Cyberbotics

Processor Architecture Laboratory, LAP

Federal Institute of Technology, Lausanne, EPFL

Master Thesis Project Report

Improving support for e-puck robot in Webots mobile robots simulation software

Author:
Nikola Veličković

Cyberbotics supervisors:
Fabien Rohrer
Olivier Michel

EPFL supervisor:
René Beuchat

March, 2012.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to Dr. Olivier Michel as well as to Mr. Fabien Rohrer from Cyberbotics for enabling me to work on this project and for their help, guidance and encouragement throughout its course. I also greatly appreciate the help provided to me by the engineering team of GCtronic, especially Gilles Caprari and Stefano Morgani. My thanks also go to my teacher prof. René Beuchat for his mentorship during my Master studies as well as on this project.

Last, but not least I would like to thank my coleagues from Cyberbotics, Yvan Bourquin and Luc Guyot for making my stay at Cyberbotics enjoyable as well as my dear friends that made my stay at EPFL and in Lausanne pleasurable and memorable. In no specific order they are, Tamara Saranovac, Miloš Balać, Marko Stojanović, Janko Katić as well as many others.

Abstract

E-puck is a mobile robot which has various sensors (camera, infra red sensors, microphones etc.) and actuators (motors, speaker, LEDs). It is useful for a broad range of applications, especially teaching and is supported in the **Webots** mobile robots simulation software. An extension for e-puck, making it a full computer on wheels was recently developed. Supporting the new extension in Webots was the goal of this project. This was done by developing a software library called **E-GO** which provides access to all of the features of the extended robot, such as sampling sensors, updating actuators, communication between e-puck and the extension, wireless communication between the extended robot and other devices. The goal was also to do this in a simple and standardized manner, providing compatibility with the Webots API and with a clean and user friendly interface. Library supports both autonomous execution of a controller on the robot as well as remote control of the robot. This approach masks the complexity of the system while still providing its functionalities to the user. Combination of the extension and the software library allows for image acquisitions with up to 10fps in maximum camera resolution (640x480) compared to a maximum of 4fps with 40x40 pixels resolution of the basic e-puck version. From the two formats in which the camera can supply the image (RGB565 and YUV422) tests showed RGB565 to be faster by about 2.5x compared to YUV422. Communication between the basic e-puck robot and the extension is via serial connection. Average transfer rate achieved is about $85\mu s/byte$, which when adjusted with the overhead of control bits gives a throughput of about $120'000b/s$ (programmed data rate of the serial connection is $230'400b/s$). Average amount of time necessary to transmit maximum possible amount of data (update all actuators and sample all sensors) through the serial bus is about $50ms$ and to acquire one image in full resolution is about $100ms$.

keywords: e-puck, autonomous, mobile, robot, Webots, E-GO, Gumstix.

Chapter 1

Introduction

In today's world robots are indispensable. They are used in many different applications and can be very complex devices, with a wide set of capabilities and configurations. There are also various classes of robots, one of which are autonomous[1] mobile[2] robots.

E-puck [1, 2] robot, used in this project and shown in figure 1.1, is a desktop size autonomous mobile robot. It was developed as an educational tool at École Polytechnique Fédérale de Lausanne (EPFL). The main idea is to be able to perform experiments quickly and cost-effectively in a small working area. It's predecessor the Khepera robot [3] has similar purpose but costs more than the e-puck robot. E-puck robot has a large community of users and is very well documented and supported with various extensions.

Fig. 1.1: E-Puck robot.

One of those extensions based on **Gumstix Overo**[3] module (figure 1.2) is used in this project. It increases the computing power and memory size of the e-puck robot along with adding features such as broadband wireless communication. The

[1] http://en.wikipedia.org/wiki/Autonomous_robot
[2] http://en.wikipedia.org/wiki/Mobile_robot
[3] https://www.gumstix.com/store/product_info.php?products_id=211

extension runs with a Linux OS, thus making the e-puck robot a full computer on wheels.

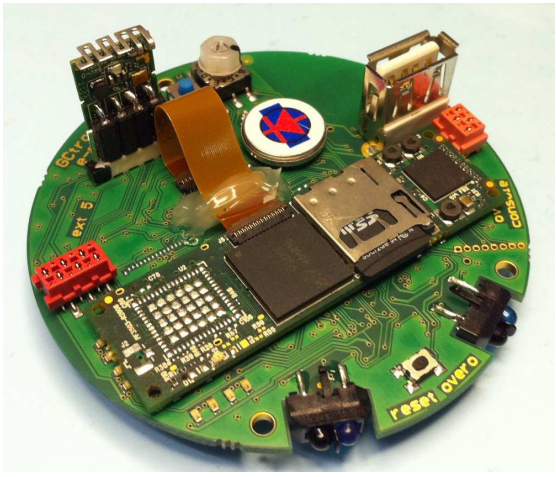


Fig. 1.2: Gumstix Overo based extension for the e-puck robot.

Webots[4] [4, 5, 6, 7] is a development environment used to model, program and simulate mobile robots. It supports the simulation and programming of the basic version of e-puck robot (without the extension).

Main goal of this project was to implement support in Webots for the e-puck robot with the Gumstix Overo based extension.

1.1 What was available at the beginning

1. Webots software support for the basic version of the e-puck robot (without Gumstix extension), illustrated in figure 1.3.

The user has the possibility of writing a software controller using application programming interface (API) provided by Webots and then do one of the following:

- simulate the behaviour of the e-puck robot inside Webots software with faithful graphical and physical representation.
- use Webots to cross-compile the controller and transfer it to the real robot.
- use Webots to remotely control the real robot.

2. Gumstix based extension for the e-puck robot.

Such extended e-puck robot can be used for more demanding applications, but is also more complex than the basic version. Further more it was not supported by the Webots software.

[4] `http://www.cyberbotics.com/overview`

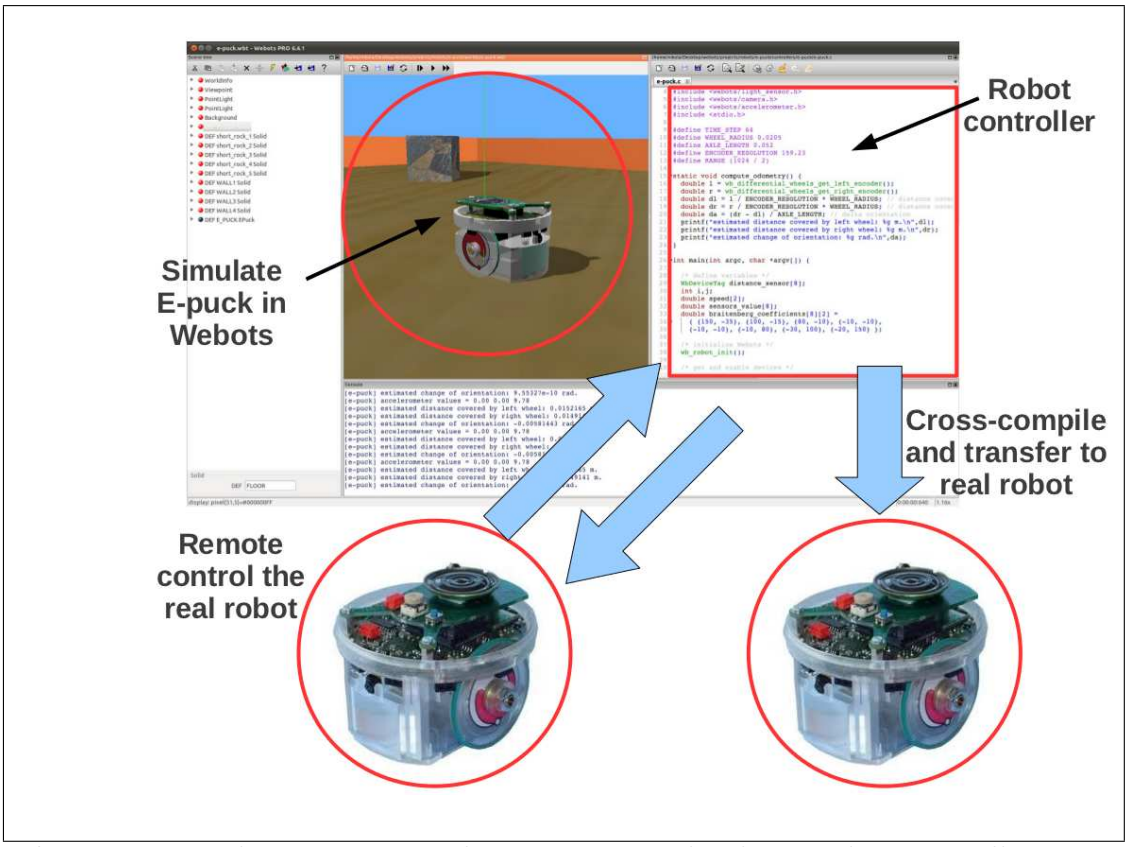


Fig. 1.3: E-puck support in Webots. User can develop a robot controller using Webots API, simulate the robot inside Webots, cross-compile the controller and transfer it to real robot or remote control the real robot from Webots.

1.2 What was done in the project

Software library called E-puck Gumstix Overo library (E-GO) was developed. Purpose of this software library is to provide support for the e-puck robot with the Gumstix extension in Webots software. This implies being able to develop a controller in Webots, cross-compile it and transfer it to the real robot as well as remotely control the real robot from Webots software. One part of the library is software executing on the Gumstix extension processing unit, thus allowing the cross-compiled controller to use robot's features. Another part of the library executes on the operating system where Webots software is running and is in charge of remote control of the robot from Webots.

Besides developing the E-GO software library there were other goals and constraints in this project.

1.2.1 Project goals

- develop a modular and easily maintainable software library.
- determine hardware constraints and adjust the E-GO library's design for them.
- make the software library compatible with the Webots API.

- optimize the implementation to achieve best results under given constraints.
- characterize the implemented solution in terms of timing parameters and memory utilization.

1.3 Report organization

The rest of this report is organized as follows.

Chapter 2 explains the hardware platform used in this project in more details. This includes overview of the e-puck robot as well as the Gumstix Overo based extension, with emphasis on the key features of the designs.

Chapter 3 explains the purpose and features of the mobile robot simulation software - Webots. It shows how the API provided by this software is used to develop robot controllers and how it influences the specifications of the software library developed in this project.

Chapter 4 gives a detailed insight in the structure of the developed library, explaining the choices made during the development.

Chapter 5 introduces relevant timing parameters and provides the results of measurements taken to obtain them. Comparisons between possible choices at several key points are given which explain the benefits and drawbacks of different solutions. Results obtained from tests verifying the functionality of the system using developed library are also presented.

Chapter 6 summarizes the report and concludes with possible future directions of the project.

Chapter 2

Hardware platform

This chapter will present the **e-puck** robot in details along with the extension used in this project, which is based on the Gumstix Overo module. Both electrical and mechanical structure of the device will be detailed. Specifications will be formulated and explained which determined the final design of the e-puck robot [1] along with the advantages and drawbacks of the used extension.

2.1 E-puck robot

Mobile robots are quite attractive objects which result out of the fusion of multiple competences. The design of such robots requires skills in disciplines such as mechanics, electronics, energy management, computer science, signal processing and automatic control. Inter-disciplinary nature of mobile robots makes them excellent educational platform which allows addressing a broad range of engineering fields in university courses, but also provides a good platform to use in robotics oriented research.

There are many examples of mobile or educational robots with varying range of prices and configurations. Robots that are both mobile and good educational tools should meet certain criteria, which according to [1] are:

- *Desktop size.* For good efficiency in robot usage in education it has to fit on the desk and still maintain good mobility, meaning a size of at maximum 10% of the working space in all directions. This imposes a limit on the maximum size of the robot of about 80mm.

- *Wide range of applications.* For this criteria to be met the robot should provide a wide set of functionalities in its basic version which in turn should make the robot applicable in various fields of education, such as embedded programming, automatic control, distributed intelligent systems design and signal processing.

- *User friendly.* User interface must be simple, efficient and intuitive.

- *Low cost.* In education for an effective usage of robots a large amount of units is needed. Considering the tight budget constraints of many institutions that

would be good candidates to use the device, this criteria has a strong impact on the final design.

- *Open platform.* Since the idea is to have a highly versatile platform that is easily upgradeable and adjustable it is important that different users can have a high degree of freedom in modifying it. Open hardware/software approach is a natural choice combined with a design that allows the users to easily customize their robots with extensions and software.

Motivation for the design of the e-puck (shown in figure 1.1) came from the fact that there was no platform available at the market that satisfied all of the above mentioned constraints. In order to be able to meet the listed requirements sensors, actuators and interfaces that are part of the e-puck robot were carefully chosen. The robot is equipped with:

- sensors for acquisition of various useful data, such as: audio, visual, light intensity, distance to objects, speed and gravity. These devices have different bandwidths varying from 10Hz to 10MHz and are shown in figure 2.1 on the left.
- actuators for interaction with the environment, such as motors, LEDs and a speaker. These are shown in figure 2.1 on the right.
- wired and wireless communication devices
- dsPIC microcontroller combining a general purpose processor and a DSP in a single core.

Detailed hardware outline of the basic configuration of the e-puck robot is shown in figure 2.2. On top of this configuration it is possible to add extension(s) which broaden the set of functionalities of the robot.

Following subsections give a closer description of the key hardware components.

2.1.1 Microcontroller

Choice of the "brain" of the robot is influenced by several factors, such as number of sensors present in the device, actuators and what communication modules exist on the system, as well as the bandwidth necessary for all the received and sent data to be processed. Specifically operating frequency and processing power of the microcontroller are important. For the e-puck robot, Microchip's 16-bit **dsPIC** microcontroller was chosen which consists of a general purpose processor and a DSP in a single core. It has an operating frequency of 64MHz and 14 MIPS of peak processing power as well as 8kB of RAM and 144kB of flash memory [8]. Additional benefits of this microcontroller are it's mostly orthogonal[1] and rich instruction set. It also contains `multiply-accumulate` and hardware `repeat`

[1] An instruction set is orthogonal if any instruction can use data of any type via any addressing mode (`http://en.wikipedia.org/wiki/Orthogonal_instruction_set`)

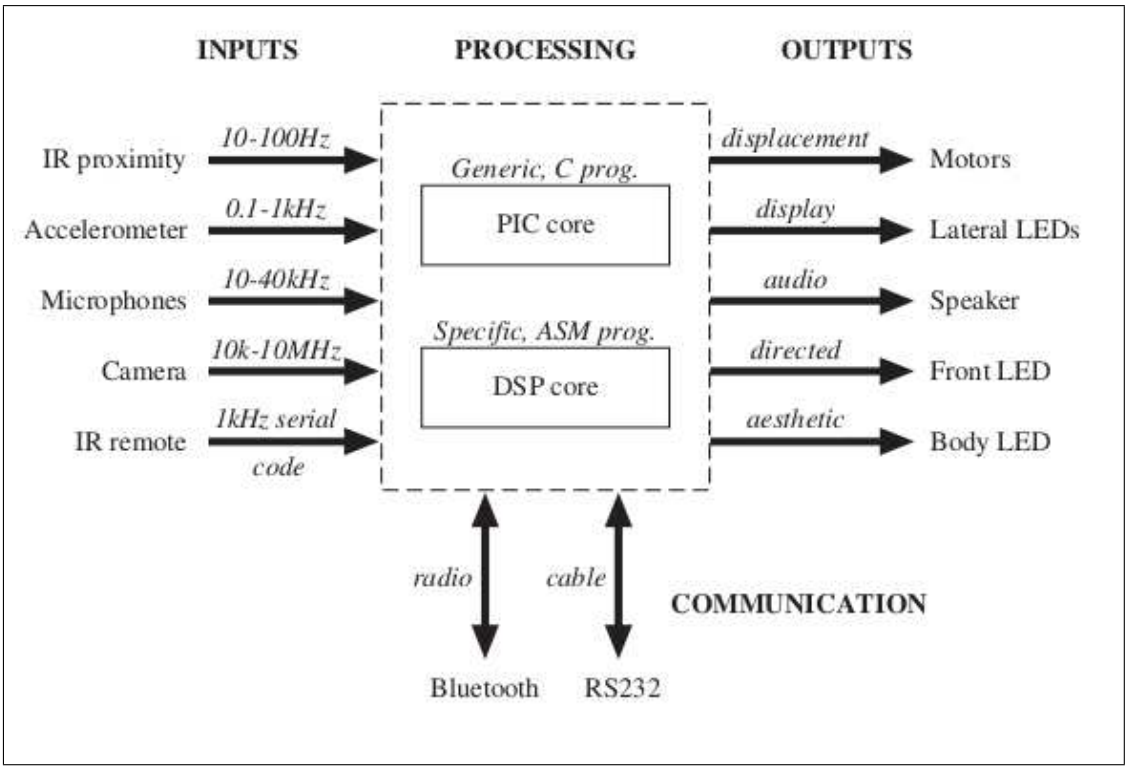


Fig. 2.1: E-puck interfaces. Variety of sensors with different bandwidths, actuators, communication modules and processing unit allow for versatile usage of the robot in numerous applications. Taken from [1].

instructions [8] which drive the DSP unit and are convenient for efficient computation of scalar products and fast Fourier transforms. For acquisition of analog signals it has at disposal 16 analog inputs connected to an internal 12-bit AD converter which is capable of sampling up to $200ksamples/s$. More details on the microcontroller can be found in the device's data sheet [8].

2.1.2 Sensors

Sensors available on the e-puck robot enable acquisition of different physical quantities, such as:

- 3D accelerometer [9] which provides acceleration on three orthogonal axes. It can be used to measure the inclination of the robot as well as the acceleration from the robot's movement. Collisions and falls can also be detected with the help of accelerometer. Examples of measurements taken with it are given in appendix.

  Figure 2.3(a) shows data acquisition. Accelerometer has 3 analog outputs giving electrical signals corresponding to each of the 3 axis acceleration. These outputs are connected to the analog inputs of the 12-bit AD converter inside the dsPIC microcontroller. Signals provided by the accelerometer are analog signals between $0 - Vdd$. When no acceleration exist on a certain axis the value of the corresponding signal is at $Vdd/2$. The resulting dig-

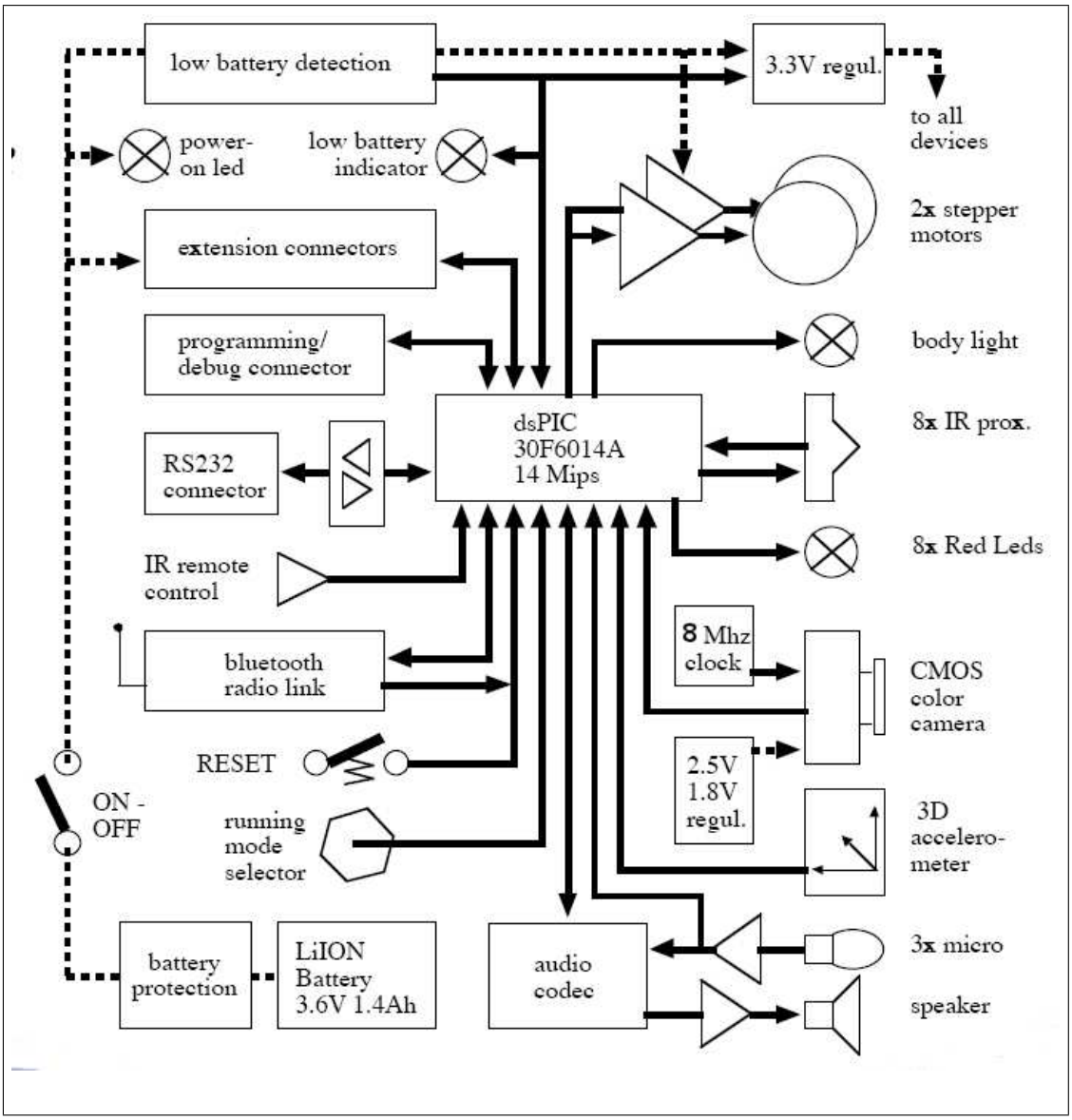


Fig. 2.2: E-puck basic configuration hardware outline. This figure describes how all elements of the robot are connected. Taken from [1].

