Univerzita Karlova v Praze
Matematicko-Fyzikální fakulta

Bakalářská práce

Ondřej Plátek

Objektově orientovaná knihovna pro řízení robota e-Puck

Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. František Mráz, CSc.
Studijní program: Obecná informatika

2010

Chtěl bych poděkovat panu RNDr. Františku Mrázovi, CSc., za jeho rady a nekonečnou podporu při vedení mé bakalářské práce. Dále jsem vděčný panu Mgr. Pavlu Ježkovi, který mi poskytl cenné informace o technologii .Net. Další díky patří kamarádum Jindřichu Vodrážkovi, Pavlu Menclovi a Adéle Čihákové, kteří mě podporovali při psaní mé bakalářské práce. Jidrovi jsem obzvlášť vděčný za jeho odvahu, když použil nedokončenou verzi *Elib* knihovny pro svůj projekt. V projektu úspěšně implementoval Breitenbergovo chování pro robota e-Puck. V neposlední řadě bych rád poděkoval rodičům za veškerou jejich péči a podporu.

V Praze dne 5. dubna 2011 Ondřej Plátek

Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Ondřej Plátek

Object Oriented Library for Controlling an e-Puck Robot

Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc.
Study branch: General Informatics

2010

I would like to thank my supervisor, RNDr. František Mráz, CSc., for his advice and endless support. I am grateful to Mgr. Pavel Ježek, who gave me valuable informations about .Net technology. My sincere gratitude belong to my friends Jindřich Vodrážka, Pavel Mencl and Adéla Čiháková, who has supported me during writing of my bachelor thesis. Jidra has the courage to use the unfinished *Elib* library developed in this thesis in his project and he successfully implemented a Breitenberg behaviour for e-Puck robot. Last but not least I would like to give thanks to my patient parents for all their efforts.

Prague, April 5, 2011 Ondřej Plátek

Contents

Název práce: Objektově orientovaná knihovna pro řízení robota e-Puck
Autor: Ondřej Plátek
Katedra: Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. František Mráz, CSc.
e-mail vedoucího: Frantisek.Mraz@mff.cuni.cz

Abstrakt: E-Puck je výukový robot s diferenciálním pohonem dvou kol, který je adekvátně vybaven senzory. Výsledkem práce je objektová *C#* knihovna *Elib*, která umožňuje ovládat robota e-Puck z hostitelského počítače přes Bluetooth rozhraní. Vzorové příklady v konsolové aplikaci *TestElib* ukazují použití knihovny *Elib* v programech pro ovládání robota e-Puck. Taktéž je přiložena sada nástrojů umožňující efektivnější ladění programů používající *Elib* library.

Klíčová slova: robot, e-Puck, asynchronní, *C#*, knihovna

Title: Object Oriented Library for Controlling an e-Puck Robot
Author: Ondřej Plátek
Department: Kabinet software a výuky informatiky
Supervisor: RNDr. František Mráz, CSc.
Supervisor's e-mail address: Frantisek.Mraz@mff.cuni.cz

Abstract: E-Puck is an educational robot with differential drive and it is sufficiently equipped with sensors. The result of present thesis is the *C#* object oriented *Elib* library, which allows to control an e-Puck robot from a host computer over the Bluetooth wireless technology. The model examples in the *TestElib* console application presents the usage of the *Elib* library in program for the e-Puck robot. Moreover, it offers a comfortable system of help and documentation. Enclosed sets of tools allows more effective debugging of programs, which use the *Elib* library.

Keywords: robot, remote control, e-Puck, asynchronous, *C#*, library

1. Introduction

Mobile robotics is nowadays a prestigious research area. Building a highly functional mobile robot is considered to be difficult. It is a fascinating process, which involves a lot of engineering disciplines.

A lot of affordable components result in a boom of simple robots used in daily life. Lawnmowers and vacuum cleaners are typical examples of mobile robots invasion into our households in these days. We also get used to robot prototypes, which are highly specialised in the space exploration or in army services.

Despite the rapid hardware development and robots increasing popularity, the research in the field of mobile robots is at its beginning. Robots dominates in industry and they are already useful in our daily life, but the true real life problems are still difficult for them. Many complex problems like driving a car in full traffic or to move in human like environment are still a real challenge for a robot.

The young scientists are often discouraged by studying hardware details during building their own robots. Luckily, several educational robots were made to help the novices. Educational robots help students focus on the other aspects of mobile robotics than the hardware one.

This thesis describes and implements a library for one of the educational robots. Its name is e-Puck and it is a typical example of an educational robot for students at the university level. E-Puck has a clean, simple and robust mechanical design, which is easy to understand. Bluetooth wireless communication enables sending data between e-Puck and PC. A camera, eight infra red (IR) sensors, an accelerometer, encoders and three microphones are enough for a robot to feel the real world. E-puck robot can reply to any type of perception by performing several actions. The robot is able to emit light from light emitting diodes (LEDs) or to play a sound. Two differential stepper motors facilitate a precise movement.

E-puck's sensors and actuators allow to solve a large scale of problems from the field of mobile robotics. On the other hand, the processing power of e-Puck does not allow to process all outputs from its sensors. For example, resolution from e-Puck's camera is 640 * 480 pixels, but the pictures taken from the camera are trimmed under 50 * 50 pixels, because there is a lack of space in e-Puck's microchip memory.

E-puck robot and its sensors and actuators can be controlled only by a low level C code, which needs to be downloaded to e-Puck's memory, and has to be run by the robot's microchip. However, a program downloaded to e-Puck can communicate with a PC and it can let the PC to control the e-Puck robot. Such a program has to define a protocol, in which it communicates between e-Puck and PC over Bluetooth. On e-Puck the program exists and is called *BTCom*.

Programming e-Puck's microchip requires knowledge of sensors and actuators interfaces. Furthermore, debugging is limited to blinking with diodes or playing sounds. The mentioned reasons and insufficient processing power hinder rapid development of even a simple program, which makes e-Puck programming inconvenient for students.

At the time of writing this thesis there was only one well developed software for e-Puck, which makes programming e-Puck easier. It is commercial software called Webots which is an useful development and simulation environment. Other simulators, which can be free alternatives for Webots, are introduced in Chapter 4.

The *Elib* library removes the drawbacks of low level programming, which is the main contribution of this work. *Elib* library runs on PC and communicates with the robot via Bluetooth. It uses *BTCom* program on e-Puck. A program, which uses *Elib* library, turns an e-Puck into a carrier of sensors and actuators. Such program sends commands to e-Puck, whereas the whole algorithmic code runs comfortably on PC.

On the other hand, the implementation of *Elib* brings new problems. This thesis introduces the *Elib's* problems and discusses under which conditions is the approach of *Elib* suitable.

Thesis structure

We introduce mobile robotics in Chapter 2.

Chapter 3 describes e-Puck design and presents its drawbacks and advantages. Chapter 4 lists software which help programmers to develop applications for robots. In this chapter we discuss the usage of a software for e-Puck.

Chapters 5 and 6 presents the design and usage of *Elib* library in detail. Conclusion sums up the properties of *Elib*. A simple installation guide of *Elib*, a list of *Elib's* functions and a list of *Elib's* exceptions are located in Appendixes A andC. An installation and a user guide for *Elib Joystick* application is included in Appendix B.

2. Introduction to mobile robotics

Human beings always created tools to make its job easier. Even the word "robot", first used by Karel Čapek in a novel RUR published in 1920, came from world "robota", which in Czech means labour.

Figure 2.1: Rover Spirit on Mars

Nowadays mobile robots are useful in many areas. Robots excel in exploration of danger places. For example rover Spirit and Delta II flew to Mars. Spirit is a robot of our imagination. It is a six wheel robot which is similar to police robots for a bomb manipulation. On the other hand, Delta II is a space rocket. Robots like Delta II and Spirit allow humankind to explore so distant places, which are too far for a man to travel there.

However, mobile robots are not narrow specialised in adventurous actions and missions impossible for humans. Their main contribution is in daily life. In many cities of the world the robots are used in public transport. They drive the trains in metro, help pilots to take off or land planes. They are used to park cars or they control cruise controls. Solutions which were studied in mobile robotics are usually incorporated in daily life use. One time a car computer helps to park a car, another time a vacuum cleaner tidies up a room. Robots are useful not only as independent machines. Most of the robots help people with difficult tasks, but people control their actions directly. Nice example is a parking robot present in many contemporary cars.

On the other hand, the research in mobile robotics tries to produce robots which are autonomous as much as possible. The developers transfer all control to a robot, which is particularly demanding if the robot interacts with people. There are areas where the research and industry already succeed.

Autonomous vacuum cleaners and lawnmowers are broadly used in households and in gardens and no one has fear from them. A number of robotic toys have been already sold. A robotic dog Aibo introduced in 1999 is a nice example of a toy robot.

Not only entertainment robots are designed for human-robot interaction. In Japan there is a big stress on creating robots for health care, which will compensate lack of medical personnel. A robot has to be a humanoid, because especially children and older people are used to communicate with other people and not with machines.

Professor Hiroshi Ishiguro from Japan reached a great success in field of humanoid robots. His goal is to build robot, which would be a good companion for a human. Among other humanoids he has successfully constructed a copy of himself.

The humanoid robots and also the Aibo robot are not wheeled robots, although the wheeled robotic systems are still dominating. The reason is in the field of the activity of the robots. The entertainment and the health care robots are in the human environment. Robots avoid a lot of different obstacles and they have to use stairs. The industrial robots are usually indoors, where the terrain is adapted

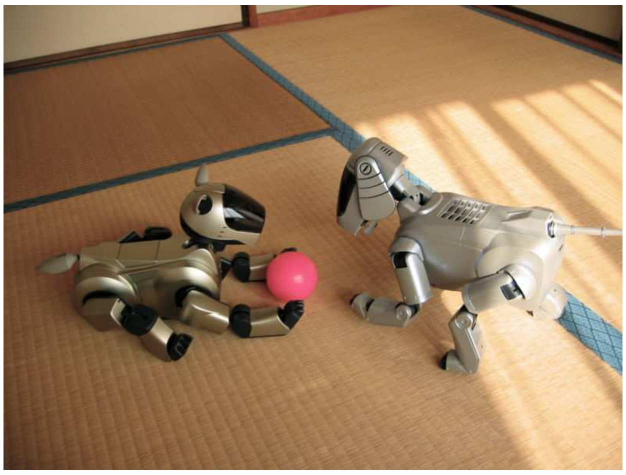


Figure 2.2: Two versions of an Aibo robot

"About 4.4 million units for domestic use and about 2.8 million units for entertainment and leisure sold up to end 2009." [9]

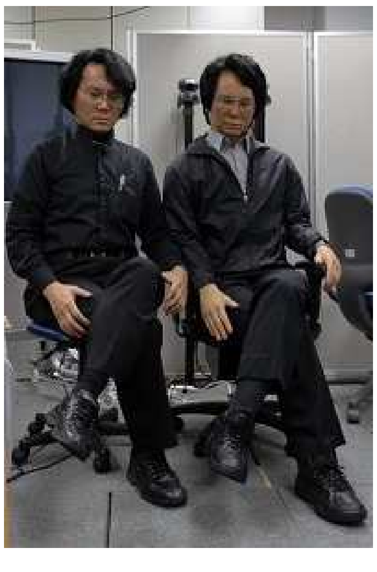


Figure 2.3: Ishiguro and his humanoid

to machines movement. Also the agricultural robots have convenient conditions for wheel movement.

Not only the design of robots, but also the control of robots differs and evolves. In the eighties the sense-plan-act model was current. Later in the nighties the behaviour model was introduced by Brooks, which was a very modern and sometimes strictly implemented. After a decade a compromise between the behaviour based robotics and the simple planning was popular. In the nighties neural networks and other modern, biologically motivated attitudes were first introduced. Genetic algorithms and neural networks have enormous success in solving complicated control problems like motion of robot with a number of legs, implementing very sophisticated behaviour and so on.

To conclude, mobile robotics is a dynamically evolving science, which has all preconditions to be one of the leading research areas in the twenty first century. The value of the market increased to 6.2 billion dollars in 2009 according to the International Federation of Robotics. [9]

> "I can envision a future in which robotic devices will become a nearly ubiquitous part of our day-to-day lives,"

says Bill Gates, who was at the beginning of the computer revolution.

3. E-Puck

An object of interest of a this thesis is a programming of the e-Puck robot. In this chapter the robot and its design is introduced. Then in Section 3.2 the chapter continues with description e-Puck's mechanical design. Section 3.3 is devoted to e-Puck's sensors and actuators.

We focus on a programmer's point of view of e-Puck qualities. For each sensor its functions, acceptable values and typical use is shortly presented. The chapter finishes with a description of a e-Puck's camera and outlining other e-Puck abilities.

3.1 Origin of e-Puck

Motivation for a new robot arose from an absence of a very small educational robot, which is can be used for education in many research areas. E-Puck was developed in summer of 2004 at the École Polytechnique Fédérale de Lausanne (EPFL) as an open tool. E-Puck designers used open software and hardware development model. E-Puck can be applied in automatic control, signal processing or distributed intelligent system research. E-Puck's structure is robust and simple to care about, because e-Puck is intended to be used by students. Designers of e-Puck tried to use manufacturing components as much as possible in order to keep the price low.

Since 2006 the first generation of e-Pucks was replaced by the second generation. So far, more than 2000 e-Pucks in version 2 have been manufactured. There are several distributors for Switzerland and America, one is in Japan [1]. The price of e-Puck should be between 450 and 550 Eurors.

3.2 Mechanical Design

The mechanical design of e-Puck is shown in Figure 3.1. It consists of a rounded transparent body, a battery, two stepper motors, wheels , a printed circuit board, plastic ring of LEDs, a camera and a default extension board. There are extensions like floor sensors, a rotating scanner or a turret with linear cameras, which can scale up e-Puck's sensors. If we are talking about e-Puck in this thesis, it is meant e-Puck with the default extension only.

A default robot is more than 60 mm high and it has a diameter of 75 mm. The battery is placed in the bottom and can be easily extracted and recharged separately. Two stepper motors with wheels are screwed to a plastic body and are located on axis of e-Puck in order to allow robot turn around on place. Wheels have the diameter of 41 mm and the perch is 53 mm. Printed circuit board is

[1] `http://www.roadnarrows.com/robotics/`, `http://www.aai.jp/`, `http://www.k-team.com/`, `http://www.cyberbotics.com/products/robots/e-puck.html`, `http://www.gctronic.com/`, `http://www.aai.ca/robots/e-puck.html`

fixed to the top of the body and a ring of LEDs' is mounted around the board. A camera is placed on the front side of robot lying on the axis between the wheels. The extension board covers the main printed circuit board.

3.3 Sensors, actuators and heart of the robot

Sensors and actuators determine the possible robot usage. Luckily, e-Puck has a lot of sensors of different kinds and is equipped with typical actuators. The following paragraphs shortly describe e-Puck's sensors and actuators and mention a few problems with transmitting data between the robot and PC using *BTCom*. *BTCom* protocol is used by *Elib* library, so *Elib* is to some extent dependent on *BTCom*. The actuators consist of motors, 8 red LEDs on perimeter, 4 green body LEDs, which are turned on/off together, a front LED and a speaker.

