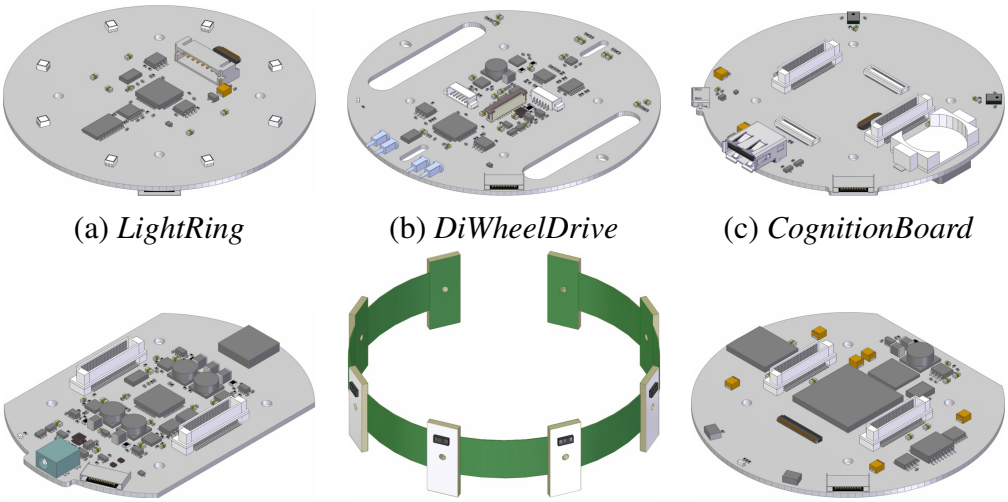
AMiRo Cheatsheet

Last Update: September 2016

Contents

1 What is the AMiRo?

The AMiRo - **A**utonomous **Mi**ni **Ro**bot - is a mini robot for research and education. Basic algorithms can be implemented and evaluated very easily due to small requirements. The construction is modular, where each module has its own processing unit. The modules are connected by a CAN bus. The basic modules are the *DiWheelDrive*, the *PowerBoard* and the *LightRing* as shown in figure 1.1(a, b, d) and in figure 1.2.



(a) *LightRing* (b) *DiWheelDrive* (c) *CognitionBoard*

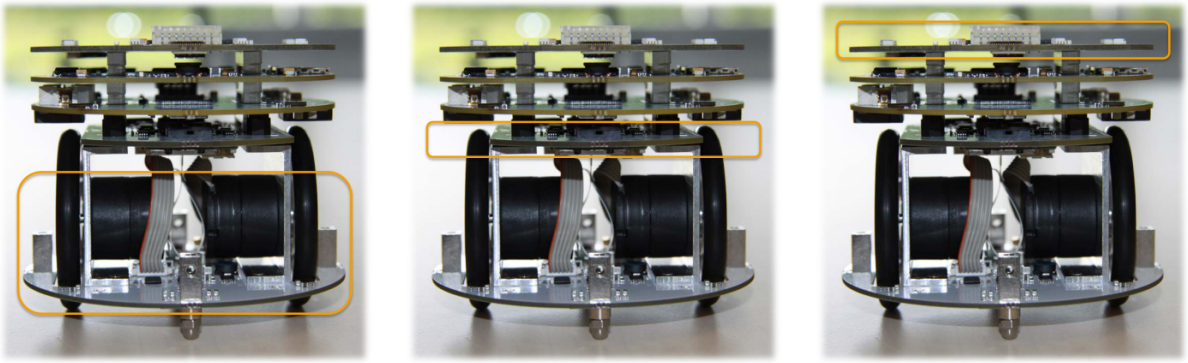
(d) *PowerBoard* (e) *ProximityRing* (f) *ImageProcessing*

Figure 1.1: Current set of AMiRo modules.

The *DiWheelDrive* (figure 1.1(b)) is for drive control, position estimation and ground detection. By using a STMicroelectronics ARM Cortex-M3 based STM32F103 MCU, it controls the motors by receiving driving commands and calculating the rotation speed for the odometry. It also gives the position of the robot by magnetometer, accelerometer and gyroscope. Additionally by using proximity sensors at the bottom, it can detect the ground.

The *PowerBoard* (figure 1.1(d)) is basically responsible for the power supply. It has the battery fuel gauge and charger and the voltage regulators for different voltage connections. Additionally it controls the *ProximityRing* around the AMiRo (figure 1.1(e)), which contains eight infrared proximity sensors and four capacity touch sensors, and the wireless communication via bluetooth. For the processes a STMicroelectronics ARM Cortex-M4 based STM32F104 MCU is used, which also consists of a FPU.

The *LightRing* (figure 1.1(a)) is for status visualization and optical communication. Additionally sensor extensions like a laser scanner, which is not included in the AMiRo, can be connected. The processing unit is the same MCU as on the *DiWheelDrive*.



(a) *DiWheelDrive* (b) *PowerBoard* (c) *LightRing*

Figure 1.2: Front view of the modules of the AMiRo. Between the *PowerBoard* and the *LightRing* there can be stacked module extensions like the *CognitionBoard*.

Between the *PowerBoard* and the *LightRing* there can be stacked module extensions as shown in figure 1.2. The most interesting module for this cheatsheet is the *CognitionBoard* (figure 1.1(c)). It contains a Gumstix Overo TidalSTORM COM, which combines an ARM Cortex-A8 based DaVinci DM3730 SoC from Texas Instruments running at 1 GHz with 1 GB RAM and a micro SD card as hard drive. An usual Linux operating system is installed. By using the CAN bus the sensor data from the other modules can be loaded and commands can be sent. Additionally it has an own wireless communication card and an additional micro USB slot, so the *CognitionBoard* can be connected to any wired or wireless IP connection.