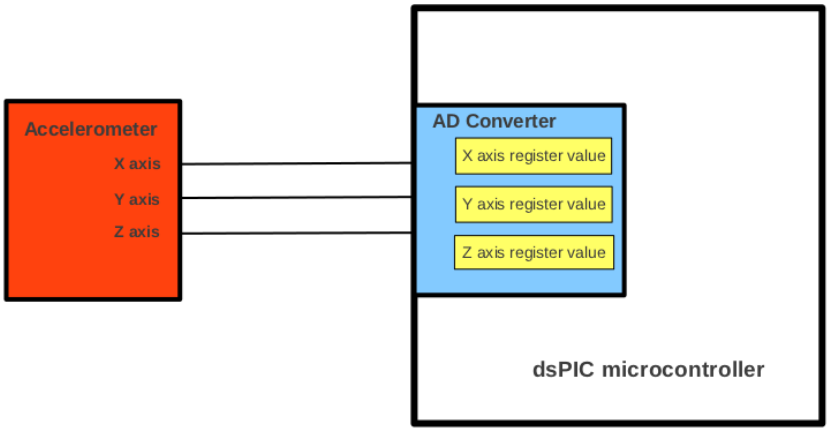
ital equivalent of the acceleration for each axis is between $0 - 4095$. AD conversion can be described by the following formula:

$$N = \frac{v}{Vdd} 4096, \tag{2.1}$$

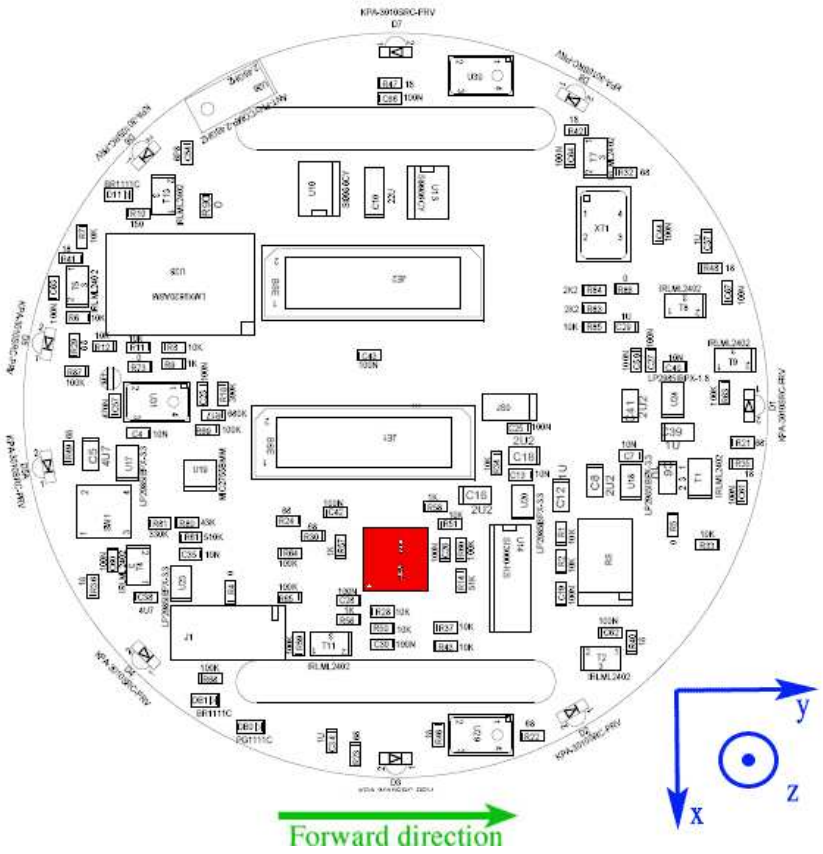
where: N - resulting digital equivalent of the analog value, Vdd - voltage span of the input analog signal(in this case between 0 and Vdd), 4096 - number of quantization levels of the AD converter (2^{12}), v - voltage level of the input analog signal.

Figure 2.3(b) shows accelerometer's position on the PCB and axis orientations.

- 8 infrared proximity sensors [10] placed around the body of the robot to



(a) Block schematics of accelerometer's data acquisition by the dsPIC microcontroller. Analog outputs of the accelerometer are connected to the analog inputs of the microcontroller connected to an internal 12-bit AD converter.

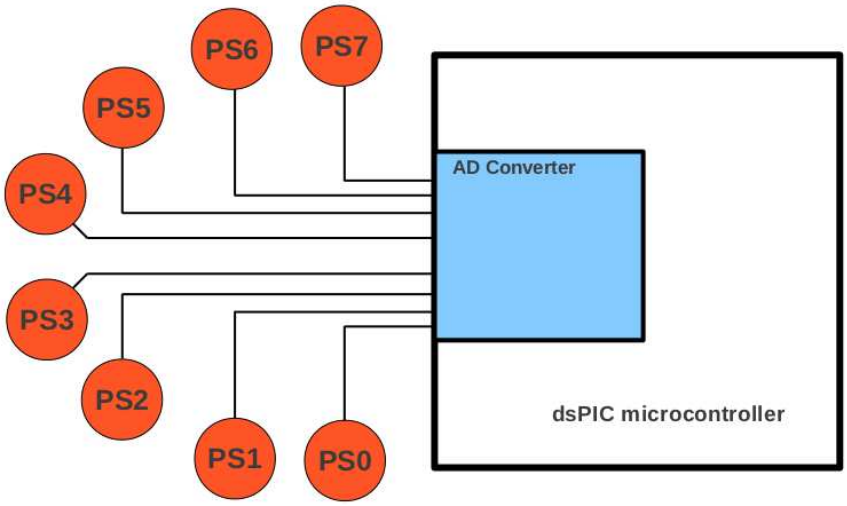


(b) Accelerometer's position on the PCB and the coordinate system orientation. Taken from [2].

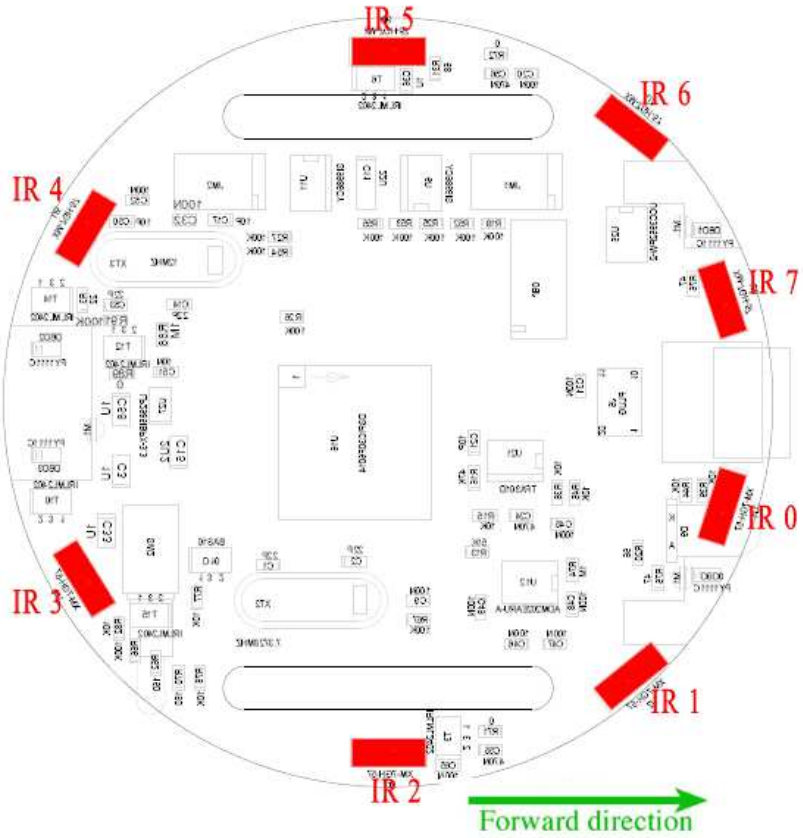
Fig. 2.3: Acceleration measurement and accelerometer

measure distance from obstacles as well as the intensity of the ambient

infrared light. Figure 2.4(b) shows the position of the sensors on the e-puck body. These are especially convenient in cluttered environments.



(a) Block schematics of proximity sensors' data acquisition by the dsPIC microcontroller. Analog outputs of the sensors are connected to the analog inputs of the microcontroller connected to an internal 12-bit AD converter.



(b) Proximity sensors' position on the PCB. Taken from [2].

Fig. 2.4: Distance measurement and distance sensor

Figure 2.4(a) shows the block schematics of infrared proximity sensors' data acquisition. Analog outputs of the sensors are connected to microcontroller's analog inputs. Resulting digital equivalents of the measurements are numbers between $0 - 4095$. Figure 2.5[2] shows an example of a measurement taken with proximity sensors, which can be used to characterize the sensor. Y-axis shows raw data acquired by the AD converter, while the X-axis shows steps of distance from the wall. 1000 steps corresponds to $12.8cm$. Robot starts against the wall and moves backwards.

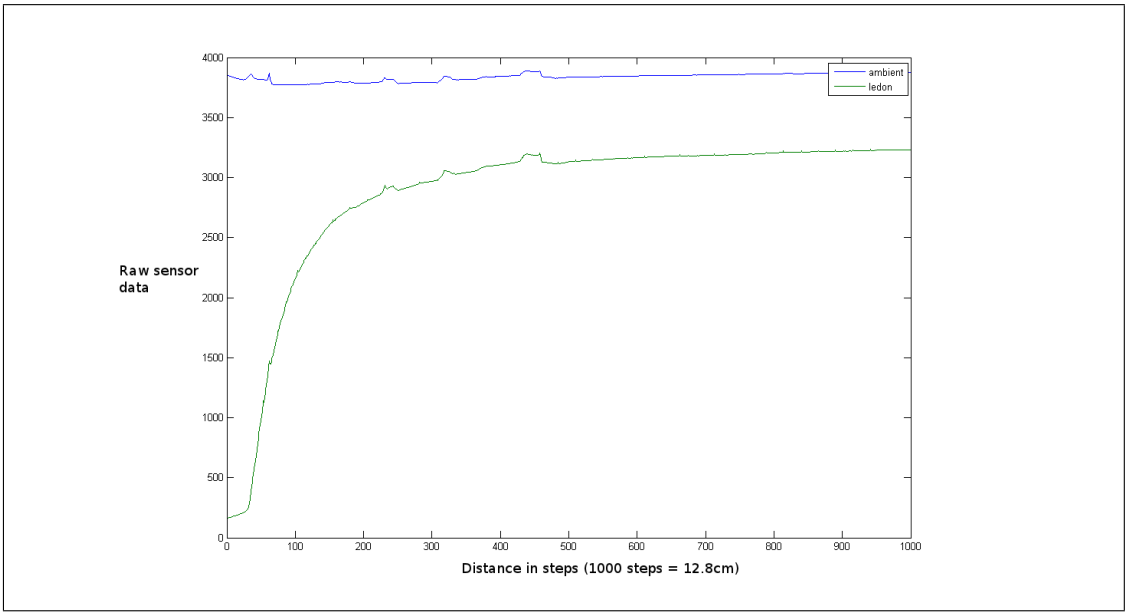


Fig. 2.5: `Ambient` measurement does not use the infra red source thereby in fact calibrating the "zero" level of the measurement. `Led on` measurement uses the infrared source and practically characterizing the sensor device. For the measurements robot starts against a wall and moves backwards. Y-axis represents raw sensor data after AD conversion and X-axis represents steps of distance from the wall. 1000 steps corresponds to $12.8cm$.

- 3 microphones [11] to capture sound. Figure 2.6[3] shows the position of the microphones. They are conveniently placed so that it is possible to triangulate the sound source, that is determine where from the sound signal is coming.

  Outputs of microphones are analog and are connected to analog inputs of the dsPIC microcontroller as for accelerometer and proximity sensors. More technical details are available in the data sheet [11].

- color CMOS camera [12] with a maximum resolution of 640x480 pixels and a maximum frame rate of 30fps at 24MHz camera clock frequency [12]. It differs from previously mentioned sensors in the sense that it needs a high bandwidth connection towards the processing unit and has much more

[2] `http://www.e-puck.org/index.php?option=com_content&view=article&id=22&Itemid=13`
[3] `http://www.e-puck.org/index.php?option=com_content&view=article&id=19&Itemid=11`

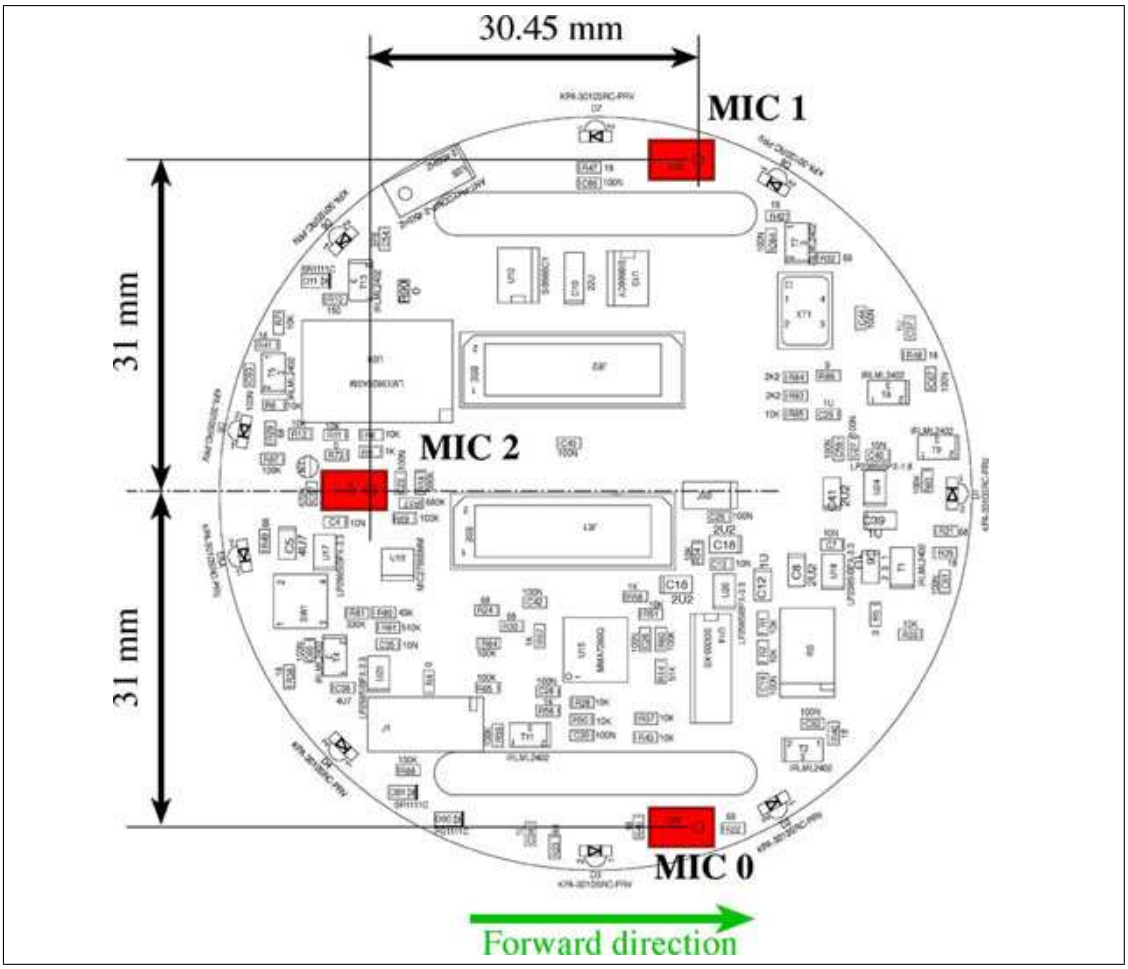


Fig. 2.6: Microphones position on the robot's PCB. Taken from [2].

options for configuration. It has a bundle of features such as: on chip image processing, programmable frame size, windows size and position, automatic $50Hz$, $60Hz$ flicker cancellation. Camera also has a low power consumption of up to $70mW$ when capturing images and a $40\mu W$ in stand by mode. On chip processing includes brightness, contrast, saturation settings as well as gamma correction, color interpolation etc. Pixel array of the camera is covered by Bayer color filters as depicted in figure 2.7.

More details are available in the data sheet [12]. In the e-puck robot the camera is connected to an 8MHz clock source. This limits the maximum frame rate to 10fps at maximum resolution. For the basic configuration this setup is logical since the dsPIC microcontroller is not capable of handling a full image from the color camera. Size of the acquisition is determined by the memory size of the dsPIC and the rate is determined by its processing power. For instance it is possible to grab a portion of a color image with the size of 40x40 pixels at 4fps. This limitation is eluded with the usage of the Gumstix Overo extension since it has enough memory and processing power but the camera clock frequency remains at 8MHz (because clock source remains at the board of the basic configuration) thus limiting the maximum frame rate at 10fps for a color image at 640x480 pixel resolution. Figure 2.8

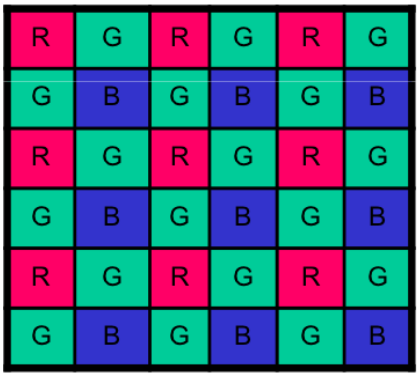


Fig. 2.7: Bayer color filters covering the pixel array of CMOS camera used on the e-puck robot.

shows the relevant signals for image acquisition from the camera and how they are connected to the system.

2.1.3 Actuators

Actuators available on the e-puck robot are:

- Two stepper motors [13], which control the movement of the wheels with a resolution of 1000 steps per wheel revolution. By counting the number of steps and measuring time passed speed of the robot can be obtained.
- A speaker, which can be used for human interaction or for example peer direction detection in a multi robot environment. This is possible because the microphones allow the e-puck to triangulate the source of the sound.
- 8 LEDs placed around the e-puck on its body.

2.1.4 Communication modules

For the communication with the e-puck robot there are two available connections which are presented in figure 2.2. One is the RS232 connector for serial communication and the other is a Bluetooth [14] radio link. The RS232 connector is useful for connecting extension boards with the dsPIC microcontroller which is on the basic board. Bluetooth radio link, on the other hand can be used to remotely program the e-puck using a bootloader as well as remotely control the robot from a desktop PC or communicate with other robots.

2.1.5 E-puck extensions

In order to satisfy the easy reconfiguration and modification design specification, the e-puck robot is developed so that it can be easily extended. Extensions physically connect through an extension bus which supplies an I^2C connection as well as connections towards most sensors.

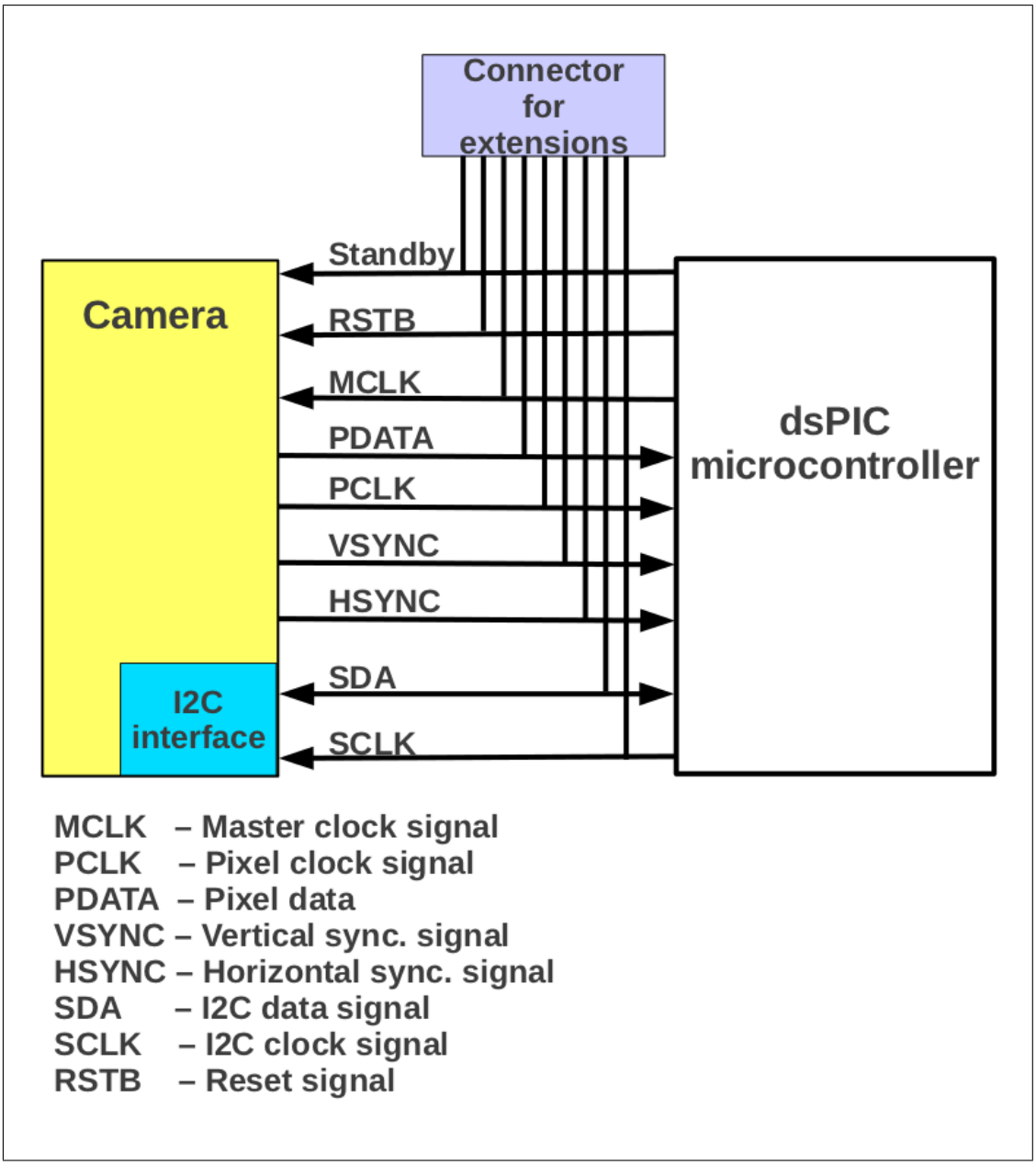


Fig. 2.8: Block schematics of image acquisition.

Various extension have been developed for the e-puck robot, expanding its functionalities in different directions. One that is used in this project adds a wireless module for broadband communication, USB connector as well as a Gumstix Overo computer-on-module card. This card features an OMAP3503 [15] applications processor with an ARM Cortex-A8 core operating on a 600MHz clock with up to 1200 Dhrystone[4] MIPS, 256MB RAM and 2GB flash memory, thus giving the e-puck robot a lot more processing power.

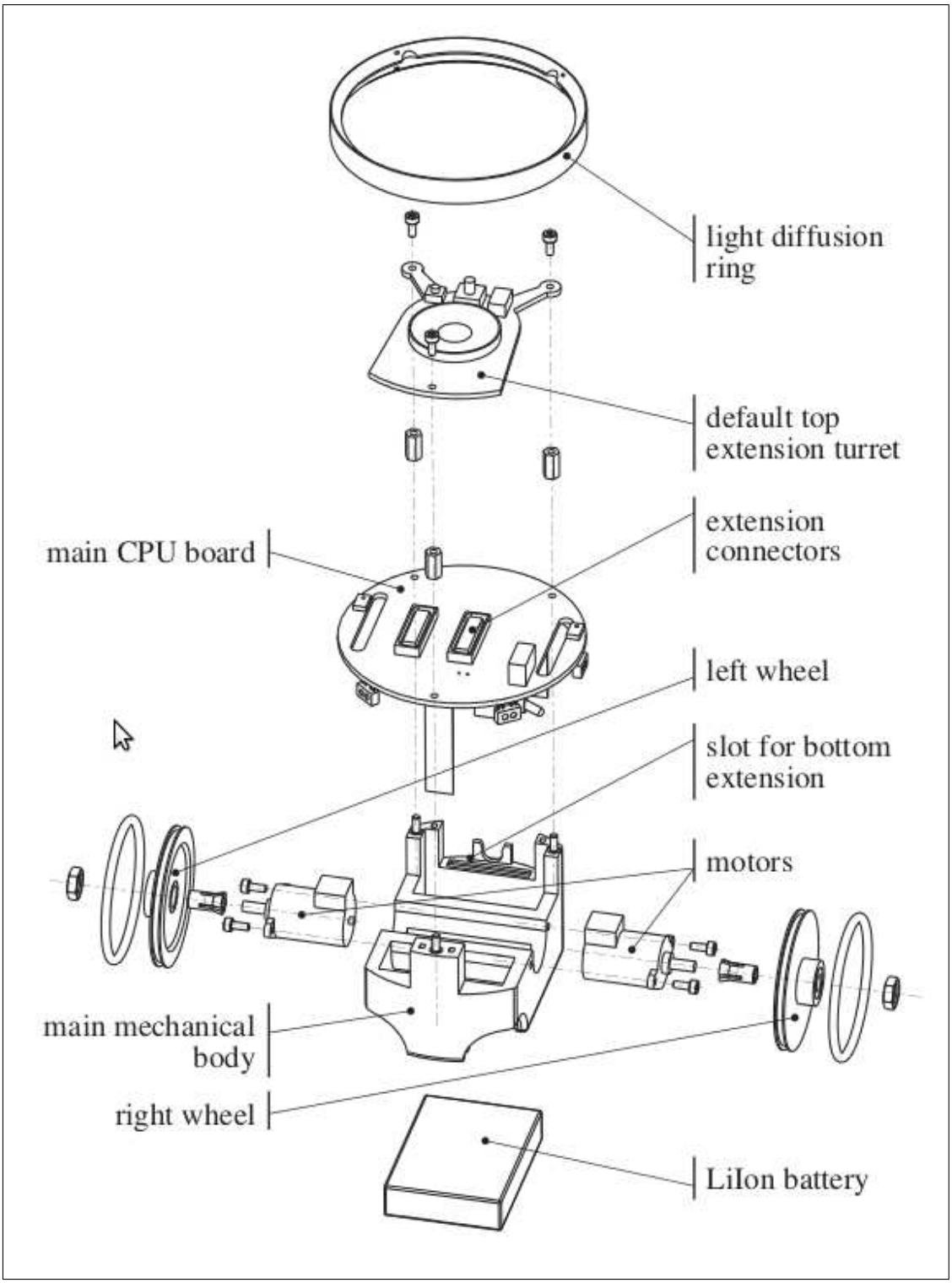


Fig. 2.9: Mechanical structure of the e-puck. Taken from [1].