Stepper motors are a great advantage of e-Puck, because the motion of wheel can be split into small steps. One wheel revolution corresponds to 1000 steps of motor. The diameter of the wheel is 41 mm. If the wheel makes one revolution, the wheel goes 128.8 mm. In conclusion a thousand of steps matches 128.8 mm of linear movement and one step is 0.1288 mm. Motors are equipped with encoders and the motors are very accurate, which in combination with simple odometry really helps in localization tasks. A nice feature of encoders is that their value can be set at any time. The maximum speed of the motors is one revolution

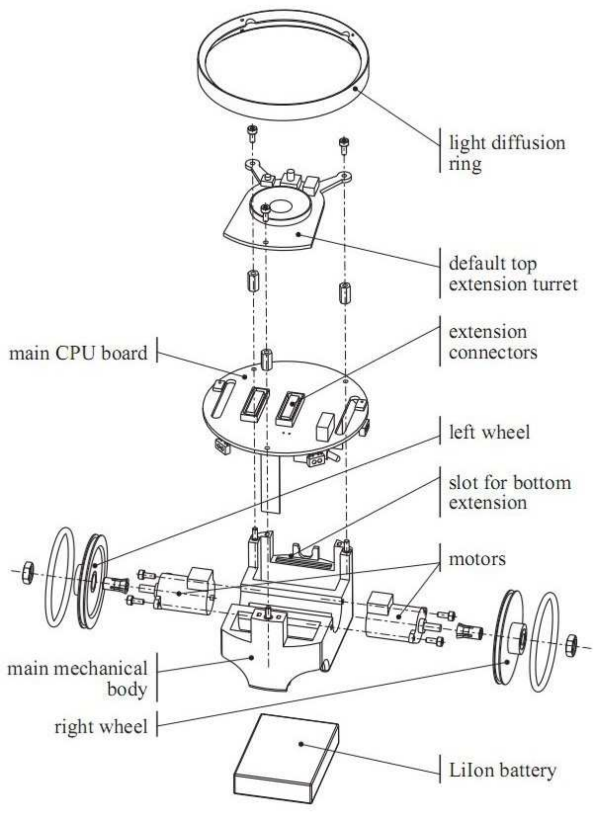


Figure 3.1: Epuck structure [7]

per second in both directions. *BTCom* allows a programmer to set the speed of motors and also get and set the values of encoders.

All e-Puck LEDs can be used for debugging or for making robot visible to other devices. Furthermore, the front LED is usually used to illuminate the terrain in front of e-Puck. Each LED can be turned off, turned on or set to an inverse state.

Three axis accelerometers are placed inside the robots body. In the rest position the accelerometers measure the slant of e-Puck. They can also measure acceleration of e-Puck and for example detect a collision or a falling state of e-Puck.

Infra Red (IR) sensors are typical sensors for mobile robotics. E-Puck has eight of them. Four are located in the front part of e-Puck, two are on both sides, two on back side. Sensors work in two modes. First they measure ambient infra red light. In the second mode IR sensors emit IR light and they measure reflected light, so they can detect close obstacles. Both functions are available in *BTCom*. Calibration of sensors increases the precision of detecting near obstacles. Infra red sensors are capable of recognising an obstacle at a distance 4 cm.

The speaker together with microphones are suitable communication tools between the people and e-Puck. On the other hand, a limited processing power of e-Puck does not allow to store and play complicated sounds. It is also impossible to use a speaker with microphones to speech recognition due to insufficient processing power. Despite the limitation of processing power the microphones can be easily used to locate the source of sound via amplitude measurement, because the microphones are placed near the perimeter in a triangle. Microphones measure current amplitude of sound. As exact distances between microphones are known, we can compute frequency of the sound using Fast Fourier Transformation (FFT). Digital Signal Processor (DSP) is suitable for computing FFT, which will be introduced below in this chapter. Maximal acquisition speed is 33 kHz. For more information see [12].

Remark. A program which uses *BTCom* can capture values of amplitude only in irregular intervals, so it is not possible to compute frequency of sound by running FFT on Personal Computer.

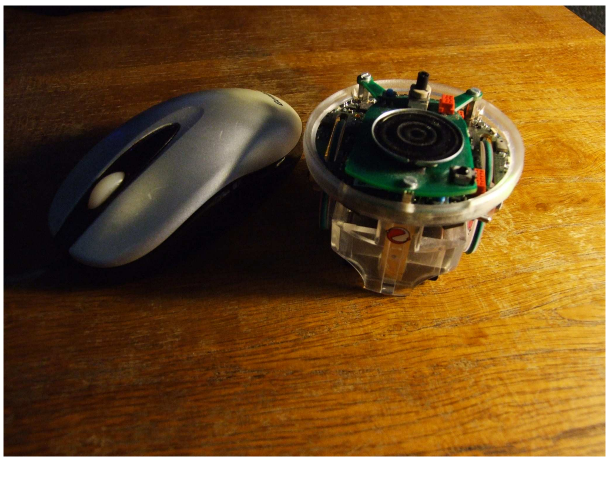


Figure 3.2: E-Puck avoiding a mouse

E-Puck's camera is a colour CMOS camera with 640 * 480 resolution. Only 8 kB of RAM memory is available, so the size of an image has to be reduced in order to save the picture in the memory. The image processing is really demanding on the processing power so it is not convenient to be run on the slow e-Puck's processor.

On the other hand, using *BTCom* solves the problem with limited processing power by sending picture to Personal Computer (PC). PC has enough resources to process the image fast, but the transport of an image takes a long time too. For example capturing and sending a colour image of size 40 * 40 pixels takes around 0.3 seconds, if it is sent over Bluetooth.

BTCom can set height, width, colour mode and zoom of camera. A colour picture is twice as big as the same picture taken in grey scale mode. In the gray scale mode, each pixel is represented by 1 byte value of intensity. Each pixel of color picture is represented as 3 values of *RGB* stored in 2 bytes. Red color is stored in first 5 bits, green is represented by the next 6 bits, finally the last 5 bits of 2 bytes represents blue color. Zoom has three acceptable values 1, 4 and 8. One is for the biggest zoom, 8 represents the smallest. Width and height are limited only by the size of the available memory in e-Puck.

Processor, dsPIC 30F6014, is the heart of e-Puck and runs at 60 MHz, which correspond to 15 MIPS. It has C oriented instructions and supports compiling from GNU compilers. Apart from standard 16 bits core unit, Digital Signal Processor brings very high performance for computation FFT or other signal processing. Programs can be downloaded to flash memory with 144 kB and are loaded to RAM memory according to the selector position. E-Puck's RAM memory has only 8 kB. A currently running program and all its data are placed in RAM memory. Communication with other devices is provided by IR port or Bluetooth and RS232 serial interface. Both, Bluetooth and RS232, can be used to download programs to e-Puck's flash memory. In addition, Bluetooth can be used to communicate with other e-Pucks or with a computer using *BTCom*. The counter part of *BTCom* on computer is connected to the serial port.

4. Available software

In this chapter we shortly introduce software that helps a programmer to debug and develop programs for a robot control. Only simulators will be introduced, because there is no available program or library for e-Puck like *Elib* that controls e-Puck remotely and asynchronously from PC. All introduced simulators run at PC and allow user to control and program a virtual copy of robot. Most of the simulators simplify physics and construction, although they try to simulate sensors and actuators as faithfully as possible. Some of the simulators support porting the program, which has been created for the virtual robot, to real robot.

4.1 Why simulators, why remote control?

Elib and the simulators benefit from moving the debugging environment from the robot to PC. PC has enough resources at disposal and the code can be debugged much more comfortably. Debugging tools depend on the structure of simulator and a programming language which is used by the simulator. *Elib* and the simulators allow to stop the program at any time.

Elib uses *C#*, which can be developed in Microsoft Visual Studio IDE or Monodevelop IDE. Both IDEs support many features like setting breakpoints with conditions, call stack table, manual switching between threads and so on.

The simulators have several advantages over robot remote control. A simulator provides the developer with a consistent behaviour of a virtual robot. A programmer has for example a battery level under control. the virtual robot measures in the same situations identical values on its sensors.

The real robot is strongly influenced by the state of the battery and measures a different values in the same situation. Great advantage of the simulators is the possibility of not only stopping the running program, but also freezing the simulated environment.

Simulators also do not need a real robot for testing a program, which is extremely useful in the class with many students where only a few robots are available. Some of the simulators are completely free, which is certainly an advantage.

On the other hand, debugging program using a remote control of a robot has some benefits too. A lot of unpredictable situations are not simulated in simulators, although many simulators have the option to involve randomness. A robot usually needs to be tested in the real environment too.

Let us note, that simulators use physical and graphical engines. These are demanding applications for PC. On the other hand, remote control is a simple fast application.

Last but not least, playing with real robot is simply much more entertaining. It motivates the users to solve real problems including hardware set up. During programming a real robot the typical problems of mobile robotics are touched on. Simulators do not motivate the users to cope with a changing environment and robot's imperfections.

In following sections the simulators listed below will be discussed:

- Microsoft Robotics Developer Studio[3] is a complex tool based on message sending.
- Player/Stage[13] (Gazebo) is an open source project developed in C++, which is widely used.
- Pyrobot(Pyro)[8] is also an open source project written in Python. It supports abstract interfaces for devices convenient for e-Puck.
- Enki[10] is a fast simulation tool for a big population of robots.
- Commercial simulator webots[5] is the only simulator, which almost fully support e-Puck.

4.2 Microsoft Robotics Developer Studio[3]

Microsoft Robotics Developer Studio (MSRS) is a set of tools including service-oriented runtime, simulator, visual programming language (VPL), tutorials and examples. MSRS is based on .Net and its runtime introduces following new technologies:

- Concurrency and Coordination Runtime (CCR), which makes asynchronous programming easier.
- Decentralised Software Services (DSS) monitor services in real time for developers.
- Common Language Runtime (CLR) 2.0 is underlying runtime, which interprets compiled code from any .Net language.

MSRS offers VPL, which is convenient for beginners or for building a structure of a new application. VPL can be later easily transformed to any .Net language.

Visual simulation environment is based on AGEIA PhysX engine. It enables a 3D simulation.

In general MSRS is a rich set of tools, which introduce numerous nice features including technologies for easy asynchronous and concurrent programming. Especially CCR helps a programmer with concurrent programming.

There are enough tutorials for beginners written in *C#*, Visual Basic or Python. (Iron Python is an implementation of Python for .Net.) on the other hand, a lot of new features and tools bring a lot of new problems to users. MSRS tutorials and documentation have been rapidly improved in last year, but still some parts of the documentation are not easy accessible.

MSRS runs only under Microsoft Windows operating systems and for a non commercial development it is for free. Neither e-Puck nor any similar robot is supported in the latest version of MSRS.

The studio would be a good solution for e-Puck, because it meets the requirements for *Elib*. On the other hand, it is a huge, not portable, complicated environment. Due to its complexity it discourages students from learning it.

4.3 Player/Stage (Gazebo)

Player[13] is a network server running on a robot and sending sensors values from the robot and accepting commands from clients. Player[13] has a codified interface. It translates commands from the interface to the implementation of actuators and translates values back from the robot sensors to Player[13] interface. Unfortunately Player[13] is not implemented for e-Puck and for its devices.

Stage is a 2D simulator, which is reflecting the sensors and actuators using the interface defined by Player. It simulates a population of mobile robots and therefore it is possible to use Player in multi agent systems. The sensors are very simplified, because all sensors of the same type have to implement the same interface. For example a sonar from Khephera and a sonar from Aibo robot has the same interface and Stage treats them equally. such simplification allows Stage to be much quicker.

Gasebo[13] is a 3D simulator. Like Stage Gazebo simulates a big population of robots. It uses a rigid-body physics like Stage too.

The whole project is an open source and it is designed for posix-compatible systems. C/C++, Python, Java are the main supported languages. Player/Stage is one of the most frequently used simulators in the research and is good for education purposes. A Khephera robot, which is similar to e-Puck is supported. It indicates, that it is worth to think about player/Stage, if a simulator for e-Puck is wanted.

4.4 Pyrobot(Pyro)[8]

Pyro is a shortcut from Python robotics. It is an open source project written in Python. Pyrobot is simulator from Pyro project and the name Pyrobot replaced the name of the project Pyro. Pyro abstracts interfaces of a robot and its devices, so each robot supported by Pyro can be treated equally. Pyro let a programmer choose from a list of simulators including Player/Stage, Gazebo, Pyrobot, Khephera simulator and Robocup soccer. After selecting a simulator there are available different worlds according to the selected simulator. A simulated robot can be driven from the command line or be controlled via loaded behaviour.

The simulated robot can be easily replaced with a real robot, Controlling the real robot is done via communication with program running on the robot similar to *BTcom* program on e-Puck.

Khephera robot is supported. There is an implementation for e-Puck, which can simulate the robot and connect to the real robot, but the implementation does not support asynchronous operations and does not implement many sensors and actuators.

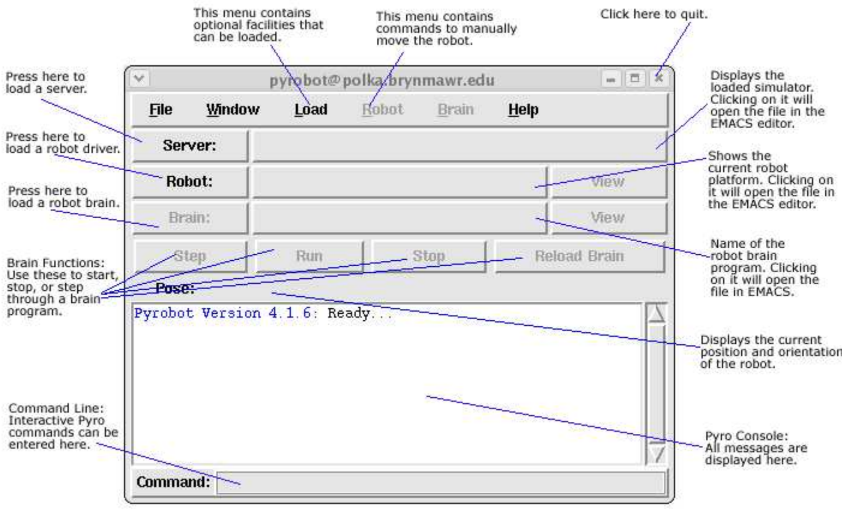
For using Pyro as simulator for *elib* a translations between *BTcom* and Pyro protocol is needed. For more information see future works in conclusion 7.

Pyro project was abandoned in 2007 and no further development is planned. The advantage of the project is its portability due to the use of Python and open source licence.

4.5 Enki & Aseba[10]

Enki is a library, which can simulate several robots hundred times faster than real time. The library is written in C++ and is still in development. It supports e-Puck including e-Puck's camera.

The e-Puck can be controlled via Aseba. Aseba is a tool for robot control. It introduce a new scripting language. The robots can be controlled only by the scripting language, which is the main drawback of Enki, because the scripting language is very simple. In addition, Enki does not support the remote control of robots.

