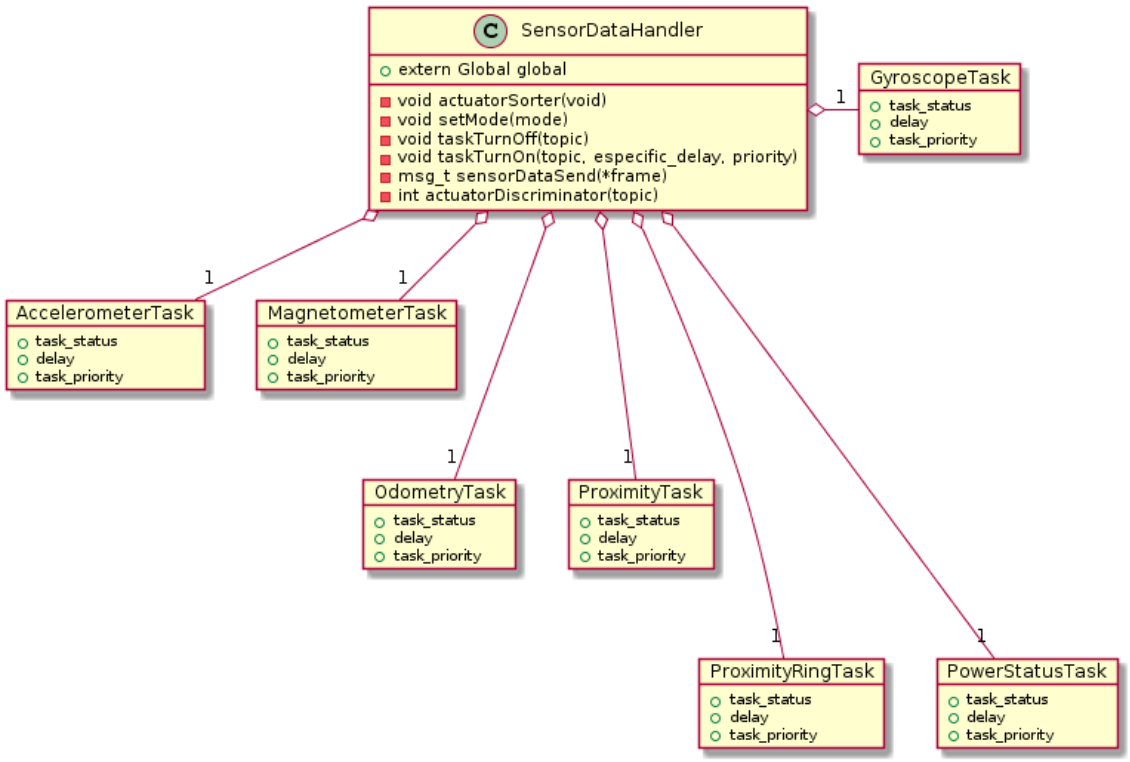


Figure 5.1: SensorDataHandler.cpp and the sensor tasks structure

5.1.1 Sensor data handling

Whenever the command corresponds to a sensor, the next step is to read the DLC 0 to know if it is a command for stopping a sensor data, if it is a command for starting a publishing sensor data, or if it is a request for a one time transmission, see figure 4.2 from section 4.2.1.

The state machine of figure 5.2 shows how the modes get handled. Whenever the DLC 0 has a '0' corresponding to a *No transmission* is detected, the id from the topic gets identified and the corresponding task gets suspended so it no longer transmits its value.

When the DLC 0 has a '1' corresponding to a *Publisher* is detected, the DLC 1, DLC 2, and DLC 3 get decoded to know the requested frequency from the id of the sensor that needs to start sending its information. If the sensor was already sending information, it needs to update its frequency and the priority, and it gets going.

Finally, when the DLC 0 has a '2' which means that the user wants a *One time transmission* of a sensor data, this value gets sent one time only. If this sensor was sending its value in a publisher, then the task gets suspended.

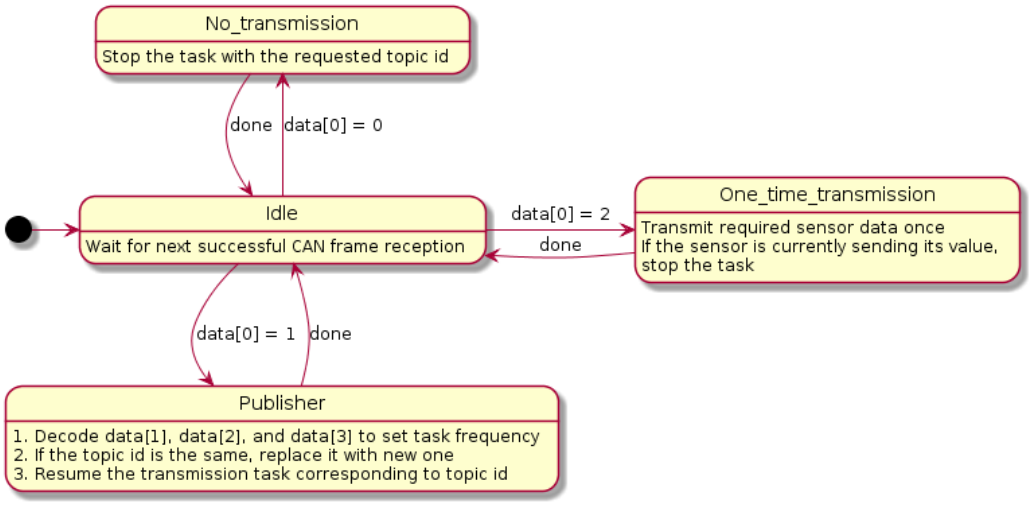


Figure 5.2: Modes implementation

5.1.2 Actuator data handling

In the other case, when the command is trying to communicate with an actuator, it goes into a sorter that separates the motor's command from the LEDs' command. If it is a command for the motors, then the motors get set with the velocity requested in the CAN frame. When its a LEDs command, the selected LED gets turn on or changes its color with the *RGB* code reviewed in section 4.2.2.

5.1.3 Real-time implementation

The tasks corresponding to the sensors can all be operating at once. The priority of the tasks is being handled as preemptive tasks. This means that if two tasks are racing for the processing time, the one with higher priority will be preempted. Also, if one task is asked to output its value at the same frequency as a task that is already outputting its value, the one with higher

priority will be on schedule and the one with less priority can be delayed some μs.