2.1.6 E-puck mechanics

As it was mentioned one of the constraints in the design of the robot was the need to have it be small enough to be able to move around on a desk. This

[4] `http://en.wikipedia.org/wiki/Dhrystone`

resulted in a robot diameter of 75mm and a height which depends on the mounted connections. To reduce the cost of production, the manufacturing of the robot is a simple assembly of several parts such as: the main body, the light ring and the two wheels which can be seen in figure 2.9 [1]. The main printed circuit board is also shown, which contains most of the electronics as well as the default top extension board.

2.2 Gumstix Overo based extension board

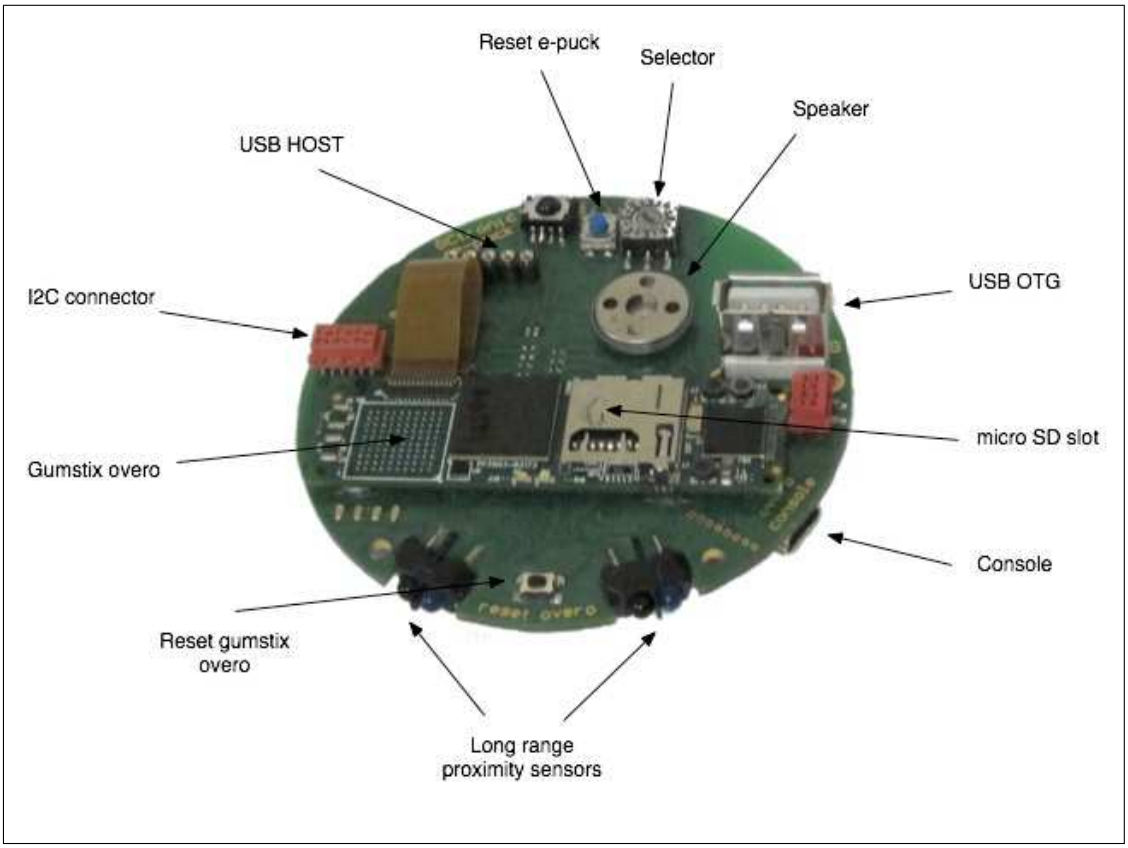


Fig. 2.10: Image of the Gumstix Overo extension with key elements pointed out.

Figure 2.10 shows an image of the extension board with key elements pointed out.

As it was previously mentioned and can be seen in the figure, the extension is based on the Gumstix Overo module with an ARM Cortex-A8 core running a Linux kernell. As far as communication with the outside world is concerned several interfaces are available, namely:

- USB host, used for a WiFi dongle which enables wireless communication. With this extension a Zyxel[5] wireless module is used. It is compatible with the 802.11n standard and when used with this extension achieves a **20Mb/s** throughput. In standby mode this device has a power consumption of 0.5W[6], while when it is active consumption goes to 1.5W[7].

- Another USB interface is available called USB OTG which can be used to extend the memory inserting a pen drive.

- A mini USB connector(console) used to enter the linux system running on the extension through a terminal.

[5] `http://www.gctronic.com/doc/index.php/Overo_Extension#Zyxel`
[6] corresponds to a current of 100mA with a power supply of 5V
[7] `http://www.gctronic.com/doc/index.php/Overo_Extension#Consumption`

- I^2C connector serving as a bus extension allowing a variety of different sensors to be used to expand the functionalities available on the robot.

Communication with the basic board of the e-puck robot is done through a serial RS232 connection. Programmed transfer rate is $230'400b/s$ with 8 data bits, 1 start bit, 1 stop bit and no parity bits. Figure 2.11 shows an image of the robot with the Gumstix Overo extension mounted on it, and figure 2.12 shows a block schematic of the entire system.

Fig. 2.11: Image of the e-puck robot with the Gumstix Overo extension mounted on it.

Other devices available are:

- Speaker[8].
- Rotary selector.
- 2 long range infrared proximity sensors. Details are available in the data sheet [16] and in appendix.

2.3 Power consumption

Since the robot is a battery powered device it is of great importance to be aware of the power consumption. Data provided by the manufacturer[9] are as follows:

- Power consumption of the robot and the extension without the wireless module is 1.5W.

[8] `http://projects.gctronic.com/Gumstix/datasheet-speaker.pdf`
[9] `http://www.gctronic.com/doc/index.php/Overo_Extension#Consumption`

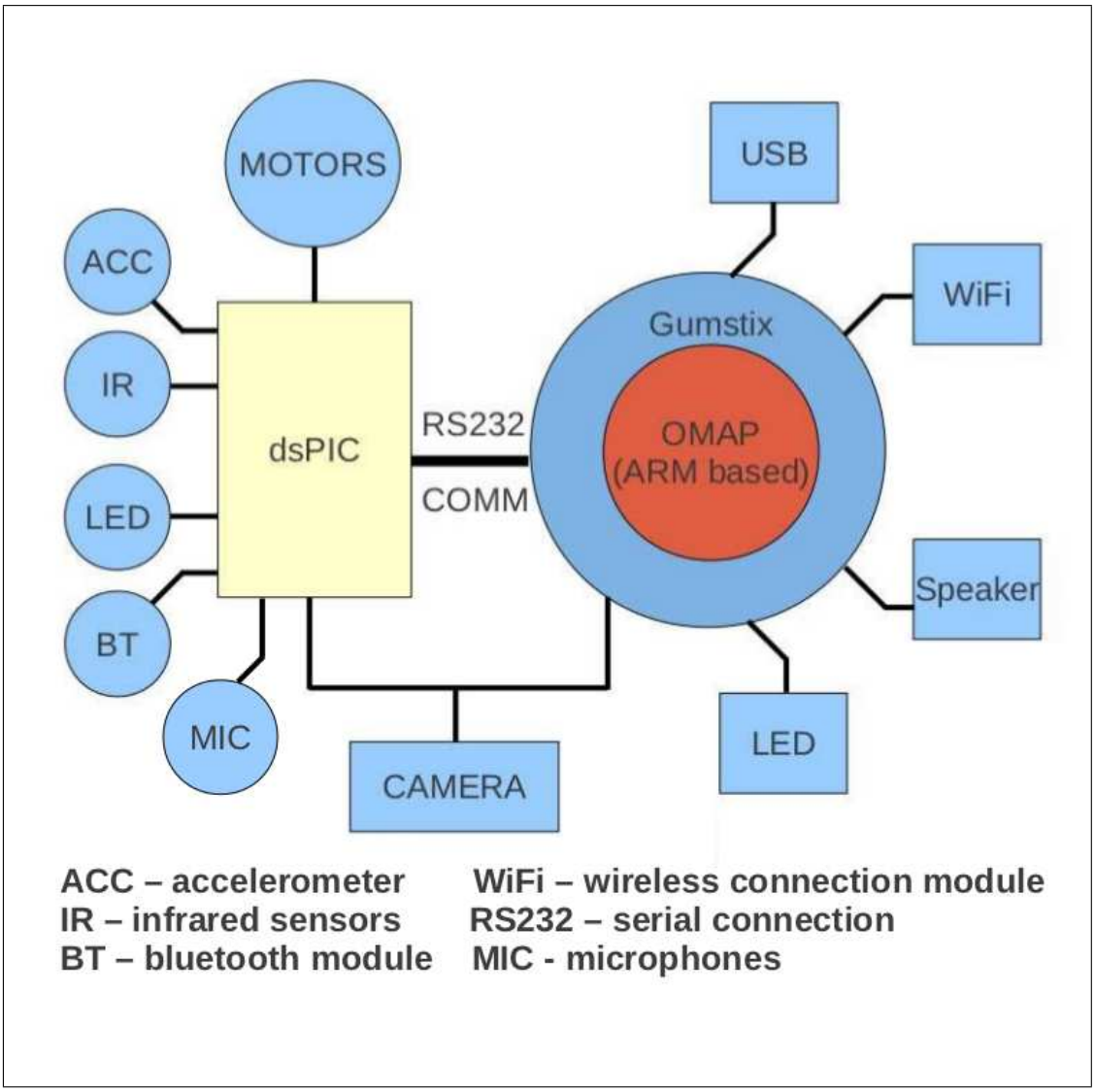


Fig. 2.12: Block schematics of the system consisted of the e-puck robot and the Gumstix Overo extension.

- With the ZyXEL wireless module in stand by mode power consumption is 2W.
- Finally power consumption of the robot and extension with the wireless module active rises up to 3.5W.

It should be noted that the nominal power supply voltage is **5V**. Battery capacity is **5Wh**. Provided data show that the wireless module alone, when active is consuming power as much as the robot and the extension together, thus reducing battery autonomy drastically. An illustration of this are the results of the following measurements provided by the manufacturer:

- e-puck and Gumstix extension active. Test conditions are: extension sends commands via serial connection to robot causing movement every 3s lasting 1s at 30% of maximum speed. This results in battery duration of **3h and 30 minutes**.

- e-puck, Gumstix and wireless module active. Test conditions are the same as in the first test with the wireless module continuously pinging. This results in battery duration of **1h and 25 minutes**.

Chapter 3

Webots software

Webots [5] is a three-dimensional, cross-platform mobile robot simulator. It was originally developed as a research tool for investigating various control algorithms in mobile robotics. It can be used as a rapid prototyping environment, allowing the user to create 3D virtual worlds with physics properties such as mass, joints, friction etc. in which one or more robots interact between each other or with the environment. Simulated robots can have different locomotion schemes, i.e. they can be wheeled robots, legged robots or flying robots. A variety of sensors and actuators can be simulated accurately such as drive wheels, cameras, servos, touch sensors, emitters, receivers etc. with a lot of freedom for the user to program each robot individually to exhibit the desired behaviour. Except simulation Webots software also enables the control of real robots. The control can be done through transferring the developed controller entirely from the Webots environment on to the robot or via remote control.

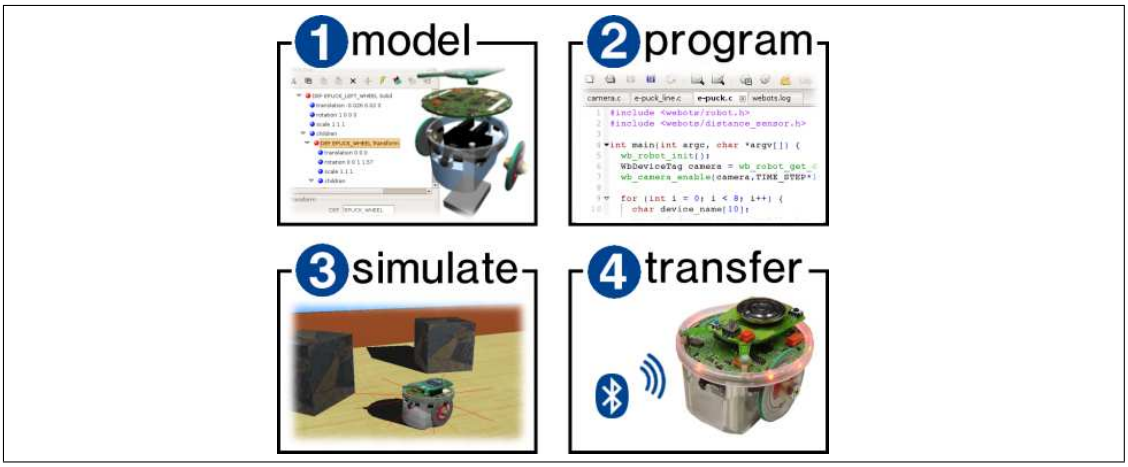


Fig. 3.1: Illustration of the features offered by the Webots software. It covers the entire design process from modelling a robot (1), through developing controllers (2) and simulating (3) to cross-compiling and transferring (4) the controller to the real robot. As opposed to cross-compiling and transferring the controller, Webots also supports remote controlling the real robot.

Figure 3.1 illustrates the possibilities of Webots software. The user has the choice to perform the entire design process from modelling a robot, through devel-

oping a program to control it and simulate it, to cross-compiling and transferring the program to a real robot. As opposed to cross-compiling and transferring the controller, Webots also supports remote controlling the real robot. There are a lot of robot examples and controllers already included into the software so the user has a choice of starting the process anywhere in between the first and last step. A wide spectrum of controller examples covering different fields of robotics and implementing different algorithms allows for greater reuse of their code. Further more a 3D environment enables visualising the progress of the simulation which is illustrated with a snapshot from a simulation involving an e-puck robot trying to follow a black line drawn on the floor. More details on the capabilities of the software is available in the Webots user guide [6].

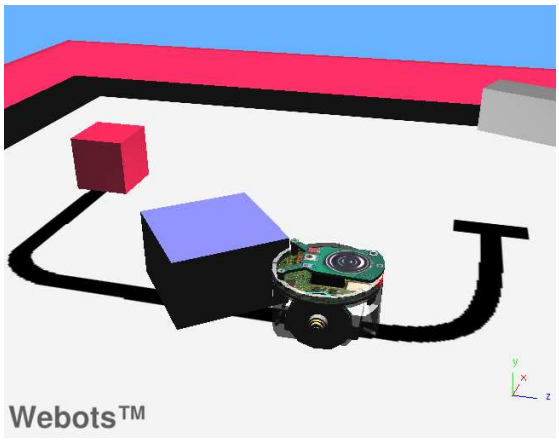


Fig. 3.2: Snapshot from a Webots simulation using an e-puck following a line on the floor.

A controller is a computer program that controls a robot. Typical organization of the controller is a loop consisting of:

- reading sensors and updating actuators
- processing acquired data and calculating new data
- performing a time step

Any two robots can be very different i.e. have different sensors, actuators as well as different processing units, for which different tools are needed to develop programs that can be executed by them. That is why the task of writing controllers is very robot specific. To overcome this Webots software provides a higher level of abstraction by providing to the users an application programming interface (API). This API is available in several versions for different programming languages, such as `C`, `C++`, `Matlab`, `Java` and `Python`. The base implementation of the API is done in C programming language and it contains functions which provide flexibility in working with robots, but also give a way of writing controllers with higher level of abstraction, not having to know details about the robot. The implementation of each of the function from the Webots API is different for each robot and is defined by the underlying libraries that handle the specifics. An example of an API function is given bellow:

```
const double* wb_accelerometer_get_values(WbDeviceTag tag);
```

This function takes the `tag` of a device as a parameter which specifies the actual device to be contacted. Return value is a pointer to an array of three variables representing the X-axis, Y-axis and Z-axis acceleration, respectively. It can be used to get the data from the accelerometer on every robot that has one, provided that there exists a library which defines how this function is implemented on a specific robot. This not only makes controller development easier for the users, but also makes them more portable since they are not robot specific. A list of Webots API functions relevant for this project, are given in appendix and can be found in the Webots reference manual [7] with much more details.

As it was mentioned the approach of enabling a higher abstraction level for the user requires a software library which will encapsulate the functionalities of a specific robot. In the case of the basic configuration of the e-puck this support already exists, but the basic version of the robot is limited in terms of memory and processing power. With the Gumstix Overo extension mounted on the e-puck these limitations are eluded and a much more powerful system is obtained. This in turn adds additional complexity to the system and a new software library which serves as a link between the Webots software and the e-puck with the extension had to be developed to be able to fully utilize the benefits of the extension on the robot. This was the main objective of the project and even though the purpose was to improve the support of the new extension in Webots software, the approach taken was to develop a library which is versatile, supports the Webots software API but is not limited to it and can be used standalone. This resulted in a solution which on its own provides a high level of abstraction, thus eluding the need for the user to know the details of the system.

Chapter 4

E-GO software library

With the addition of the Gumstix Overo extension to the e-puck robot the complexity of the system has increased. There are now two quite different processing units (dsPIC microcontroller on one side and ARM based OMAP application processor running a Linux kernel on the other), with quite different capabilities. Controllers are now executed on the extensions processing unit because its much more powerful. Communication with the outside world is also handled by the extension whilst most of the sensor acquisition and actuator control is handled by the dsPIC microcontroller. On the other hand, communication between the extension and the basic robot's configuration is handled through a serial connection.

All of these varieties make the system versatile and configurable but also more complicated for the user to handle. Developing controllers for this robot requires from the user to know the details of the system, such as how to configure the serial connection between the two processing units, what is the protocol used to communicate between them, and to which unit each sensor and actuator is connected. These demands in turn require to know how the Linux OS handles serial connections and implement that in the controller, what firmware is running on the dsPIC microcontroller and also how are the drivers of the Linux kernel configured.

As it can be seen quite a lot of details of both hardware and software (firmware, OS) of the system are needed to develop a controller for the e-puck robot with the Gumstix extension. The robot is intended to be used for education and by a broad range of users with different level of understanding of hardware and low level software. Having to know all of the previously mentioned details deflects them from using this system. Needless to emphasize that complex user interface of a device, in this case the e-puck robot, generally makes users avoid it.

Therefore there is the need to encapsulate all of the functionalities of the described system in a form that provides a simple and standardized access to its features. This demand can be fulfilled by implementing a software library, to serve as an interface between the user and the robot, providing a simple and clean interface, well documented and easy to understand, on one side and an efficient implementation of all the robots functionalities on the other. This approach also enables greater portability of the written code since it is less device specific.

This project implements E-GO software library to respond to the afore men-

tioned demands. The library is divided into two parts. One part operates on the Gumstix extension processing unit and is developed using `C` programming language. The other part is executed on the OS on which Webots software runs and is developed using `C++` programming language for non-GUI parts and `Qt` interface for GUI parts.

Several aspects have to be considered as far as the implementation is concerned, such as: constraints imposed by the project specification and the robot system, structure of the E-GO library, efficiency of the implementation, maintainability and modularity of the code etc.

4.1 System constraints

Before starting the development of software the constraints coming from the specifications of the project and the robot system have to be considered. Main project specification is to implement the solution in such a way so that Webots API functions can be used to control the robot. To examine the constraints imposed by the system a good starting point is to take another look on the block schematics of the system consisted of basic e-puck and Gumstix extension shown in figure 4.1.

Basic configuration board of the e-puck containing the dsPIC microcontroller has a software library developed which handles low level sensor data acquisition, actuator data update as well as serial communication. As it can be seen most of the sensors and actuators are connected to the dsPIC microcontroller (working at about $60MHz$) on the basic board of e-puck, while the controller executes on the Gumstix extension (working at about $600MHz$). This means that sensor and actuator data must be exchanged through the serial connection operating at a rate of $230'400b/s$. It is obvious that the serial connection is the bottle neck of the system, thus limiting the maximum rate at which data can be refreshed. Even more so, the E-GO library must have internal functions which handle the configuration of the serial connection and exchange of data, thus introducing overhead.

Communication interface between the two processing units has to be simple and minimalistic transferring as much as useful data as possible per unit of bandwidth. This basically means how many bits have to be transferred over the serial connection per one bit of useful data. To characterize the communication interface it will be described first.

4.1.1 Serial connection communication interface

Serial connection between the basic e-puck and the extension follows the RS232 communication protocol allowing the user to transfer data in byte granularity. Communication is done in such a way that first all the necessary commands in a particular time step are sent to the basic board of the e-puck and then all the responses are received. This is faster than to send a single command then wait for the response before sending the next command.

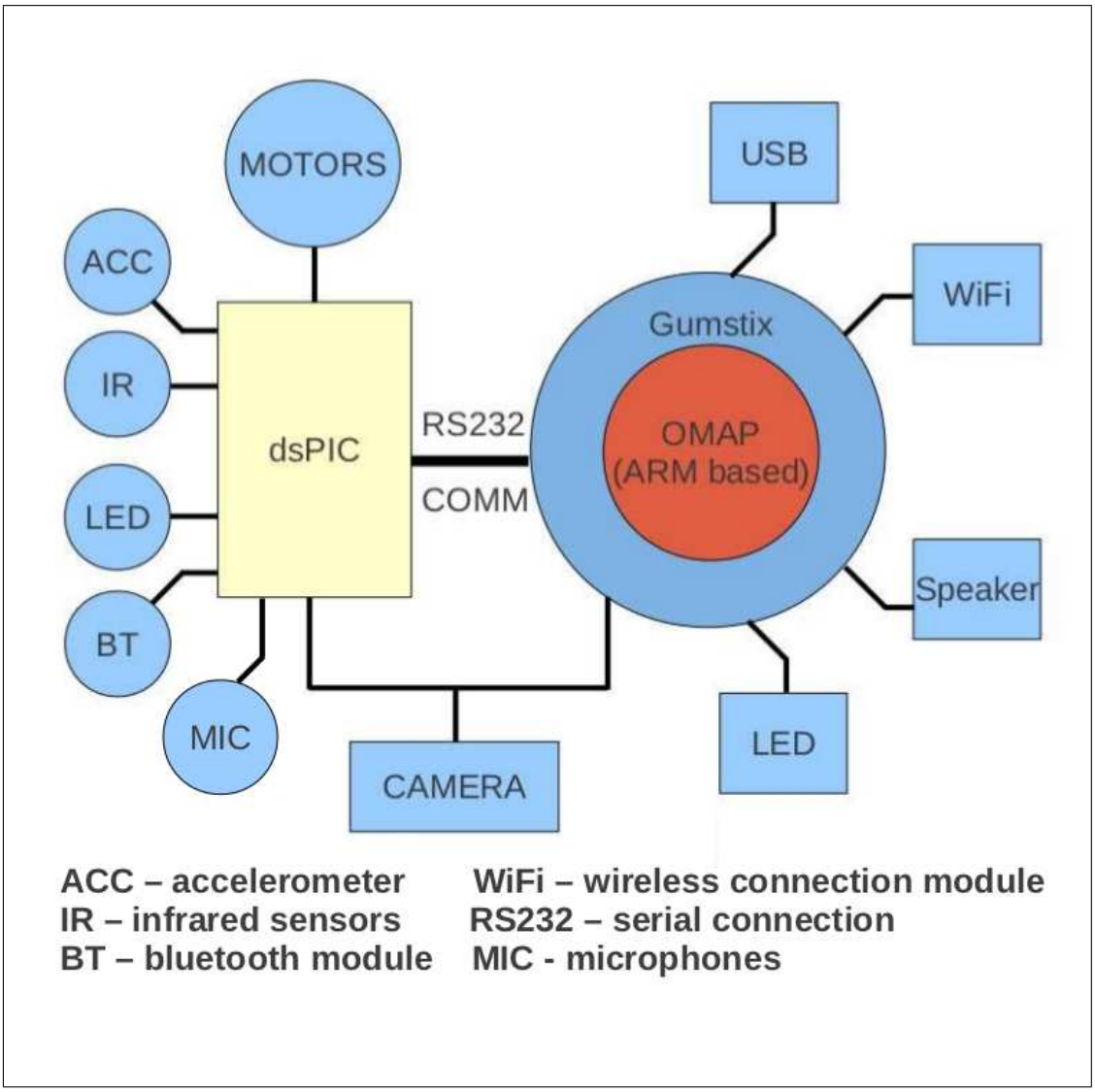


Fig. 4.1: Block schematics of the system consisted of the e-puck robot and the Gumstix Overo extension.