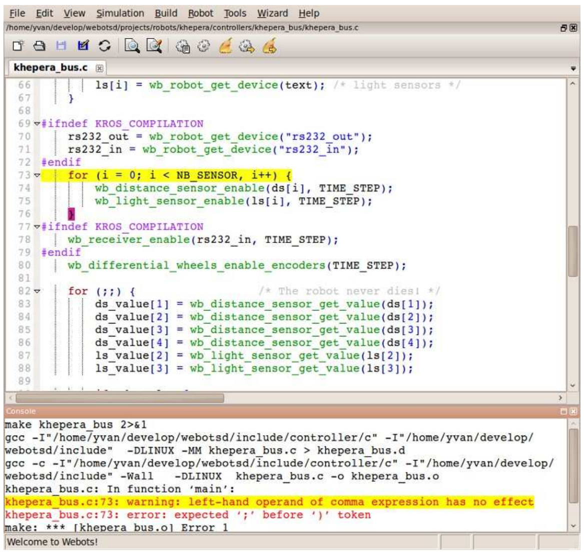


This menu contains optional facilities that can be loaded.
This menu contains commands to manually move the robot.
Click here to quit.
pyrobot@polka.brynmawr.edu
File
Window
Load
Robot
Brain
Help
Press here to load a server.
Press here to load a robot driver.
Press here to load a robot brain.
Server:
Robot:
Brain:
View
View
Step
Run
Stop
Reload Brain
Pose:
Displays the loaded simulator. Clicking on it will open the file in the EMACS editor.
Shows the current robot platform. Clicking on it will open the file in the EMACS editor.
Name of the robot brain program. Clicking on it will open the file in EMACS.
Brain Functions: Use these to start, stop, or step through a brain program.
Displays the current position and orientation of the robot.
Command Line: Interactive Pyro commands can be entered here.
Pyro Console: All messages are displayed here.
Command:

```
Pyrobot Version 4.1.6: Ready...
```

Figure 4.1: Starting window of Pyrobot

4.6 Webots[5]


File Edit View Simulation Build Robot Tools Wizard Help
/home/yvan/develop/webotsd/projects/robots/khepera/controllers/khepera_bus/khepera_bus.c
khepera_bus.c
Console
Welcome to Webots!

```
66 |       ls[i] = wb_robot_get_device(text); /* light sensors */
67 |     }
68
69 #ifndef KROS_COMPILATION
70     rs232_out = wb_robot_get_device("rs232_out");
71     rs232_in = wb_robot_get_device("rs232_in");
72 #endif
73     for (i = 0; i < NB_SENSOR, i++) {
74       wb_distance_sensor_enable(ds[i], TIME_STEP);
75       wb_light_sensor_enable(ls[i], TIME_STEP);
76     }
77 #ifndef KROS_COMPILATION
78     wb_receiver_enable(rs232_in, TIME_STEP);
79 #endif
80     wb_differential_wheels_enable_encoders(TIME_STEP);
81
82     for (;;) {                  /* The robot never dies! */
83       ds_value[1] = wb_distance_sensor_get_value(ds[1]);
84       ds_value[2] = wb_distance_sensor_get_value(ds[2]);
85       ds_value[3] = wb_distance_sensor_get_value(ds[3]);
86       ds_value[4] = wb_distance_sensor_get_value(ds[4]);
87       ls_value[2] = wb_light_sensor_get_value(ls[2]);
88       ls_value[3] = wb_light_sensor_get_value(ls[3]);
89
```

```
make khepera_bus 2>&1
gcc -I"/home/yvan/develop/webotsd/include/controller/c" -I"/home/yvan/develop/
webotsd/include"  -DLINUX -MM khepera_bus.c > khepera_bus.d
gcc -c -I"/home/yvan/develop/webotsd/include/controller/c" -I"/home/yvan/develop/
webotsd/include" -Wall   -DLINUX  khepera_bus.c -o khepera_bus.o
khepera_bus.c: In function ‘main’:
khepera_bus.c:73: warning: left-hand operand of comma expression has no effect
khepera_bus.c:73: error: expected ‘;’ before ‘)’ token
make: *** [khepera_bus.o] Error 1
```

Figure 4.2: Debugging in Webots environment

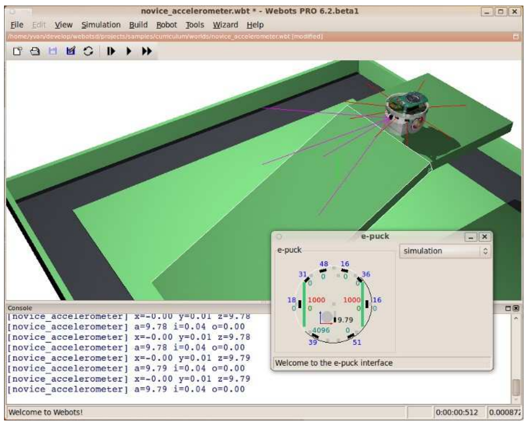
Webots is a simulation and prototyping tool for mobile robots. We will describe the simulation part. Prototyping tools allow to create a virtual robot, which will correspond to the real robot. Virtual e-Puck robot has already been designed by Webots’s developers, so users of Webots do not have to design its own virtual e-Puck.

A development of a program under Webots is divided into four stages. The first stage designing a virtual robot can be skipped, because the developers of Webots prepared a virtual e-Puck. The second phase consists of programming robot in C language. The third phase is the simulation. In the fourth stage a compiled program is flashed to e-Puck’s memory.

Webots has also the option of controlling e-Puck via Bluetooth. The robot can be programmed with graphical programming language for beginners or with C/C++ or Java, but only C/C++ can be compiled for the real e-Puck.

The simulated worlds are created and saved in VRML and can be imported from 3D Max, Cinema 4D. A lot of virtual worlds are already prepared, so it is not necessary to create them.

Webots is a complex environment, in which it is easy to program, compile and load a compiled hex file to e-Puck. The simulation runs in 3D and Webots supports all sensors and actuators except for microphones. A great advantage is that the real robot can be controlled by Bluetooth or programmed by loading a compiled program. Webots is also well documented and good tutorial are available. A significant drawback of Webots is its price and its annual fees.


novice_accelerometer.wbt * - Webots PRO 6.2.beta1
File
Edit
View
Simulation
Build
Robot
Tools
Wizard
Help
/home/yvan/develop/webotsd/projects/samples/curriculum/worlds/novice_accelerometer.wbt [modified]
e-puck
e-puck
simulation
48
16
31
36
18
1000
1000
16
9.79
4096
39
51
Console
Welcome to Webots!
0:00:00:512
0.000872

```
[novice_accelerometer] x=-0.00 y=0.01 z=9.78
[novice_accelerometer] a=9.78 i=0.04 o=0.00
[novice_accelerometer] x=-0.00 y=0.01 z=9.78
[novice_accelerometer] a=9.78 i=0.04 o=0.00
[novice_accelerometer] x=-0.00 y=0.01 z=9.79
[novice_accelerometer] a=9.79 i=0.04 o=0.00
[novice_accelerometer] x=-0.00 y=0.01 z=9.79
[novice_accelerometer] a=9.79 i=0.04 o=0.00
```

Welcome to the e-puck interface

Figure 4.3: Simulation of e-Puck behaviour

5. Elib design

The *Elib* library is a *C#* asynchronous library for a remote control of e-Puck. The goal of the library is to make developing programs for the robot easier. This chapter presents several topics, which influenced the design of *Elib*. The Examples on usage of *Epuck* class and its interfaces are introduced is in Chapter 6.

In the first place the possibilities of programming e-Puck are depicted. Next section covers the advantages and the drawbacks of asynchronous remote control programming of e-Puck. Furthermore, key features of *Elib* are presented together with crucial decisions, which lead to properties of *Elib*. *Elib's* features are confronted with the alternatives. The differences are discussed and the main reasons for choosing the current implementations are mentioned.

After that, the logic of the main classes *Sercom* and *Epuck* are introduced. Sections 5.5 and 5.6 describe implementation of the classes in detail. In the part devoted to *Sercom* the main stress is on the design of Bluetooth communication processing. The implementation details of the two *Epuck's* interfaces are the main topic of the rest of this chapter.

The last Section 5.7 sums up qualities of interfaces implemented by *Epuck* and *Sercom* class. Section 5.7 also describes *Elib's* performance results.

This chapter assumes that a reader understands the use of callbacks and *EventWaitHandle* class in .Net. However, the callbacks and *EventWaitHandle* class together with other advanced .Net features are introduced in section 6.1.

5.1 Approaches to e-Puck programming

E-Puck is distributed with several programs in its flash memory. Most of the programs implement a simple behaviour, which introduces e-Puck sensors and actuators. There is a behaviour, which made e-Puck go to the source of light. If another behaviour is running, e-Puck reacts to clapping and it turns around to the source of a clap. For example one of the behaviours makes e-Puck cry, if it is falling.

The *BTCom* program and e-Puck bootloader are also distributed on e-Puck, but belong to different classes of programs. *BTCom* allows e-Puck to communicate with other devices via Bluetooth.

It defines a BTCom protocol, which works in two modes. The text mode accepts commands for all sensors and actuators except for the fact that it does not allow to take a picture with the e-Puck's camera. In the text mode the accepted commands and replies of BTCom protocol are short text messages. On the other hand, *BTCom* in the binary mode can send a picture from e-Puck.

If *BTCom* runs in the binary mode, single byte commands are accepted instead of text messages. The binary mode is more economical, because the commands are sent in single bytes and also the integer values from sensors are not converted to a textual form. Unfortunately, the binary mode of *BTCom* does not control all sensors and actuators, because it accepts fewer commands than in the text mode. The absence of commands for controlling some sensors and actuators is the main reason for sending all commands from *Elib* to *BTCom* in the textual form. The single exception is sending an image back to *Elib* in the binary mode.

Bootloader, officially called Tiny Bootloader[1], is the only wired program on e-Puck. It downloads other programs to e-Puck via Bluetooth. Tiny Bootloader has its counterpart, a graphical PC application, which allows users to download hex files to e-Puck. A hex file[1] for e-Puck is compiled C program for e-Puck's dsPic microchip.

All mentioned programs including Tiny Bootloader use a C library, which interface the devices on e-Puck. The library and the programs are published under open source licence at `http://www.e-puck.org`.

How to run a program

If e-Puck is turned on, e-Puck starts running one of the downloaded programs according to its selector position. E-Puck's selector switch is located on the main board of e-Puck. If a programmer wants to run another program, he has to switch selector to another position and hold a second a reset button on e-Puck. Selected program is loaded from flash memory to RAM memory and immediately executed.

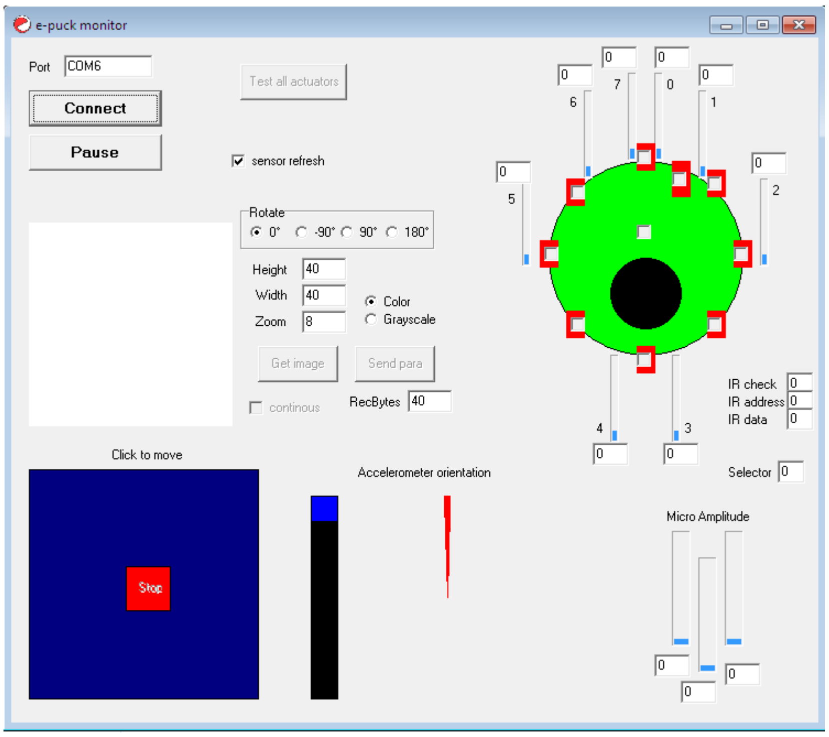
Suppose there is correct C code for e-Puck's processor saved on PC. What has to be done to run the program? Let us describe the procedure. Firstly it has to be compiled and linked. After that, a transformation to a hex file is necessary, because e-Puck's dsPic is 16 bit processor. In the next step Bluetooth and e-Puck are turned on. The robot and your computer are paired together. The user has to enter the PIN, which is needed to pair the devices. PIN is a number, which is on e-Puck's body. The OS opens a serial port for Bluetooth communication. Graphical counter part of Tiny Bootloader is needed. It is used to select the hex file and choose the opened port. Furthermore, the Tiny Bootloader guides user through the downloading of the hex file. The instructions tell the user to press a reset button on e-Puck. It has to be pressed only for about a second. Tiny Bootloader announces when the download finishes. To run the program it is necessary to reset the e-Puck.

The program is written to e-Puck's flash memory according to e-Puck's selector position. If another program had been saved under the same selector position, it is overwritten by the new program. Downloading hex file to an e-Puck lasts approximately 30 seconds and it is sometimes unsuccessful. Programming e-Puck via Tiny Boatloader is not very comfortable. Luckily, there is an another way, which uses *BTCom* program downloaded on e-Puck.

In order to use *BTCom* turn e-Puck's selector to position, under which it is downloaded, and press reset button. Now *BTCom* is running. Pair your PC with e-Puck as described above and use the open port to control e-Puck. *BTCom* starts running in text mode, so its possible to use a terminal to communicate with

[1]`http://en.wikipedia.org/wiki/Hex_file`

e-Puck. E.g Hyperterminal on Windows. On Linux write directly to the serial port using a command line. A program can control e-Puck via BTCom protocol too, but it must process the commands and their replies.



Port COM6

[✓] sensor refresh

Rotate [✓ 0°] -90° 90° 180°

Height 40
Width 40
Zoom 8

[✓ Color] Grayscale

[ ] continous

RecBytes 40

IR check 0
IR address 0
IR data 0

Selector 0

Figure 5.1: Epuck Monitor 2.0

Nice example of accessing sensors and actuators over Bluetooth is e-Puck Monitor[2]. E-Puck Monitor is an open source graphical application written in C++, which uses BTCom protocol. Its User Interface presents values of all e-Puck's sensors in one window and user can easily control e-Puck's actuators. E-puck Monitor's serious drawback is its freezing. The application does not respond, because it waits synchronously to answers from *BTCom*. If a Bluetooth connection with e-Puck is broken or the answer is lost, the application remains unresponsive.

Summary

Downloading a hex file to e-Puck's microchip and remote control over BTCom protocol are two alternatives of controlling e-Puck. Using *BTCom*, a user does not have to download any file to e-Puck's microchip. It is great advantage, because the length of developing cycle of program for e-Puck is reduced. The developing cycle consist of:

- writing a code,
- compiling it,
- downloading it,

[2]Download E-Puck Monitor from `http://www.e-puck.org`.

- debugging it,
- and correcting it.

In following section the advantages of remote control will be described in detail.

5.2 Advantages of remote control

Applications, which use *BTCom* and its protocol, run a whole algorithm on PC and *BTCom* just executes the commands and gets the values from sensors. Imagine you have a library, which process the commands and their replies. A program for controlling e-Puck uses the library and just asks the library for sensor values and gives the library commands what to do. Important is that download part is missing.

Leaving out the item "downloading the program to e-Puck" from the life cycle in List above saves programmer 20 seconds or more, because the process of downloading a program to e-Puck is unreliable. Last but not least the quality of debugging is significantly improved by using a library for remote control.