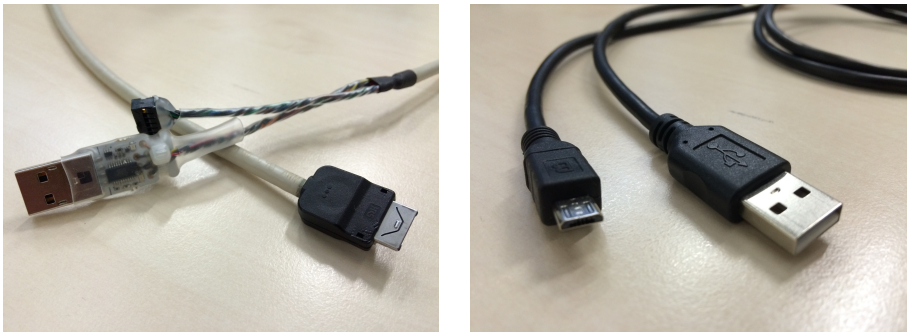
Additionally the *ImageProcessing* (figure 1.1(f)) could be added, which isn't finally developed yet. It shall contain a FPGA for parallel image processing.

On the title page different AMiRos are shown. The left AMiRo in the foreground, which is touched by a hand, is the basic AMiRo only consisting the modules *DiWheelDrive*, *PowerBoard* and *LightRing* with its lights turned on. The right AMiRo in the foreground (its front is to the left side) doesn't have its chasis and its *ProximityRing*. The wheels at its sides and the battery packs in its front and back can be seen. Additionally it consists of the extension modules *CognitionBoard* and *ImageProcessing* between the *PowerBoard* and the *LightRing*, so it is higher than the left AMiRo. At its front, the small camera, which is connected to the *CognitionBoard*, can be seen. In the background there are several AMiRos with different sensor extensions like a laser scanner or a RGBD camera (camera with pixel values consisting of **r**ed, **g**reen and **b**lue channels and **d**epth) on top of the robots.

In the following chapters the descriptions and algorithms are designed for the *CognitionBoard*. The basic modules *DiWheelDrive*, *PowerBoard* and *LightRing* will only be used, but not programmed or changed.

2 What do we need at first?

For basic communication between the computer and the AMiRo, the *programming cable* which is shown in figure 2.1(a) is needed. It can be used for basic console commands on all boards and for flashing new programms on the microcontrollers of the *DiWheelDrive*, *PowerBoard* and the *LightRing*.



(a) (b)

Figure 2.1: The *programming cable* (a) and micro USB cable (b).

For copying programs on the *CognitionBoard* a ssh connection is needed. This can be accomplished in different ways:

- Using a *micro USB cable*:

  The *micro USB cable*, which is shown in figure 2.1(b), has to be pluged into the micro USB connector of the *CognitionBoard*. The basic IP address for the ssh connection for every AMiRo is 192.168.3.1. On Linux machines there has to be initialized an *usb0 interface* with the IP address 192.168.3.<2-255>.

- Using the *citopenrobotix* network:

  With the internal network card on the *CognitionBoard* the AMiRos connect themselfs to the *citopenrobotix* network automatically. The computer just has to be in the same network. By typing *ifconfig* into the console of the *CognitionBoard* the IP address of the AMiRo can be read.

2.1 Access to murox Repository

The programs for the Linux evironment of the *CognitionBoard* are stored in the murox repository. Please contact the supervisor for the link and the access data. The murox repository is a git repository, which has to be cloned recursively on the computer. The structure is described in chapter 4. The programs have to be copied on the *CognitionBoard* via a ssh connection.

To build the projects of the murox repository, it has to be initialized:

1. Open the console on the computer.
2. Go to the cloned murox repository.
3. Got to the *usability* folder: `$ cd project/utilities/`
4. Execute initialization script: `$ ./initMurox`
5. Go to home directory: `$ cd`
6. Source the bashrc: `$ source .bashrc`

Now global variables for murox path specifications have been added to the bashrc. Do not change the additions manually. If the murox repository has been moved, the paths have to be reinitialized. In this case the murox lines in the bashrc just have to be removed and the described initialization has to be done again.

2.2 Initialization of GTKTerm

GTKTerm is a program for serial communication. It is needed to have access to the consoles of the AMiRo boards. For the connection with the AMiRo, the configuration file of GTKTerm has to be edited:

1. Open the configuration file: `$ nano ~/.gtktermrc`
2. Change the following lines as shown:
   ```
   > port = /dev/ttyUSB0
   > speed = 115200
   > crlfauto = True
   ```

2.3 Start and Connect the AMiRo

For starting and connecting with the AMiRo there is a basic procedure:

1. Plug the *programming cable* into the computer.
2. Start GTKTerm: `$ gtkterm`
3. Reset RTS flag by pressing F8 (the RTS flag in the right bottom corner has to be gray).

4. Plug the *programming cable* into the serial slot of the *DiWheelDrive*.

If the AMiRo is shutdowned, the RTS flag has to be set and reset by double pressing F8 again.

The RTS flag is the reset flag for the AMiRo. If it is set, all processing units on the boards are turned off. If the flag is reset, all processing units are running. GTKTerm automatically sets the RTS flag. If the AMiRo is turned on and the *programming cable* is beeing pluged in, the AMiRo will be turned off without any save shutdown procedure. This is the reason, why the flag has to be reset befor pluging in the *programming cable*.