Since each character is represented by one byte of data the communication interface uses characters to identify different commands. Detailed description of the commands and their usage is given in table 4.1.

Each command is specified by one byte. There can be 0, 1 or 2 additional parameters to the command depending on the command, corresponding to 0, 2 or 4 additional bytes. Response to the commands can be 0, 4, 6 or 16 bytes depending on the command. Numbers sent to update actuator and those received from the sensors are considered useful data while bytes representing commands are considered overhead. In the case of the communication interface used in this project, overhead of each command is shown in table 4.2. Presented overhead for each command is one special case and another is sending all of the command in one time step. Such a command buffer is shown in figure 4.2 and the overhead in that case is **15.3%** (19 command bytes and 105 bytes of sent and received useful data).

Table 4.1: Commands of the serial communication interface

Commands	**First arg.**	**Second arg.**	**Command size [byte]**	**Response size [byte]**	**Description**
a	-	-	1	6	Returns accelerometer values for x, y and z axis.
D	left motor	right motor	5	0	Set motor speeds.
E	-	-	1	4	Returns left and right motors speed, respectively.
L	number	value	4	0	Set the *number* LED with *value* which can be 0=off, 1=on or 2=toggle.
M	-	-	1	6	Returns values of floor sensors.
N	-	-	1	16	Returns values of proximity sensors.
O	-	-	1	16	Returns values of light sensors.
P	left encoder	right encoder	5	0	Set encoder values.
Q	-	-	1	4	Returns left and right encoder position, respectively.
u	-	-	1	6	Returns microphone amplitudes.
0	-	-	1	0	Marks termination of the command buffer.

Table 4.2: Commands overhead

Commands	**Useful bytes (U)**	**Overhead bytes (O)**	**Overhead[%]=$\frac{O}{O+U}$**
a	6	1	14.3%
D	4	1	20.0%
E	4	1	20.0%
L	2	1	33.3%
M	6	1	14.3%
N	16	1	5.9%
O	16	1	5.9%
P	4	1	20.0%
Q	4	1	20.0%
u	6	1	14.3%

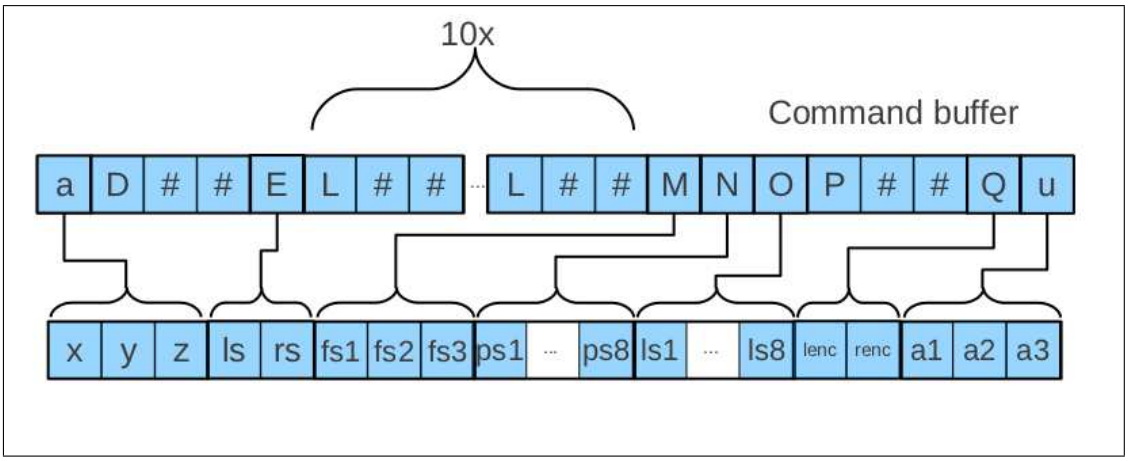


Fig. 4.2: Example of transferred data when issuing all commands. Upper buffer shows sent commands according to table 4.1 and lower buffer shows the response to those commands. (**x, y, z** - axis accelerations; **ls, rs** - left and right motor speed; **fs1, fs2, fs3** - floor sensors values; **ps1...ps8** - proximity sensors values; **ls1...ls8** - light sensors values; **lenc, renc** - left and right encoder values; **a1, a2, a3** - microphone amplitude values)

Solution chosen for the communication interface in this case can be compared to an alternative one which eliminates the overhead of issuing commands. In the simplest form this alternative solution would be to assume all the commands are always issued thus avoid sending bytes which mark them. Comparison of the two approaches is given in table 4.3.

Considering the advantages and drawbacks of the two mentioned approaches it is justified to continue with the first one which is already implemented in basic e-puck and Webots. This makes further implementation easier and even though the performance in the worst case scenario of the first approach is about 15% worse, it should be kept in mind that in most applications such a scenario would rarely occur.

Important parameter of the serial connection is the effective speed at which data can be transferred. For this purpose a test was conducted which measured the time needed to transfer data of 10′000 transfer cycles (i.e. time steps). Total amount of transferred data is 954′940$bytes$ and the total time it took for the transfers is 80.24s[1], yielding an effective speed of $\mathbf{84.03\mu s/byte}$.

Measuring time is also an important aspect of the implementation. A robot controller is in fact consisted of a series of time steps and sampling of sensor data and updating actuators with a defined frequency requires support for efficient time measurement on the robot system. This is dealt with in the following subsection.

4.1.2 Time measurement

There exist three options for measuring time in the case of this project. One is to measure it with the timers from the dsPIC microcontroller on the basic board

[1]Time measurement was done from the processing unit of the Gumstix extension using Linux OS built-in function (clock_gettime()).

Table 4.3: Comparison of two command interfaces

Command interface description	**Advantages**	**Disadvantages**
First approach: Issuing commands for each transfer cycle (used in this project)	• Only those commands that are needed are issued at each transfer cycle. • Already implemented in the e-puck library and Webots software thus nothing would have to be changed there.	• Overhead of sending extra bytes which mark the commands is introduced (124 bytes in worst case scenario). • Additional code to keep record of commands issued and the size of the expected response.
Second approach: Assuming all commands issued at each transfer cycle (hence not explicitly sending them)	• No overhead of sending commands, only data is transferred. • No need for additional code to keep record of transfer buffers size.	• All possible data of all sensors and actuators transferred at each cycle even when not needed (105 bytes). Depending on the application this can introduce significant overhead. • Besides the implementation for the extension it would have to be implemented on the basic e-puck and Webots as well.

of the e-puck, another to measure it from the Gumstix extension processing unit and the third and most reliable one is to measure it with a logic analyser or an oscilloscope. Because of the hardware configuration (difficult access to pins) and large amount of measurements that needed to be done, using logic analyser or an osciloscope was not practical. The two other options were implemented in this project to analyse their advantages and drawbacks. Time values were held in program variables and displayed using a `printf` function. It should be kept in mind that the robot controller is executed on the Gumstix extension processing unit and all of the timings will need to be available to it.

Time measurement with dsPIC microcontroller timers

Test implementation in this case was the following:

- From the Gumstix extension a signal is sent to the dsPIC microcontroller to start measuring a time interval. Also a value specifying the size of the interval is sent.

- This results in calling a function which starts the timer on the dsPIC microcontroller. This function including the settings for the timer already exists in the e-puck library.
- When the time step expires a signal is sent back to the Gumstix extension to mark its ending.

The actual passed time is measured from the Gumstix extension. It is defined as the time from initiating the process for a time step described above until a confirmation is received that the time step has expired. Table 4.4 shows measurement results. Each result is an average of 100 measurements. There we can see that there is an overhead appearing when measuring time this way in the range of $(5-8)ms$. This is caused by several cumulative factors. One factor for of the overhead is the time needed to send a signal from Gumstix extension to the dsPIC microcontroller through the serial interface to start measuring a time step, then the time needed for the firmware application running on the dsPIC to decode the command (this is computationally heavy operation because there are a lot of possible commands and therefore a lot of checks to determine which command is sent) and also the time it takes for the dsPIC to respond to the extension that a time step has finished. Timer on the dsPIC measures time with millisecond precision. This means that every millisecond an interrupt is generated by the timer in which a program variable is incremented meaning that another millisecond has passed. Time step function starts the timer when called and then polls the value of this program variable once it finishes all of the other tasks inside it. This way if tasks are done before the wanted time step has expired it will wait for the full time step. The other factor causing overhead is the imprecision of time measurement which can add up to $1ms$.

Table 4.4: Time measurement with dsPIC timer

Time step [ms]	**Total time [ms]**	**Overhead [ms]**
8.0	15.6	7.6
16.0	23.4	7.4
32.0	39.0	7.0
64.0	71.2	7.2
128.0	133.8	5.8
256.0	263.0	7.0
512.0	518.8	6.8

dsPIC microcontroller used on the basic board of the e-puck offers a variety of configuration options for its timers. On one hand the low level access allows for a much greater degree of freedom but on the other it requires knowledge of the details about the device. Table 4.5 gives an overview of the advantages and disadvantages of this solution.

Time measurement with Gumstix extension

Gumstix extension has an OMAP application processor on it running a Linux kernel. Time measurement in this case is somewhat different than with the dsPIC

Table 4.5: Analysis of the dsPIC time measurement solution

Advantages	**Disadvantages**
• Fine (low level) control over highly configurable timer parameters. • Already implemented code for time measurement and serial connection interface in the e-puck software library.	• Low level control of the timer requires knowledge of details of the device. • Time overhead introduced. Could be ignored for time step values larger than $128ms$ because of its rather constant nature, but not for smaller ones. • Occupying bandwidth of the serial interface. This is a serious drawback since time step requests would be on the serial bus most often potentially clogging the bus.

microcontroller. There is no more low level access to the timers of the device, but a higher level of abstraction is implemented. There is an OS which provides timing functions (e.g. clock_gettime()) through a higher level library than what we have with the dsPIC. It should also be noted that the Linux OS running on the extension is not a real-time OS. Therefore there is no explicit support for real-time design and no guarantees regarding any of the relevant timing parameters such as process execution time. There are also other aspects which influence how the code will be executed, such as: what is the priority of the process, how many processes are executing concurrently, how is scheduling handled, is a process multi-threaded or not etc.

Test conditions for Gumstix extension time measurement evaluation were: custom application, which has a single-thread and a multi-thread version, runs on the Linux OS and has a time step function inside it. This function takes a time parameter specifying how long it should last exactly. Separately, time of the function execution is measured. Results are classified in 4 different groups. Specifically results falling into the area of 1% difference from the specified time (for which the mentioned functions is supposed to last) are counted as well as those falling into the 5%, 10% and 20% groups. This way the precision of measuring time this way is characterized.

For example, if a time step of 10 units is desired, number 10 is passed as a parameter to the time step function. Then the actual time of execution is measured. If the measured time is 10.7 units, this would mean that the relative difference between the wanted and measured time is $|10.7 - 10.0|/10.0 = 0.07$ (7%). This result would be counted in both the 10% and 20% groups, but not in the 1% and 5% groups since the offset falls into the 10% and 20% range. Specified times used in this test varied from $1ms$ to $500ms$.

Table 4.6 shows how many of all the measurements (in percentages) fell into each of the afore mentioned groups. In this case a single-threaded version of the

application was used. Table 4.7 shows the same results but for a multi-threaded version of the application. Number of samples was in excess of 200 for each measurement. Real time was measured with the `clock_gettime()` function.

Table 4.6: Time measurement with Gumstix extension (single thread)

Time values [ms]	1% offset	5% offset	10% offset	20% offset
1-10	98.58%	98.58%	99.53%	99.53%
10-100	99.53%	100.00%	100.00%	100.00%
100-500	100.00%	100.00%	100.00%	100.00%

Table 4.7: Time measurement with Gumstix extension (multi thread)

Time values [ms]	1% offset	5% offset	10% offset	20% offset
1-10	63.68%	63.68%	64.15%	65.09%
10-100	58.02%	68.87%	79.25%	90.57%
100-500	74.06%	95.75%	99.53%	100.00%

Comparing the measurements results it can be noticed that with a single-thread the execution time of the function is much closer to the specified time than with multiple-threads in the application. For smaller specified time values this difference is much more emphasized than for higher values.

Finally the choice for where will time measurement be handled falls to the Gumstix extension since the alternative solution with the dsPIC has significant drawbacks, such as potentially clogging the serial bus and greater time overhead.

4.2 Structure of the software library

The structure of the developed E-GO software library will be described here. Main goals were to design a modular, easily maintainable, efficient high level of abstraction library which provides an interface to all of the robot's functionalities. Figure 4.3 illustrates the structure of the developed library. It can be seen that the software library is in fact divided into two sub parts one of which is the cross-compilation library and the other remote control library. This is because there are two basic ways a robot can be controlled, by running a controller on it directly, thus the developed controller's code has to be cross-compiled to run on the robot or remotely from a remote station such as a PC. In both situations the support of the cross-compilation library is necessary. The remote control library adds those functionalities needed on the remote station's side to form an interface towards the controlled robot.

In the cross-compilation library each class of sensors and actuators have their own module where their functionalities are handled. Besides those, there are additional modules which are handling the serial communication between the extension and the basic e-puck robot, providing utilities for features such as keeping track of issued commands and expected response, measuring time and using the **V4L2** (Video for Linux 2) [17] driver capabilities.

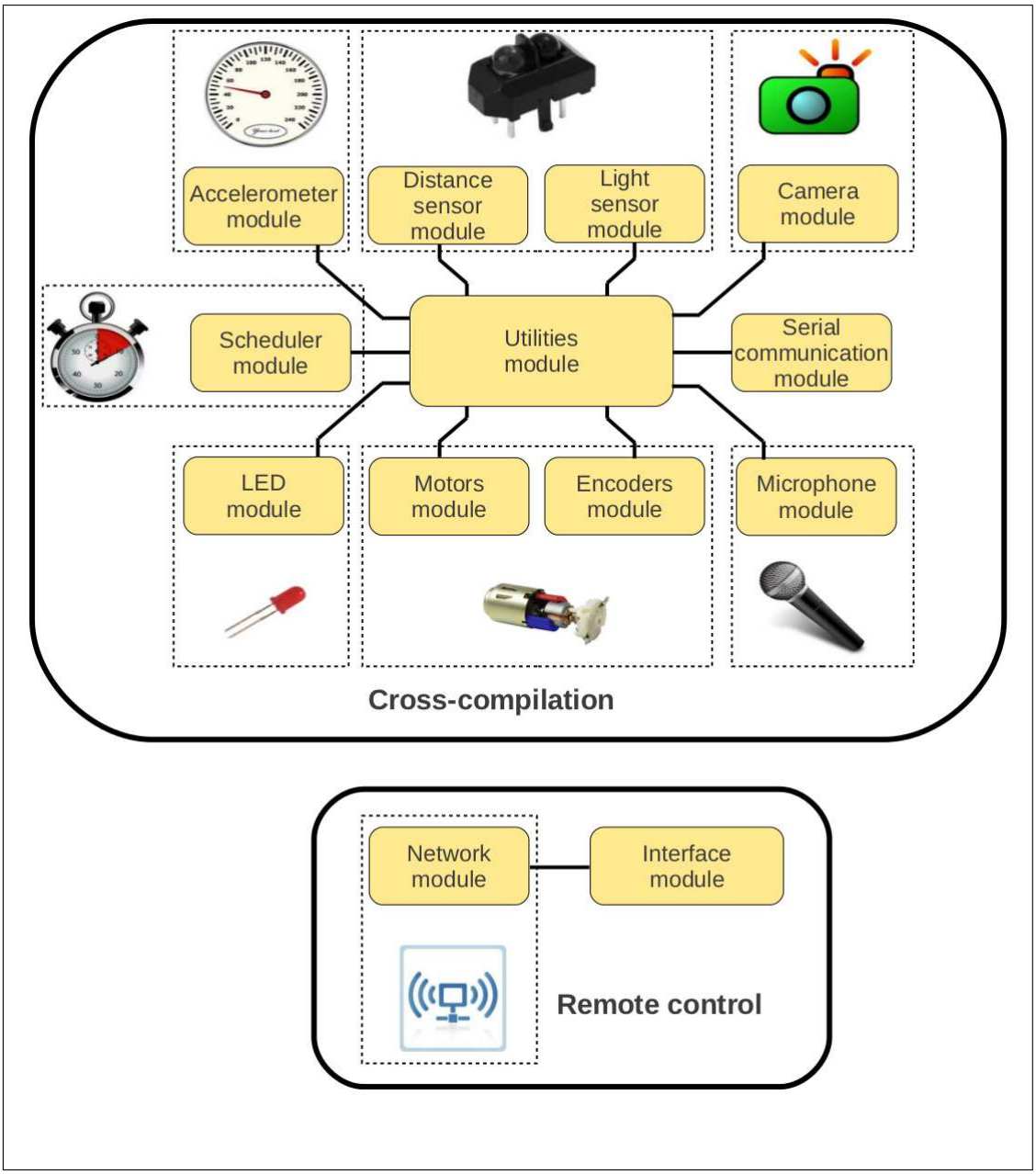


Fig. 4.3: Structure of the developed software library

Camera

Figure 4.4 illustrates the process of getting one single image from the camera. Before it is delivered to the user an image is handled by several application layers. First the video driver inside the kernel space of the Linux OS running on the extension handles the lowest level acquisition on one side and provides the data conformed to the V4L2 API specification on the other. The V4L2 API level provides a somewhat higher level of abstraction while still allowing for the large portion of the low level control and flexibility to be used. Next the image is passed to the E-GO software library through the V4L2 API and provided to the user with an even higher level of abstraction, masking the details of the configu-

ration and setup of the camera device. These details are handled by the E-GO library. Obviously such an approach allows for a complicated system with a lot of parameters to be used by the users in a simple manner, but on the other hand passing the image through several layers of software introduces timing overhead which degrade the performance. It is important to characterize this overhead and identify points which can be optimized and whose optimization would have the greatest impact on the performance improvement.

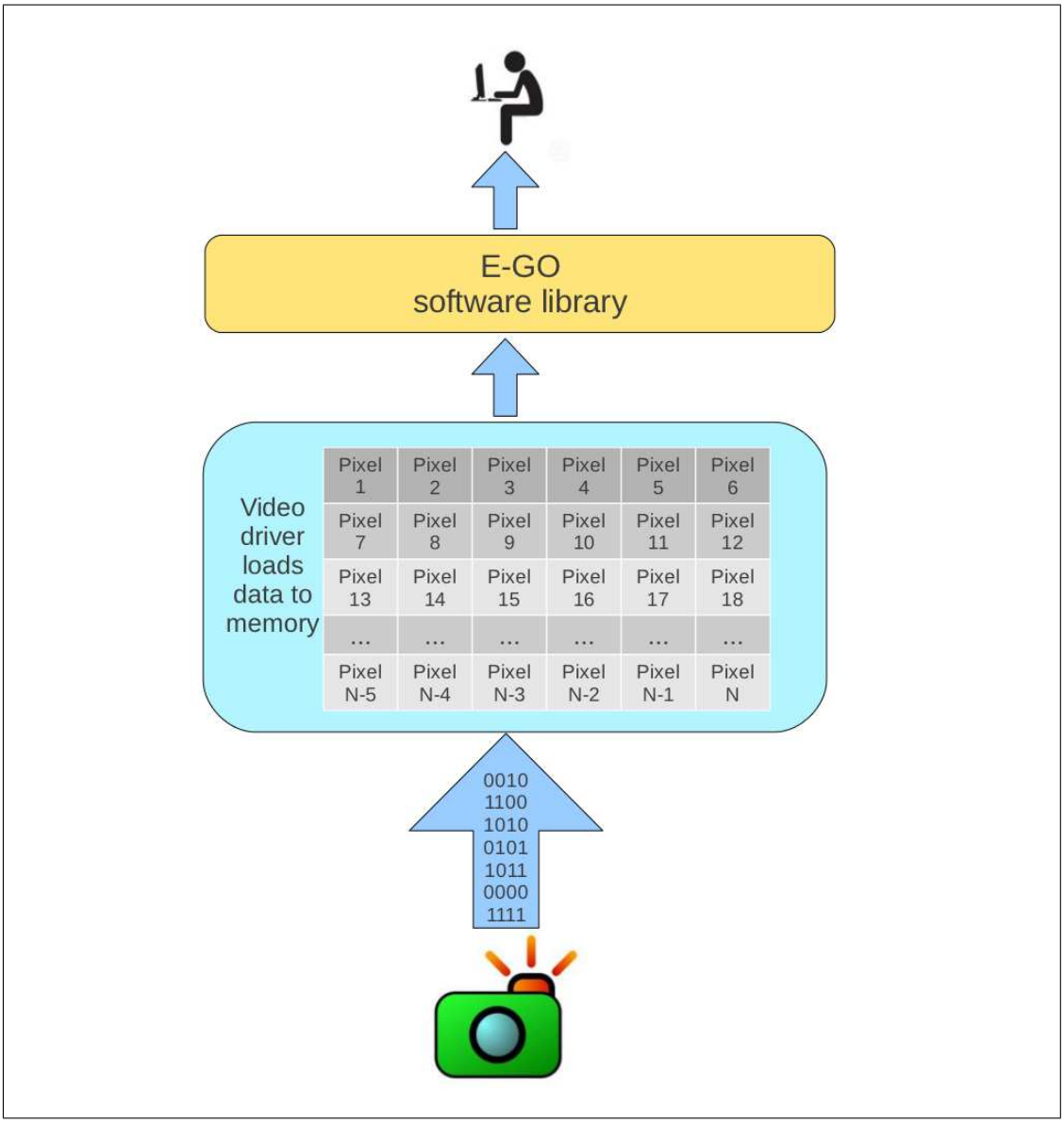


Fig. 4.4: Illustration of the process of acquiring a single image from the camera device and providing its data to the user. Data from the camera is handled by the video driver which stores them in memory and provides access to the image following the V4L2 API specification. E-GO's library functions retrieve the information about the image (image pointer) and pass them to the user.

Figure 4.5 shows what are the responsibilities of the developed software library once it takes over the image from the Linux driver.

• The driver supplies the image written in RGB565 or YUV422 format which needs to be converted to RGB888 format.