If a program is downloaded to e-Puck, there are only e-Puck's actuators for a feedback. The actuators are a very poor debugging tool, because if something goes wrong, it is impossible to say whether the program stopped running, or e-Puck is waiting to sensors values, or battery is down. Discovering the problem, even if it is the low battery, takes one or two loops of downloading and compiling the program. E-Puck has an indicator of low battery, but it is not reliable. Downloading a program takes a long time and successful download could last even a couple of minutes. It drives a lot of people crazy.

A programmer, which controls a robot remotely, can debug with tools of his programming environment and he still controls e-Puck. All this can be utilised, because a computer has much more resources than e-Puck. Let us mentioned a few techniques for debugging. Logging of robot actions is often underestimated, but it is very useful for performance testing. Flow control by setting breakpoints is the great advantage of remote control. However, in applications, which call procedures asynchronously, the breakpoints should be set carefully. Breakpoints do not stop the asynchronous operation and do stop the main thread. Therefore breakpoints should be set only in place, where no asynchronous methods have been invoked with timeout or the methods have been already finished.

Another point of view is where the computation is performed. If the remote control is used, it can be said that the intelligence is located on a computer. However if you load a program to e-Puck via Tiny Bootloader the whole algorithm runs on e-Puck. The resources of PC are also useful for a computation of the program and not only for its debugging. E-Puck for example takes pictures smaller than 3200 bytes and still complicated image processing requires too much work for e-Puck's microchip. It is faster to process the picture on a computer. Let us note that also implementing a genetic algorithm in 8 kB memory is very difficult.

5.3 Design of the *BTCom* program. What does it mean for *Elib*?

The interfaces of *Elib* is designed for *BTCom* version 1.1.3, but *Elib* is also compatible with *BTCom* 2.0. The complete *BTCom* source code version 2.0 is located on enclosed CD in file "btcom.c".

BTCom is a mainly textual protocol, which is defined by *BTCom* program. The answer of *BTCom* always begins with the first letter of the relevant command. If the command requires some sensor values, the value is appended to the letter and sent. Otherwise the first letter of a command is sent back alone. Each answer ends by \r\n escape sequence. See below the list of all text commands printed from *BTCom* below.

```
"A"          Accelerometer
"B,#"        Body led 0=off 1=on 2=inverse
"C"          Selector position
"D,#,#"      Set motor speed left,right
"E"          Get motor speed left,right
"F,#"        Front led 0=off 1=on 2=inverse
"G"          IR receiver
"H"           Help
"I"          Get camera parameter
"J,#,#,#,#"  Set camera parameter mode,width,heigth,zoom↩
   (1,4,8)
"K"          Calibrate proximity sensors
"L,#,#"      Led number,0=off 1=on 2=inverse
"N"          Proximity
"O"          Light sensors
"P,#,#"      Set motor position left,right
"Q"          Get motor position left,right
"R"          Reset e-puck
"S"          Stop e-puck and turn off leds
"T,#"        Play sound 1-5 else stop sound
"U"          Get microphone amplitude
"V"          Version of BTCom
```

Two questions

Let A be a PC and B be an e-Puck robot. A controls B over *BTCom* protocol. *BTCom* program located on B replies to all commands sent from A. The answers of *BTCom* from B allows A to know that the sent command was received and executed.

Two main questions, which affect design of *Elib*, arise from the *BTCom* structure. Should *Elib* limit the waiting time for an answer from B? How fast should *Elib* allow A sending the commands to B?

Remark. Sending commands from *Elib* in the following paragraphs does mean sending textual or binary commands using *BTCom*. It does not mean calling any functions of the *Elib* library, which wrap the sending.

It is necessary to understand, that not only executing a *BTCom* command, but also a transfer of a command and a transfer of a reply take significant amount of time. The time is measured from a computer processor's point of view, because it is the processor, who is waiting for serial port, which is sending the message.

Back to the question how should *Elib* limit sending of the commands. As we have said, sending commands takes a while and therefore sending is performed asynchronously. Due to asynchronous sending, users do not have to wait to the end of sending previous command and can send next one before the first command has finished. It leads to queueing the commands in a serial port output buffer of PC. Unfortunately the buffer is usually not very large and can easily overflow. This is the reason for implementation *Elib's* buffer, a queue, which slows down the sending and does not allow overwriting of sent commands by each other in the output buffer.

Is the problem with sending solved? Not for e-Puck. If the commands were sent too fast, the input buffer of e-Puck would have the same problem, because e-Puck's processor does not keep up emptying e-Puck's input buffer. The buffer would be flooded and the commands would be overwritten. *Elib* has to check, if the Bluetooth is not able to send commands too fast for e-Puck and possibly slow down sending of commands even more.

How long *Elib* should wait for the answer of a sent command?

Clearly we have to wait more than is the minimum transfer time from a computer to e-Puck and back. The transfer time depends on amount of transfered data. Sensors messages carry more data on the way back, therefore sending sensors answers takes longer time. An extreme is a command for taking a picture. Sending command to e-Puck last about 0.02 s, but the answer needs more than 0.2 s for a transfer.

As we mentioned, waiting for the answers takes even longer time than sending a command, so the waiting is performed in *Elib* asynchronously too.

Possible variability of waiting time force *Elib* to let the user decide how long *Elib* should wait for an answer. The *Elib* library does not limit the minimum waiting time too, because the convenient values differ according to the state of a battery on e-Puck, a strength of Bluetooth signal and so on.

Explanation (Timeout)**.** The waiting times for answers are in *Elib* called *timeouts*. In examples, which are introduced in Chapter 6, the waiting times are set to the bottom values, which allow commands to be confirmed without problems under good conditions.

summary

Let us sum up the facts. Sending of commands takes significant amount of time. Sent commands are queued, because otherwise they could be overwritten in the buffers. Receiving an answer takes even more time than sending a command from computer. We concluded to implement sending commands and receiving answers asynchronously, because the communication with e-Puck can be pointlessly block the processor of the host PC.

5.4 Asynchronous programming model (APM) for the *Elib* library

It was already mentioned that an asynchronous call of a function is convenient if the classical synchronous call let the processor waiting for instance to a device. In e-Puck's case the device is serial port.

Epuck Monitor [3] mentioned above implements synchronous communication with e-Puck using *BTCom*. Synchronous communication leads to freezing of Epuck Monitor.

On the other hand, asynchronous programming is much more complicated than synchronous programming. The goal of *Elib* is to hide the complications of APM and offer an interface which performs sending and receiving messages asynchronously and which allows a programmer to easily create applications for e-Puck. For example applications like Epuck Monitor. What means asynchronous implementation in case of *Elib*? What is done if the answer arrives in time and what if not? These questions are answered in the next section called Asynchronous programming model for *Elib*.

Remark (Callbacks)**.** Callbacks are functions, which are called after some procedure has finished.

Elib contains two main classes and it implements two interfaces.

- *Sercom*: public class, wraps serial communication.
  - $Sercom's$ basic interface – $Write(..)'s$ function arguments:
    - string command – specifies a type of sending command
    - an integer – the *timeout* for the command
    - Okf callback – function called if the answer is delivered in time.
    - Kof callback – function called if the *timeout* has elapsed and the answer did not arrive
    - class object – instance used to pass custom information to Okf callback
- *Epuck*: presents an instance of a e-Puck robot, uses *Sercom* internally.

[3] Download at `http://www.gctronic.com/files/e-puckMonitor_2.0_code.rar`

  - *Epuck's* basic interface
    - Based on *Sercom's* basic interface
    - the only difference to *Sercom's* interface – it implements string argument for each command implicitly
  - *IAsyncResult* interface
    - .Net interface
    - based on *Epuck's* basic interface

IAsyncResult interface also accepts *timeout*, an instance of *object* class and one callback, but it returns an implementation of *IAsyncResult* interface which will be introduced later. Simpler interface logic and the fact that *IAsyncResult* is broadly used interface are reasons for its implementation.

5.5 The *Sercom* class

An instance of *Sercom* allows a computer to asynchronously communicate with other devices over serial communication. The class accepts textual commands and parses textual answers. The commands and answers can be distinguished according to its first letter. Let us describe its structure.

Explanation. *Sercom* waits after each sent command until the answer arrives or its *timeout* elapses. Let us called this behaviour of waiting a *handshaking*.

Handshaking guarantees that the commands would not be sent too fast, because *Sercom* waits on e-Puck's answer. E-Puck picks up the command from its input buffer and then sends the reply. The short answers under good conditions do not slow down sending with *handshaking*, because they arrive under 0.04 s.

At the beginning of an *Elib's* development we thought about implementing both the *handshaking* and the *nonhandshaking* sending. The *nonhandshaking* sending does not wait for answer of the previous command. It sends the next command immediately. Motivation for *nonhandshaking* is to save time which is spent on waiting for the answers of the sent commands. Consequently, *nonhandshaking* has to ensure that the input buffer does not overflow. We have experienced that it is a non-trivial task to set the correct time gap after different types of commands. The final observation is that the *nonhandshaking* sending is even slower than the *handshaking* sending, because the gaps after sent commands have to be set at maximum in order to avoid a failure under all conditions. The gaps in *nonhandshaking* mode has too be unnecessarily large, because it must not let the input buffer of e-Puck overflow.

Handshaking solves the problem with synchronising sending and receiving the commands and their answers. It made us choose *handshaking* as the best implementation. E-Puck processes a received message in three steps. It picks the command from input buffer. It processes the command and then replies with an answer. For commands with long sending or processing phase it means the following consequence. If such command is sent, next commands has to wait longer time, because there is a danger of e-Puck's input buffer overflow.

Next advantage of *handshaking* is its simple and elegant implementation. It also does not depend on *BTCom* implementation so much. If *BTCom* had improved the times of processing the answers, in *nonhandshaking* mode it would have resulted into a change of a table, which stores minimal gaps after each command.

```
public class Sercom {
  public void Write(string command, RCallback okf, NRCallback kof,object state, double timeout) {
        //... the body of function
  }
}
public class Epuck {
  public void Stop(OkfActuators okf, NRCallback kof, object state, double timeout) {
        //example of interface directly based on BTCom's interface
  }
  public IAsyncResult BeginStop(double timeout, AsyncCallback callback, Object state) {
        //example of IAsyncResult interface implementation
  }
}
```

Figure 5.2: Public methods of *Sercom* and *Epuck*

5.5.1 The logical problems with a confirmation of commands

In *nonhandshaking* mode a lot of commands can be sent before the *timeout* of the first elapses. Let us introduce a common situation, which causes a serious problem.

Let us suppose, that *Elib* library has to send a lot of commands of one type. For example all the commands can control the front LED light. The problem consists of unwanted postponing a call of the *Kof* callback. Let us imagine that the answer for the first command is lost, but the second answer manage to be processed under *timeout* of the first command. It means that *Okf* callback is called and the user assumes that everything is fine for the first command. Furthermore, the problem can be postponed to the last command of the same type, because the reply to the second command can replace the reply of the third command and so on.

The commands can be logically different, although they are of the same type. The *Kof* callback can be called on a different command, because the first command can turn the front LED on with message "F,1", however the last command can switch the front led off with "F,0" command and the reply to both commands is "F$\backslash r \backslash n$. The reason is that *BTCom* replies to commands without their arguments.

Handshake mode does not suffer from calling the wrong callback.

Example. Let us imagine a situation where a user of Elib receives a *Kof* callback for "SetSpeed(0.5,0.5)" command. He decides to stop with the same command "SetSpeed(0,0)". He successfully receives the *Okf* callback. Unfortunately, the user does not know if the callback confirms "SetSpeed(0.5,0.5)" or "SetSpeed(0,0)".

The simple solution is to paste a command of a different type which prevents interchanging of the answers.

5.5.2 *Sercom's* inner implementation

The current *Sercom's* implementation has only four public functions. The constructor, the *Start* method, the *Dispose* method and the public method *Write*. See its interface in Figure 5.2.

The previous versions of *Elib* in addition have implemented public method for changing from *nonhandshake* mode to *handshake* mode and vice versa. The implementation of *nonhandshaking* mode uses one thread for each unconfirmed command in handshaking mode. It leads to instability of a program, if the Bluetooth communication breaks up.

Previous version

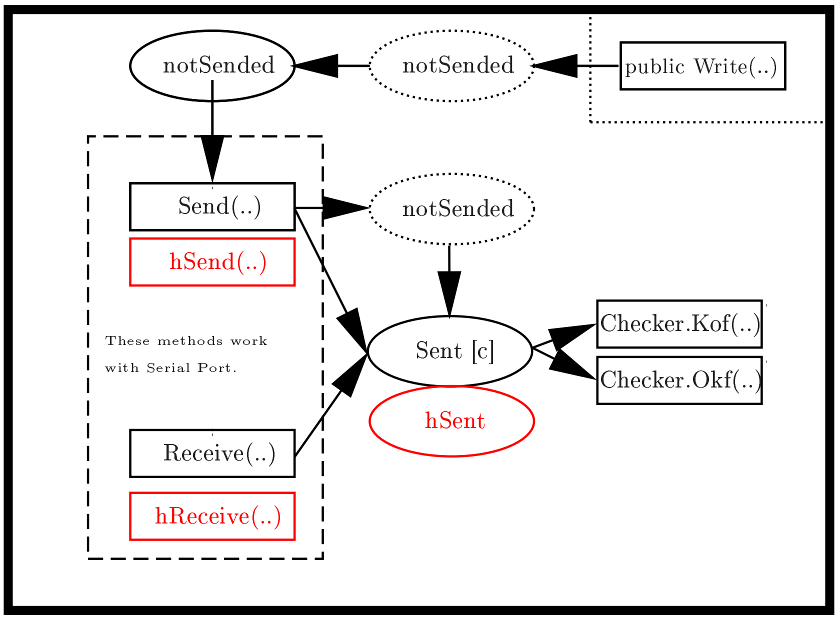


Figure 5.3: Old structure

In Figure 5.3 the schema of the inner implementation from the previous versions of *Elib* is illustrated. It shows functions and objects, which are used for sending and receiving the messages. Some methods have *nonhandshaking* and *handshaking* versions. *Nonhadshaking* methods, which are drawn in red in Figure 5.3, implement the same logic as *handshaking* methods except they allow to send and receive more commands at once.

The current *Sercom*

The current version abandoned the implementation of the *nonhandshaking* mode, because of the problems with receiving answers described in Section 5.1. It also solves the problem with redundant threads.