Everything up to this point works as an independent sensor data handler. This means that the internal controller is encapsulated and complete. The only thing that can interact with the functioning system are commands in the CAN bus. For this purpose the MuRox board has to be set and running the Linux based Operating System. See next chapter for more information about how to debug and the functionality of the *Reflective Operator* in general.

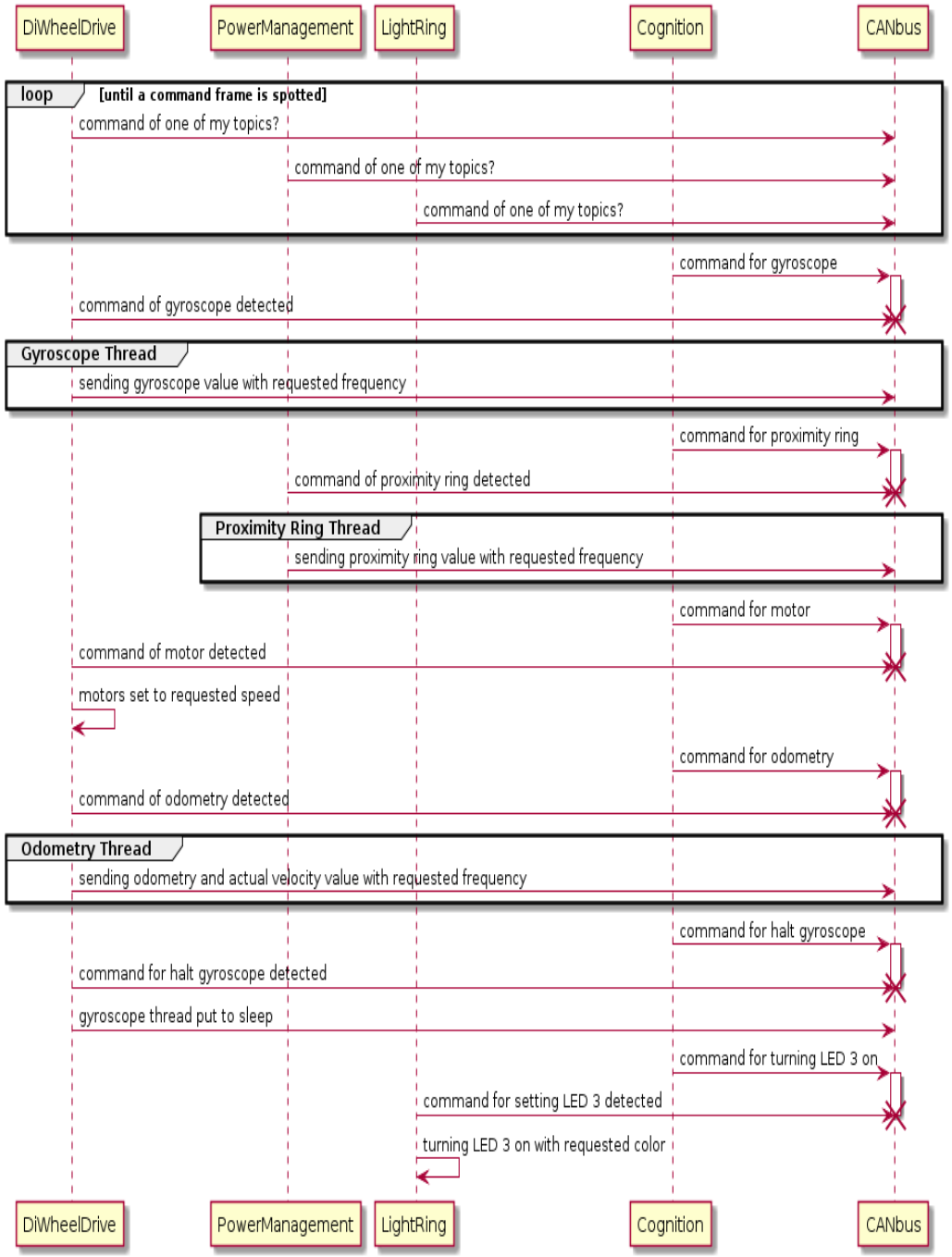
An example of how the controller and the *Reflective Operator* are communicating can be seen in figure 5.3. The three boards are constantly checking the CAN bus for a CAN id corresponding to a *command*. Whenever the *Reflective Operator* sends a command asking for the gyroscope value with a specific frequency, this CAN frame gets fetched by the *DiWheelDrive* board and starts the thread of the gyroscope with the requested frequency and priority.

In the image the *Reflective Operator* asks then for a proximity ring value. This sensor is located on the *Power Management* board so this thread gets started by this board. Afterwards, the *Reflective Operator* asks to set the motor with a specific speed. The *DiWheelDrive* board where the motors are connected, fetches this information and sets the motors accordingly.

The same process gets carried out for the odometry value request, which is managed also by the *DiWheelDrive* board. The *Reflective Operator* asks to stop the gyroscope sensor from sending its value. This command also gets fetched by the same board and puts the corresponding thread to sleep.

Finally the *Reflective Operator* asks for the LED number 3 to turn on with a specific RGB value. This information is fetched by the *Light Ring* board and turns on the LED with the requested information.

Figure 5.3: Functionality of an example application



Chapter 6

Reflective Operator

The board responsible of performing as the *Reflective Operator* of the system is the MuRoX board. This board is the one that asks for a certain sensor data from the internal controller or also it can control the actuators. The user can connect to this board through ssh, with the address 192.168.1.1 with password *root*.

The board already has a package called 'candump' which interacts with the CAN module in the board and can be used to debug the internal controller's functionality. The way of sending CAN frames from the MuRoX board has to follow the next convention:

```
cansend can0 --identifier=CANid DLC0 DLC1 DLC2 DLC3 DLC4 ...
```

Listing 6.1: CAN send format

6.1 Compiling for MuRoX

For creating an application that the *Reflective Operator* can run, it is necessary to use the *cortexa8hf-vfp-neon-poky-linux-gnueabi* version of the cross-compiler. Once the application is compiled, the resulting binaries can be moved into the board using the *scp* command.

```
scp anyFile root@192.168.0.1:~
```

Listing 6.2: moving files into MuRoX board [20]

6.2 Compiling from a Simulink model for MuRoX

Alternatively, an application for the *Reflective Operator* can be also compiled form a Simulink model. For this purpose, the user can modify the Simulink model that is already a part of the project, inside the *Reflective_Operator/Matlab_Reflective_Operator* from figure 3.1. Once the model has been edited, the user has to generate the code using the Matlab solver with the current settings.