After the described procedure the AMiRo turns on and in the console output of GTKTerm there should be running an interactive console. By for example typing *help* a list of all console commands of the *DiWheelDrive* will be shown. For getting access on the console of the *CognitionBoard*, the following steps have to be done:

1. Unplug the *programming cable* from the *DiWheelDrive* only.
2. Plug the *programming cable* into the serial slot of the *CognitionBoard*.

If this is done directly after turning the AMiRo on, there will be the setup output of the Linux operating system. Soon the login request will be shown. If there is no console output showing by GTKTerm, please press return. The login request should be shown now.

If the supervisor doesn't give any login information, the basic login is *root*. If it is asked for a password, just press return. Now basic Linux commands can be used for navigation and execution like in any other Linux console.

2.4 Stopping and Shutting down the AMiRo

If a program of the *CognitionBoard* is broken or has been stopped, all functions, which are controlled by this program, will be stopped on the *CognitionBoard*. But there are also functionalities, which are not controlled by any program of the *CognitionBoard* like the motor control, which is done by the *DiWheelDrive*. If a program for example sets motor speeds and breaks, the motors will continue driving.

For really stopping the AMiRo, it is important to execute the stop tool *stopAMiRo* in *project/-tools/robotTools/* on the *CognitionBoard*. It stops the motors and resets the lights to the initial colors of the *LightRing*.

For shutting down the AMiRo, please do the following steps:

1. Plug the *programming cable* into the *DiWheelDrive* (the RTS flag has to be reset!)
2. Type the shutdown command for deep sleep: `$ shutdown d`

The *programming cable* can be unplugged now. The AMiRo starts the save shutdown procedure. Please, do not use the RTS flag for shutting down. If the flag is set, the AMiRo is turned off, but when the *programming cable* will be unplugged, the AMiRo restarts.

2.5 Building and Running first Program

At first the program has to be build. Please follow the following steps:

1. Open the console on the computer.
2. Go to the cloned murox repository.
3. Go to the project: `$ cd project/demo/braitenberg`
4. Load the murox environment: `$ module load murox/env`
5. Build the project: `$ ./build.sh`

Now the project is build for the AMiRo. For copying and starting the program example the ssh connection to the AMiRo is needed:

1. Copy the project on the AMiRo: `$ ./copyprograms <AMiRo IP address>`
2. Open console of the *CognitionBoard* of the AMiRo.
3. Go to the project: `$ cd root/braitenberg`
4. Start program: `$ ./runSimple.sh`

Now the AMiRo behaves like a very simple Braitenberg vehicle. It drives forward and turns if there is an object by using the sensors of the *ProximityRing*. It doesn't stop in front of edges! To change the sensitivity open the script *runSimple.sh* and change the integer value of the parameter *-f*. The behavior of the proximity sensors in the *ProximityRing* is described in chapter 5.1.2.

For stopping the program press strg+C and execute *stop.sh* afterwards. The execution of the stop tool *stopAMiRo*, which has been described in chapter 2.4, is already included in the stop script, so it doesn't have to be executed manually.

3 AMiRo's Setup

Some sensors of the AMiRo need a setup before they can be used efficiently. For sensor details please have a look into chapter 5.1. For the sensor initialization there is a project, which has to be build and copied on the AMiRo (by using ssh connection):

1. Open the console on the computer.
2. Go to the cloned murox repository.
3. Go to the initial project: `$ cd project/demo/initial`
4. Load the murox environment: `$ module load murox/env`
5. Build the project: `$ ./build.sh`
6. Copy the project: `$ ./copyPrograms.sh <AMiRo IP address>`
7. Open the console on the *CognitionBoard* of the AMiRo.
8. Go to the initial project: `$ cd root/initial`

Now there are different scripts for the initialization of the sensors. Please have a look into chapter 5.1 for the initialization details of the needed sensor.

3.1 Program Example with Setup

At first execute the initialization of the *ProximityRing*, which is described in chapter 5.1.2. Afterwards just build and copy the Braitenberg program as described in chapter 2.5.

Before execution, please check, if in *root/initial* there exist the files *irConfig.conf* and *irEmpty.conf* and if the ground offsets belong to the ground the AMiRo shall now drive on. Then execute the script *runStandard.sh*.

The AMiRo behaves like a simple Braitenberg vehicle again, but now it is more sensitive to objects and it can detect table edges. Due to the initialization of the *ProximityRing*, an obstacle and an edge model can be used for precise distance calculations.

4 Structure of murox Repository

The murox repository contains several programs for the *CognitionBoard*. These are sorted into different subdirectories based on their functionality. All programs and the structure is documented in the murox documentation:
www.multirobotix.de/dokuwiki/doku.php
Please contact your supervisor for login information.

Basically there are seven subdirectories:

Subdirectory	**Description**
demo	Demos of the AMiRo. A demo is a conclusion of programs, which are implemented in the other directories. In demos there are only scripts and executables, but there isn't any program code.
sense	All sensing tools (sensors, cameras, etc.).
act	All tools for executing actions (motors, lights).
process	All processing tools including exploration, navigation, object detection, statemachines, etc.
tools	Tools for testing or stopping procedures on the AMiRo or tools for communication between the computer and the AMiRo.
sandbox	Development directory. All programs and demos, which are still in development and are not runnable or functional, are stored here.
includes	All programs and headers, which are included in the other programs, for example headers for AMiRo and CAN constants.