Inner structure

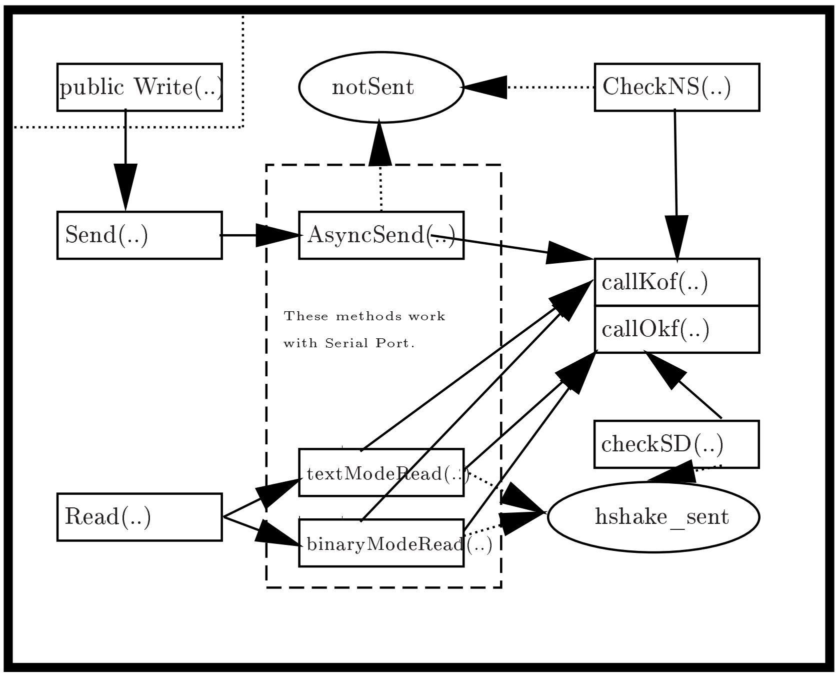


Figure 5.4: Methods for changing state of *ansGuard* instance

Elib currently uses two working threads and implements only the *handshake* mode. One thread running in the *checkNS* function checks whether the *timeout* of sent commands has not elapsed during waiting in *notSent* queue. The second thread checks whether the sent command stored in the *hshake_sent* variable has a valid *timeout* or the *timeout* has already elapsed. *Send* calls asynchronously *AsyncSend* function, which dequeues an instance of *ansGuard* from *notSent* queue. The *ansGuard* class is a wrapper of commands. The answers are read asynchronously in the *DataReceived* procedure.

Remark. Both methods *Send* and *DataReceived* use a so called .Net asynchronous delegate invocation. It enables simple asynchronous invocation using a minimum resources.

Explanation (Delegate)**.** Delegate[6] is a .Net strictly typed wrapper for function instances. An example of asynchronous function invocation using .Net delegate is depicted in Figure 5.5.

Remark. The *ThreadPool* class, which is created at start of the application, prevents from useless thread creation. The instance of the class keeps the threads after a function termination. The threads are handed over among different functions, which are called by asynchronous delegate invocation. *ThreadPool* uses a delegate to invoke the functions in different threads.

```
// The form of delegate declaration garantees, that only functions
// with no arguments and with void returning value
// can be saved in SendAsync delegate.
delegate void SendAsync();
void Send(object Sender, EventArgs ev) {
  //SendAsyncCall is a function, asyncCaller a delegate variable
  SendAsync asyncCaller = new SendAsync(SendAsyncCall);
  asyncCaller.BeginInvoke(null, null);
}
// This function match requirements of SendAsync delegate.
void SendAsyncCall(){
  //the body
}
```

Figure 5.5: Example of Asynchronous function invocation

All commands are wrapped by the *ansGuard* class. An instance of *ansGuard* is created in the *Write* method. The references of callbacks are stored in *ansGuard* together with their *state* argument. The *timeout* value in seconds is added to the current time and stored in *ansGuard*. The instance of *ansGuard* is immediately enqueued in the *notSent* queue.

The queue *notSent* implements an event *NonEmpty*, which calls the *Send* method whenever something is written to *Sercom* by the *Write* method. The *Sent* method is also called, if the *hshake_sent* variable is set to *null*. *Send* just calls *SendAsyncCall*. See a snippet in Figure 5.5 .

The *Read* method which is called asynchronously just calls *textModeCall* in the text mode and *binaryModeRead* in the binary mode. The only command, which is sent in the binary mode in *Elib*, is the command to get a picture. The picture is the only binary answer of *BTCom* too. A description of *binaryModeRead* is postponed after introducing *textModeCall* method.

Does *SendAsyncCall* implement *handshaking* mode? *SendAsyncCall* checks whether the previous command is received by checking whether *hshake_sent* is set to *null*. If the command is received and the *notSent* queue contains some *ansGuard*, then *SendAsyncCall* moves *ansGuard* from *notSent* to *hshake_sent* queue and sends the command.

The answers of *Sercom* are processed in the *Read* function which is the *DataReceived* handler of .Net *SerialPort*. The *DataReceived* handler is called asynchronously in .Net implementation of the *SerialPort* class if at least one new character is delivered to serial port input buffer.

Every time the *textModeCall* is raised, the new characters are step by step added to the *ans* container. If the whole answer is stored in *ans*, *ans* is cleared. The found *answer* is checked against *ansGuard* stored in *hshake_sent*. If it matches, the *Okf* callback is called and the *hshake_sent* variable is set to *null*. Also the *Send* function is called via the *Received* delegate. If the answer does not match, it is thrown away.

If the binary mode is on, the *binaryModeRead* procedure is called from the *Read* function. The *binaryModeRead* does not read bytes step by step. At first it reads first three bytes where width, height and mode of a picture is stored. The total size of a picture is computed from the first three bytes of the picture. After receiving the first three bytes the function reads remaining bytes. If the picture is transferred before the *timeout* elapsed, then the *Okf* callback is called. The *Kof* callback is called otherwise. If *Sercom* is receiving the picture and the time out elapses, *Sercom* continues receiving the picture. *Sercom* stops the receiving only if the image is complete or if no bytes have been received longer than 0.1 seconds. The *Kof* callback of "GetImage" command is the only *Kof* callback that can have the answer at disposal, so the picture is passed to *Kof* callback too.

The *Elib* library receives image after *timeout*, because of implementation of *BTCom*, which can not receives command during sending a picture. After *Kof* or *Okf* call in binary mode instance of *ansGuard* is removed from *hshake_sent* and *Send* method is invoked using *Received* event. The last action of *binaryModeRead* is to switch the flag from binary mode to text mode.

Remark. The *BTCom* switches back to the text mode automatically, because the "GetImage" command is sent in binary mode to e-Puck together with an empty byte, which switches *BTCom* on e-Puck immediately after sending the picture back to the text mode.

Sending and receiving commands answers works well until an answer is lost. *Timeouts* ensure that the stacked instance of *ansGuard* are cleared after its *timeout* elapses. The instance of *ansGuard* is stacked either in *notSent* queue or in *hshake_sent*. Both methods, which implement the removal of an *ansGuard*, run in worker threads and also implement sophisticated system of waking and putting their threads to sleep, which prevents the threads from wasting the CPU time during their waiting to the *timeout* expiration. The *checkSD* function looks after the *ansGuard* in *hshake_sent* and the *checkNS* function guards the *ansGuards* in the *notSent* queue.

Remark. For details of *Sercom* implementation see the code and the reference documentation in enclosed CD.

5.5.3 *Sercom's* interface

Let us present the interface of *Sercom* and its public members. Figure 5.6 lists all public functions of *Sercom*. The *Write* function can be used only after the *Start* function is called. The *Start* function is called only once and it opens serial port and initialise the *BTCom* communication. The serial port is opened with parameters set in the constructor of *Sercom* or with default parameters specified in *Sercom* in *Sercom's* constants. The *Dispose* method closes the serial port and prepares *Sercom* for a garbage collection.

```
public class Sercom:IDisposable{
  public void Write(string command, OkfCallback okf, KofCallback kof,object state, double timeout);
  public void Start();
  public Sercom(string portName, int serialPortWriteTimeout, int serialPortReadTimeout);
  public Sercom(string portName) : //with default constants
        this(portName, defWriteTimeout, defReadTimeout) { }
  public int NotAnswered{get;}
  public int NotSent{get;}
  public byte[] LastImgBytes{get;}
  public bool FullImgBytes{get:}
  public int ModeImg{ get:}
  public int HeightImg{ get;}
  public int WidthImg{ get;}
  public void Dispose();
}
```

Figure 5.6: Public functions of *Sercom*

There are also several properties available. The *NotAnswered* and *NotSent* properties show how much is the instance *Sercom* class occupied. *NotSent* returns only the length of the *notSent* queue. *NotAnswered* is at most by one greater than *NotSent*, because it returns number of *ansGuards* in *notSent* plus one if is an *ansGuard* in *hshake_sent*.

The remaining properties are attributes of the last taken image. They are valid, if the *FullImgBytes* is set to true. Otherwise the properties do not have to be from the last image. If the *Okf* was called on the last taken image then *FullImgBytes* is set to *true*. The property comes in use if the *Kof* callback was called, because it can be called from *notSent* queue when the bytes of image got lost, or the image is all right, but it was received too late. Width, height, mode and bytes are used as parameters for bitmap, which is created in the *Epuck* class.

5.6 The *Epuck* class

An instance of *Epuck* offers a virtual representation of e-Puck, which interfaces all sensors and actuators of e-Puck to user. The *Epuck* class also hides Bluetooth communication from the user.

Epuck uses *Sercom* internally and makes controlling of robot much more comfortable. *Sercom* treats only the command "GetImage" specially, because it is the only one command processed in binary mode and it has big time requirements. *Sercom* does not differentiate between other commands. *Epuck* differentiates between commands a lot. It tights *Epuck* with concrete implementation of *BTCom*, but it also allows to process the answers and to offer a typing of the returning values, which increases applicability. Main goal of *Epuck* class is to offer a good interface.

In this chapter the usage of *Sercom* in *Epuck's* class is presented together with additional methods for implementation of the interfaces. The reasons, which lead to implementing two independent interfaces for controlling e-Puck robot through *Epuck* class, are presented. Last but not the least the used technology of .Net is mentioned.

The *Epuck* class brings nothing more than interface and debugging tools. We describe the interface on examples as much as possible. There are several introductory examples in these chapter, which explain the properties of the interface. Furthermore, guide lines for the *Epuck* class and the debugging tools can be found in Chapter 6. The following sections present the internals of *Epuck's* methods.

5.6.1 *Basic* interface: Okf and Kof callbacks

Epuck's basic interface just wraps the *Sercom* class. Methods of *Epuck's* basic interface differ in callback parameters. Types of callbacks differ according commands, which were sent by functions of *Epuck's* basic interface. We divide callbacks and *Epuck's* methods in following categories.

- Functions for actuators and sensors with string return values
- Functions for sensors returning array of integers
- Function for sensors returning an image

commands for actuators, sensors with string return value and sensors, which return an array of integers. *Epuck* treats pictures differently too, because it passes to callback a *Bitmap* instance.

Although there are four kinds of commands, the implementation differs only in processing the answer. The processing functions *actuators*(..), *intArrSensors*(..), *stringSensors*(..) are all similar to *GetImage*(..) method.

There are only two differences between *GetImage*(..) and other functions. Other functions have to parse their answers in Okf(..) functions differently and they do not parse answer in Kof(..) callback at all.

The differences between *GetImage*(..) and the processing functions consist in using different delegates for Okf(..) functions. All Kof(..) functions fit to the same delegate *KofCallback*. Only the *GetImage*(..) function defines its own delegate *OkfKofCamsSensor*, which *GetImage*(..) also uses for the Okf(..) functions.

Remark. As you can see in Figure 5.7 on lines 14-18 and 19-25, the callbacks are wrapped in lambda functions [2]. Instead of passing Okf(..) and Kof(..) directly

```
public class Epuck{
  public void Stop(OkfActuators okf, NRCallback kof, object state, double timeout) {
        actuators(Commands.c_Stop(),okf,kof, state, timeout, "Stop(..)");
  }
  public void GetIR(IntSensorsOk okf, NRCallback kof, object state, double timeout) {
        IntArrSensors(Commands.c_Proximity(),8,okf,kof,state,timeout,"GetIR(..)");
  }
  public void GetBTComVersion(stringSensors(..)Ok okf, NRCallback kof, object state, double timeout) {
        stringSensors(..)(Commands.c_Version(), okf, kof, state, timeout, "BTComVersion(..)");
  }
  public void GetPicture(CamSensor okf, CamSensor kof, object state, double timeout) {
        checkArgs(okf, kof, timeout);
        logf(Action.call, f_name);
        ser.Write(Commands.c_GetImage(),
        (ans, data) => {
          okf(parsingBitmap(ser.LastImgBytes, ser.WidthImg, ser.HeightImg, ser.ModeImg), data);
          logf(Action.ok, f_name);
        },
        (data) => {
          if (ser.FullImgBytes)//a whole img was captured
            kof(parsingBitmap(ser.LastImgBytes, ser.WidthImg, ser.HeightImg, ser.ModeImg), data);
          else
            kof(null, data);//img is demaged
          logf(Action.ko, fname);
        },
        state,timeout);
  }
}
```

Figure 5.7: Four types of *Epuck* control functions

to $Sercom.Write(..)$ function, $Okf(..)$ and $Kof(..)$ are called within the lambda functions. The lambda functions allow logging and parsing the answers.

A function *checkArgs* on line 12 of *GetPicture*(..) does not allow to pass invalid arguments to *Epuck* functions. *Timeout* has to be a positive *double* value, Okf and Kof delegates has to be defined and not set to *null*.

$Okf(..)$ callbacks should be used for implementing the desired algorithm. On the other hand, $Kof(..)$ callbacks should perform repair actions in order to get into a valid state. Users of *Elib* should always have in mind, that $Kof(..)$ callbacks can be raised very often for low *timeouts*, but the call of $Kof(..)$ callback signalizes a serious error state. See Section 6.2 for more examples of how to use $Okf(..)$ and $Kof(..)$ callbacks.

5.6.2 The *IAsyncResult* interface

Two callbacks and a schizophrenic logic of the $Okf(..)$ and $Kof(..)$ implementation are not convenient for exploring e-Puck's sensors and actuators, however they allow e-Puck to easily recover from every situation.

```
interface IAsyncResult{
  public Object AsyncState { get; }
  public Boolean CompletedSynchronously{get; }
  public WaitHandle AsyncWaitHandle { get; }
  public Boolean IsCompleted { get; }
}
```

Figure 5.8: *IAsyncResult* interface

Motivation for implementing the *IAsyncResult* is its clear usage and its proven usability. See Chapter 6 for *IAsyncResult* introduction and examples. The requirements of interface is shown in Figure 5.8

See the *IAsyncResult* example in Figure 5.9, which uses an instance of *Epuck* to get IR sensor values from a real e-Puck. The *timeout* is the only parameter of *BeginGetIR*(..), which has nothing to do with *IAsyncResult* interface.

```
IAsyncResult ar = ada.BeginGetIR(timeout, callback, state);
int[] IRSensors = ada.EndGetFtion(ar); //if callback == null
```

Figure 5.9: Usage of *IAsyncResult*

An asynchronous operation implemented by *IAsyncResult* needs two functions. The first function usually starts with "*Begin*" prefix and the second starts with "*End*" prefix. This convention is strictly respected in *Elib*. The function *BeginGetIR*(..) is example of the first function and *EndGetFtion*(..) of the second function.

If the *callback* delegate is null, then we need the *EndGetFtion*(*ar*) function in order to be sure that the real e-Puck has sent the values of IR sensors. See Figure 5.9. The call of *EndGetFtion*(..) blocks the current thread and waits synchronously until the real e-Puck is stopped.

If the *callback* functions are defined, they are raised after the *BeginGetIR*(..) function has finished. Usually, if *callback* delegate is used, then the *EndFtion*(..) function call is not necessary.

The only possibility of *IAsyncResult* interface to signal an error is raising an exception. The exception is passed to callback or the "End" function raises it.

The *ar* instance of the *IAsyncResult* interface allows the *EndGetFtion*(..) to wait to the end of *BeginGetIR*(..) function. The instance is also used for passing user defined data in *state* object to the callback function, because *ar* instance has an *ar.AsyncState* reference to *state* object. Another important feature of *ar* is, that it allows *EndGetFtion* function to receive the data from *ar*. For example an integer array of IR sensors' values can be extracted from *ar*, which illustrates Figure 5.9.