After the code has been successfully generated, the user must go into the *Reflective_Operator/Demo_Applications/Demo_Matlab_Reflective_Operator* directory where a bash file with the name of *matlab_amiro_setup.sh* is. When executed, this shell script copies the necessary files into the current folder, copies a custom makefile that was created to cross-compile with the requirements for the MuRoX board, and compiles the application. The generated binary can be copied into the MuRoX board with the method described in section 6.1.

6.3 Example applications

For testing purposes, four demo applications were created. These applications are already inside the *Reflective Operator*, but the user can modify and recompile the source code which is obtainable from the DA_AMiRo git repository [18]. The modified versions of the source code can be compiled and uploaded to the MuRoX board using the method already described in section 6.1. The three demos are:

- *Demo Sender Reflective Operator*
- *Demo Receiver Reflective Operator*
- *Demo Standalone Reflective Operator*
- *Demo Matlab Reflective Operator*

The first one is called *Demo Sender Reflective Operator* and it is a simple application where the user can send commands to the internal controller with assistance of a set of menus. The application asks for the priority, the topic id, the mode to be set, and all the needed fields for each case. It also separates actuators from sensors. This application facilitates the sending of

commands, because the user does not have to memorize the topic ids or the CAN frame conventions.

In figures 6.1 and 6.2 a possible usage of the demo is shown. The menus can be seen, as the first one asks if the user wants to talk with an actuator or a sensor. If the answer is an actuator, like in figure 6.1 the second menu is to select which actuator to use and finally to set the according values to the desired actuator. If the user wants to talk with a sensor like in figure 6.2, the menus ask for the mode: stop, publisher, and one time transmission. Then it proceeds to ask the frequency if it was publisher, the topic and the priority.

```
root@192.168.1.1's password:
root@amiro:~# ./Demo_Sender_Reflective_Operator
*** AMiRo DEMO ***
Send command
Send command to actuator (0) or to sensor (1) ?
0
1. Led 1
2. Led 2
3. Led 3
4. Led 4
5. Led 5
6. Led 6
7. Led 7
8. Led 8
9. Motor command
5
Red value?(0-255)
 100
Green value?(0-255)
 50
Blue value?(0-255)
 120
```

Figure 6.1: Example of a *Demo Sender Reflective Operator* usage, part 1

The second application is called *Demo Receiver Reflective Operator* and it is basically a thread that outputs the information that the CAN bus receives corresponding to the sensors. The information is translated in English so the user can read it easily without the necessity of the convention shown in figures 4.1 and 4.2. The output shows the priority, a time stamp upon the reception of the CAN frame, the topic of the sensor's data, the value, and its units. This can be seen in figure 6.3, where an example of how the receiver displays the information of the gyroscope can be seen.

The idea behind these two applications *Demo Sender Reflective Operator* and *Demo Receiver Reflective Operator* is that the user runs both at the same time in two different terminal windows to send commands and to see how they are being answered in real time.

The third application is called *Demo Standalone Reflective Operator*. It has a determined functionality so the user needs only to run it in order to see its results with the help of the *Demo Receiver Reflective Operator* or the *candump* package. This Demo has a set of commands to be sent to the internal controller and shows the outputs of the sensors. The demo has the following expected functionality which can be seen in figures 6.4 and 6.5.

1. At 1s the CAN bus has a command frame asking for the accelerometer values every 500ms.
2. At 5s the CAN bus has a command frame asking for the proximity ring values every second.
3. At 10s the CAN bus has a command frame asking for the power status one time.
4. At 15s the CAN bus has a command frame asking for the gyroscope values every 5s.
5. At 16s the CAN bus has a command frame asking for the proximity ring values to stop being transmitted.
6. At 17s the CAN bus has a command frame asking for the magnetometer values every 800ms.
7. At 18s the CAN bus has a command frame asking for the accelerometer values to stop being transmitted.
8. At 21s the CAN bus has a command frame asking for the odometry values every 900ms.
9. At 23s the CAN bus has a command frame asking to set the motors speed at $50\mu m/s$.
10. At 25s the CAN bus has a command frame asking to turn LED 1 on.
11. At 27s the CAN bus has a command frame asking to stop the motors.

12. At 30s the CAN bus has a command frame asking for the odometry values to stop being transmitted.

13. At 31s the CAN bus has a command frame asking for the power status one time.

14. At 33s the CAN bus has a command frame asking for the proximity floor values every 2s.

15. At 35s the CAN bus has a command frame asking for the magnetometer values to stop being transmitted.

16. At 38s the CAN bus has a command frame asking for the proximity ring values to stop being transmitted.

17. At 40s the CAN bus has a command frame asking to turn LEDs 2, 4, 6, and 8 on.

The fourth demo is the *Demo Matlab Reflective Operator* and it is the resulting application from the Simulink model. The current demo turns on three LEDs on the AMiRo but the user can modify the Simulink model to fulfill any desired functionality. The way of compiling a modified version of the Simulink model was discussed in section 6.2.

Lastly, the user can also send independent CAN frames using the *candump* package installed in the MuRoX board, as it was already discussed in chapter 6. For this method, the user has to decode previously the topic id and the CAN data frames according to images 4.1 and 4.2.

```
root@amiro:~# ./Demo_Sender_Reflective_Operator
*** AMiRo DEMO ***
Send command
Send command to actuator (0) or to sensor (1) ?
1
Set mode:
0. Stop
1. Publisher
2. One time transmission
1
Set frequency value
1
Select time units:
0. miliseconds
1. seconds
2. microseconds
1
Select topic:
3. Odometry
4. Proximity Floor sensors
5. Magnetometer
8. Gyroscope
9. Accelerometer
1e. Power status
1f. Proximity ring
1f
Set priority: 0, 1, 2, or 3
1
root@amiro:~# ./Demo_Sender_Reflective_Operator
*** AMiRo DEMO ***
Send command
Send command to actuator (0) or to sensor (1) ?
1
Set mode:
0. Stop
1. Publisher
2. One time transmission
0
Select topic:
3. Odometry
4. Proximity Floor sensors
5. Magnetometer
8. Gyroscope
9. Accelerometer
1e. Power status
1f. Proximity ring
1f
Set priority: 0, 1, 2, or 3
1
```