• Because of the way camera is mounted on the robot the image acquired is rotated 180° so it needs to be rotated back.

• Depending on whether a compression is requested by the user a corresponding compression algorithm is applied to the image.

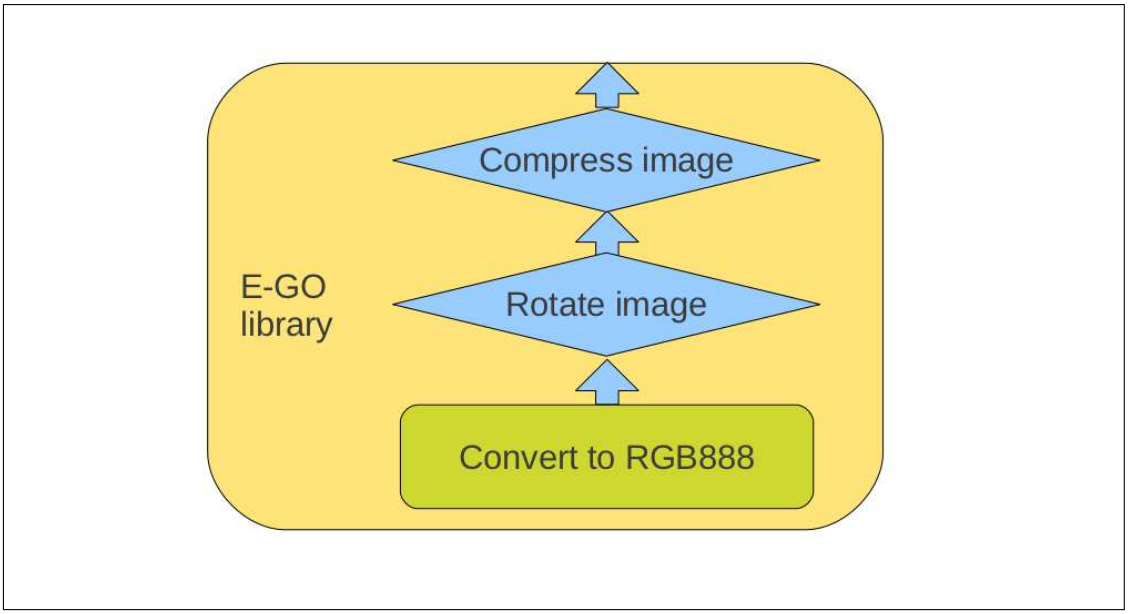


Fig. 4.5: Shows the tasks which need to be performed on the image received from the video driver. Conversion to RGB888 always occurs. Rotation is not mandatory. If not needed it is skipped. Angle is determined by the users choice and it can be 90°, 180° or 270°. Compression is chosen by the user. It can be none, JPEG or PNG format.

RGB565 to RGB888 conversion

RGB565[2] image format represents a single pixel with 2 bytes of data where as RGB888 represents it with 3 bytes of data. Figure 4.6 shows both representations as well as how the conversion is handled in this case. Conversion between the two formats is done with the help of a look-up table where each value of one formats component has an equivalent value in the other formats corresponding component. This solution can be implemented very computationally efficient by using the values of the RGB565 components as indexes to get their equivalent values from the RGB888 format. Figure 4.7 shows the look up tables used for this purpose. There are two of them since for the red and blue component 5 bits are converted to 8 bits, where as for the green component 6 bits are converted to 8 bits.

[2] `http://en.wikipedia.org/wiki/RGB_color_model`

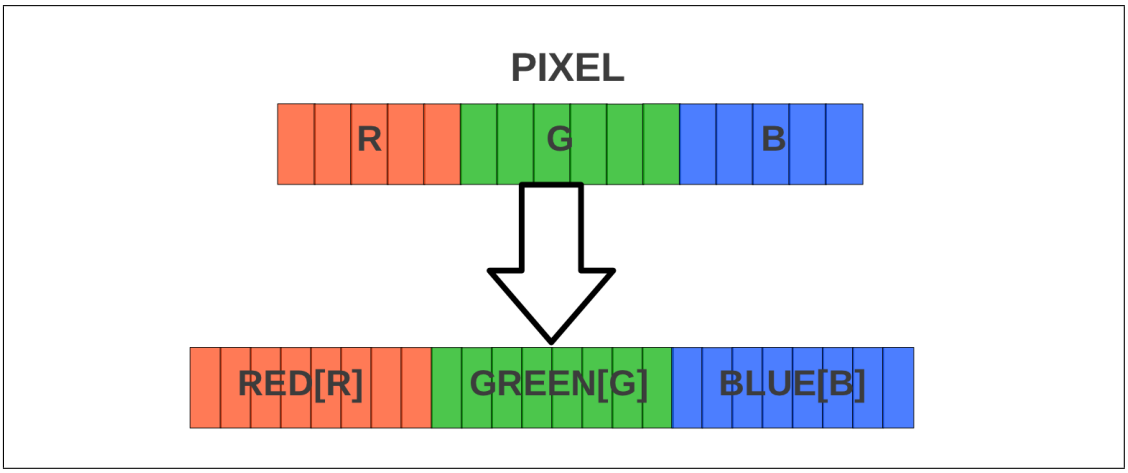


Fig. 4.6: Illustrates the conversion process from RGB565 image format to RGB888 image format as well as how these formats represent a pixel of an image. R, G and B are values of the red, green and blue components in RGB565 format. They act as indexes to search their corresponding value in the look-up tables, which give the values for the RGB888 format.

5 bits to 8 bits

0	8	16	25	33	41	49	58
66	74	82	90	99	107	115	123
132	140	148	156	165	173	181	189
197	206	214	222	230	239	247	255

6 bits to 8 bits

0	4	8	12	16	20	24	28
32	36	40	45	49	53	57	61
65	69	73	77	81	85	89	93
97	101	105	109	113	117	121	125
130	134	138	142	146	150	154	158
162	166	170	174	178	182	186	190
194	198	202	206	210	215	219	223
227	231	235	239	243	247	251	255

Fig. 4.7: Shows the look-up tables used to convert an image from RGB565 to RGB888. RGB565 component values are used as indices to get the value after conversion. Upper table is used for red and blue components and lower for the green component. (E.g. a value for red component in RGB565 of 10 corresponds to a value in RGB888 format of 82)

YUV422 to RGB888 conversion

YUV422[3] image format represents a pixel of an image with 2 bytes of data on average. This type of conversion is somewhat more complicated than the previous one since the two image formats differ greatly. Figure 4.8 shows how an image is represented in YUV422 image format. Y values represent pixel luminance (brightness) while U and V represent colour components. Each pixel has its own Y component while the U and V components are shared between two pixels (in figure 4.8 pixel with Y1 luminance uses U0 and V0 colour components), thus reducing necessary bandwidth per pixel. Equations used to convert from YUV422 image format to RGB888 depending on the standard are given bellow:

ITU-R float version (cost: 4 multiplications, 8 additions and 3 comparisons)

$$
\begin{aligned}
R &= clamp(Y + 1.402 * (V - 128)) \\
G &= clamp(Y - 0.344 * (U - 128) - 0.714 * (V - 128)) \\
B &= clamp(Y + 1.772 * (U - 128))
\end{aligned}
$$

NTSC version (cost: 6 multiplications, 14 additions, 3 comparisons and 3 shift operations)

$$
\begin{aligned}
R &= clamp((298 * (Y - 16) + 409 * (V - 128) + 128) >> 8) \\
G &= clamp((298 * (Y - 16) - 100 * (U - 128) \\
&\qquad -208 * (V - 128) + 128) >> 8) \\
B &= clamp((298 * (Y - 16) + 516 * (U - 128) + 128) >> 8)
\end{aligned}
$$

where `clamp()` function denotes clamping a value to the range of 0 to 255.

ITU-R integer version (cost: 14 additions and 12 shift operations)

$$
\begin{aligned}
R &= Y + V + (V >> 2) + (V >> 3) + (V >> 5) \\
G &= Y - ((U >> 2) + (U >> 4) + (U >> 5)) \\
&\qquad - ((V >> 1) + (V >> 3) + (V >> 5)) \\
B &= Y + U + (U >> 1) + (U >> 2) + (U >> 6)
\end{aligned}
$$

All three possibilities are implemented in this project and it was determined that ITU-R integer version of the YUV422 to RGB888 comparison runs faster than the other two possibilities on the Gumstix extension. ITU-R integer was also used to compare the YUV422 to RGB888 with the RGB565 to RGB888 conversion times.

[3] `http://en.wikipedia.org/wiki/YUV`

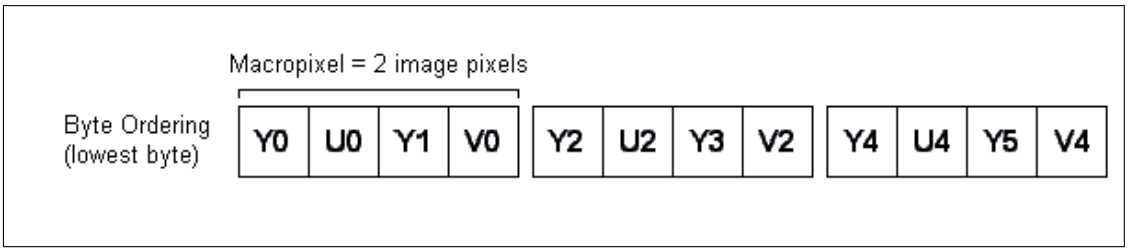


Fig. 4.8: Shows how image is represented with YUV422 format.

Remote control library, on the other hand has two parts, one of which handles the communication interface and the other handles graphical representation of the robot and transmitted data. Communication interface includes modules responsible for interfacing the e-puck robot and Webots software. Gumstix extension has a wireless module allowing the e-puck to communicate through a broadband wireless network. This is particularly suitable for remote control. For that purpose the remote control library has modules which implement network protocols such as TCP and UDP as well as modules which implement the interface protocol which is used to communicate between the remote station and the robot. Modules handling communication interface protocol were developed also.

Network

Remote control is basically a client-server interface between two software applications, one running on the robot and the other on the remote station. Which of the applications will act as a server and which as a client has to be determined. Since the robot is the one providing data in response to a request as well as processing received data also on a request it is convenient to have a small and simple server application running on the robot. This is also beneficial because there is need for very little to no intrusion on the robot side when establishing a connection since the request is made from the client application which is running on the remote station, usually a PC. Having multiple robots to control is also easier since all the connections are made from one single device (e.g. users PC). In the opposite scenario it would be necessary to manually access every single robot to set the parameters of the connection, such as the IP address and the port to connect to.

Collection of three classes handle network functionalities as shown in figure 4.3. `Broadband` which is responsible for collecting information about available networks and getting the information about the IP address and port towards which to establish a connection. This class also determines from the users choices which internet protocol to use and call the appropriate objects constructor. `TCP` class handles the specifics of establishing a connection which uses `tcp` networking protocol. `UDP` has the same purpose but is called when `udp` network protocol is used.

The reason why an option of these two network protocols are supported is because of their very different characteristics. The choice of the protocol to be used can be greatly influenced by the application for which it is intended.

TCP[4] uses a handshaking protocol to establish a connection between two nodes. A "connection" is basically a set of state variables ensuring all transmitted data reached their end destination by requesting confirmation for each packet of data. This makes it optimized for accurate delivery at the cost of transmission overhead and increased

[4] `http://en.wikipedia.org/wiki/Transmission_Control_Protocol`

latency. Because of its reliability it is convenient for applications which need guarantees of accurate delivery. It is however not convenient for applications which are time constrained such as most real-time applications.

UDP[5] on the other hand is relieved of any handshaking dialogue and follows a simple transmission model which allows it to avoid transmission overhead of TCP and thus be convenient for time-sensitive applications. The cost is reduced reliability which means packets may arrive out of order or not at all. Any necessary error checks have to done by the application or use another protocol which provides reliability.

Communication interface & GUI

Class `Communication` implements the communication interface on the remote control station's side. Handles for Webots API are provided in `Robot` class and `RemoteControl` class collects information of all available interfaces (serial, broadband). Graphical user interface (GUI), shown in figure 4.9 handles the graphical representation of the robot and the transmitted data. An existing library `libepuckwindow` was reused for this purpose in this project.

Technical documentation related to the implementation of the software library are available in the appendix.

[5] `http://en.wikipedia.org/wiki/User_Datagram_Protocol`

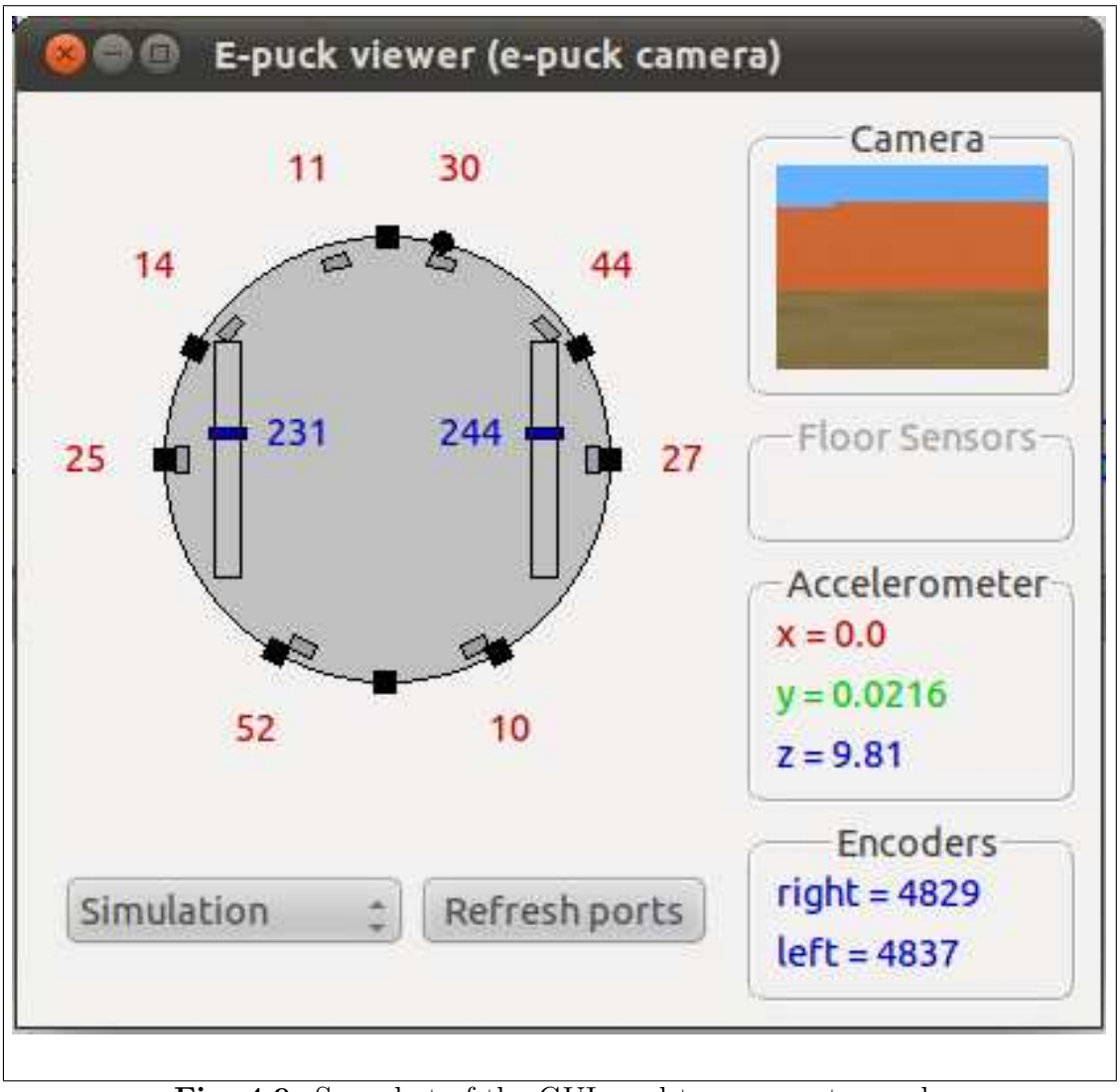


Fig. 4.9: Snapshot of the GUI used to represent e-puck.

Chapter 5

Results

This chapter presents results obtained from characterization of the developed software library. It is consisted mainly of profiling results, measuring execution time of critical pieces of code. As it was shown in the previous chapter a bottle neck of the system is the serial bus between the basic board of the e-puck robot and the Gumstix extension because of its limited bandwidth. Measurement have also shown that the average time for transmitting a byte of data through the serial bus is about $\mathbf{85}\boldsymbol{\mu s}$.

On the other hand another limitation on the bandwidth of the total system is image acquisition subsystem, because of inherently high amount of data that are transferred. For this reason parts of the code handling image acquisition were tested extensively.

5.1 Image acquisition

It was mentioned in the previous chapter that the image is written in either RGB565 or YUV422 format when it is received from the driver. The choice of format is selectable and both use 2 bytes per pixel. Compared to RGB888, which uses 3 bytes per pixel there is considerable savings in bandwidth and buffer size when the image is sampled by the sensor and acquired from its buffers by the driver. Nevertheless for further processing the image has to be converted to RGB888 format. Acquisition of the image using both RGB565 and YUV422 formats was tested to see which one is faster with the combination of the image sensor and video driver available in the system used in this project.

Table 5.1 shows the times needed to acquire a single image for both formats for 10 different resolutions. Time was measured from the moment an image was requested to the moment a pointer to an array containing the image was returned. In between additional step of conversion to RGB888 is performed. Each result in the table is an average of 7 independent measurements and each measurement is an average of 100 image acquisitions. Along with the average value, maximum offset in percentage of the average value is also given.

Figure 5.1 shows graphically the dependency between the times needed for image acquisition for both formats and the number of pixels in an image. It can be noticed that the times change linearly with the number of pixels in an image. It can also be noticed that the times for the RGB565 image format are significantly lower than those for YUV422 format. Acquisition for RGB565 format is about 2.5x faster than for YUV422 format.

This can be explained by a more efficient implementation of the conversion from

Table 5.1: RGB565 and YUV422 acquisition times for different resolutions

Number of pixels	RGB565 time [ms]	RGB565 max. offset [%]	YUV422 time [ms]	YUV422 max. offset [%]
307'200	62.4	0.03	158.6	0.04
276'480	56.2	0.07	142.8	0.02
245'760	50.0	0.07	127.0	0.07
215'040	43.7	0.06	110.9	0.13
184'320	37.5	0.14	95.2	0.08
153'600	30.8	0.16	78.8	0.2
122'800	24.6	0.10	62.9	0.06
92'160	18.3	0.09	47.2	0.11
61'440	12.0	0.15	31.3	0.09
30'720	5.9	0.15	15.5	0.38
1'024	0.2	1.00	0.5	2.22

RGB565 to RGB888 than of YUV422 to RGB888. Namely a conversion from RGB565 to RGB888 costs three accesses to arrays representing the look-up tables (32 or 64 table size depending on the component) for three components (red, green and blue), where as YUV422 to RGB888 conversion costs 12 shift and 14 addition operations for the fastest implementation (ITU-R integer).

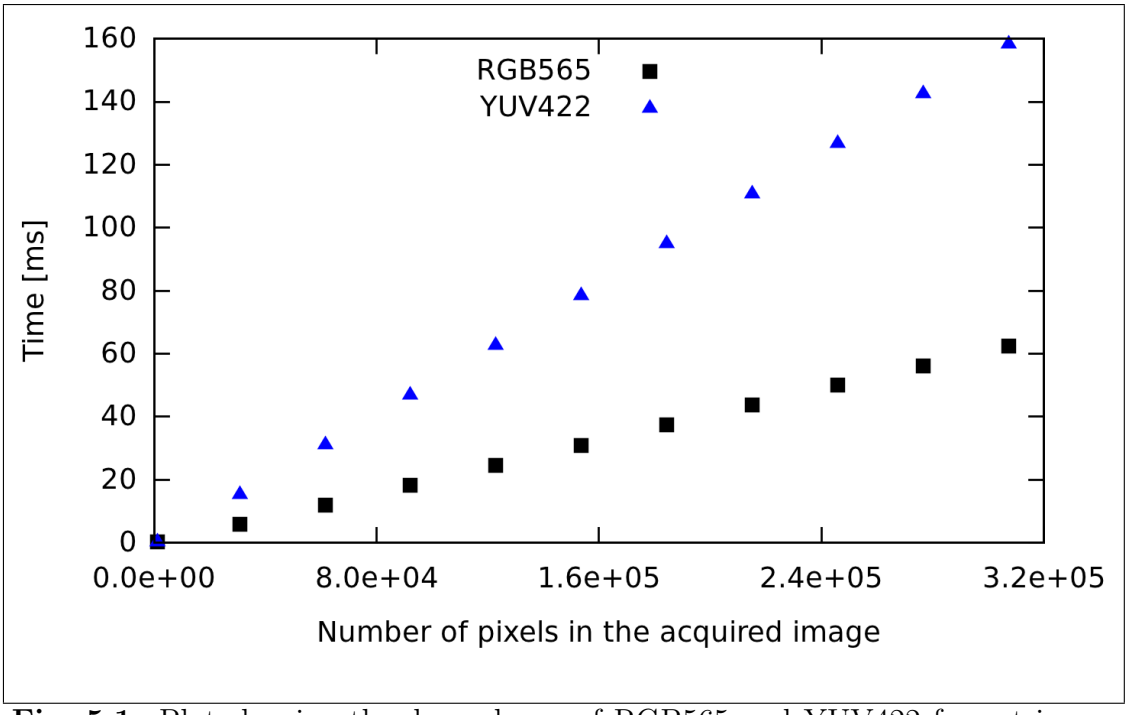


Fig. 5.1: Plot showing the dependency of RGB565 and YUV422 format image acquisition time with the number of pixels in the acquired image.

Therefore the images are acquired from the camera in the RGB565 format. Following step is to rotate the acquired image if necessary depending on the users choices. After this step an image pointer can be returned to the user or another step involving image compression and saving can be performed.

Table 5.2 shows times needed to both acquire the image and perform a rotation. Since the image is delivered with an offset of 180° it has to be canceled if the user does not want any rotation in the image. If however the user specifies an angle of rotation the actual angle for which the image will be rotated is calculated including the already existing offset angle, therefore avoiding double rotation. Every rotation angle introduces the same overhead because each rotation requires exactly one pass through the image buffer. Each time value is an average of 7 independent measurements and each measurement is an average of 100 image acquisitions.

Table 5.2: Image acquisition and rotation times for different resolutions

Number of pixels	**Time [ms]**	**Max. offset [%]**
307'200	151.2	0.02
276'480	136.4	0.02
245'760	121.3	0.02
215'040	106.1	0.01
184'320	90.3	0.40
153'600	74.7	0.60
122'800	60.1	0.04
92'160	45.4	0.04
61'440	30.8	0.08
30'720	16.2	0.11
1'024	2.4	0.3

Comparing results from table 5.1 specifically those from the RGB565 times column (representing times of image acquisition and conversion to RGB888) and results from table 5.2 where times needed for acquiring an image, convert it to RGB888 and rotate it are shown, significant overhead introduced by the rotation operation can be noticed. This is caused by doing another pass through the image buffer to reorganize its pixels.

Figure 5.2 plots the times needed to acquire the image and convert it to RGB888 vs. the times needed to acquire the image, convert and rotate it. This gives a graphical representation of the overhead introduced by the rotation operation.

To reduce the overhead of rotation operation instead of doing another pass to reorganize the pixels, a solution was implemented which performs this action during the conversion. When a pixel is converted from RGB565 to RGB888 it is immediately placed to its final position, one to which it is supposed to be on after rotation. This way both conversion and rotation are performed in one pass through the image buffer.

Table 5.3 shows the times needed to acquire, convert and rotate the image with the improved implementation. All times are an average of 7 independent measurements and each measurement is an average of 100 image acquisitions.