5.6.3 *IAsyncResult* implementation

Jeffrey Richter [11] made a nice example of two classes, which implement *IAsyncResult* interface. I used his classes and modified them according to *Elib*'s needs. The first class is *AsyncNoResult* and implements *IAsyncResult* for actuators. The second class is generic *AsyncResult* $< T >$ subclass of *AsyncNoResult*. It implements *IAsyncResult* interface for sensors.

Explanation (Generic class)**.** Generic class $C < T >$ with type T means, that the type T is chosen at compile time.

Instances of *AsyncResult* $< T >$ are used for *String*, *Bitmap* as well as integer array answers with a different generic parameter T.

The idea of the *IAsyncResult* interface is that a programmer does not have to know which class implements the *IAsyncResult*. They use *IAsyncResult* type in a code. See Figure 5.9. On the other hand, the functions *BeginGetIR* and *EndGetFtion* have to know the type, which is passed to *IAsyncResult* object. "Begin" function in *Elib* creates the instances *AsyncNoResult* for actuators and *AsyncResults* $< T >$ for sensors. "End" functions in *Elib* wait until the end of "Begin" functions and throw exceptions if the commands have not been delivered in time. If the answer is delivered in time and if the command has requested a sensor value, the "End" function returns the desired sensor's values.

Obligatory members of *IAsyncResult*

Let us examine in detail the *IAsyncResult* interface from Figure 5.8.

- *AsyncState* is an object used as argument for a callback function, which is passed to "Begin" function. If no callback is passed, the *AsyncState* is not used. *AsyncState* is useful for passing information to the callback.

- *AsyncWaitHandle* is a synchronisation primitive which allows waiting until the operation started in "Begin" function is done. *AsyncWaitHandle* is used in "End" function if the "End" function is called and the operation is still running.
- Flag *CompletedSynchronously* tells whether *AsyncWaitHandle* has been used to wait to the end of the operation. *CompletedSynchronously* is always set to false if callback is used.
- *IsCompleted* tells whether the operation from "Begin" function terminated or not.

The "Begin" and "End" functions

Let us describe public methods of *AsyncNoResult*, which are used to implement "Begin" and "End" functions. *IAsyncResult's* properties are not enough to implement *BeginGetIR*, *EndGetFtion* or any other "Begin" and "End" functions. *BeginGetIR* method uses a constructor of *AsyncNoResult*. It also uses the *SetAsCompleted* function and the *EndInvoke* function both from *AsyncResult* subclass. We will explain the interesting and crucial part of both classes. For implementation details see code in Figure 5.12 and 5.11.

Remark. A constructor of *AsyncNoResult* only sets the field members, which are not changing during *AsyncNoResult* existence. The implementation of *SetAsCompleted*(..) method from *AsyncNoResult* raises an exception to *AsyncNoResult*. The function *SetAsCompleted*(..) from *AsyncResult* overloads *SetAsCompleted*(..) method from its base class *AsyncNoResult* and adds a possibility to set the results instead of raising the exception. Third overload is implemented in *AsyncResult* so that the *GetPicture*(..) command can set both exception and the result at one time. *EndInvoke*(..) is called from "End" function and does all its logic. It checks whether the called operation has finished. If the operation is still pending, it sets up a new *AsyncWaitHandle* and waits to its termination. At the end it throws an exception if one has been set up, or it returns a result if *EndInvoke*(..) from *AsyncResult* was called.

An important feature of *AsyncNoResult*, which is inherited by *AsyncResult*, is the creation of an *EventWaitHandle's* instance. It is not created in all cases and it is created at most once during an *AsyncNoResult's* life cycle. An instance of *EventWaintHandle* is created only if the user explicitly called get method for *WaitHandle* or if the *EndInvoke* method was called and the operation is still pending. It means that if you use callback and do not call "End" function then no instance of *EventWaitHandle* is created. Let us stress that *EventWaitHandle* is provided from an operating system and its instantiation is relatively slow.

The *name* field seems useless, but it allows "Begin" function to put its name in it. The name can be used in logging or debugging, which will be described in following paragraphs devoted to design of "Begin" and "End" functions.

Let us note that *AsyncResult* adds no crucial logic except setting a result in *SetAsCompleted* function and returning the result in *EndInvoke* function. *AsyncResult* uses its base class *AsyncNoResult* to implement the logic.

```
public class AsyncResultNoResult : IAsyncResult {
  // Fields set at construction which never change
  readonly AsyncCallback m_AsyncCallback;
  readonly Object m_AsyncState;
  // Fields which do change after operation completes
  const Int32 c_sp = 0;//StatePending
  const Int32 c_scs= 1;// StateCompletedSynchronously
  const Int32 c_sca = 2;//StateCompletedAsynchronously
  Int32 m_CompletedState = c_sp;
  // Field that may or may not get set depending on usage
  ManualResetEvent m_AsyncWaitHandle;
  // Fields set when operation completes
  Exception m_exception;
  // Name makes debugging easier in Elib. It shows, which ↩
     command was used.
  string name;
  public string Name {
    get { return (name != null) ? name : ""; }
    set { name = value;     }
  }
  public AsyncResultNoResult(AsyncCallback asyncCallback, ↩
     Object state, string name_) {
    m_AsyncCallback = asyncCallback;
    m_AsyncState = state;
    name = name_;
  }
  public AsyncResultNoResult(AsyncCallback asyncCallback, ↩
     Object state) :
    this(asyncCallback, state, null) { }
  public void SetAsCompleted(Exception exception, Boolean ↩
     completedSynchronously) {
    // Passing null for exception means no error occurred.
    // This is the~common case
    m_exception = exception;
    // The~m_CompletedState field MUST be set prior calling ↩
       the~callback
    Int32 prevState = Interlocked.Exchange(ref ↩
       m_CompletedState,
      completedSynchronously ? c_scs : c_sca);
    if (prevState != c_sp)
      throw new InvalidOperationException("You can set a ↩
         result only once");
    // If the~event exists, set it
    if (m_AsyncWaitHandle != null) m_AsyncWaitHandle.Set();
    // If a callback method was set, call it
    if (m_AsyncCallback != null) m_AsyncCallback(this);
  }
```

Figure 5.10: *AsyncNoResult's* members and constructors

```
public void EndInvoke() {
  // This method assumes that only 1 thread calls EndInvoke
  // for this object
  if (!IsCompleted) {
    // If the~operation isn't done, wait for it
    AsyncWaitHandle.WaitOne();
    AsyncWaitHandle.Close();
    m_AsyncWaitHandle = null;  // Allow early GC
  }
  // Operation is done: if an exception occured, throw it
  if (m_exception != null) throw m_exception;
}

//region Implementation of IAsyncResult
public Object AsyncState { get { return m_AsyncState; } }
public Boolean CompletedSynchronously{get {
  return Thread.VolatileRead(ref m_CompletedState) == ic_scs; }}
public WaitHandle AsyncWaitHandle {
  get {
    if (m_AsyncWaitHandle == null) {
      Boolean done = IsCompleted;
      ManualResetEvent mre = new ManualResetEvent(done);
      if (Interlocked.CompareExchange(ref m_AsyncWaitHandle,
         mre, null) != null) {
        // Another thread created this object's event; dispose
        // the~event we just created
        mre.Close();
      } else {
        if (!done && IsCompleted) {
          // If the~operation wasn't done when we created
          // the~event but now it is done, set the~event
          m_AsyncWaitHandle.Set();
        }
      }
    }
    return m_AsyncWaitHandle;
  }
}
public Boolean IsCompleted {
  get {return Thread.VolatileRead(ref m_CompletedState) != c_sp; }}
//endregion Implementation of IAsyncResult
}
```

Figure 5.11: *AsyncNoResult* internals

```
public class AsyncResult<TResult> : AsyncResultNoResult {
  // Field set when operation completes
  TResult m_result = default(TResult);
  public AsyncResult(AsyncCallback asyncCallback, Object ↩
      state,string name) :
    base(asyncCallback, state,name) { }
  public AsyncResult(AsyncCallback asyncCallback, Object ↩
      state) :
    this(asyncCallback, state, null) { }
  //enable to set Result. AsyncNoResult enables to set ↩
      exception.
  public void SetAsCompleted(TResult result, Boolean ↩
      completedSynchronously) {
    // Save the asynchronous operation's result
    m_result = result;
    // Tell the base class that the operation completed
    // sucessfully (no exception)
    base.SetAsCompleted(null, completedSynchronously);
  }
  // Allows to set both the exception and the result. Added ↩
      for GetImage(..).
  public void SetAsCompleted(TResult result, Boolean ↩
      completedSynchronously, Exception exception) {
    m_result = result;
    base.SetAsCompleted(exception, completedSynchronously);
  }
  new public TResult EndInvoke() {
    base.EndInvoke();// Wait until operation has completed
    return m_result; // Return the result (if above didn't ↩
        throw)
  }
}
```

Figure 5.12: *AsyncResult* $< T >$

Implementation of "Begin" and "End" functions

The "Begin" and "End functions are the key functions of the *IAsyncResult* implementation. All the "Begin" and "End" functions' implementations are similar. They are implemented similarly to functions *BeginGetIR's* and *EndGetFtion's* from Section 5.9. The implementation of *BeginGetIR's* and its "End" function is shown in Figure 5.13. All "Begin" functions use *Epuck's* interface with the *Okf* and *Kof* callbacks introduced in Section 5.1.

```
public IAsyncResult BeginGetIR(double timeout, AsyncCallback ↩
    callback, object state) {
  AsyncResult<int[]> a =
  new AsyncResult<int[]>(callback, state, logFunctionNames["↩
     BeginGetIR(..)"]);
  GetIR(receivedSensors<int[]>, failed, a, timeout);
  return a;
}
static T EndSensors<T>(IAsyncResult ar) {
  AsyncResult<T> a = (AsyncResult<T>)ar;
  return a.EndInvoke();
}
public int[] EndGetFtion(IAsyncResult ar) {
  return EndSensors<int[]>(ar);
}
```

Figure 5.13: An example of "Begin" and "End" function for sensors.

The "Begin" function creates an instance *a* of the class *AsyncNoResult* respectively an *AsyncResult* instance. The function passes the supplied callback with its *state* object to *a*. Third argument of *a's* constructor is a name of the used function.

In Figure 5.13 *BeginGetIR*(..) instantiates *AsyncResult* < *int*[] >, because *BeginGetIR*(..) expect array of *ints* as its answer. The *Name* field of *AsyncResult* is filled with the *string* value from *logFunctionNames* dictionary. The dictionary values are by default set to their key values, so the *Name* in *BeginGetIR*(..) function is filled with "BeginGetIR(..)" value. The *a* instance is passed as the *state* argument and is at disposal to the *failed*(..) and *receiveSensors*(..) functions. The function *receiveSensors*(..) is called with generic parameter *int*[], because the *GetIR* values expect an array of *ints* from sensor. Sensor's methods as well as actuator's methods use the *failed*(..) function as its *Kof*(..) callback. An exception is the *GetImg*(..) function which uses the *failedBitmap*(..) callback instead. The last action of each "Begin" function is returning the instance *a*. Functions, which control actuators, call the *received*(..) function as the *Okf*(..) callback. See an example in Figure 5.14.

Let us shortly introduce the "End" functions before we will focus on the *Okf*(..) and *Kof*(..) callbacks used in the "Begin" functions. "End" functions have two tasks of the "End" functions. The first one is to cast the *IAsyncResult* argument to the correct type of *AsyncResult* or to the *AsyncNoResult* class. The second task is to call the *EndInvoke*(..) method. If the "End" function is called with an *IAsyncResult* argument from a sensor command, the value of the sensor is returned from *EndInvoke*(..). See Figure 5.13. If the *IAsyncResult* argument is passed from the "Begin" function, which controls an actuator, then *EndInvoke*(..) is called and nothing is returned. See Figure 5.14.

```
public IAsyncResult BeginMotors(double leftMotor, double ↩
   rightMotor,
       double timeout, AsyncCallback callback,Object state) {
 AsyncResultNoResult a =
   new AsyncResultNoResult(callback, state,logFunctionNames["↩
      BeginMotors(..)"]);
 Motors(leftMotor, rightMotor, received, failed, a, timeout);
 return a;
}
public void EndActuators(IAsyncResult ar) {
 AsyncResultNoResult a = (AsyncResultNoResult)ar;
 a.EndInvoke();
}
```

Figure 5.14: An example of "Begin" and "End" function for actuators.

5.6.4 Callback: Alternative to the "End" function

The callbacks are called after the delivery of the answer or after the *timeout* expiration. The callback allows to process the result of the operation. In the *IAsyncResult* interface the results are set via calling the *SetAsCompleted* method on the *AsyncNoResult* respectively *AsyncResult* instance. If the operation fails, the *Kof* callback *failed* is called. In the *failed* function the *SetAsCompleted* method is called and an exception is passed to the first argument of *SetAsCompleted* overridden function. See the function *failed* in Figure 5.15 on the first line.

The only *Epuck's* method that can return both the exception and the result is the *BeginGetPicture*(..) function. The function $receivedSensors < Bitmap >$ is used in the *Okf* delegate *BeginGetPicture*(..), and the special function *failedBitmap*(..) is called if an exception has been raised. The *Kof*(..) callback raises a special exception, if the image was captured and has been delivered after timeout expiration. Its implementation is shown in Figure 5.15.

If the answer is delivered in time, then the *Okf* callback is called. The sensor's callback $receivedSensors < T >$ passes the answer of the type T to its first argument. The actuators callback *received* just passes *null* to its first argument, which indicates no exception was raised. See the code below in Figure 5.16.

```
static void failed(object asyncNoResult) {
  AsyncResultNoResult ar = (AsyncResultNoResult)asyncNoResult;
  ar.SetAsCompleted(new ElibException(ar.Name + " command hasn't been confirmed in timeout"), false);
}
static void failedBitmap(Bitmap pic,object asyncResult) {
  if (pic != null) {
    AsyncResult<Bitmap> ar = (AsyncResult<Bitmap>)asyncResult;
    ar.SetAsCompleted(pic, false, new ElibException(ar.Name +
      " command GetPicture has not been confirmed in timeout, but picture is still available in AsyncResult<Bitmap>"));
  } else
    failed(asyncResult);
}
```

Figure 5.15: Kof callbacks for $IAsyncResult$

```
static void received(object asyncNoResult) {
  AsyncResultNoResult ar = (AsyncResultNoResult)asyncNoResult; ar.SetAsCompleted(null, false);
}
static void receivedSensors<T>(T b, object asyncResIntArr) {
  AsyncResult<T> a = (AsyncResult<T>)asyncResIntArr;
  a.SetAsCompleted(b, false);
}
```

Figure 5.16: Okf callbacks for $IAsyncResult$

5.7 A summary of Elib interfaces

In the previous three subsections the implementations of the two interfaces of *Epuck* and one *Sercom's* interface have been described.

IAsyncResult is built on the basic *Epuck's* interface with the typed *Okf*(..) and *Kof*(..) callbacks. The basic *Epuck* interface is itself built on *Sercom's* *Okf* / *Kof* interface. The purpose of the interfaces is to make programming e-Puck over Bluetooth easier. The *IAsyncResult* is the most comfortable of the three interfaces. The use of *Sercom's* interface needs good knowledge of BTCom protocol and it forces user to process every answer of *BTCom*.

Let us compare the possibilities and limitations of the interfaces against each other. The *Sercom's* interface and *Epuck* basic interface with *Okf* and *Kof* have the same limitations and possibilities for the e-Puck running *BTCom* version 1.1.3.