Figure 6.2: Example of a *Demo Sender Reflective Operator* usage, part 2

```
PRIO: HIGH, Time: 111.480000 seconds
TOPIC: Gyroscope_x, DATA: 70 dps
TOPIC: Gyroscope_y, DATA: 39 dps
TOPIC: Gyroscope_z, DATA: 5 dps
```

Figure 6.3: Example of a *Demo Receiver Reflective Operator* for the gyroscope

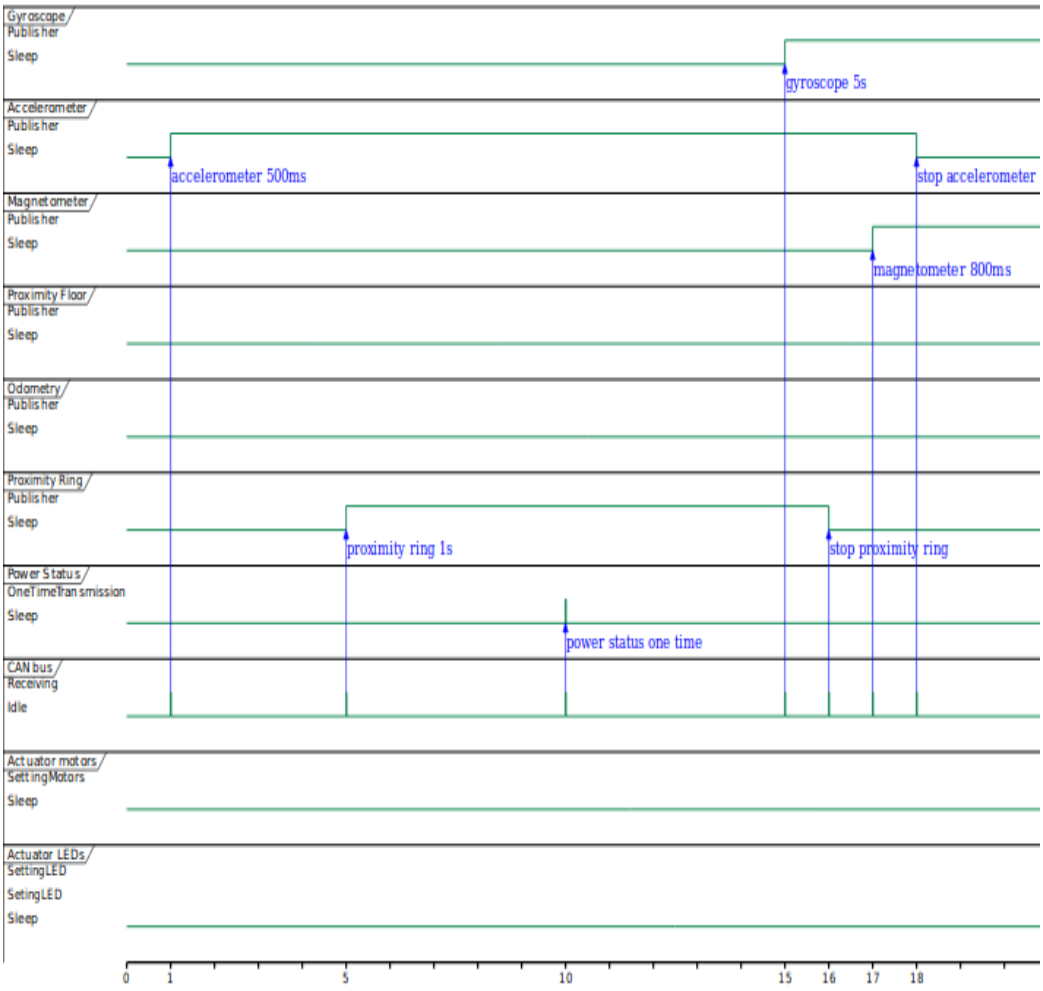


Figure 6.4: Demo Standalone sequence part 1

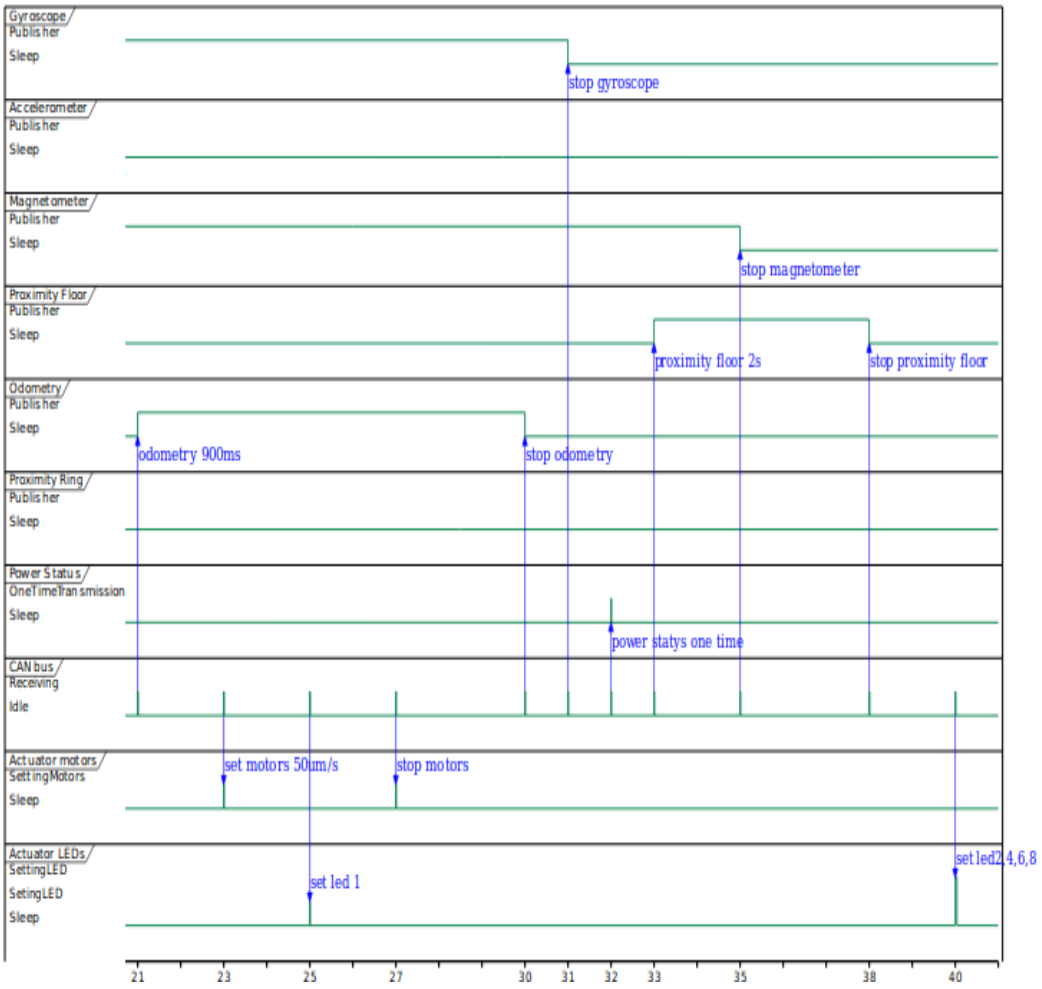


Figure 6.5: Demo Standalone sequence part 2

Chapter 7

Results

7.1 DA_AMiRo performance

In this section the performance of the AMiRo will be revised based the results of some testing conducted to the AMiRo. From all the testing conducted, the AMiRo's response to the *Demo Standalone Reflective Operator* will be the one discussed in this section, because its desired functionality was already explained in section 6.3.

In figure 7.1 the process described in figures 6.4 and 6.5 is shown and it displays the expected behavior of the *Demo Standalone Reflective Operator*. This figure also have time stamps corresponding to the time at which each CAN frame was sent to the CAN bus. Among the listing of commands that are sent to the CAN bus by this application, the reader can also look at the actuators commands such as the motor's velocity and the turning on and off of the LEDs.

Now lets have a look at the results. In figure 7.2 the values of the accelerometer and the proximity ring sensors are displayed. In figure 7.3 values for the magnetometer, gyroscope, velocity, and encoders are displayed. The velocity and encoder CAN frames show that the motors were on-going at the moment of the CAN frame being sent. And finally in figure 7.4 the power status gets shown, the magnetometer values appear as well, and also the proximity floor sensors are displayed.

The functionality and frequency of the expected values is very satisfactory, because the delays between every value from the same sensor are exactly as the requested in the *Demo Standalone Reflective Operator* seen in figure

7.1. And even though as seen on top of figure 7.3, the CAN frames for the magnetometer values are not sent in a row but the value of the velocity gets sent before the magnetometer 'y' value. This behavior is expected as it adheres to the requested frequencies. In the case from the velocities requested frequency, the $900ms$ between two velocity CAN frames is precise.

As it can be seen, the delay between two CAN frames of the same sensor, correspond to the frequencies requested from the *Demo Standalone Reflective Operator* as seen in figure 7.1. There is however, an expected delay between the sensors that get sent together in different CAN frames. Such is the case of the magnetometer values, that are sent in three different CAN frames and the proximity floor ring sensors that are sent in eight different CAN frames as explained in section 4.2.1. This delay is of $20ms$ and affects only the frequencies of said sensor values. So for example if the magnetometer value gets requested at a period of time shorter than every $20ms$, then the CAN frames will not be sent appropriately. This delay does not affect other possible parallel tasks.