For the AMiRo there can be implemented **programs** and there can be designed **demos**. A program contains program code and can be built directly. A demo concludes some programs to one project, but there isn't any program code. In a demo the referenced programs can be built and only their executables are stored in this project. Additionally there can be added some scripts for faster executions and communications.

If a new project shall be created with new programs, the new demo and the new programs will be stored in *sandbox* at first. Here are stored all not runnable or functional programs and demos. Later, if it's running stable, the program will be moved to one of the directories *sense*, *act*, *process* or *tools* based on its functionality. A stable demo will be moved to the directory *demo*.

4.1 Robotic Service Bus (RSB)

The **R**obitic **S**ervice **B**us (RSB) is a message-oriented, event-driven middleware. It is the basic communication framework for all murox programs, which are cooperating with each other. For example the sense tools, which are described in chapter 4.2, are publishing their sensor data via RSB. For a full documentation of RSB and for program examples in different programming languages, please visit the Cor-Lab's documentation website:
`docs.cor-lab.de`

If many programs are publishing and receiving data via RSB, it is very important to get an overview about the published scopes by the RSB logger. The following commands are possible:

RSB logger command	Description
`rsb-loggercpp0.11 --style monitor <scope>`	Gives an overview about all scopes and their publication frequency, which are under the given scope.
`rsb-loggercpp0.11 --style detailed <scope>`	Gives all details about the published packages of all scopes, which are under the given scope.

The Braitenberg demo, which has been presented in the chapters 2.5 and 3.1, consists of two basic programs: the main Braitenberg behavior calculation in *braitenberg* and the reading of the proximity sensors of the *ProximityRing* by *senseRingProximity*. The last tool publishs the read sensor data via RSB, which can be received by the main program. In the following subsections the RSB logger output for both demos will be presented.

4.1.1 RSB Logger of Simple Braitenberg Demo

If the simple Braitenberg demo will be started like in chapter 2.5, an RSB communication overview can be called by the following command:

```
$ rsb-loggercpp0.11 --style monitor /
```

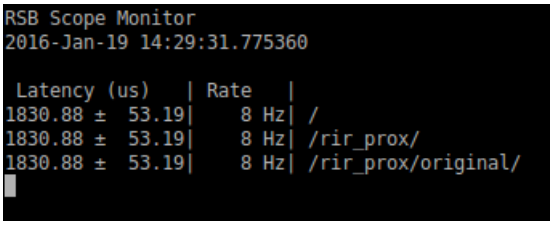


Figure 4.1: Monitor style output of the RSB logger of the simple Braitenberg demo.

As shown in figure 4.1 all RSB communication scopes are printed, which start with "/". For every scope the latency in micro seconds and the frequency is listed.

If there are needed more detailed information about for example the sensor data publications, the RSB logger can be called as follows:

```
$ rsb-loggercpp0.11 --style detailed /rir_prox/original
```

Now the RSB logger gives every detailed information about every package that is published via the given scope. This example is shown in figure A.1. The most important information is the scope name, the type and the payload. In this case it is a byte array of 36 bytes. The values themselves are hard to read, but in this case relative value changes can be checked.

4.1.2 RSB Logger of Standard Braitenberg Demo

The standard version of the braitenberg demo, which has been built and started in chapter 3.1, can also be analized like the simple version in the foreign chapter:

```
$ rsb-loggercpp0.11 --style monitor /
```

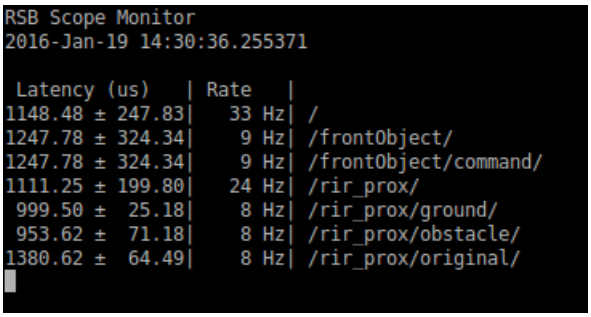


Figure 4.2: Monitor style output of the RSB logger of the standard Braitenberg demo.

But this time the scope list of the RSB logger, which is shown in figure 4.2, is longer. The *senseRingProximity* tool doesn't send the original proximity sensor values only, but the sensor values for the obstacle and the ground model, too. This is the reason for the three subscopes for */rir_prox*. Additionally there is a scope, which is used by the main program itself. It publishs commands.

For getting more information about the RSB packages of the *senseRingProximity* tool, the main scope can be choosen:

```
$ rsb-loggercpp0.11 --style detailed /rir_prox
```

Now all packages of the three subscopes are shown. For getting information about just the packages of only one scope, the full scope name can be used:

```
$ rsb-loggercpp0.11 --style detailed /rir_prox/obstacle
```

As shown in figure A.2(a) and A.1, the packages sent by *senseRingProximity* have all the same structure.

For getting more information about te command packages of the main program, the following command has to be typed:

```
$ rsb-loggercpp0.11 --style detailed /frontObject/command
```

The type of the package, which is shown in figure A.2(b), is a standard string and the payload consists of 4 readable chars: STOP. This is the command, which is sent, when the robot is in front of an object. It can be received and used by for example an object detection, which shall only detect objects, if there is something in front of the robot. Behaviors like this can for example save energy.