Every program written using *Epuck's* basic interface can be written in *Sercom* by implementing *Epuck's* interface as it is in *Elib*. On the other hand, every program, which uses *Sercom* interface and communicates with *BTCom* version 1.1.3 on e-Puck, can be written using *Epuck's* basic interface. *Epuck's* basic interface is a specialization of *Sercom* for concrete version of *BTCom* protocol.

IAsyncResult limits the role of the *Kof* callback, but does not limit the role of the *Okf*(..) callback. If a user wants to implement an *Okf* logic using the *IAsyncResult* interface, he uses callback from *IAsyncResult* interface, which is passed to the "Begin" function. The callback is called after the *received* respectively *receivedSensor* $< T >$ (..) function call the *SetAsCompleted*(..) method on an instance *a* of *IAsyncResult*. Remind the implementation in Figure 5.16. The *received*(..) and *receivedSensor* $< T >$ (..) functions are called as the *Okf* callbacks from the *Epuck's* basic interface. See Figure 5.14 and Figure 5.13 for the confirmation. To conclude the *IAsyncResult* callback is invoked in the *Okf*(..) callback of *Epuck's* basic interface.

```
static void SimulatingKof_over_IAsResult(Epuck ada) {
  //the timeout is too small!
  ada.BeginGetImage(0.001, okf, ada);
}
```

Figure 5.17: Start of the behaviour.

The limitation of *IAsyncResult* arises from setting an exception in the *Kof*(..) callbacks, which are used in the *IAsyncResult's* implementation. The only difference is, that the functions in Figures 5.17 and 5.18 use the *kof*(..) callback function from Figure 5.19 which implements the *Kof* logic in spite of the functions use *IAsyncResult* interface.

The *Kof* logic should be implemented if an exception is raised. The exception is raised in callback passed to *IAsyncResult* if the answer is not delivered in timeout. The nearest place, where it can be caught is in the call of the *EndInvoke* method. The called *EndInvoke* method is associated with the *IAsyncResult* instance. The common way for a user to invoke the *EndInvoke* method is to call the "End" method on the instance of *Epuck* with the *IAsyncResult* object as its parameter. This technique is used in Figure 5.18. The call of *ada.EndGetImage*(*ar_*) gets the image and the callback function continues or the *EndGetImage*(..) function throws an exception which is caught by *try−catch* block. In *catch* block the user defined *kof* function is invoked.

```
 1 //It can be only a callback of BeginGetImage,
 2 //because the ar paramater has to contain a Bitmap
 3 //See 8. row!
 4 static void okf(IAsyncResult ar_) {
 5   if (!endf) {
 6     Epuck ada = (Epuck)ar_.AsyncState;
 7     try{
 8       Bitmap b = ada.EndGetImage(ar_);//no EventWaitHandle created
 9       IAsyncResult ar = ada.BeginMotors(-1, 1, 0.1, null, null);// some work
10       //simulate image processig
11       Thread.Sleep(20);
12       ada.EndFtion(ar);
13       //the timeout is too small!
14       ada.BeginGetImage(0.01, okf, ada);
15     } catch (ElibException) {
16       //has to be fixed in kof
17       ada.BeginStop(0.1, kof, ada);
18     }
19   } else
20     endconfirmed.Set();
21 }
```

Figure 5.18: The *Okf* callback with too small timeout.

The logic of the *Kof*(..) function from the basic *Epuck's* interface is preserved, because the user-defined *kof*(..) is called in the same case when the *Kof*(..) is called. Both are invoked if the *timeout* expires. On the other hand, the *kof*(..) uses an extra command to be invoked. In Figure 5.18 it is *ada.BeginStop*(..) command, which invokes *okf*(..). The *Stop*(..) function would be the first command in *Kof*(..) logic implementation of *Epuck's* basic interface. The second difference is that the same *Kof*(..) callback is called from *EndGetImage*(..) and from the commands in the body. In this example the same *kof*(..) is called for *EndGetImage*(..) and for *EndActuators*(..), which is the pair function of the *BeginMotors*(..) command from the body of the *okf*(..) function. The exception's origin can be found out from the *message* property of the exception, where the *Name* property of the *AsyncNoResult* instance is used.

Summary

The unknown source of an exception as well as the extra invocation command are minor problems, because there is usually single $Kof(..)$ callback that consists of more than one command. Therefore, *IAsyncResult* interface of *Epuck* class can be considered as powerful as *Epuck's* basic interface.

Remark. The lost answer usually completely breaks the logic of Okf implementation. If it happens, the appropriate solution is to put e-Puck into the starting position of the behaviour. Usually, commands to stop e-Puck are sent and a message is sent to the user or to the log file. See Figure 5.19.

```
 1 //It can be called from any function,
 2 //because we call only EndFtion!
 3 //It can be applied to every IAsyncResult in Elib.
 4 //See 7. row!
 5 static void kof(IAsyncResult ar_) {
 6   if (!endf) {
 7     Epuck ada = (Epuck)ar_.AsyncState;
 8     ada.EndFtion(ar_);
 9     try {
10       IAsyncResult ar = ada.BeginStop(to, null, null);//do ↩
            the repair actions
11       ada.EndFtion(ar);
12       Console.WriteLine("The~problem is fixed. Lets call okf!↩
            ");
13       ada.BeginGetImage(0.1, okf, ada);
14     } catch (ElibException) {
15       ada.BeginStop(0.1, kof, ada);
16     }
17   } else
18     endconfirmed.Set();
19 }
```

Figure 5.19: Simulation of Kof using *IAsyncResult*

Demands of *Epuck* and *Sercom* class on operating system

Let us explore the demands and the load of operating system (OS) when we use these interfaces. The more sophisticated the interface is, the bigger demands on OS it has. It is a consequence of building one interface upon the other. Let us stress that every program written using *IAsyncResult* interface, can be written with less or equal system resources using *Sercom's* interface. On the other hand, the programmers, who use more complicated interface, for example the *Sercom* class, do not usually find better way than the one that is designed by the richer interface like *IAsyncResult*.

In fact, *IAsyncResult* is not much more demanding than *Sercom* interface, if it is not misused. The only field, where *IAsyncResult* does not keep up with *Sercom's* interface, is an extremely frequent calling of *Kof* callbacks. *IAsyncResult* throw an exception for every *Kof* callback invocation, whereas *Sercom's* interface just needs to invoke a callback. *IAsyncResult* also needs to call other functions, but this is quite fast. The raising and catching an exception is significantly slower. On the other hand, such behaviour that extensively uses *Kof* callbacks has very special purpose or is a bad programming style. See Chapter 6 for guidelines how to use the interfaces.

The only resource, which can be extensively used in *IAsyncResult*, is *EventWaitHandle*. *EventWaitHadndle* is a synchronisation primitive provided by OS. There is at most one *EventWaitHandle* created for each instance of *IAsyncResult* in *Elib*. On the other hand, by using callbacks in *Elib's* *IAsyncResult* implementation, the creation of *EventWaitHandles* is completely avoided.

Let us note that *Sercom's* and the simple *Epuck's* interface are almost identical and therefore they have almost identical performance. The *Epuck* class adds answers processing and allows logging. The user would do the processing of the answers anyway. If the logging is off, the one additional "if statement" in method adds no overhead. Preferring one interface over another in *Elib* does not significantly influence the performance of application.

Remark. Let us remind, that we use *Threadpool* for callbacks, so each callback does not start new thread. The *Threadpool* distributes the free threads to functions.

We have tested the performance of *Elib*. Let us introduce the results and parameters of *Elib* library. The size of the compiled *Elib* library is less 50 KB. Using *IAsyncResult* in "Bull" behaviour *TestElib* consumes 8512 KB of memory. Compare it with a simple console application, which needs 4580 KB of memory, or with Google Chrome, which consumes more than 30000 KB of memory. During three performance tests of *SimulatingKof_over_IAsResult*(..), the number of threads of the *TestElib* application did not exceed 10. See Section 5.19. Moreover, the threads were usually unused in *ThreadPool* system class which has used only one or two. The two worker threads were running of course all the time.

The Table 5.1 was obtained from measuring using the *ConsoleTestSensorsTimeout* and *ConsoleTestActuatorsTimeout* functions with *timeout* 10 seconds. It shows that most of commands were delivered under 0.1 seconds. On the contrary, the commands to take a picture or to set camera parameters lasted almost half a second. The reason is that the camera is very demanding device and it produces a lot of data. The special commands for calibrating IR sensors and for resetting e-Puck lasted even longer. The "reset" command has even exceeded the *timeout*, which means that part of its answer was lost.

The values in the table were measured from 10 independent measurements split into two tests. The first test consists of the first six measurements which used intentionally e-Puck with a not fully charged battery. The second test comprises another 4 measurements performed on e-Puck with a recharged battery. The first six measurements used e-Puck with a not fully charged battery. The last four tests were performed on e-Puck with a recharged battery.

Command string	Average (1. test)	Variance (1. test)	Average (2. test)	Variance (2. test)
GetHelpInfo(..)	0.13873	0.00036	0.12162	0.00007
GetAccelerometer(..)	0.07017	0.00009	0.06393	0.00009
GetVersionInfo(..)	0.04153	0.00028	.05715	0.00055
GetCamParams(..)	0.04927	0.00001	0.05073	0
GetEncoders(..)	0.0481	0.00012	0.04015	0.00026
GetImage(..)	0.30715	0.00118	0.31232	0.00698
GetIR(..)	0.06543	0.00011	0.05977	0.00008
GetIRInfo(..)	0.06002	0.00055	0.05053	0.00088
GetLight(..)	0.08305	0.00083	0.0919	0.00106
Microphones(..)	0.0319	0.00045	0.01868	0
ada.GetSelector(..)	0.07517	0.00296	0.04993	0
GetSpeed(..)	0.01298	0.00066	0.00245	0.00001
Reset(..)	*1.3942*	0	*1.39023*	0.00007
BodyLight(..)	0.03253	0.00009	0.03057	0.00004
CalibrateIRSensors(..)	3.69635	0.00006	3.69173	0.00013
FrontLight(..)	0.03123	0.00002	0.02998	0.00004
LightX(..)	0.02515	0.00015	0.02347	0.00012
Motors(..)	0.03432	0.00029	0.0252	0.00015
PlaySound(..)	0.04403	0.00017	0.03488	0.00032
SetCam(..)	0.16965	0.00028	0.16708	0.00026
SetEncoders(..)	0.04013	0.00051	0.03243	0.00021
Stop(..)	0.02375	0.00004	0.0265	0.00017

Table 5.1: Times between sending commands and receiving their answers

6. Usage of *Elib*

This chapter is aimed at a programmer who wants to start working with an e-Puck using *Elib*. *Elib* offers a lot of tools, which make e-Puck programming easier. The tools are useless without a proper use. This chapter introduces guidelines and numerous examples how to use *Elib* and avoid problems.

Elib and e-Puck's properties is presented on thirteen samples from *TestElib* project. The samples are written in *C#* and are densely commented. We suggest a programmer, who wants to program his e-Puck, to start with one of the examples from *TestElib* project and modify it according his needs. *Elib* is built robustly and improper use does not destabilise the operating system.

Let us note that this chapter describes the use of *Elib* and its tools, but does not explain basic .Net or *C#* features. The basic knowledge of .Net and *C#* language is required for understanding *Elib* examples. On the other hand .Net *delegates*, an *EventWaitHandle* class and usage of lambda functions in *C#* are shortly introduced. A great part of this chapter is devoted to *IAsyncResult* interface and its usage in *Elib*.

Elib library requires .Net 2.0 on Windows and requires Mono 2.0 and higher on Linux. It also requires *BTCom* [1] version 1.1.3 running on e-Puck. As an example of a graphical application, which uses *Elib*, we present *Elib Joystick* at the end of this chapter. *Elib Joystick* is a graphical application, which uses Windows Presentation Foundation. It requires .Net 3.5 or higher and is not portable to Linux. *Elib* can control every actuator and can get every sensor value with *BTCom*'s on e-Puck.

The source codes are deployed using a solution of Microsoft Visual Studio 2008 (MSVS) for Windows and MonoDevelop solution for Linux. If you use higher version of MSVS or MonoDevelop, then the solution can be easily upgraded.

The contents of this chapter is divided into 6 sections. The first section is devoted to advanced .Net techniques, which are used in *Elib*. The Section 6.2 shortly introduces features of each *Elib's* interfaces and presents e-Puck's sensors and actuators. The next section goes through samples of implemented behaviours from *TestElib* project and explains the crucial part of the examples. The Section 6.4 presents the *Elib Tools* console application, which is useful during developing a program for e-Puck. Finally the last section sums up the most important guidelines for *Elib* library. It also shortly points out, which algorithms and applications can profit most from *Elib* design and which applications bring problems.

6.1 Advanced .Net techniques

Elib interface requires knowledge of *delegates*. Lambda functions and

[1] *BTCom* is a simple program running on e-Puck. It receives commands from *Elib*, performs the relevant actions and sends confirmation answer back. If we ask for a sensor value, the sensor value is returned together with the confirmation message.

EventWaitHandle class are used in *TestElib's* samples. Asynchronous programming is much more comfortable with them. If you are familiar with terms above, their usage in *Elib* will be described in Section 6.3.1. This section describes them in general.

Delegates are .Net wrappers for functions. Every .Net language is strongly typed and even functions are typed in .Net. A delegate has two meanings in .Net. The first meaning is the placeholder type for functions of given type. See the *C#* implementation in Figure 6.1, where the placeholder type has name *OkfActuators*. The second meaning is the placeholder itself. In the sample code below the placeholder is called *a*.

Delegate variable can contain only functions, which exactly match delegate definition. In the case of *OkfActuators* delegate from Figure 6.1 the function must return *void* and must have one argument of type object. All classes in *C#* are inherited from the class *Object* and therefore every object can be passed to this delegate. The class *Object* contains method *ToString*(), so every object has its own string representation.

therefore the number as well as the string are printed. Delegate function can contain more functions. On the other hand this feature is not used in *Elib*, because it reduces readability of the code.

The console output of the *Example*() function shows, that the functions are called in the order, in which the functions were added to the delegate variable. The functions have implemented void returning value, because it is problematic to return a value from the first function. Delegates allow the last function to return its value, but it is considered a bad manner, because another function added to delegate can overwrite the returning value of the delegate call.

Let us focus on the function, which is used in the *OkfActuators* declaration. The function was defined and declared in place. Functions defined in place are called lambda functions and have several advantages. They can omit types of arguments, because they are inferred from delegate definitions. Lambda functions can also directly use variables from the scope of its declaration. The lambda function printed 8 in the enclosed output above, although we have not passed it as an argument to the delegate.