For starters, lets look into the DA_AMiRo efficiency. From the programming of the three basic boards, a $545\mu s$ delay of timeouts in every transmission is expected. This number can be explained with the following equation:

$$1us(1 + 11 + 1 + 1 + 1 + 4 + 64 + 15 + 1 + 1 + 1 + 7) * 5 = 545us$$

$$1/(1MHz) * (SOF + ID + RTR + IDE + RES + DLC + DATA + CRC +$$
$$DELIM + ACK + DELIM + EOF) * No.RETRIES$$

taken from the original code of the repository [26].

7.2 DA_AMiRo vs. AMiRo OS

In this section the structure and functionality of the AMiRo OS and the OCM approach for the DA_AMiRo project are compared.

The AMiRo OS is a very complete system that has a lot of possible applications for different purposes, from line followers, to a LIDAR that can detect obstacles. It also has a command line shell in which the user can communicate with the AMiRo in very simple commands. This feature is very helpful to get an immediate status of a sensor or to get a quick response from the AMiRo.

Every board has a user thread where a functionality using the sensors of the board is implemented. Everyone of these boards gets to broadcast its sensor values via CAN with a pre-selected frequency. A good way the user can interact with the sensors and the boards themselves, is via the *Cognition Board* that also has pre-compiled demos in the Wiki [20] for different possible applications.

The AMiRo OS has all of the possible functionalities already implemented and this takes a lot of the processing power from the boards, this is one of the main differences between this and the DA_AMiRo project. The latter has a specific purpose and the processing power is focused towards this goal, making it more resource effective. A good example of this is that the DA_AMiRo does not have possible demos implemented like the line follower, the motor corrections, and the obstacle avoidance for mentioning a few.

What appears to be similar between these two projects is the CAN broadcasting of the AMiRo OS and the CAN sensor handler of the DA_AMiRo project. The AMiRo OS sends the values of the sensors all together via the CAN bus at a specific frequency of $16Hz$, while the DA_AMiRo project has a much more complex sensor value handling as it has been already explained through the past chapters.

One of the biggest advantages of the DA_AMiRo project is the compatibility of the Simulink models from Matlab to generate applications for the Reflective Operator of the AMiRo. The students can easily edit the given model to operate according a desired goal. This method is easy because with the Simulink model the students can change logic in a visual fashion rather than a code one. Instead of having to remember the CAN frame conventions or worrying about a programming language issue, they can simply change the values of the model or create logic with simple block logic in Simulink.

7.3 Outlook

The real-time functionality in the DA_AMiRo project is works as expected: it fulfills the performance deadlines and does not loose packages. After performing a stress test on the system, the error range or delays found are still small enough to make the project fulfill the requirements of a soft real-time system.