4.2 Useful Sense and Act Tools

For the most sensors there already exist sense tools, which read the sensors' values and publish them via RSB:

- **sendOdometry<publish type>:**
  Reads the odometry from the *DiWheelDrive* and publishs the values via RSB based on the RSB type included in the tool name.
- **senseCam<publish type>:**
  Catchs the images of the internal camera. The image will be compressed and published via RSB based on the RSB type included in the tool name.
- **senseFloorProximity:**
  Reads the floor proximity sensors, which are at the bottom of the AMiRo. The values are published via RSB as an integer vector (see chapter 5.1.2).
- **senseHokuyo:**
  Reads the values of the Hokuyo Laser Scanner. The published RSB type is `rst::vision::LocatedLaserScan`.
  This type includes the laser scans and all parameters of the laser scanner like laser count, angular ranges, etc. Basically this sense tool can be used for different laser scanners with different laser scan parameters. For now it is specialized in the Hokuyo Laser Scanner.
- **senseRingProximity:**
  Reads the proximity sensors of the *ProximityRing*. The values are published in three different variants via RSB as an integer vector: As original values, values for the obstacle model and as values for the edge model. For the models the configuration files *irConfig.conf* and *irEmpty.conf* are loaded from the directory *root/initial*. While the air offsets can only be read at the beginning, the ground offsets can be changed at runtime by giving a path to another ground offset file. For more detailed information about the offsets, please have a look into chapter 5.1.2.
- **setLights:**
  This tool sets the lights to given colors. Additionally it cannot let the LEDs only shine, but it can let them blink in different ways in a given period time. Please have a look into chapter 5.2.2 for more detailed information.

4.3 Useful Runnable Demos

The following demos in the directory *project/demo/* can be build and run on the AMiRo easily.

4.3.1 initial

This demo is the most important one. It contains all setup scripts for sensor initializations and setup tests. For detailed information about the sensor initializations, please have a look into chapter 5.1. The generated configuration files will be stored in this project. Finally any other project, which needs the configuration information, has access to this directory.

4.3.2 braitenberg

This demo already has been presented in the chapters 2.5 and 3.1. The AMiRo behaves like a simple Braitenberg vehicle by only using the proximity sensors of the *ProximityRing*. There are three different variants the braitenberg demo can be started:

- **simple:** By executing the script *runSimple.sh* the AMiRo drives forward and turns in front of objects only by comparing the measurements with a simple threshold. There isn't any initialization needed.
- **standard:** By executing the script *runStandard.sh* the AMiRo drives forward and turns in front of objects and table edges. For this behavior the initialization of the *ProximityRing* is needed, which is described in chapter 5.1.2.
- **detect objects:** By executing the script *runObstacleStop.sh* the AMiRo drives forward and turns in front of table edges. But this time it just stops in front of objects. By using the camera the AMiRo sends an image via RSB, where the object is marked. Only if the object is taken the AMiRo continues driving. Due to the obstacle and edge model based on proximity sensors, the *ProximityRing* has to be initialized (chapter 5.1.2).

4.3.3 follow_ProximitySensors

In this demo the AMiRo follows an object, another AMiRo or something else, which is in front of it, by only using the proximity sensors of the *ProximityRing*. The distances are calculated by the obstacle model of the proximity sensors, wherefore the *ProximityRing* has to be initialized, which is described in chapter 5.1.2.

By executing the script *run.sh* all programs for the following behavior will be started, but the AMiRo won't start following yet. By executing the script *follow.sh* the following command will be sent via RSB and the AMiRo starts following the object, which is in front of it. By executing the script *wait.sh* the AMiRo stops following due to the given stop command via RSB. The

following and stop commands can be repeated. Finally for stopping the program, the script *stop.sh* has to be executed.

5 Sense and Act with the AMiRo

The basic structure of the AMiRo consisting of *DiWheelDrive*, *PowerBoard* and *LightRing* contains many sensors and some actors. By using the *CognitionBoard* the internal camera can also be used. Of course, there can be added external sensor and actor systems, but in this chapter only sensor and actor systems included in the AMiRo will be presented.

5.1 Sensors

The AMiRo is a small sensor platform. While driving, the *DiWheelDrive* calculates the movement by measuring the motor rotation speed with motor encoders. The *DiWheelDrive* also contains gyroscope, accelerometer and magnetometer. The information of these sensors ca be used for example for odometry and position calculation.

For measuring the close environment there are used infrared proximity sensors. Four proximity sensors are at the bottom of the AMiRo for ground detection (e.g. line detection), which are controlled by the *DiWheelDrive*, and eight proximity sensors, which are controlled by the *PowerBoard*, are placed circular at the side of the chasis in the *ProximityRing* for a 360 ° view around the robot.

The *ProximityRing* also includes four capacity touch sensors, two on each side. They can be used for human interaction very easily. The capacity touch sensors are also controlled by the *PowerBoard*.

Finally there is an included camera. It can only be controlled by the *CognitionBoard* directly, so the basic AMiRo structure, which only consists of *DiWheelDrive*, *PowerBoard* and *LightRing*, doesn't contain the camera.

In the following subsections the sensors will be explained briefly. Additionally the usage for the Linux operating system on the *CognitionBoard* will be described, including initialization information. The *CognitionBoard* gets the sensor information of the other modules via the CAN bus.

5.1.1 Odometry

The odometry is only based on the motor encoders yet. An integration of gyroscope, accelerometer and magnetometer is still in development. The odometry can be set and read by the *Cog-*

nitionBoard via the CAN bus. But there isn't any access to the encoders. The odometry will always be actualized by the *DiWheelDrive*.

Additionally for odometry correction there can be set the kinematic calibration constants E_d and E_b based on the calibration by Jung and Chung [**?**], which correct variation in wheel distance and wheel size proportion. These constants can be different for each AMiRo.