Figure 5.3 compares the times needed to acquire the image, convert and rotate it before and after the optimization. This gives a graphical representation of the improvement introduced by the implemented optimization.

Comparing the results from tables 5.2 and 5.3 it can be seen that the implemented optimization reduced the total time needed to acquire an image up to about 50% depending on the number of pixels in the image. This is presented in table 5.4.

After acquisition, conversion and rotation have been performed an image buffer is obtained whose pointer can be provided to the user which can then do further processing

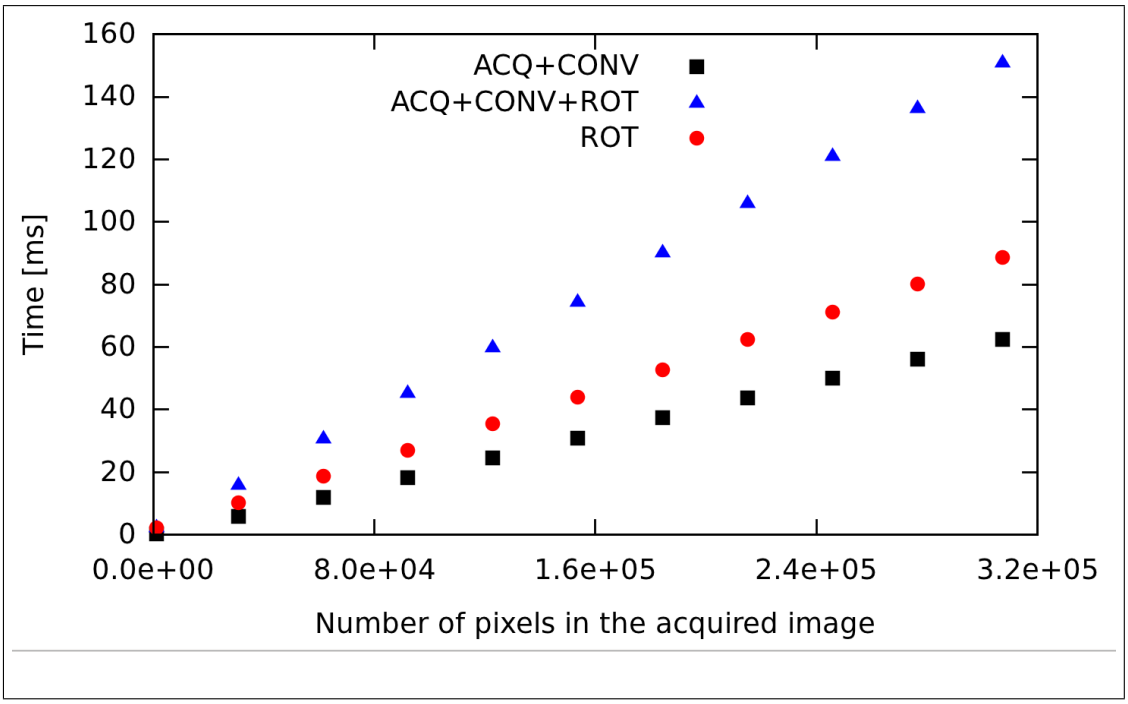


Fig. 5.2: Plot showing acquisition + conversion + rotation time (ACQ + CONV + ROT), acquisition + conversion time (ACQ + CONV) and just rotation time against number of pixels in the acquired image.

Table 5.3: Optimized image acquisition and rotation times

Number of pixels	**Time [ms]**	**Max. offset [%]**
307'200	80.3	0.02
276'480	72.6	0.04
245'760	63.7	0.02
215'040	56.5	0.02
184'320	48.3	0.03
153'600	37.9	0.03
122'800	30.7	0.06
92'160	23.4	0.1
61'440	15.9	0.06
30'720	8.6	0.12
1'024	2.3	4.4

on the image. By users choice instead of returning a pointer to the image it can be supplied to one of the compression algorithms, JPEG or PNG.

JPEG compression is implemented using an already available library which provides necessary functionalities for lossy compression. A quality parameter is available which determines the trade-off between the quality of the compressed image and its final size. This parameter can be set by the user of the library.

PNG compression is implemented also using an already available library which provides necessary functionalities for lossless compression.

Compression process reduces the size of the resulting image thus having to use less

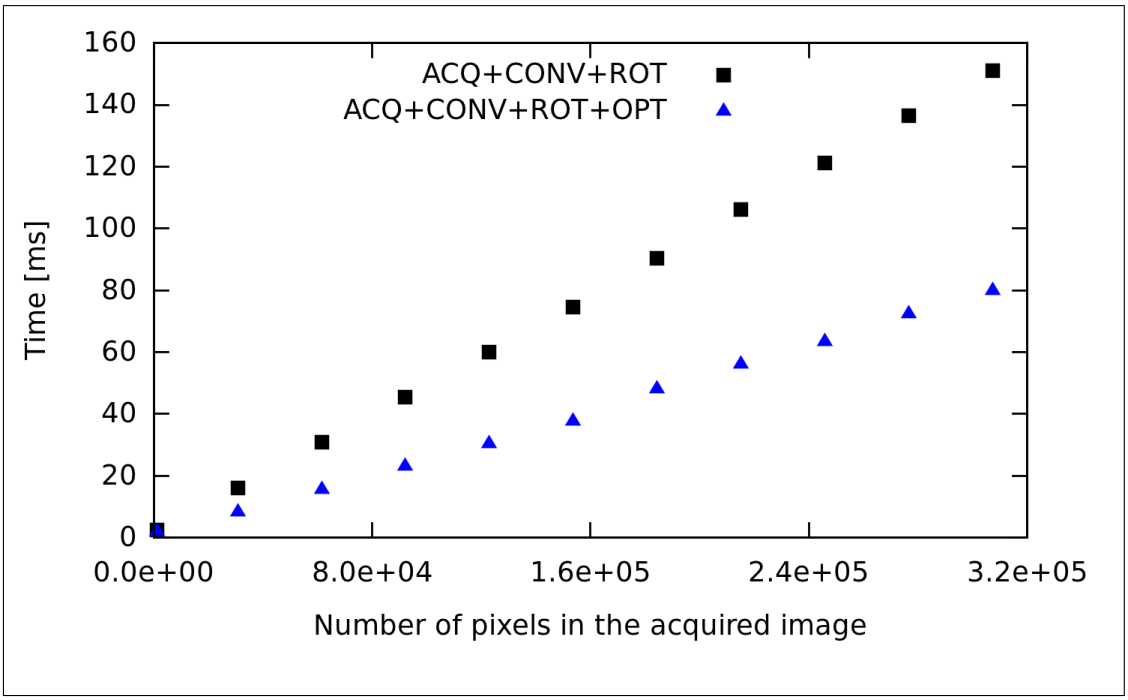


Fig. 5.3: Plot showing acquisition + conversion + rotation time (ACQ + CONV + ROT) and optimized acquisition + conversion + rotation time (ACQ + CONV + ROT + OPT) against number of pixels in the acquired image.

Table 5.4: Results from non-optimized and optimized implementation compared

Number of pixels	**Non-optimized time [ms]**	**Optimized time [ms]**	**Improvement [%]**
307'200	151.2	80.3	46.9
276'480	136.4	72.6	46.8
245'760	121.3	63.7	47.5
215'040	106.1	56.5	46.7
184'320	90.3	48.3	46.5
153'600	74.7	37.9	49.3
122'800	60.1	30.7	48.9
92'160	45.4	23.4	48.5
61'440	30.8	15.9	48.4
30'720	16.2	8.6	46.9
1'024	2.4	2.3	4.2

bandwidth when transmitting or less memory when storing. On the other hand the very process of compression introduces time overhead.

Tables 5.5 and 5.6 show the cumulative times necessary to acquire an image, convert it to RGB888, rotate and compress to either PNG and JPEG format, respectively. Each value is an average of 7 measurements and each measurement is an average of 100 image acquisitions.

It can be noticed that times for PNG format are quite higher than for JPEG. This is also influenced by the characteristics of the PNG format. It is useful to compress im-

Table 5.5: Image acquisition and PNG compression times

Number of pixels	**PNG time [ms]**
307'200	1461.0
276'480	1316.7
245'760	1141.9
215'040	916.4
184'320	781.5
153'600	671.2
122'800	532.1
92'160	388.7
61'440	274.3
30'720	141.1
1'024	7.0

Table 5.6: Image acquisition and JPEG compression times

Number of pixels	**JPEG time [ms], quality parameter $10 - 100$**									
	10	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
307'200	184.8	185.7	186.4	187.0	187.5	188.6	190.0	192.3	198.8	247.5
276'480	165.8	166.7	167.5	168.0	168.5	169.6	170.9	173.1	179.4	223.8
245'760	147.7	148.2	149.0	149.5	149.8	150.5	151.6	153.5	158.6	198.3
215'040	128.7	128.9	129.7	130.0	130.4	131.0	131.7	133.8	138.5	173.6
184'320	110.9	111.4	111.7	112.0	112.2	112.9	113.6	115.2	119.2	149.4
153'600	90.9	91.4	91.7	91.9	92.1	92.6	93.3	94.4	97.8	123.1
122'800	72.0	72.2	72.6	72.9	73.0	73.4	73.8	74.8	77.5	97.7
92'160	54.4	54.6	54.7	54.9	55.0	55.4	55.8	56.4	58.4	73.1
61'440	36.9	37.1	37.2	37.3	37.4	37.6	38.0	38.5	39.8	49.5
30'720	19.9	20.0	20.1	20.2	20.2	20.3	20.5	20.7	21.5	26.4
1'024	4.1	4.1	4.1	4.1	4.1	4.1	4.1	4.1	4.1	4.2

ages with sharp transitions such as black and white text while the tests were performed acquiring images of the scenery, which inherently has much milder transitions. Surprisingly JPEG times are almost constant in a wide range of used values for the `quality` parameter. This allows for more freedom when choosing the value of the parameter since higher quality can be achieved with not big of an increase in acquisition time.

Figures 5.4 and 5.5 graphically show the results of image acquisition with PNG and JPEG compression, respectively. Results show that the total time for acquisition and compression changes linearly with the number of pixels, while for the JPEG compression times are changing only slightly for quality parameter between $10 - 80$.

Another important parameter is the size of the compressed image. This impacts the necessary bandwidth for image transmission as well as storage memory. Tables 5.7 and 5.8 show the sizes of the PNG and JPEG compressed images depending on the number of pixels and `quality` parameter for JPEG, respectively. Each value from the tables is an average of 1000 compressed images.

Figures 5.6 and 5.7 graphically show PNG and JPEG compressed image sizes.

Figure 5.8 shows examples of images acquired with the camera on the e-puck robot

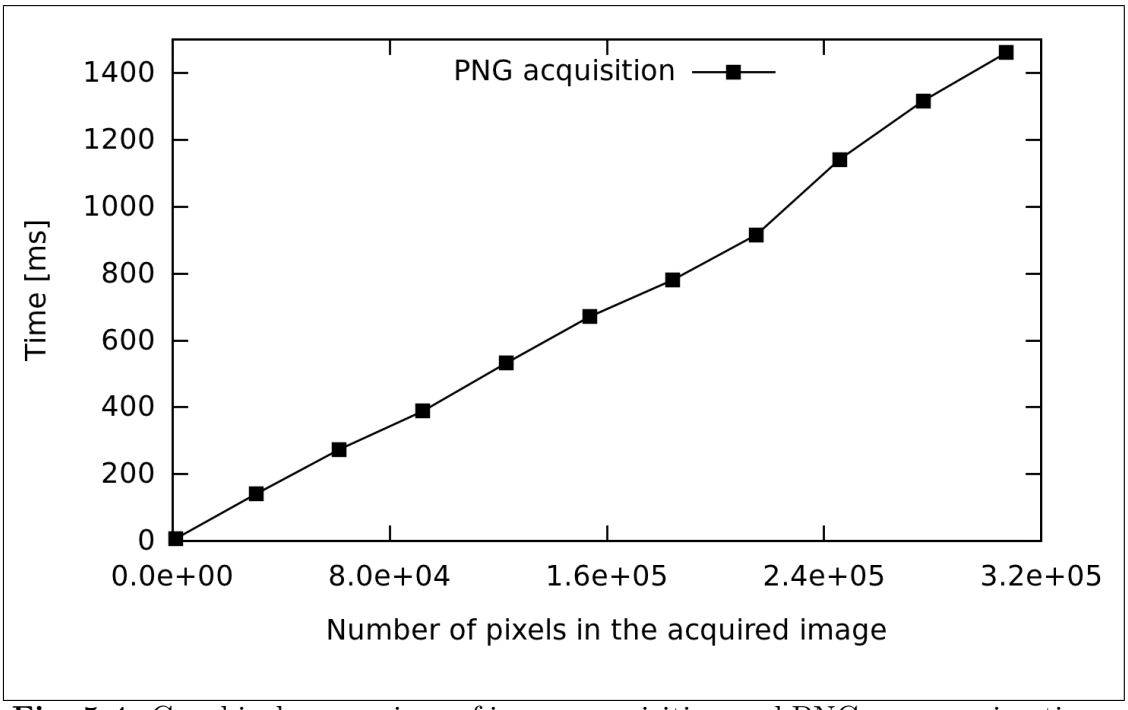


Fig. 5.4: Graphical comparison of image acquisition and PNG compression times against number of pixel

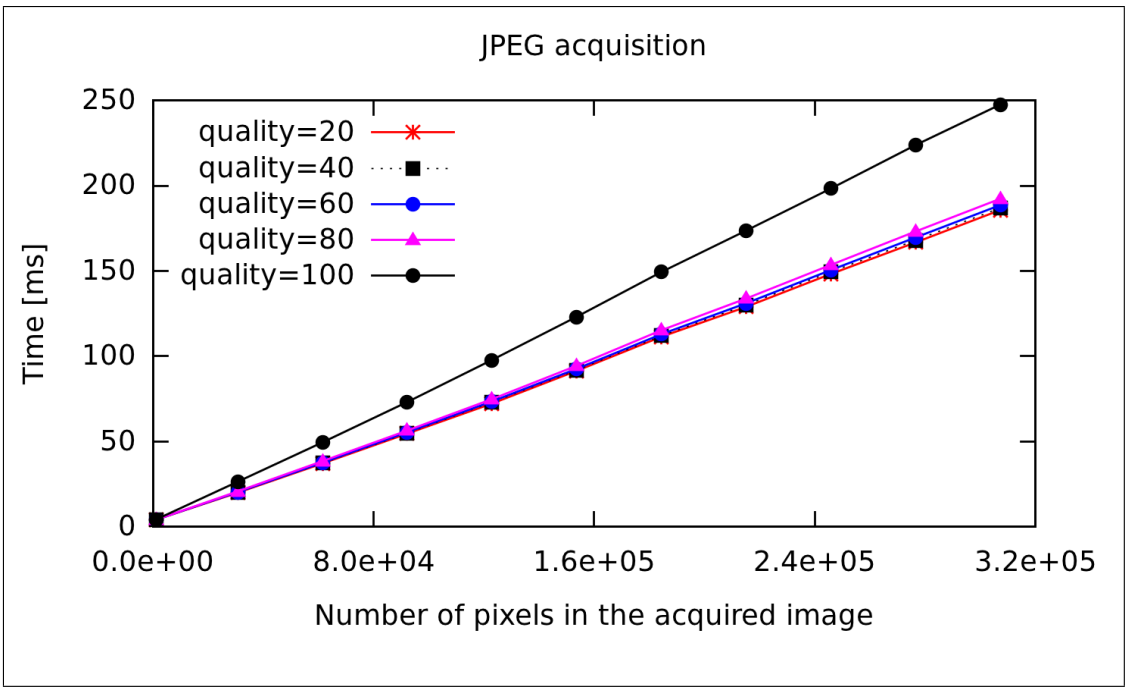


Fig. 5.5: Graphical comparison of image acquisition and JPEG compression times for different values of the `quality` parameter.

and compressed to JPEG format with various values of the `quality` parameter. The images were acquired in maximum available resolution for the camera (640x480). From the visual appearance point of view, quality of 40 is satisfactory. For larger value of the `quality` parameter there is no visually noticeable improvement.

Table 5.7: PNG compressed image sizes

Number of pixels	**Image size [kB]**
307'200	167.9
276'480	159.1
245'760	141.9
215'040	139.8
184'320	117.6
153'600	97.6
122'800	81.9
92'160	60.1
61'440	41.8
30'720	22.0
1'024	0.6

Table 5.8: JPEG compressed image sizes

Number of pixels	**Image size [kB], quality parameter 10 − 100**									
	10	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
307'200	7.3	9.2	10.8	12.4	14.3	15.5	19.6	24.7	40.9	185.3
276'480	6.7	8.4	10.3	11.7	13.4	14.8	18.2	24.7	40.1	172.2
245'760	5.9	7.4	8.7	10.1	11.4	12.8	15.7	20.7	34.5	150.8
215'040	5.4	7.4	8.8	9.5	11.8	13.4	16.9	21.5	35.2	142.5
184'320	5.1	6.6	7.6	8.7	10.4	11.9	14.5	18.7	30.0	122.0
153'600	4.1	5.5	6.5	7.1	8.2	9.5	11.7	15.2	24.5	101.4
122'800	3.5	4.6	5.4	6.4	7.4	8.2	9.8	13.1	20.7	82.6
92'160	2.9	3.7	4.3	5.0	5.6	6.5	7.6	9.8	15.6	63.0
61'440	2.0	2.4	2.8	3.1	3.5	3.9	4.6	5.9	9.6	39.7
30'720	1.4	1.8	2.1	2.3	2.4	2.5	2.7	3.6	5.7	21.5
1'024	0.6	0.6	0.6	0.6	0.6	0.6	0.7	0.7	0.7	1.2

5.2 Sensors and actuators communication

This section presents timing results for a worst case scenario when maximum amount of data is transmitted through the serial bus at one time step. This result impacts the minimum value of the time-step for which the controller behaves correctly.

Testing conditions were: maximum amount of transmitted data (124 bytes in each time step), 5 different runs and 1000 time steps each run. Result obtained is $\mathbf{53.5}\boldsymbol{ms}$ as the time needed to perform all of the operation of updating sensors and actuators data. This is the maximum time that the operations of updating and actuating sensors (camera excluded) can take. It should be noted that if a time step less than $\mathbf{53.5}\boldsymbol{ms}$ is specified the behaviour of the controller will be functionally correct but the specified timing will not be honoured.

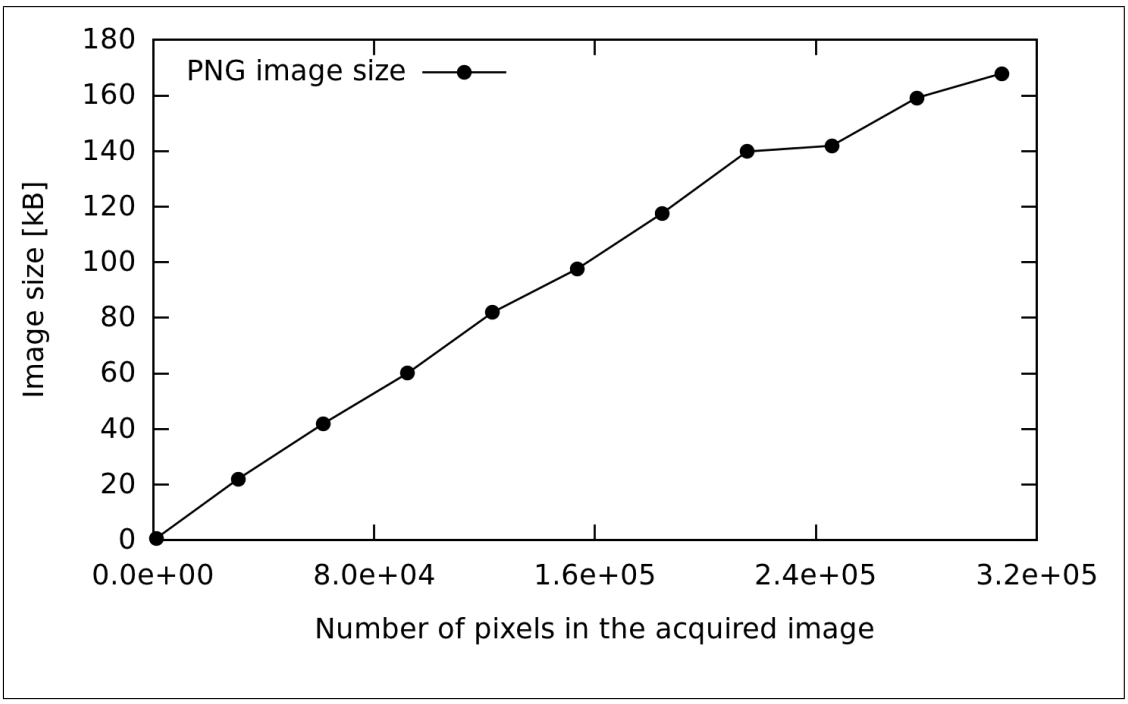


Fig. 5.6: PNG compressed image sizes against number of pixels

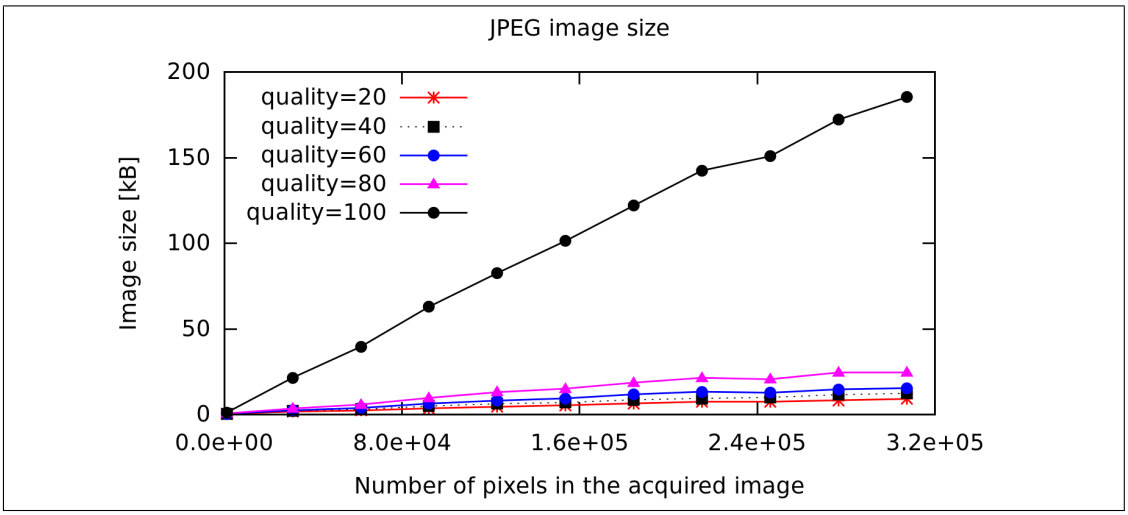
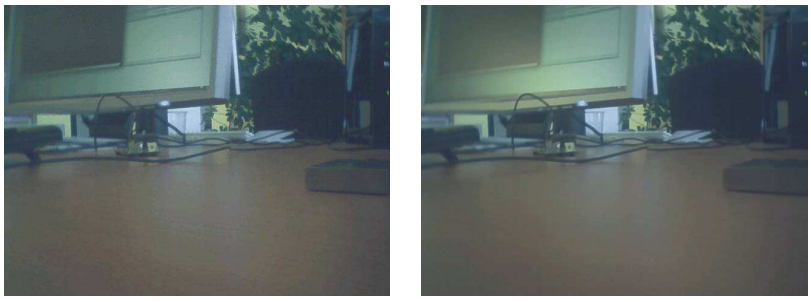


Fig. 5.7: JPEG compressed image sizes against number of pixels, given for different values of the `quality` parameter.