The system is also capable of handling all the available tasks from the sensors at once, and communicates successfully with the actuators, changing

its values at the will of the user. On top of this, the DA_AMiRo project is fulfilling the CAN frame conventions established by Uwe Jahn.

With the help of the Simulink models, the usage of the Reflective Operator becomes very simple. This feature makes the DA_AMiRo project a good tool for students to work with real-time systems and learn from them in a simple and visual way.

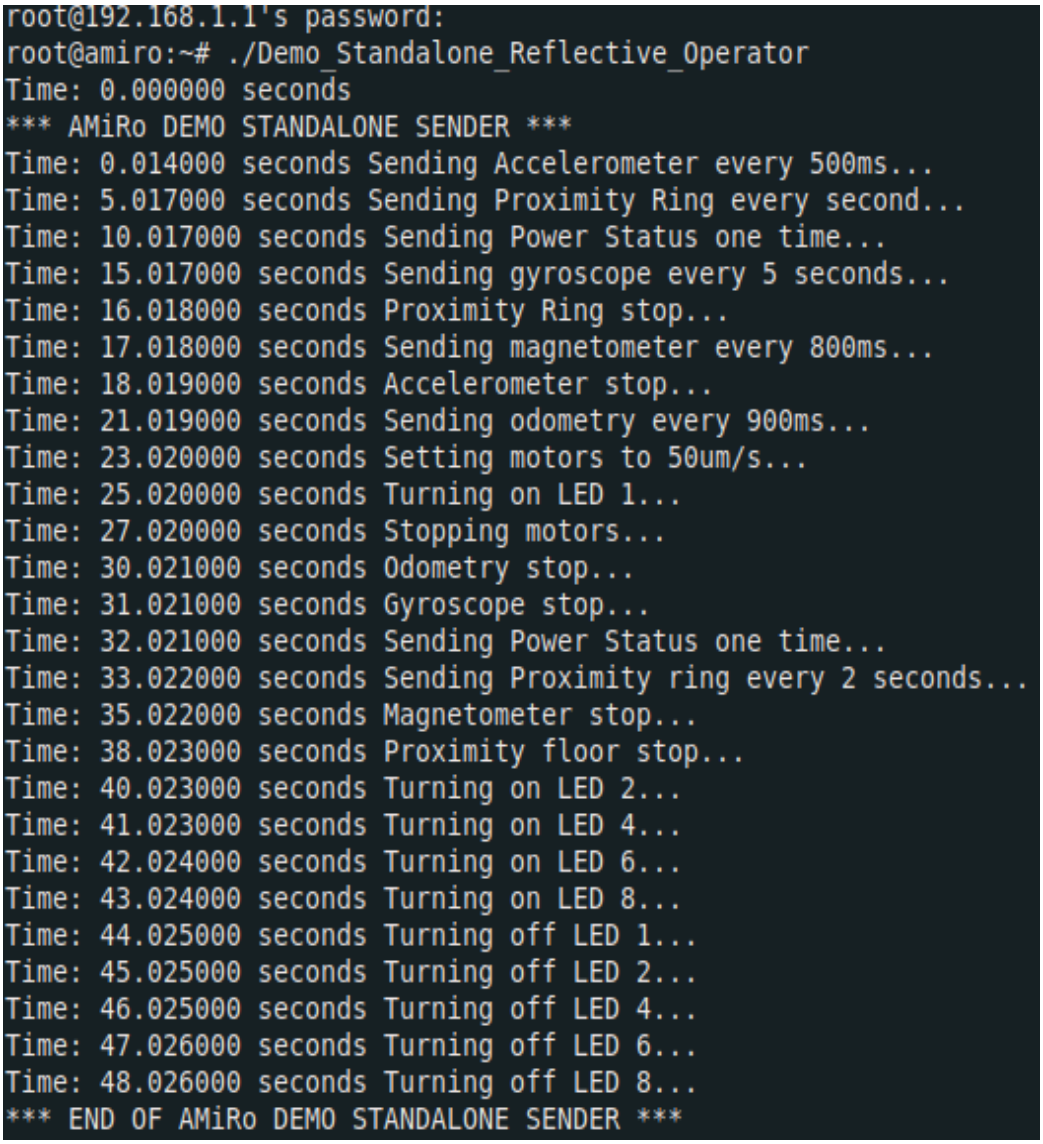


Figure 7.1: Demo Standalone results sender part

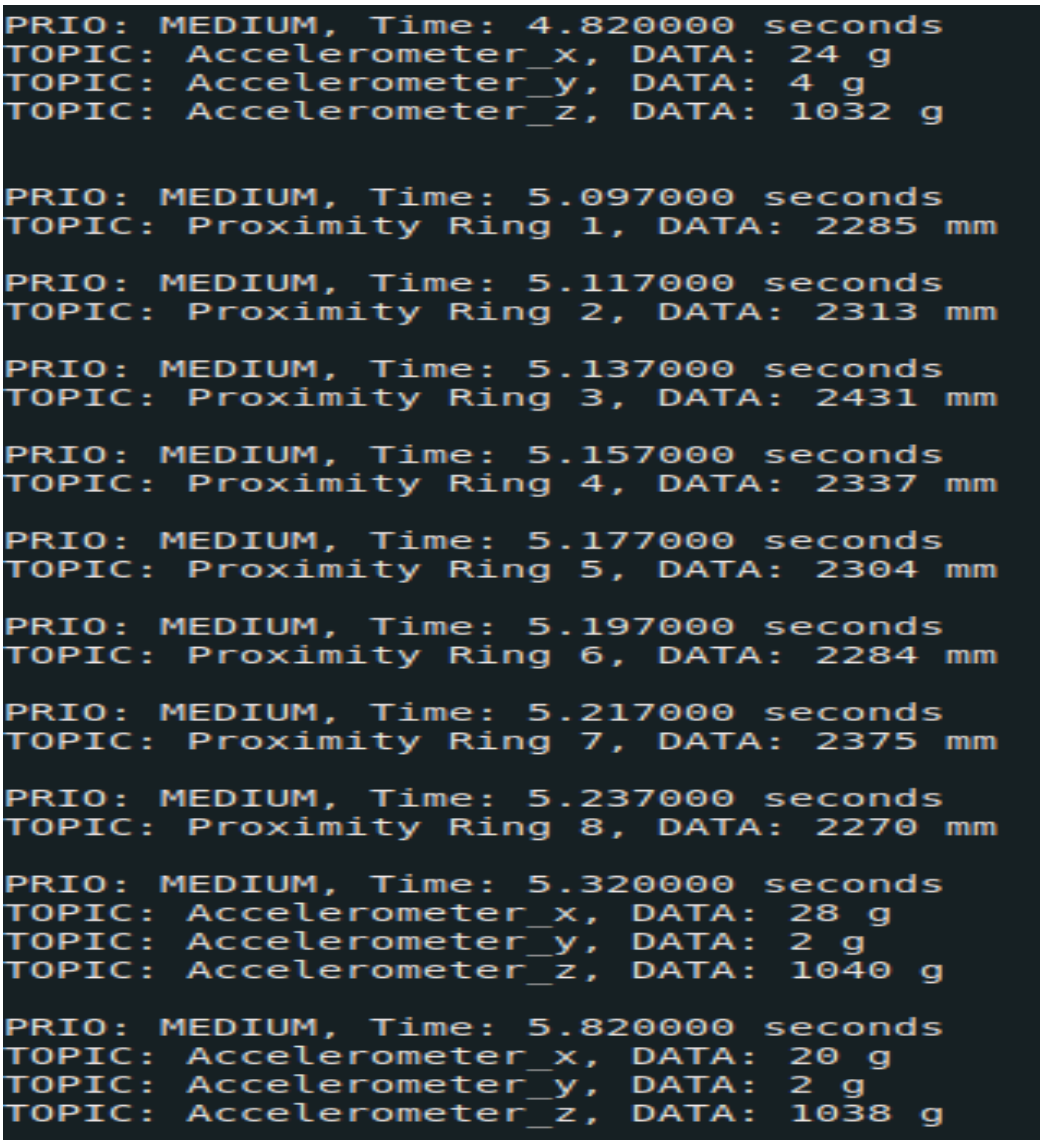


Figure 7.2: Demo Standalone results part 1

```
PRIO: MEDIUM, Time: 23.880000 seconds
TOPIC: Magnetometer_x, DATA: -10240 µGAUSS

PRIO: MEDIUM, Time: 23.900000 seconds
TOPIC: X velocity, DATA: 899884 µm/s
TOPIC: Angular velocity, DATA: 549573 µrad/s

PRIO: MEDIUM, Time: 23.920000 seconds
TOPIC: Magnetometer_y, DATA: -788480 µGAUSS

PRIO: MEDIUM, Time: 23.960000 seconds
TOPIC: Magnetometer_z, DATA: 1574400 µGAUSS

PRIO: MEDIUM, Time: 23.980000 seconds
TOPIC: Encoder_x, DATA: 2504 µm
TOPIC: Encoder_y, DATA: 614 µm
TOPIC: Orientation of the wheels, DATA: 1681 µrad

PRIO: MEDIUM, Time: 24.680000 seconds
TOPIC: Magnetometer_x, DATA: -5120 µGAUSS

PRIO: MEDIUM, Time: 24.720000 seconds
TOPIC: Magnetometer_y, DATA: -783360 µGAUSS

PRIO: MEDIUM, Time: 24.760000 seconds
TOPIC: Magnetometer_z, DATA: 1576960 µGAUSS

PRIO: MEDIUM, Time: 24.800000 seconds
TOPIC: X velocity, DATA: 905718 µm/s
TOPIC: Angular velocity, DATA: 380473 µrad/s

PRIO: MEDIUM, Time: 24.880000 seconds
TOPIC: Encoder_x, DATA: 5101 µm
TOPIC: Encoder_y, DATA: 2425 µm
TOPIC: Orientation of the wheels, DATA: 3018 µrad

PRIO: MEDIUM, Time: 25.059000 seconds
TOPIC: Gyroscope_x, DATA: 65522 dps
TOPIC: Gyroscope_y, DATA: 65493 dps
TOPIC: Gyroscope_z, DATA: 16 dps


PRIO: MEDIUM, Time: 25.480000 seconds
TOPIC: Magnetometer_x, DATA: -12800 µGAUSS