For reading the odometry the sense tool *sendOdomtry* should be used. The usage and variants are described in chapter 4.2.

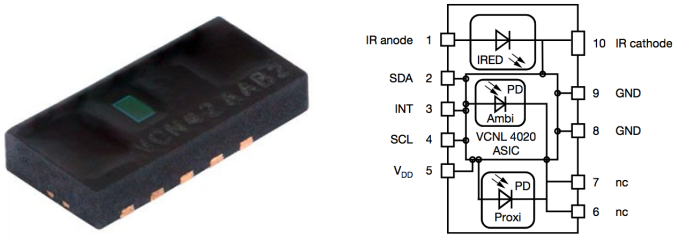
The CAN commands are:

CAN command	Description
`types::position getOdometry`	Gives the odometry as a TWB position type.
`void setOdometry` `  types::position robotPosition`	Sets the odometry to the given position *robotPosition*.

`void setKinematicConstants`
`  float Ed`
`  float Eb`
	Sets the kinematic calibration constants E_d and E_b.

5.1.2 Proximity Sensors

The proximity sensors at the bottom of the AMiRo and in the *ProximityRing* are VCNL4020 infrared proximity sensors from Vishay. As shown in figure 5.1 the sensor consists of an infrared LED and the suitable infrared emitter for proximity calculations and additionally an ambient light emitter.



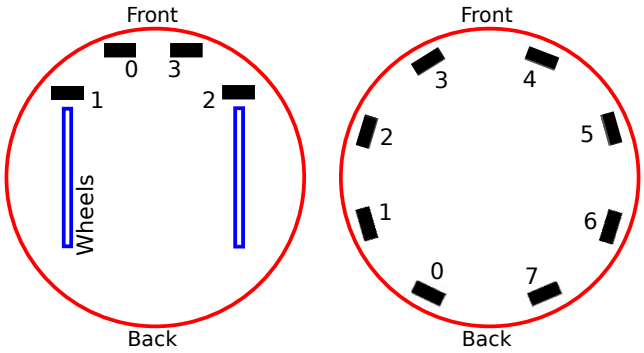
(a) (b)

Figure 5.1: The VCNL4020 sensor from Vishay (a) and its schematic (b).

The ambient functionality isn't implemented yet. The sensors are only used for proximity calculations based on the measured infrared light reflections in the environment. Due to an internal integration of measurements while the infrared LED is turned on and off, the measured proximity value is nearly independed on the changing infrared light of the environment.

By using the floor proximity sensors at the AMiRo's bottom, there can be done a ground detection. By using the proximity sensors of the *ProximityRing*, an obstacle avoidance can be imple-

mented. For reading the proximity values the sense tools *senseFloorProximity* and *senseRingProximity* should be used only (see chapter 4.2). These tools give vectors with integer values. The ID assignment of the proximity sensors at the bottom of the *DiWheelDrive* and in the *ProximityRing* is defined in figure 5.2.



(a) Bottom view of *DiWheelDrive*

(b) Top view of *ProximityRing*

Figure 5.2: ID assignments of the proximity sensors on the *DiWheelDrive* (a) and on the *ProximityRing* (b).

5.1.2.1 Interpretation of a Proximity Value

A proximity value doesn't give any precise information about the environment. The reflection of the infrared light depends on the object's surface and color and, of course, on the objects count. One close object can effect the same proximity value than two objects of the same type farer away.

Additionally the position of one object isn't precise, either. An object, which is directly in front of the sensor, effects the same proximity value than an closer object of the same type at the side of the sensor. This is defined as the sensor's cone, which is shown in figure 5.3.

In the subdirectory *includes* of the murox repository there is a sensor model for the VCNL4020 proximity sensors based on the proximity sensor model of Benet et. al. [**?**]. It can calculate the distance of an object by known angle and calculate the distance of a table edge. But this model needs an initialization of the proximity sensors, because the sensors' offsets are different.

5.1.2.2 Initialization of the ProximityRing

The *ProximityRing* has to be initialized. There are two offsets, which have to be measured for each sensor: The air offset (infrared light in the air) and the ground offset (infrared light reflected by the ground the robot stands on). Please build and copy the project *project/demo/initial* for the initialization (see chapter 3). The needed scripts are stored in this project.

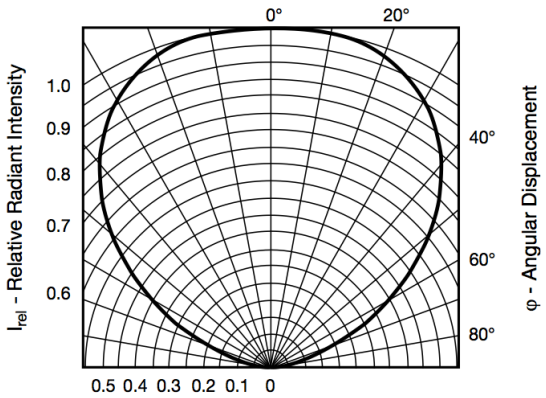
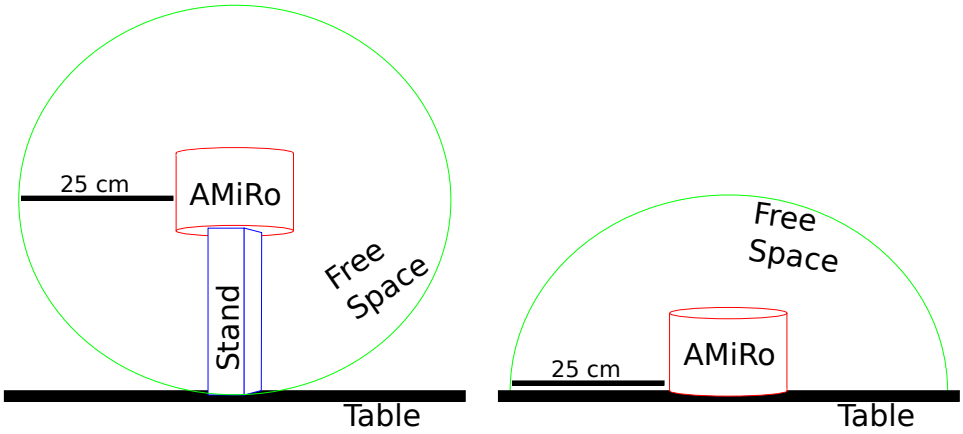