(a) quality = 10 (b) quality = 20

(c) quality = 30 (d) quality = 40

(e) quality = 50 (f) quality = 60



(g) quality = 70 (h) quality = 80

(i) quality = 90 (j) quality = 100

Fig. 5.8: JPEG compressed images

Chapter 6

Conclusion

Almost all of the project goals were achieved. Mainly a modular, Webots API compatible software library which provides support for the extended e-puck robot in Webots software is developed. Hardware constraints of the system were taken into account during development. Performance of the software library was measured also. Optimizations of some modules, mainly camera module have been done.

Nevertheless, there is always room for improvement. As far as image acquisition is concerned, due to the constraints from the Webots API, conversion to RGB888 format always occurs introducing overhead in image size and in processing time. If the constraints are relaxed this overhead could be avoided.

Another image acquisition optimization can be the following. Instead of acquiring the image in the wanted resolution WxH, acquire it with the resolution $(\frac{W}{2}$x$\frac{H}{2})$ and then apply resizing algorithm to achieve wanted resolution. The idea here is to reduce the time of acquisition, in this case approximately 4x (because acquired image has 4x less pixels and acquisition time changes linearly with number of pixels in image). Application of a resizing algorithm introduces some overhead, but it should be investigated how much overhead is that, and whether the quality of obtained image is satisfiable.

The implemented solutions in this project provide a base which can be helpful to all of those using the e-puck robot with the Gumstix extension and open new possibilities to develop interesting applications for it.

Future work regarding this project could go in various directions. The support for handling sound sensors and audio processing is not very evolved. Improving this support can be one future direction of the project.

With the Gumstix extension a lot more processing power and memory is available. This allows for much better image processing capabilities. Characterizing the capabilities of the developed software library for image processing can be another future direction of the project. The way this characterization could be done is by implementing various image processing algorithms using the E-GO software library.

Bibliography

[1] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J. christophe Zufferey, D. Floreano, and A. Martinoli, "The e-puck, a robot designed for education in engineering," in *In Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, 2009, pp. 59–65.

[2] "E-puck website." [Online]. Available: http://www.e-puck.org

[3] F. Mondada, E. Franzi, and P. Ienne, "Mobile robot miniaturization: A tool for investigation in control algorithms." 1994.

[4] O. Michel, "Webots: Professional mobile robot simulation," *Journal of Advanced Robotics Systems*, vol. 1, no. 1, pp. 39–42, 2004. [Online]. Available: http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf

[5] Webots, "http://www.cyberbotics.com," commercial Mobile Robot Simulation Software. [Online]. Available: http://www.cyberbotics.com

[6] "Webots User Guide," Cyberbotics Ltd. [Online]. Available: http://www.cyberbotics.com/guide/

[7] "Webots Reference Manual," Cyberbotics Ltd. [Online]. Available: http://www.cyberbotics.com/reference/

[8] "dsPIC30F6011A/6012A/6013A/6014A data sheet," Microchip Technology Inc.

[9] "1.5g - 6g Three Axis Low-g Micromachined Accelerometer data sheet," Freescale Semiconductor, Freescale Semiconductor Technical Information Center, CH370 1300 N. Alma School Road Chandler, Arizona 85224.

[10] "Reflective Optical Sensor with Transistor Output, TCRT1000 data sheet," Vishay Semiconductors.

[11] "Zero Height "Ultra-Mini" SiSonic^{tm} Microphone Specification With MaxRF Protection - Halogen Free data sheet," Knowles Acoustics.

[12] "PO6030K CMOS camera data sheet," Pixel Plus Co., LTD., 6th Floor, Gyeonggi RDB Center, 906-5 Iui-dong, Yeongtong-gu, Suwon-si, Gyeonggi-do, 443-766, Korea.

[13] "PG15S-020 stepper motor data sheet," Minebea Motor Manufacturing Corporation.

[14] "LMX9820A Bluetooth Serial Port Module data sheet," Texas Instruments Inc.

[15] "OMAP 3515/03 Applications Processor data sheet," Texas Instruments Inc., Post Office Box 655303, Dallas, Texas 75265.

[16] "Reflective Optical Sensor with Transistor Output, TCRT5000 data sheet," Vishay Semiconductors.

[17] "Linux Media Infrastructure API," LinuxTV Developers. [Online]. Available: http://linuxtv.org/downloads/v4l-dvb-apis/index.html

Appendix A

Hardware

This section contains details which are not crucial for understanding the topics elaborated in this report, but provide a more complete picture of them. All of the details mentioned in this section can be found in the relevant documentation provided in the references, but are also condensed here for the sake of completeness.

A.1 Accelerometer

An example of a measurement taken with the accelerometer is given here. The robot moves in steps and at each step there is a strong micro-acceleration and micro-deceleration occurring explaining the acceleration pattern visible in figure A.1. Figure shows the accelerations on all 3 axis in a situation where the stepper motors are moving at a rate of $70\frac{steps}{s}$ and the sampling is done at a rate of $7kHz$. Y-axis shows the digital equivalent of the acceleration after AD conversion. For a normalization to a physical quantity we have to use the following equation:

$$a = (a\_dig - 2000) * 0.0123[\frac{m}{s^2}] \tag{A.1}$$

where: a_dig is the Y-axis value.

X-axis shows the sample number. Considering the sample frequency of $7kHz$ the 1000 samples shown in figure A.1 correspond to about $0.143s$.

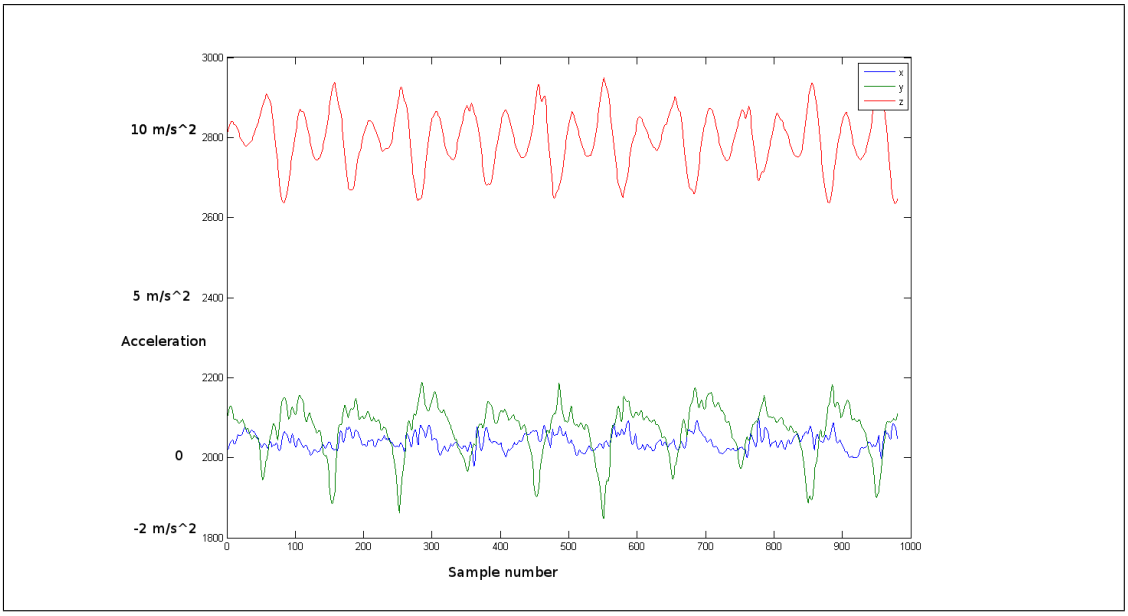


Fig. A.1: Example of acceleration measurement. Accelerations for all 3 axis of the coordinate system are shown with the orientation as in figure 2.3(b). Y-axis of the figure represents a digital equivalent of the acceleration after AD conversion. Scale for reading acceleration as a physical quantity is also given (in bold face). X-axis represents sample numbers. Sampling rate is $7kHz$ and stepper motor speed is $70\frac{steps}{s}$.

A.2 Long range proximity sensors

This device is consisted of a photo-transistor sensitive to infra red light and an emitter which is emitting light with a wavelength of $950nm$. When active, power consumption of photo-transistor is about $0.1W$ and $0.2W$ when emitter is also active. Figure A.2 illustrates the long range proximity sensor's response towards a white surface.

Figure A.3 shows a zoomed area of the characterization chart in the area of higher distance from the wall. As it can be seen the change in the values returned by the sensor are still usable to react to obstacles.

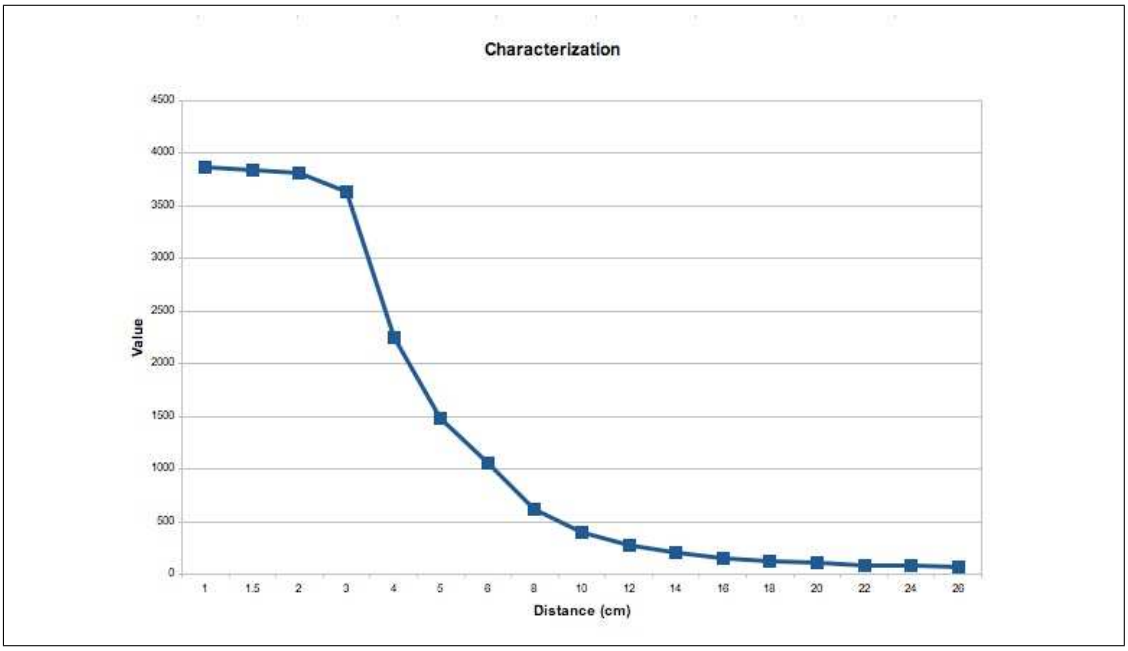


Fig. A.2: Characterization of the long range proximity sensor. Y-axis shows a digital equivalent value of the sensor's output which can be in the range of $0 - 4095$. X-axis shows the distance from the wall.

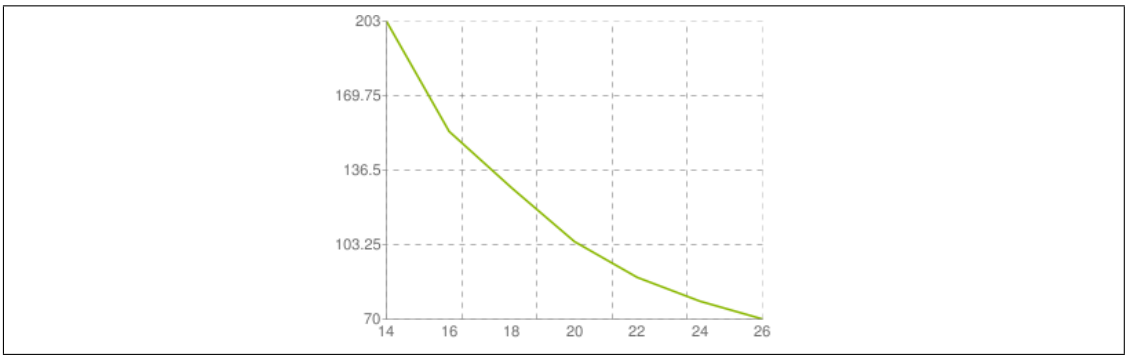


Fig. A.3: Shows the area of higher distances from the wall. Y-axis shows a digital equivalent value of the sensor's output which can be in the range of $0 - 4095$. X-axis shows the distance from the wall.

Appendix B

Webots API

Material presented here for the sake of completeness may also be found in the Webots reference manual [7]. The API functions presented here are only those implemented in the E-GO library. The full API includes much more functions and the details can be found in Webots reference manual.

B.1 Accelerometer functions

```
#include <webots/accelerometer.h>

void wb_accelerometer_enable (WbDeviceTag tag, int ms);

void wb_accelerometer_disable (WbDeviceTag tag);

const double *wb_accelerometer_get_values (WbDeviceTag tag);
```

Function `wb_accelerometer_enable()` allows the user to enable acceleration measurement each `ms` milliseconds.

Function `wb_accelerometer_disable()` turns off acquiring acceleration data that in turn saves computation time.

Function `wb_accelerometer_get_values()` returns the current values measured by the accelerometer.

B.2 Camera functions

```
#include <webots/camera.h>

void wb_camera_enable (WbDeviceTag tag, int ms);

void wb_camera_disable (WbDeviceTag tag);

double wb_camera_get_fov (WbDeviceTag tag);

void wb_camera_set_fov (WbDeviceTag tag, double fov);

int wb_camera_get_width (WbdeviceTag tag);

int wb_camera_get_height (WbdeviceTag tag);

const unsigned char *wb_camera_get_image (WbDeviceTag tag);

int wb_camera_save_image (WbDeviceTag tag, const char *filename, int quality);

unsigned char wb_camera_image_get_red (const unsigned char *image, int width,
int x, int y);

unsigned char wb_camera_image_get_green (const unsigned char *image, int width,
int x, int y);

unsigned char wb_camera_image_get_blue (const unsigned char *image, int width,
int x, int y);

unsigned char wb_camera_image_get_grey (const unsigned char *image, int width,
int x, int y);
```

Function `wb_camera_enable()` allows the user to enable a camera update each `ms` milliseconds.

Function `wb_camera_disable()` turns off camera saving computation time.

Function `wb_camera_get_fov()` returns the camera field of view (fov).

Function `wb_camera_set_fov()` sets the camera field of view (fov).

Functions `wb_camera_get_width()`, `wb_camera_get_height()` return the width and height of the camera image, respectively.

Function `wb_camera_get_image()` read the last image grabbed by the camera and returns a pointer to it. Image is coded in RGB888 format, which means as a sequence of three bytes representing the red, green and blue levels of a pixel.

Functions `wb_camera_get_red()`, `wb_camera_get_green()`, `wb_camera_get_blue()` and `wb_camera_get_grey()` return the red, green, blue and grey components of a specified pixel, respectively.

Function `wb_camera_save_image()` allows the user to save an image obtained by the camera of a `tag` device in either PNG or JPEG format which is specified by the `filename` extension. The `quality` parameter $(1 - 100)$ defines the quality of the saved JPEG image and has no influence on images saved in PNG format.

B.3 Differental wheels functions

```
#include <webots/differential_wheels.h>

void wb_differential_wheels_set_speed (double left, double right);

void wb_differential_wheel_enable_encoders (int ms);

void wb_differential_wheels_disable_encoders();

double wb_differential_wheels_get_left_encoder();

double wb_differential_wheels_get_right_encoder();

void wb_differential_wheels_set_encoders (double left, double right);
```

Function `wb_differential_wheels_set_speed()` allows the user to specify the speed of the robot.

Functions `wb_differential_wheels_enable_encoders()` and `wb_differential_wheels_disable_encoders()` allow the user to enable or disable the incremental wheel encoders for both wheels.

Functions `wb_differential_wheels_get_left_encoder()`, and `wb_differential_wheels_get_right_encoder()` return the values of the left and right encoder, respectively.

Function `wb_differential_wheels_set_encoders()` allows the user to set values for the encoders.

B.4 Distance sensor functions

```
#include <webots/distance_sensor.h>

void wb_distance_sensor_enable(WbDeviceTag tag, int ms);

void wb_distance_sensor_disable(WbDeviceTag tag);

double wb_distance_sensor_get_value(WbDeviceTag tag);
```

Function `wb_distance_sensor_enable()` allows the user to enable distance sensor measurement each `ms` milliseconds.

Function `wb_distance_sensor_disable()` turns distance sensor off, saving computation time.

Function `wb_distance_sensor_get_value()` returns the last value measured by the specified distance sensor.

B.5 LED functions

```
#include <webots/led.h>

void wb_led_set (WbDeviceTag tag, int value);
```

Function `wb_led_set()` switches an LED on or off, possibly changing its colour. If the `value` parameter is 0, the LED is turned off. Otherwise it is turned on.

B.6 Light sensor functions

```
#include <webots/light_sensor.h>

void wb_light_sensor_enable (WbDeviceTag tag, int ms);

void wb_light_sensor_disable (WbDeviceTag tag);

double wb_light_sensor_get_value (WbDeviceTag tag);
```

Function `wb_light_sensor_enable()` enables a light sensor measurement each `ms` milliseconds.

Function `wb_light_sensor_disable()` turns off the light sensors and saves CPU time.

Function `wb_light_sensor_get_value()` returns the most recent value measured by the specified light sensor.

Appendix C

E-GO software library

Details of the implementation of the developed software library are presented here. These details include functions signatures and description of actions caused by them.

C.1 Accelerometer

```
#include <overo/accelerometer.h>

int accelerometer_enable;
int accelerometer_raw;
int accelerometer_spherical;
int accelerometer_ms;
double accelerometer_last_value[];
int offset_accelerometer_raw;
int offset_accelerometer_spherical;

int overo_accelerometer_get_offset_raw(void);

void overo_accelerometer_set_offset_raw(int offset_raw);

int overo_accelerometer_get_offset_spherical(void);

void overo_accelerometer_set_offset_spherical(int offset_spherical);

void overo_accelerometer_enable_raw(DeviceTag dt, int ms);

void overo_accelerometer_enable_spherical(DeviceTag dt, int ms);

void overo_accelerometer_command_get_values_raw(void);

void overo_accelerometer_command_get_values_spherical(void);

void overo_accelerometer_update_state(void);

double* overo_accelerometer_get_value(void);
```

```
void overo_accelerometer_disable(deviceTag dt);
```

`accelerometer_enable` - variable describing the state of the accelerometer. 0 indicates the sensor is disabled and 1 indicates the sensor is enabled.

`accelerometer_raw` - variable indicating whether data is supplied as acceleration of each Decart's coordinate system axis a_x, a_y, a_z (raw format). 1 indicates data are supplied this way, 0 indicates this format is not used.

`accelerometer_spherical` - variable indicating whether data is supplied as acceleration in the format of a spherical coordinate system[1] a_r, a_θ, a_ϕ. 1 indicates data are supplied this way, 0 indicates this format is not used.

`accelerometer_ms` - variable holding the refresh period of the sensor. It defines the minimum time between two samplings of the sensor.

`accelerometer_last_value[]` - variable holding the last sampled value of the sensor. It has 3 members each holding acceleration for one axis.

`offset_accelerometer_raw` - variable holding the position of the first byte of acceleration data in the buffer received as a response to issued commands when received in Decart's coordinate system format.

`offset_accelerometer_spherical` - variable holding the position of the first byte of acceleration data in the buffer received as a response to issued commands when received in spherical coordinate system format.

`overo_accelerometer_get_offset_raw()` - returns the value of the `offset_accelerometer_raw` variable.

`overo_accelerometer_set_offset_raw()` - sets the value of the `offset_accelerometer_raw` variable.

`overo_accelerometer_get_offset_spherical()` - returns the value of the `offset_accelerometer_spherical` variable.

`overo_accelerometer_set_offset_spherical()` - sets the value of the `offset_accelerometer_spherical` variable.

`overo_accelerometer_enable_raw()` - enables accelerometer sampling in raw format with a device tag `dt` and a period of sampling `ms`.

`overo_accelerometer_enable_spherical()` - enables accelerometer sampling in spherical format with a device tag `dt` and a period of sampling `ms`.

`overo_accelerometer_command_get_values_raw()` - issues a command to acquire a sample of acceleration data in raw format from the sensor.

`overo_accelerometer_command_get_values_spherical()` - issues a command to acquire a sample of acceleration data in spherical format from the sensor.

`overo_accelerometer_update_state()` -updates the `accelerometer_last_value` array with new data.

`overo_accelerometer_get_value()` - returns a pointer to the `accelerometer_last_value` array.

`overo_accelerometer_disable()` - turns off sampling of the accelerometer and resetting state variables. This in turn saves computing time and serial bus bandwidth.

[1] not fully implemented in the current version of the software

C.2 Camera

Camera of the e-puck robot has a direct connection with the extension. Driver for the camera conforms to the V4L2 specification and its API. Maximum resolution of the camera is 640x480 pixels and minimum 32x32 pixels.

```
#include <overo/camera.h>

int camera_enable;
int camera_ms;
unsigned int camera_image_width;
unsigned int camera_image_height;
int camera_brightness;
int camera_contrast;
int camera_saturation;
int camera_auto_white_balance;
const int nb_of_camera_parameters;
double camera_field_of_view;

static void overo_camera_extract_parameters_from_configuration_
file(void);

void overo_camera_init(void);

void overo_camera_enable(DeviceTag dt, int ms);

const unsigned char* overo_camera_get_image(DeviceTag dt);

int overo_camera_get_width(DeviceTag dt);

void overo_camera_set_width(DeviceTag dt, int width);

int overo_camera_get_height(DeviceTag dt);

void overo_camera_set_height(DeviceTag dt, int height);

int overo_camera_get_fov(DeviceTag dt);

void overo_camera_set_fov(DeviceTag dt, int width);

void overo_camera_brigthness_control(int brightness_value);

void overo_camera_contrast_control(int contrast_value);

void overo_camera_saturation_control(int saturation_value);

int overo_camera_save_image(deviceTag dt, const char* filename, int quality);

unsigned char overo_camera_get_red(DeviceTag dt, const unsigned char *image,
int width, int x, int y);
```

```
unsigned char overo_camera_get_green(DeviceTag dt, const unsigned char *image,
int width, int x, int y);

unsigned char overo_camera_get_blue(DeviceTag dt, const unsigned char *image,
int width, int x, int y);

unsigned char overo_camera_get_grey(DeviceTag dt, const unsigned char *image,
int width, int x, int y);
```

`camera_enable` - variable describing the state of the accelerometer. 0 indicates the sensor is disabled and 1 indicates the sensor is enabled.

`camera_ms` - variable holding the refresh period of the sensor. It defines the minimum time between two samplings of the sensor.

`camera_image_width` - variable holding the `width` of the acquired images. Default is 640, the maximum possible value for the camera.

`camera_image_height` - variable holding the `height` of the acquired images. Default is 480, the maximum possible value for the camera.

`camera_brightness` - variable holding the `brightness` parameter of the acquired images. Default is 1.

`camera_contrast` - variable holding the `contrast` parameter of the acquired images. Default is 64.

`camera_saturation` - variable holding the `saturation` parameter of the acquired images. Default is 64.

`camera_auto_white_balance` - variable holding the `auto_white_balance` flag. Default is 0.

`nb_of_camera_parameters` - Image parameters to be set to the camera such as resolution, brightness, contrast, saturation and auto white balance are loaded from a configuration file. This variable holds the number of these paramters to be loaded from the file.

`camera_field_of_view` - variable holding the field of view parameter of the camera.

`overo_camera_extract_parameters_from_configuration_file()` - extracts parameters from a configuration file and load the values to the corresponding variables.

`overo_camera_init()` - handles initiations for the camera device.

`overo_camera_enable()` - enables image acquisition from the camera for a device with tag `dt` and with a period of sampling `ms`.