PRIO: MEDIUM, Time: 25.519000 seconds
TOPIC: Magnetometer_y, DATA: -780800 µGAUSS

PRIO: MEDIUM, Time: 25.559000 seconds
TOPIC: Magnetometer_z, DATA: 1582080 µGAUSS
```

Figure 7.3: Demo Standalone results part 2

```
PRIO: MEDIUM, Time: 32.083000 seconds
TOPIC: charging flag, DATA: OFF
TOPIC: state of charge, DATA: 100 % remaining
TOPIC: Time remaining, DATA: 274 minutes remaining
TOPIC: Power consumption, DATA: 64585 mW

PRIO: MEDIUM, Time: 32.139000 seconds
TOPIC: Magnetometer_y, DATA: -803840 µGAUSS

PRIO: MEDIUM, Time: 32.199000 seconds
TOPIC: Magnetometer_z, DATA: 1579520 µGAUSS

PRIO: MEDIUM, Time: 32.959000 seconds
TOPIC: Magnetometer_x, DATA: -5120 µGAUSS

PRIO: MEDIUM, Time: 33.019000 seconds
TOPIC: Magnetometer_y, DATA: -793600 µGAUSS

PRIO: MEDIUM, Time: 33.080000 seconds
TOPIC: Magnetometer_z, DATA: 1579520 µGAUSS


PRIO: MEDIUM, Time: 33.123000 seconds
TOPIC: Floor sensor 1, DATA: 5969 luminance
TOPIC:  Floor sensor 2, DATA: 0 luminance
TOPIC:  Floor sensor 3, DATA: 0 luminance
TOPIC:  Floor sensor 4, DATA: 0 luminance

PRIO: MEDIUM, Time: 33.803000 seconds
TOPIC: Magnetometer_x, DATA: -12800 µGAUSS

PRIO: MEDIUM, Time: 33.843000 seconds
TOPIC: Magnetometer_y, DATA: -791040 µGAUSS

PRIO: MEDIUM, Time: 33.883000 seconds
TOPIC: Magnetometer_z, DATA: 1579520 µGAUSS

PRIO: MEDIUM, Time: 34.603000 seconds
TOPIC: Magnetometer_x, DATA: -2560 µGAUSS

PRIO: MEDIUM, Time: 34.643000 seconds
TOPIC: Magnetometer_y, DATA: -798720 µGAUSS

PRIO: MEDIUM, Time: 34.683000 seconds
TOPIC: Magnetometer_z, DATA: 1576960 µGAUSS


PRIO: MEDIUM, Time: 35.083000 seconds
TOPIC: Floor sensor 1, DATA: 5959 luminance
TOPIC:  Floor sensor 2, DATA: 0 luminance
TOPIC:  Floor sensor 3, DATA: 0 luminance
TOPIC:  Floor sensor 4, DATA: 0 luminance
```

Figure 7.4: Demo Standalone results part 3

Bibliography

[1] *Amiro cheatsheet.*

[2] FreeRTOS. (2015). What is an rtos?, [Online]. Available: `http://www.freertos.org/about-RTOS.html`.