Figure 5.3: The sensor's cone of the VCNL4020 sensor. Given a proximity value, an object (with known characteristics) can be anywhere on this cone.

For the air offset the robot has to be placed without anything in its range (about 25 cm). As shown in figure 5.4(a) the AMiRo can be placed on a tall object, which has a hight of about 25 cm. Now *runIREmpty.sh* has to be executed. This will take several minutes. Afterwards in the project folder there should be a file named *irEmpty.conf*, which contains the air offsets. This initialization has only to be done once per robot. Ideally the air offsets never change.



(a) Setup for air offsets (b) Setup for ground offsets

Figure 5.4: Initialization setup for measuring the air offsets (a) and the ground offsets (b).

For the ground offset the AMiRo has to be placed on the ground without any objects or edges around it (about 25 cm) as shown in figure 5.4(b). Now *runIRConfig.sh* has to be executed. This will take several minutes. Afterwards in the project folder there should be a file named *irConfig.conf*, which contains the ground offsets of the actual surface the robot is standing on. Of course, the ground offsets depend on the ground, so this initialization has to be done for every ground, which shall be used.

For checking the offsets the following tools can be used: By executing *senseRingProximity* with the parameter *-p* the original values and the values for the obstacle and the edge model are printed. If the robot is on the ground without any object or edge in range, the obstacle values should be near zero, while the edge values should be near 10000. By executing *edgeMeasurement* (without any parameters) the edge distances for each sensor will be printed. The maximum distance is about 6.5 cm.

5.1.3 Capacity Touch Sensors

>>> **topic in progress** <<<

5.1.4 Gyroscope

>>> **topic in progress** <<<

5.1.5 Accelerometer

>>> **topic in progress** <<<

5.1.6 Magnetometer

>>> **topic in progress** <<<

5.1.7 Camera

>>> **topic in progress** <<<

5.2 Actors

The basic structure of the AMiRo consisting of the modules *DiWheelDrive*, *PowerBoard* and *LightRing* doesn't have much action. There are only driving actions and giving light signals.

5.2.1 Motors

The *DiWheelDrive* controls the driving behavior of the AMiRo. By giving CAN commands the driving velocity can be set and read. The motors cannot be commanded directly, but a forward velocity and an angle velocity can be given, which will be transformed in motor velocities by the *DiWheelDrive*.

CAN command	Description

`void setTargetSpeed`
`  int v`
`  int w`
	Sets the driving speed consisting of the forward velocity *v* and the angle velocity *w*

`int getActualSpeed`
`  int v`
`  int w`
	Reads the actual forward velocity *v* and the actual angle velocity *w*. If the values could be read, the return value will be zero.

There are still other CAN commands for driving like *setTargetPosition*, but these functions are deprecated! Please, only use the functions listed in the table.

5.2.2 Lights

The *LightRing* consists of eight RGB LEDs which are placed directly above the proximity sensors in the *ProximityRing*. The lights can only be set, but not read. The ID assignments for the LEDs are defined in figure 5.5.

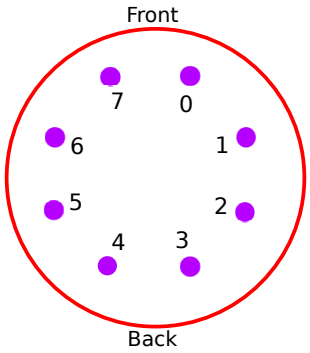


Figure 5.5: Top view of the *LightRing* with the ID assignments of the LEDs.

There are two different possibilities to set the colors of the LEDs. The first one is setting the colors directly via CAN by the following functions:

CAN command	Description

`void setLightColor`
`  int index`
`  Color color`
	Sets the LED with the ID *index* on the given color.

`void setLightBrightness` `  int brightness`	Sets the brightness of all LEDs.

The second way is to use the *setLights* tool in *project/act/*. It listens to an integer vector via RSB. Additionally to setting the colors, this tool can also blink the LEDs in different lighting types for given period times, so this tool is recommended, when the LEDs shall show special behaviors:

Lighting Type	Description
SINGLE_INIT	Sets the initial colors.
SINGLE_SHINE	Sets the given colors and lets it just shine.
SINGLE_BLINK	Sets the given colors and blinks with all LEDs turned on or off.
SINGLE_WARNING	Sets the given colors and blinks with always the left or right half turned on or off.
SINGLE_CROSSED	Sets the given colors and blinks with only 4 crossed LEDs at once turned on.
SINGLE_CIRCLELEFT	Sets the given colors and lets circle two LEDs turned on to the left (counter clockwise).
SINGLE_CIRCLERIGHT	Sets the given colors and lets circle two LEDs turned on to the right (clockwise)
CHANGE_INIT	Changes the colors without turning off the LEDs. The color set equals the initial colors.
CHANGE_SHINE	Changes the given colors without turning off the LEDs.
CHANGE_BLINK	Changes the given colors and blinks with all LEDs turnes on or off.