`overo_camera_get_image()` -returns a pointer to an image buffer which was last acquired with the camera. Image is represented in RGB888 format, which means that one pixel is represented with three bytes of data. Each of the red, green and blue components is represented with one byte.

`overo_camera_get_width()` -returns the value of `camera_image_width` variable of a device with tag `dt`.

`overo_camera_set_width()` -sets the value of `camera_image_width` variable of a device with tag `dt`.

`overo_camera_get_height()` - returns the value of `camera_image_height` variable of a device with tag `dt`.

`overo_camera_set_width()` -sets the value of `camera_image_height` variable of a device with tag `dt`.

`overo_camera_get_fov()` - returns the value of `field of view`.
`overo_camera_set_fov()` -sets the `field of view` parameter of the camera.
`overo_camera_brightness_control()` - sets the value of `brightness` parameter.
`overo_camera_contrast_control()` -sets the value of `contrast` parameter.
`overo_camera_saturation_control()` -sets the value of `saturation` parameter.
`overo_camera_save_image()` - saves an image either in JPEG or PNG format, specified by the extension of the `filename` parameter. Quality of the JPEG compressed image is determined by the `quality` parameter and has no effect on PNG images.
`overo_camera_get_red()` - returns red component of the specified pixel.
`overo_camera_get_green()` - returns green component of the specified pixel.
`overo_camera_get_blue()` - returns blue component of the specified pixel.
`overo_camera_get_grey()` - returns grey component of the specified pixel.

C.3 Distance sensors

Infra red proximity and floor sensors fall into the group of distance sensors. E-puck roboot is equipped with 8 proximity sensors and optionally with 3 floor sensors.

```
#include <overo/distance_sensors.h>

static int offset_floor_sensors = 0;
static int offset_proximity_sensors = 0;

static int proximity_sensors_enable[] = {0, 0, 0, 0, 0, 0, 0, 0};
static int proximity_sensors_ms[] = {0, 0, 0, 0, 0, 0, 0, 0};
static double proximity_sensors_last_value[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0};
static int proximity_sensors_trigger[] = {0, 0, 0, 0, 0, 0, 0, 0};

static int floor_sensors_enable[] = {0, 0, 0};
static int floor_sensors_ms[] = {0, 0, 0};
static double floor_sensors_last_value[] = {0.0, 0.0, 0.0};
static int floor_sensors_trigger[] = {0, 0, 0};

int overo_distance_sensors_get_offset_proximity(void);

void overo_distance_sensors_set_offset_proximity(int offset_proximity);

int overo_distance_sensors_get_offset_floor(void);

void overo_distance_sensors_set_offset_floor(int offset_floor);

void overo_distance_sensors_enable(DeviceTag dt, int ms);

void overo_floor_sensors_command_get_values(void);

void overo_proximity_sensors_command_get_values(void);

void overo_distance_sensors_update_state(void);

double overo_distance_sensors_get_value(DeviceTag dt);

void overo_distance_sensors_disable(DeviceTag dt);
```

`proximity_sensors_enable` - array variable describing the state of each sensor. 0 indicates sensor sampling is off and 1 indicates sampling is on.

`proximity_sensors_ms` - array variable holding sampling periods of all proximity sensors.

`proximity_sensors_last_value` - array variable holding the last acquired data from the sensors.

`proximity_sensors_trigger` - array variable holding flags which signal if a sensor needs to be sampled (1) or not (0).

`floor_sensors_enable` - array variable describing the state of each sensor. 0 indicates sensor sampling is off and 1 indicates sampling is on.

`floor_sensors_ms` - array variable holding sampling periods of all proximity sensors.

`floor_sensors_last_value` - array variable holding the last acquired data from the sensors.

`floor_sensors_trigger` - array variable holding flags which signal if a sensor needs to be sampled (1) or not (0).

`offset_proximity_sensors` - variable holding the position of the first byte of proximity sensors data in the buffer received as a response to issued commands.

`offset_floor_sensors` - variable holding the position of the first byte of proximity sensors data in the buffer received as a response to issued commands.

`overo_distance_sensors_get_offset_proximity()` - returns `offset_proximity_sensors` value;

`overo_distance_sensors_set_offset_proximity()` - sets `offset_proximity_sensors` value;

`overo_distance_sensors_get_offset_floor()` - returns `offset_floor_sensors` value;

`overo_distance_sensors_set_offset_floor()` - sets `offset_floor_sensors` value;

`overo_distance_sensors_enable()` - enables sampling of distance sensors with tag `dt` and with a period of sampling `ms`.

`overo_proximity_sensors_command_get_values()` - fills the command buffer with a command for acquiring a sample from the proximity sensors.

`overo_floor_sensors_command_get_values()` - fills the command buffer with a command for acquiring a sample from the floor sensors.

`overo_distance_sensors_update_state()` - updates variables `proximity_sensors_last_value` and `floor_sensors_last_values` with the latest acquired sample.

`overo_distance_sensors_get_value()` - returns the value of a specified distance sensor device held by either `proximity_sensors_last_values` or `floor_sensors_last_values`.

`overo_distance_sensors_disable()` - disables sampling of the specified distance sensor, saving computation time and serial bus bandwidth.

C.4 Encoders

Event though there are no physical encoders on the e-puck robot, there are low level library functions which count the steps of the stepper motors from the start of their operation. This emulates incremental encoders on left and right motor. Current values of these counters can be set and read by the user.

```
#include <overo/encoders.h>

static int offset_encoder = 0;

static int encoder_enable = 0;
static int encoder_ms = 0;
static double encoder_last_value[] = {0.0, 0.0}; // {left encoder, right encoder}
static int encoder_set_value[] = {0, 0}; {left encoder,
right encoder}
static int encoder_trigger = 0;

int overo_encoders_get_offset(void);

void overo_encoders_set_offset(int offset);

void overo_encoders_enable(DeviceTag dt, int ms);

void overo_encoders_command_set_values(void);

void overo_encoders_command_get_values(void);

double overo_encoders_get_left(void);

double overo_encoders_get_right(void);

void overo_encoders_set(double left, double right);

void overo_encoders_update_state(void);

void overo_encoders_disable(DeviceTag dt);
```

`encoder_enable` - variable describing state of the sensor. 0 value means sensor is off, 1 means it is on.

`encoder_ms` - variable holding sampling period of the sensor.

`encoder_last_value` - array variable holding the last acquired data from the sensor.

`encoder_set_value` - array variable holding values to be set as starting points for the encoders.

`encoder_trigger` - variable serving as a flag to say when variable `encoder_set_value` was changed and encoders starting values have to be updated. Value 0 means no change and 1 means the variable has changed.

`overo_encoders_get_offset()` - returns `offset_encoder` value.

`overo_encoders_set_offset()` - sets `offset_encoder` value;

`overo_encoders_enable()` - enables sampling of encoders with a period of sampling `ms`.

`overo_encoders_command_get_values()` - fills the command buffer with a command for acquiring a sample from the encoders.

`overo_encoders_command_set_values()` - fills the command buffer with a command for setting new values to the encoders.

`overo_encoders_get_left()` - returns the value of left encoder.

`overo_encoders_get_right()` - returns the value of right encoder.

`overo_encoders_set()` - sets the values of `encoder_set_value` variable.

`overo_encoders_update_state()` - updates variable `encoder_last_value` with latest sample of sensor data.

`overo_encoders_disable()` - disables sampling of encoders, saving computation time and serial bus bandwidth.

C.5 LEDs

E-puck robot has 8 LEDs positioned in a ring around it, a body LED and a front LED. Gumstix extension has another 8 LEDs on it.

```
#include <overo/led.h>

// Enable by default. Currently 9th and 10th value have no meaning.
// In the future these will replace body and front LED variables.
static int led_enable[] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
static int led_trigger[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
static int led_ms[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
static int led_value[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
static int body_led_enable = 1; // Enable by default
static int body_led_trigger = 0;
static int body_led_ms = 0;
static int body_led_value = 0;
static int front_led_enable = 1; // Enable by default
static int front_led_trigger = 0;
static int front_led_ms = 0;
static int front_led_value = 0;

void overo_led_enable(DeviceTag dt, int ms);

void overo_led_command_set(void);

void overo_led_set(DeviceTag dt, int value);

void overo_led_disable(DeviceTag dt);
```

`led_enable`, `body_led_enable`, `front_led_enable` - variable describing the state of each LED. 0 indicates LED cannot be controlled, 1 indicates control of LED is enabled. This feature is currently not used.

`led_ms`, `body_led_ms`, `front_led_ms` - variable holding updating periods of all LEDs. This feature is currently not used.

`led_value`, `body_led_value`, `front_led_value` - variable holding LEDs state. 0 means LED is turned off and 1 means it is turned on.

`led_trigger`, `body_led_trigger`, `front_led_trigger` - variable holding flags which signal when a member of `led_value` has changed and the corresponding LED needs to be updated.

`overo_led_enable()` - enables control of LED with a device tag `dt` and updating period of `ms`. Currently actuators are always enabled and this function is not used.

`overo_led_command_set()` - issues a command to set the LED specified with `dt` with `value` value.

`overo_led_set()` - updates the state of variables `led_value`, `body_led_value` and `front_led_value`.

`overo_led_disable()` - disable the control of LEDs. Currently actuators are always enabled and this function is not used.

C.6 Light sensors

```
#include <overo/light_sensors.h>

static int offset_light_sensors = 0;
static int light_sensors_enable[] = {0,0,0,0,0,0,0,0};
static int light_sensors_ms[] = {0,0,0,0,0,0,0,0};
static double light_sensors_last_value[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0};
static int light_sensors_trigger[] = {0,0,0,0,0,0,0,0};

int overo_light_sensors_get_offset(void);

void overo_light_sensors_set_offset(int offset);

void overo_light_sensors_enable(DeviceTag dt, int ms);

void overo_light_sensors_command_get_values(void);

void overo_light_sensors_update_state(void);

double overo_light_sensors_get_value(DeviceTag dt);

void overo_light_sensors_disable(DeviceTag dt);
```

`light_sensors_enable` - array variable describing the state of each sensor. 0 indicates sensor sampling is off and 1 indicates sampling is on.

`light_sensors_ms` - array variable holding sampling periods of all light sensors.

`light_sensors_last_value` - array variable holding the last acquired data from the sensors.

`light_sensors_trigger` - array variable holding flags which signal if a sensor needs to be sampled (1) or not (0).

`offset_light_sensors` - variable holding the position of the first byte of light sensors data in the buffer received as a response to issued commands.

`overo_light_sensors_get_offset()` - returns `offset_light_sensors` value;

`overo_light_sensors_set_offset()` - sets `offset_light_sensors` value;

`overo_light_sensors_enable()` - enables sampling of light sensors with tag `dt` and with a period of sampling `ms`.

`overo_light_sensors_command_get_values()` - fills the command buffer with a command for acquiring a sample from the light sensors.

`overo_light_sensors_update_state()` - updates variable `light_sensors_last_values` with the latest acquired sample.

`overo_light_sensors_get_value()` - returns the value of a specified light sensor device held by `light_sensors_last_values`.

`overo_light_sensors_disable()` - disables sampling of the specified light sensor, saving computation time and serial bus bandwidth.

C.7 Microphone

```
#include <overo/microphone.h>

static int offset_microphone = 0;
static int offset_microphone_buffer = 0;
static int microphone_enable[] = {0,0,0};
static int microphone_get_buffer_enable[] = {0,0,0};
static int microphone_ms[] = {0,0,0};
static double microphone_volume_last_value[] = {0.0, 0.0, 0.0};
static int microphone_last_scan_id = -1;
static int microphone_trigger[] = {0,0,0};
static int microphone_buffer_trigger[] = {0,0,0};

int overo_microphone_get_offset(void);

void overo_microphone_set_offset(int offset);

int overo_microphone_get_offset_buffer(void);

void overo_microphone_set_offset_buffer(int offset);

void overo_microphone_enable(DeviceTag dt, int ms);

void overo_microphone_command_get_last_volumes(void);

void overo_microphone_command_get_buffer(void);

void overo_microphone_update_state(void);

double overo_microphone_get_volume(DeviceTag dt);

void overo_microphone_disable(DeviceTag dt);
```

`offset_microphone` - variable holding the position of the first byte of microphones amplitudes data in the buffer received as a response to issued commands.

`offset_microphone_buffer` - variable holding the position of the first byte of microphones data in the buffer received as a response to issued commands.

`microphone_enable` - array variable describing the state of each microphone sensor. 0 indicates sampling of microphone's amplitude is off and 1 indicates sampling is on.

`microphone_get_buffer_enable` - array variable describing the state of each microphone sensor. 0 indicates sampling microphone sensor data is off and 1 indicates sampling is on.

`microphone_ms` - variable holding sampling periods of all microphone sound sensors.

`microphone_volume_last_value` - array variable holding the values of microphone last sampled volume amplitudes.

`microphone_last_scan_id` - variable holding the id of the last sampled microphone.

`microphone_trigger` - array variable whose members are flags indicating new microphone volume data are expected in the response buffer.

`microphone_buffer_trigger` - array variable whose members are flags indicating new microphone data are expected in the response buffer.

`overo_microphone_get_offset()` - returns `offset_microphone` value;

`overo_microphone_set_offset()` - sets `offset_microphone` value;

`overo_microphone_get_offset_buffer()` - returns `offset_microphone_buffer` value;

`overo_microphone_set_offset_buffer()` - sets `offset_microphone_buffer` value;

`overo_microphone_enable()` - enables sampling of microphone with tag `dt` and with a period of sampling `ms`.

`overo_microphone_command_get_last_volumes()` - fills the command buffer with a command for acquiring microphone's volume amplitude.

`overo_microphone_command_get_buffer()` - fills the command buffer with a command for acquiring microphone's sound data.

`overo_microphone_update_state()` - updates variable `microphone_volume_last_value` with the latest acquired sample.

`overo_microphone_get_volume()` - returns the value of a specified microphone held by `microphone_volume_last_value`.

`overo_microphone_disable()` - disables sampling of the specified microphone, saving computation time and serial bus bandwidth.

C.8 Motors

```
#include <overo/motors.h>

static int offset_motors = 0;
static int motors_enable_set = 1; // left motor, right motor.
                                  // enable by default.
static int motors_enable_get = 0;
static int motors_trigger = 0;
static int motors_ms = 0;
static double motors_speed[] = {0.0,0.0};
static int motors_speed_last_value[] = {0,0};

int overo_motors_get_offset(void);

void overo_motors_set_offset(int offset);

void overo_motors_enable(DeviceTag dt, int ms);

void overo_motors_command_set_speed_values(void);

void overo_motors_command_get_speed_values(void);

void overo_motors_set_speed(DeviceTag dt, double left, double right);

void overo_motors_update_state(void);

void overo_motors_disable(DeviceTag dt);
```

`offset_motors` - variable holding the position of the first byte of motors speed data in the buffer received as a response to issued commands.

`motors_enable_set` - variable indicating whether control of motor's speed is enabled. Value 1 means it is and 0 means it is not enabled. Currently actuators are enabled by default and this feature is not used.

`motors_enable_get` - serves as a flag indicating whether reading the speed from `motors_speed_last_value` variable. This is NOT speed measured by encoders, but just a reflection of the last speed set to the robot.

`motors_trigger` - variable indicating whether a new speed was specified. Value 0 indicates it was not and 1 indicates new speed was specified.

`motors_ms` - variable holding updating periods of motors. This feature is currently not used.

`motors_speed` - array variable that holds the wanted speed of the robot specified by the user.

`motors_speed_last_value` - array variable which reflects the speed actually sent to the robot. This is NOT speed measured by encoders, but just a reflection of the last speed set to the robot and can serve for consistency checks.

`overo_motors_get_offset()` - returns `offset_motors` value;

`overo_motors_set_offset()` - sets `offset_motors` value;

`overo_motors_enable()` - enables sampling of motors with tag `dt` and with a period

of sampling `ms`.

`overo_motors_command_get_speed_values()` - fills the command buffer with a command for acquiring speed previously set.

`overo_motors_command_set_speed_values()` - fills the command buffer with a command for setting new speed data.

`overo_motors_update_state()` - updates variable `motors_speed_last_value` with the last set speed.

`overo_motors_disable()` - disables sampling of the specified motor, saving computation time and serial bus bandwidth.

C.9 Scheduler

A collection of time measurement related functions.

```
#include <overo/scheduler.h>

static int last_ms = 0;
static int step_counter = 0;
static double s_counter = 0.0;
static double interval_start_timepoint = 0.0;
static double interval_stop_timepoint = 0.0;

int overo_scheduler_get_last_ms(void);

int overo_scheduler_get_step_counter(void);

void overo_scheduler_start_time(void);

int overo_scheduler_robot_step(int ms);

// Returns local time in secs
double overo_scheduler_local_get_time(void);
```

`last_ms` - temporary variable used to store intermediate values of measured time.

`step_counter` - counts millisecond steps. If exceeds predefined value signals faulty operation.

`s_counter` - counts number of seconds past from the start time of the controller.

`interval_start_timepoint` - marker to hold a time point.

`interval_stop_timepoint` - marker to hold a time point.

`overo_scheduler_get_last_ms()` - returns the value of `last_ms` variable.

`overo_scheduler_get_step_counter()` - returns the value of `step_counter` variable.

`overo_scheduler_robot_step()` - handles calls to a function which in turn performs all the updates of each sensor and actuator. It also ensures a consistent time step duration.

`overo_scheduler_local_get_time()` - returns the value of `s_counter`.

C.10 Serial communication

Collection of functions which handle the serial communication between the e-puck robot and the extension.

```
#include <overo/serial_communication.h>

static int port_fd = -1; // serial port file descriptor

int overo_serial_get_port_fd(void);

void overo_serial_init(unsigned int baudrate, char* dev_name);

void overo_serial_close(int port_fd);

void overo_serial_send_data(int port_fd, char* tx_buffer, unsigned int tx_size);

void overo_serial_receive_data(int port_fd, char* rx_buffer, unsigned int
rx_size);
```

`port_fd` - handle to access serial port.
`overo_serial_get_port_fd()` - returns the value of `port_fd`.
`overo_serial_init()` - handles all of the initiation details. Opening a port, configuring its speed and protocol.
`overo_serial_close()` - closes the specified serial port.
`overo_serial_send_data()` - responsible for transmitting specified data.
`overo_serial_receive_data()` - responsible for receiving data through the serial port and storing them in a buffer.

C.11 Utilities

```
#include <overo/utilities.h>

// TX and RX buffers pointers
static char tx_buffer[TXMAXLENGTH];
static char rx_buffer[RXMAXLENGTH];

// TX and RX buffer size values
static unsigned int tx_size = 0;
static unsigned int rx_size = 0;

// Holds information about issued sensor commands,
// so we can now what to expect in the response buffer.
static unsigned int issued = 0;

char* overo_utilities_get_rx_buffer(void);

void overo_utilities_add_character_to_tx_buffer(char c);

void overo_utilities_set_issued_bit(int mask);

void overo_utilities_increase_rx_size(int increment);

int overo_utilities_get_issued_value(void);

int overo_utilities_extract_integer_from_buffer(char* buffer, unsigned int
position);

void overo_utilities_update_state(void);
```

`tx_buffer` - command buffer which is transmitted to the basic e-puck board.

`rx_buffer` - response buffer containing information requested with the command in `tx_buffer`.

`issued` - each bit of this variable corresponds to a specific command. It reflects the commands which are issued in a specific time step.

`overo_utilities_get_rx_buffer()` - returns a pointer to the `rx_buffer`.

`overo_utilities_add_character_to_tx_buffer()` - adds one character to the command buffer.

`overo_utilities_set_issued_bit()` - updates the value of `issued` variable according to what commands are issued.

`overo_utilities_increase_rx_size()` - updates the expected size of response buffer which is used later in parsing received data.

`overo_utilities_get_issued_value()` - returns the value of `issued` variable.

`overo_utilities_extract_integer_from_buffer()` - extracts an integer with a specified position from a specified buffer.

`overo_utilities_update_state()` - this function is called in each time step and it handles calls to all of the functions needed to correctly update the states of all sensors and actuators.

C.12 Remote control

So far modules and their functions were presented which enable access to the functionalities of the e-puck robot with the Gumstix extension. Those are enough to be able to run robot controller on the extension's processing unit. However, to be able to remotely control the robot from a remote station, such as PC some supplemental functionalities are needed which will enable the robot to interface with the remote station. These functionalities are encapsulated in several modules which handle interfacing over a broadband network. Specifics for these modules are given here.

C.12.1 Network

```
#include <webots/Broadband.hpp>

class Broadband {

  public:
    Broadband(string protocol, string ip, string port);
    ~Broadband();
    static void refreshAvailableNetworks (bool ethernet=true, bool wireless=true);
    static std::vector<std::string> getAvailableNetworks() { return portNames;
}
    TCPClient* getTcpHandle() {return tcpClient;}
    UDPNode* getUdpHandle() {return udpNode;}
    bool hasFailed() {return failed;}
  private:
    static std::vector<std::string> portNames;
    TCPClient *tcpClient;
    UDPNode *udpNode;
}


#include <webots/TCPClient.hpp>

class TCPClient: public QObject
{
   Q_OBJECT
   public:
      TCPClient(QObject* parent = 0, string ip, int port);
      ~TCPClient();
      void start();
   public slots:
      void startWrite();
      void startRead();
   private:
      QTcpSocket client;
      QByteArray buffer;
      QString qIP;
      quint16 qPort;
};
```

```
#include <webots/TCPServer.hpp>

class TCPServer: public QObject
{
   Q_OBJECT
   public:
      TCPServer(QObject * parent = 0, int port);
      ~TCPServer();
   public slots:
      void acceptConnection();
      void startRead();
      void startWrite();
   private:
      QTcpServer server;
      QTcpSocket* serverConnection;
      QByteArray buffer;
};
```

```
#include <webots/UDPNode.hpp>

class UDPNode : public QObject
{
  QOBJECT
  public:
    UDPNode(QObject* parent = 0, string ip, int port);
    virtual ~UDPNode();
    void setNewData(bool value) {newData = value;}
    bool getNewData() {return newData;}
    char* getReceivedDatagram() {return receivedDatagram->data();}
  public slots:
    void processPendingDatagrams();
    void sendDatagram();
  private:
    QUdpSocket *udpSendSocket;
    QUdpSocket *udpReceiveSocket;
    QString *qIP;
    quint16 *qPort;
    bool newData;
    QByteArray *receivedDatagram;
}
```

C.12.2 Communication interface

```
#include <webots/Robot.hpp>

int wbr_init(void args*);
```

```
int wbr_robot_step(int ms); /* ms = time step in milliseconds */

void wbr_robot_cleanup();

void wbr_stop_actuators();

bool wbr_has_failed();


#include <webots/Communication.hpp>

class Communication {
public:
           Communication(void *args, EPuck *epuck, bool broadband, bool serial,
string protocol, string ip, string port);
  virtual ~Communication();
  virtual bool isInitializedSerial();
  virtual bool isInitializedBroadband();
  virtual int step(int s);
private:
  bool initializedSerial;
  bool initializedBroadband;
  Serial *serial;
  Broadband *broadband;
  EPuck *epuck;
  int retry;
  WbDeviceTag accelerometer;
  WbDeviceTag ps[8];
  WbDeviceTag ls[8];
  WbDeviceTag fs[3];
  WbDeviceTag leds[10];
  WbDeviceTag camera;
  Accelerometer *acc;
  Camera *c;
  DistanceSensor *ds[11];
  LightSensor *lts[8];
  LED *led[10];
  DifferentialWheels *dw;
  bool accelerometerCommandIssued;
  bool proximitySensorCommandIssued;
  bool floorSensorCommandIssued;
  bool lightSensorCommandIssued;
  bool encodersCommandIssued;
  bool cameraCommandIssued;
};


#include <webots/RemoteControl.hpp>
```

```
int wbr_remote_control_get_number_of_available_ports();

const char **wbr_remote_control_get_available_ports();
```