[3] T. Graydon and B. Flanagan. (Aug. 30, 2016). Poky linux, [Online]. Available: `http://www.pokylinux.org/`.

[4] C. Henke, M. Tichy, T. Schneider, J. Böcker, and W. Schafer, "System architecture and risk management for autonomous railway convoys", pp. 1–8, May 2008.

[5] S. Herbrechtsmeier, "Amiro – autonomous mini robot for research and education", Cluster of Excellence Cognitive Interaction Technology 'CITEC', Bielefeld University, Bielefeld, Germany, Project presentation. [Online]. Available: `http://www.ks.cit-ec.uni-bielefeld.de/`.

[6] ——, "Amiro – autonomous mini robot for research and education (extensions)", Cluster of Excellence Cognitive Interaction Technology 'CITEC', Bielefeld University, Bielefeld, Germany, Project presentation. [Online]. Available: `http://www.ks.cit-ec.uni-bielefeld.de/`.

[7] T. Hestermeyer, O. Oberschelp, and H. Giese, "Structured information processing for self-optimizing mechatronic systems", in *Proceedings of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*, (published), IEEE Computer Society Press, Aug. 2004.

[8] M. Hill, *Concise Encyclopedia of Engineering.* 2002.

[9] Honeywell. (). 3-axis digital compass ic hmc5883l, [Online]. Available: https://cdn-shop.adafruit.com/datasheets/HMC5883L_3-Axis_Digital_Compass_IC.pdf.

[10] HyperPhysics. (). Hall effect, Department of Physics and Astronomy, Georgia State University, [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/magnetic/Hall.html#c2.

[11] N. Instruments. (2018). What is a real-time operating system (rtos)?, [Online]. Available: http://www.ni.com/white-paper/3938/en/.

[12] T. Instruments. (2018). Bq27500 (nrnd) system-side impedance track™ fuel gauge, [Online]. Available: http://www.ti.com/product/BQ27500#.

[13] U. Jahn. (2017). Daebot wiki, [Online]. Available: https://gitlab.idial.institute/DAEbot/DAEbot/wikis/home.

[14] S. M. ltd. (2018). Stm32f3discovery overview, [Online]. Available: http://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-discovery-kits/stm32f3discovery.html.

[15] C. Mathas. (). Ambient-light sensors mimic the human eye, [Online]. Available: https://www.digikey.com/en/articles/techzone/2014/feb/ambient-light-sensors-mimic-the-human-eye.

[16] S. mems and sensors. (2011). L3g4200d: Three axis digital output gyroscope, [Online]. Available: https://www.elecrow.com/download/L3G4200_AN3393.pdf.

[17] ——, (). Mems digital output motion sensor ultra low-power high performance 3-axes "nano" accelerometer, [Online]. Available: https://www.st.com/resource/en/datasheet/cd00213470.pdf.

[18] H. G. Munoz Hernandez. (2018). Da_amiro repository, [Online]. Available: https://gitlab.idial.institute/DAdevs/DA_AMiRo.git.

[19] ——, "Daebot, internal controller", Fachhochschule Dortmund, Research project, 2014.

[20] (2016). Murox wiki, [Online]. Available: http://www.multirobotix.de/dokuwiki/doku.php?id=start.

[21] Plantuml.com. (2018). Plantuml home page, [Online]. Available: http://plantuml.com/.

[22] R. Queinell, "Embedded market study", *In ARM TechCon*, p. 34, 2015.

[23] U. Rückert, "Amiro: A mini robot as versatile teaching platform", Cluster of Excellence Cognitive Interaction Technology 'CITEC', Bielefeld University, Bielefeld, Germany, paper, 2018. [Online]. Available: http://www.ks.cit-ec.uni-bielefeld.de/.

[24] ——, "Amiro: A mini robot for scientific applications", Cluster of Excellence Cognitive Interaction Technology 'CITEC', Bielefeld University, Bielefeld, Germany, paper, 2015. [Online]. Available: http://www.ks.cit-ec.uni-bielefeld.de/.

[25] ——, "Amiro: A modular & customizable open-source mini robot platform", Cluster of Excellence Cognitive Interaction Technology 'CITEC', Bielefeld University, Bielefeld, Germany, paper, 2016. [Online]. Available: http://www.ks.cit-ec.uni-bielefeld.de/.

[26] T. Schöpping. (2018). Amiro-os, [Online]. Available: https://openresearch.cit-ec.de/projects/amiro-os/wiki.

[27] G. D. Sirio. (2018). Chibios home page, [Online]. Available: http://www.chibios.org/dokuwiki/doku.php.

[28] ——, (2018). Chibios rt manual, [Online]. Available: http://chibios.sourceforge.net/docs3/rt/index.html.

[29] C. Sondermann-Wölke and W. Sextro, "Integration of condition monitoring in self-optimizing function modules applied to the active railway guidance module", Jan. 2010.

[30] Sparkfun. (). What is a gyroscope, [Online]. Available: https://learn.sparkfun.com/tutorials/gyroscope/all.

[31] ——, (). What is an accelerometer, [Online]. Available: https://learn.sparkfun.com/tutorials/accelerometer-basics.

[32] Vishay. (). Fully integrated proximity and ambient light sensor with infrared emitter, i2c interface, and interrupt function, [Online]. Available: https://www.vishay.com/docs/83476/vcnl4020.pdf.

[33] F. Voorburg. (Aug. 30, 2016). Openblt, [Online]. Available: http://feaser.com/en/openblt.php.

[34] W3C. (). Magnetometer, [Online]. Available: https://www.w3.org/TR/magnetometer/.