If the suffix SHINE is used, then there can be given 1 color for all LEDs or 8 colors for each LED. While the chosen action the color of a single LED won't change. If the suffix CHANGE is used, then there can be given many colors. In this case all LEDs will have the same color at the same time. The colors of every LED will change with each step of the chosen action.

Please use the *LightType* enumeration in the *LightModel.h* in *project/includes/actModels/*. There can be also find a more detailed documentation of the *LightType* in the file *startLightModel.txt*. As an example code the tool *sendLights* in *project/tools/examples/* shows, how to use the *LightModel.h* and how to generate and send the light command to the *setLights* tool.

A RSB Logger Output for Detailed Style

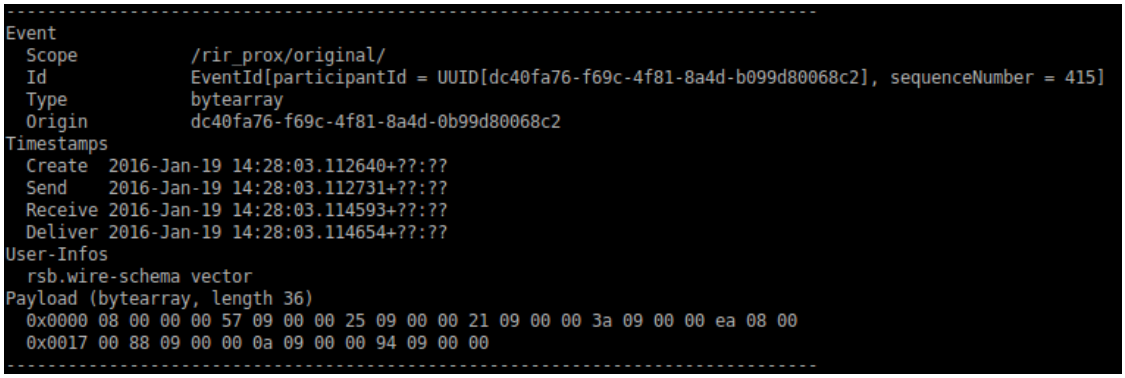
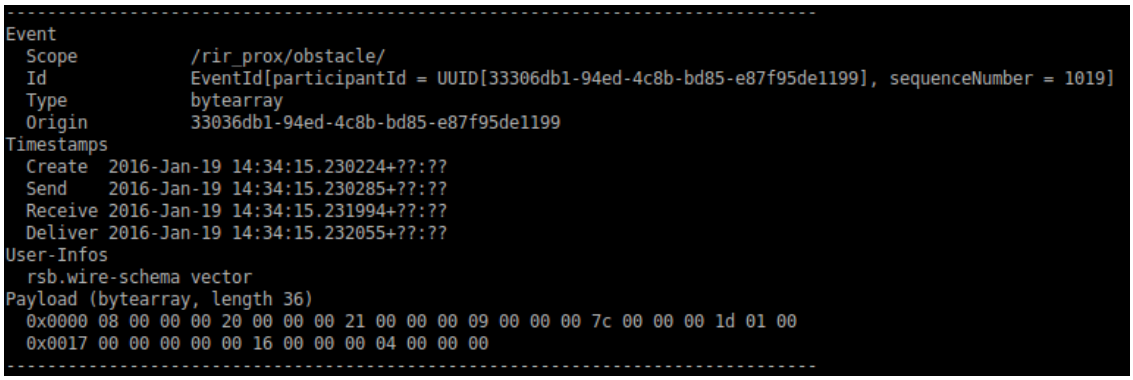
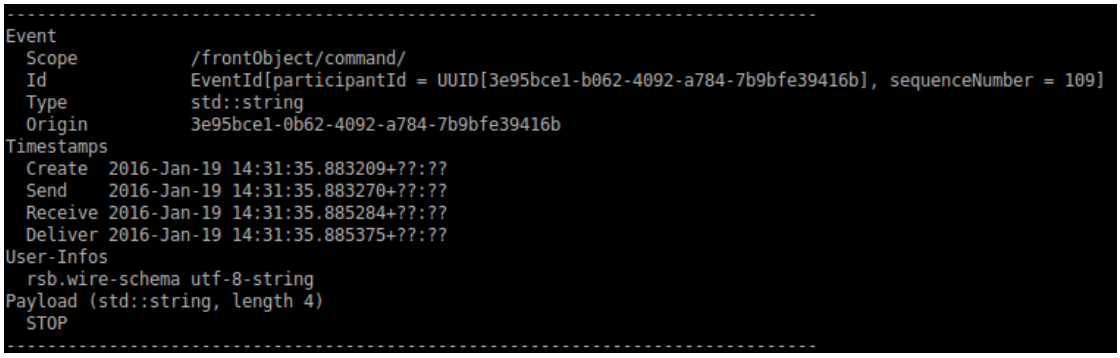


Figure A.1: Detailed style output of the scope */rirprox/original* of the RSB logger of the simple Braitenberg demo.



(a)



(b)

Figure A.2: Detailed style output of the scope */rirprox/obstacle* in (a) and of the scope */frontObject/command* in (b) of the RSB logger of the standard Braitenberg demo.