Explanation (*EventWaitHandle*)**.** The *EventWaitHandle* class

EventWaitHandler class is not tricky itself, but it is usually used in multi thread programming, which is usually complicated. An instance of the class is used to synchronise two threads. It is usually used to signal from one thread to another, that some work has been done. In *Elib* there are used two methods per *EventWaitHandle* instance. Let us suppose we have a thread *A* running. Let *B* be the thread, which should perform a long task. The thread *A* wants after a while to wait until the work of *B* is finished. Before the job *B* finishes. *A* creates *EventWaitHandle* *e* with parameters *false* and *ManualReset*. *False* parameter sets *e* to a blocking state. In the blocking state all threads, which have called method *e.WaitOne*, are blocked in *WaitOne* method. *ManualReset* means that the state of *EventWaitHandle* can be changed only by calling its methods and *EventWaitHandle* does not perform any action itself. Let us return

```
// Delegate definition in some class e.g Program
delegate void OkfActuators(object data);
void Example(){
  int i=8;
  //declaration of delegate variable and initialization with lambda function
  OkfActuators a = new OkfActuators((sth) => { Console.WriteLine("Lambda f{0},{1}",sth,i); });
  //second function is added to delegate
  a += new OkfActuators(suitableFunction);
  //invocation of 2 functions with string parameter
  a("Hurray!");
  //third function is added to delegate
  ExampleClass c=new ExampleClass();
  a+= new OkfActuators(c.suitableMethod);
  //invocation 3 functions with int parameter.
  a(-333);
}
private static void suitableFunction(object sth) {
  Console.WriteLine("suitable Function {0}",sth);
}
class ExampleClass{
  public void suitableMethod(object sth) {
          Console.WriteLine("suitable method {0}",sth);
  }
}
/
```

Figure 6.1: Definition of a delegate

```
Lambda fHurray!,8
suitable Function Hurray!
Lambda f-333,8
suitable Function -333
suitable method -333
```

back to threads A and B. A creates an instance e of $EventWaitHandle$ with mentioned parameters, passes e to the second thread. Finally A calls $WaitOne$ and blocks on this call. B thread works and after the job is done, it just invokes *Set* method on e. *Set* method releases all threads, which are blocked in $WaitOne$ method. Run the code from Figure 6.2 to understand it.

```
static EventWaitHandle e = null;
static void Athread() {
  Thread t = new Thread(Bthread);
  e = new EventWaitHandle(false, EventResetMode.ManualReset);
  t.Start();
  e.WaitOne();
  Console.WriteLine("Finally someone pressed the button!!!");
}
static void Bthread() {
  //simulate the work
  Console.Readline();
  e.Set();
}
```

Figure 6.2: Definition of a delegate

$EventWaitHandle$ is used in the function *endBehaviour* described in Section **??** used in behaviour implementation. Lambda functions can be seen for example in $ConsoleTestSensorsTimeout$ function, which presents all sensors from e-Puck. Delegates are used in every command invocation to specify types of callback functions. Callback functions are functions, which are called after finishing an operation.

6.2 Explore *Elib* through examples

There are three public classes in *Elib*. *Sercom*, *Epuck* and *Stamp*. *Stamp* is for time measurement. *Sercom* and *Epuck* are classes, where all the algorithms are located. *Epuck* class uses *Sercom* internally. Let us focus only on $Epuck's$ class, because *Epuck*'s basic interface is a specialisation of *Sercom*'s interface for version 1.1.3 of $BTCom$. For more information about interfaces and implementation of *Sercom* and *Epuck* see Section 5.

Epuck class itself has two interfaces. Let us name them the basic interface and $IAsyncResult$ interface. $IAsyncResult$ interface is used widely through .Net. We will introduce it in the examples from *TestElib* project. All examples from this section come from *TestElib* project. We suggest reading this chapter with *TestElib* project opened and explore the samples from *TestElib* in detail.

The examples are listed from the simplest to the more complex. At first, function, which tests the communication between e-Puck and your computer is described. Then Section 6.2.1 focuses on starting session using *Epuck*. Later in Section 6.2.2

simple functions describe all e-Puck's sensors and actuators using simple *Epuck's* interface.

Four methods, which invoke different behaviours using *IAsynResult* interface are described in Subsections 6.3.1, 6.3.3, 6.3.4 and 6.3.5. One behaviour implemented by *Epuck's* basic interface is presented. After the main section, which covers the behaviours, some "tips and tricks" are shown. There is a behaviour in *TestElib*, that emulates *Epuck's* basic interface via *IAsyncResult*. It is depicted in the summary of Chapter 5. We will skip the behaviour, because it is implemented only for theoretical purposes. Apart from minor functions there is an example of image processing in "Tips and Tricks" part and a logging example. Logging *Epuck's* actions is the last example.

Remark. In the following paragraphs an exception means a subclass of *ElibException* if it is not said otherwise.

6.2.1 Setup and disposal of a session

Let us start with a function, which does not use *Elib*. *TestPortTurnAround* opens a given serial port and sends some commands to move and stop, then it ends. It does not require any feedback and throws no exceptions. It is useful to run it, because a lot of errors are not hidden in the code, but in the hardware set up.

Function *startEpuck* returns an instance of *Epuck*, which allows you later to control an e-Puck.

```
return new Epuck(port, "Ada"); //Name it. It is useful for ↩
    logging and debugging.
```

The function throws exception only if other application blocks the port. Both *Kof* callback and exception let user know, that a command was not delivered in time. *Kof* callback from *Epuck's* basic interface or an exception from *IAsyncResult* are raised after sending a command to e-Puck, which is not confirmed in timeout. Sending a command to e-Puck is done via calling one of *Epuck's* functions. Let us mentioned the closing of session and then we introduce both interfaces. The function *endEpuckSession* sends a command to stop your robot, but it primarily closes the session by calling *Dispose* method. Dispose method releases serial port for another application. After disposal *LogStream*, *Port* and *Name* are the only properties of *Epuck* instance, which can be accessed. They can be used to reconnection, see below.

```
//dispose can take a while (under 500ms)
ada.Dispose();
TextWriter t= ada.LogStream;
t.WriteLine("We are trying to reconnect");
ada.BeginStop(0.1,null,null); //throws an exception
ada=new Epuck(ada.Port,ada.Name);
ada.LogStream=t;
ada.StartLogging();
ada.Stop();//does not throw an exception
```

6.2.2 Touching e-Pucks's sensors and actuators

ConsoleTestActuatorsTimeout function presents all actuators. Let us choose to control the motors of e-Puck.

```
ada.Motors(1,-1,
  (nth) => { Console.WriteLine("Motors(..) OK "); end = true;↩
      },
  (nth) => { Console.WriteLine("Motors(..) KO"); end = true; ↩
     },
  null, myto);
wait(0);
ada.PlaySound(3,
// ...and so on
```

Motors method controls the speed of wheels. See code above. Maximum forward speed is +1, backward speed -1. Both motors are controlled at once via first two parameters. Next two parameters are functions, which match *OkfActuators* delegates, respectively *KofCallback* delegates. Both delegates have only one *object* parameter. In this example *null* value is passed to the lambda function through the fifth argument of *Motors*. The last parameter is the *timeout*, which tells *Epuck* how long it can wait to answer of *BTCom* in seconds. The first delegate is called if the answer from *BTCom* is confirmed before *timeout* has elapsed. *KofCallback* is called otherwise. *ConsoleTestActuatorsTimeout* function wants to present the reaction time of all commands, therefore it does not allow commands to be called asynchronously one after another. Function *wait*() forces the current thread to wait until answer is delivered or *timeout* elapsed. Simple synchronization is done via *end* flag.

```
static void wait(int gap) {
  ///<remarks> Simple but inefficient way of waiting. See ↩
      KofOkfWaiting(..) in Behaviour for usage of ↩
      EventWaitHandle.</remarks>
  while (!end) {
    Thread.Sleep(5);
  }
  end = false;
  Console.WriteLine("Ended: {0}",Stamp.Get());
  Thread.Sleep(gap);
  Console.WriteLine("Start: {0}", Stamp.Get());
}
```

ConsoleAsynchronousSensors function asks for sensors too, but does use synchronous waiting only at the end. Run the example and note the huge gap between the time after asynchronous calls and the time after synchronisation.

The time is measured in seconds!

The last function in the introductory section is *GetImage* function. The function creates a window after an image is captured. The *TestElib's* application is blocked until the windows is opened. To see the picture from e-Puck's camera switch from the running *TestElib's* console to the new window.

```
IAsyncResult ar = e.BeginSetCam(40, 40, Zoom.Small, CamMode.↩
    Color, toSetCam, null, null);
e.EndFtion(ar);
ar = e.BeginGetImage(toImg, null, null);
Bitmap bm = e.EndGetImage(ar);
```

ShowImage uses an *IAsyncResult* interface to set the camera and get the picture. Let us shortly introduce the interface on *BeginSetCam* method, which called on an instance *e* of e-Puck. *IAsyncResult* interface allows easily to start asynchronous operation and after that wait for the result. *BeginSetCam* starts the asynchronous operation, which sends the relevant command, and *EndFtion* waits to it. The *EndFtion* can wait for *BeginSetCam* result, because *ar IAsyncResult* instance was created in *BeginSetCam*. Every *IAsyncResult* asynchronous function start with "*Begin*" prefix. If it asks for a sensor value, then it starts with "BegingGet" as you can see on *BeginGetImage*. *BeginGetInfoVersion* and *BeginGetInfoHelp* functions start with "*BeginInfo*" prefix, their return values never change.

The "End" functions differ according to the type, which is returning from their invocation. Sensors do return a value. Most of functions return *int*[] array. They start with "BeginGet" followed by the name of the sensor. However,*BeginGetImage* returns a *Bitmap*. Last type of the functions, which ask for *BTCom's* help and version, return *string*. *EndFtion* can be applied on every *Elib's IAsyncResult's* invocation. In other words *EndFtion* can be used to wait on every "Begin" function, but it is not capable of returning values. Functions *EndGetFtion*, *EndGetImage* and *EndInfoFtion* returns relevant values, but these "Get" functions can be called only in pair with "Begin" functions, which ask for a sensor's value of the same type as a return value from "Get" function.

An *IAsyncResult* "Begin" function has always three arguments. If a function has more than three arguments, the obligatory arguments are the last. Let us look at *BeginSetCam's* arguments. Width and Height of image are first two parameters and together with zoom and colour is specific for *BeginSetCam*. Next comes *timeout*, which tells how long we are willing to wait to the answer. Instead of the *null* values callback and its parameter of *object* type can be specified. If we use *EndFtion*(*ar*) we do not need callback.

Callbacks are last not introduced feature of *IAsyncResult*. They are presented in the next section devoted to simple behaviours of e-Puck robot.

6.3 Behaviours

A behaviour is a program, which controls a robot. It can be described as an finite automaton, where the transition from one state to another is based on sensor values. States represent actions, which robots perform.

We present behaviours invoked by functions *Bull*, *Billiard*, *GoAndTurn*, *Go2Light* and *KofGoXcm*. The robot running *Bull* behaviour is supposed to act like a bull. If there is a red obstacle, robot attacks it. In *Billiard* behaviour e-Pucks goes from an obstacle to an obstacle and when it is very near it turns around. *GoAndTurn* let the robot drive along a square with 15 cm long sides. There are other functions like *goXcm* or *turnAround* available. All mentioned behaviours are implemented using *IAsyncResult*. *KofGoXcm* is the only one behaviour, that is implemented by using simple *Epuck's* interface. It allows robot to go almost exactly a given amount of centimeters, even if the connection breaks during the ride. All behaviour used callbacks, because it is a natural way how to switch between states of a behaviour.

Callbacks of "Begin" functions like *BeginStop* take one *IAsyncResult* instance as an argument and return *void* value. The *state* parameter, which is passed to "Begin" function can be found in *AsyncState* property from *IAsyncResult* argument. In Figure 6.3 *BeginGetIR* function is invoked on the third line with *EasyDisposal*(..) callback and *ada* instance as parameter for callback. On line 7 of Figure 6.3 the *ada* parameter is extracted from *IAsyncResult ar*. If we use "BeginGet" functions to get a sensor values, then the values are returned by "EndGetFtion" call on *ar* as we can see on the line below.

In *Elib* callbacks are called always after the operation invoked by a "Begin" function finishes, therefore the *EndFtion*(..) called in callback function does not blocks the thread. See again line 7 of Figure 6.3. Furthermore no *EventWaitHandle* is created, which results in a better performance if we use callback. For information how this feature is implemented see Section 5.6.2.

Callbacks are invoked in separate threads. It complicates the code, if we invoked more than one behaviour from one function as we do in the *Main*(..) function of *TestElib*. The problem is that *Epuck* instance gets commands from different behaviours running in parallel at the same time. The behaviours in *TestElib* let the thread with *Main*(..) function wait, until they are finished. After end of one behaviour the next is invoked.

Next we introduce a guideline for ending a behaviour, which uses *Epuck* class. *Epuck* class use a serial port, which is an operating system resource. It is good practice to free the serial port as soon as our behaviour does not need it. We free the serial port by calling *Dispose* method or by letting the garbage collector collects our instance of *Epuck* class.

On the other hand, do not call *Dispose* method if callbacks in different threads are running. Nor let the garbage collector to call *Dispose* method if you other threads can access the *Epuck* instance. Such situation is introduced in *Example* class bellow. The *Dispose* method can be called before *BeginMotors* method from callback on line 10.

```
class Example{
  static void GoCallback(IAsyncResult ar) {
    Epuck r = (Epuck)ar.AsyncState;
    ar = r.BeginMotors(0.5,-0.5,1.0,null,null);
    r.EndFtion(ar);
  }
  static int Main(string[] args){
   Epuck r= new Epuck("COM1");
   r.BeginStop(1.0,GoCallback,r);
  }// Here can be called r.Dispose() before r.BeginMotors() ↩
     from GoCallback()
}
```

We advice to control e-Puck during the whole program from one thread at the moment. In such situation it is easy to wait until the thread finishes communication with e-Puck by calling the *Dispose* method in last callback invocation. See Figure 6.3 and the last callback *SafeDispose*. We suggest to return to this example after exploring some of the behaviours.

```
static int Main(){
  //some work, Epuck ada created
  ada.BeginGetIR(0.1, EasyDisposal,ada);
  // no code using ada is allowed after BeginStop call
}
static void EasyDisposal1(IAsyncResult ar){
  Epuck ada = (Epuck)ar.AsyncState;
  int[] irSensors = ada.EndGetFtion(ar);
  // synchronous work e.g to stop using EndFtion
  ar=ada.BeginStop(0.1, null, null);
  ada.EndFtion(ar);
  //asynchronous call has to be the last!
  ada.BeginPlaySound(4,SafeDispose, ada);
}
static void SafeDispose(IAsyncResult ar{
  Epuck ada = (Epuck)ar.AsyncState;
  ada.Dispose();
}
```

Figure 6.3: Safe disposal for single behaviour

6.3.1 *IAsyncResult* Behaviours

Let us finally introduce the *Bull* behaviour. Running this behaviour e-Puck behaves like a bull and bumps into red obstacles. *Bull* behaviour switches between three states. In the first state it travels randomly. After a random time it switches to observation state. In the observation state it takes 4 images from 4 directions. If the images are red enough it switches to aggressive behaviour and goes to the red obstacle, otherwise it starts a random ride again.

In this snippet *CountDown* is a struct, which wraps the *Epuck* and the number of rotations. The start function firstly sets e-Puck's camera to desired settings and secondly it invokes the behaviour.

```
public static void Bull(Epuck ada) {
  startBehaviour();
  try {
    IAsyncResult ar = ada.BeginSetCam(40,40, Zoom.Small, ↩
       CamMode.Color, toSetCam, null, null);
    ada.EndFtion(ar);
    ada.BeginGetImage(0.5, searchAround, new CountDown(ada, ↩
       3));
    endBehaviour();
  } catch (ElibException e) {
    exceptionOccured_End("Exception was thrown in Bull(..)", ↩
       e);
  }
}
```

All behaviours from *TestElib* run until the user presses a key or communication with e-Puck is broken. The waiting is performed in the main thread. The *Bull* behaviour as all other behaviours runs in worker threads. If an exception, that signals undelivered command, is thrown, its message is written to a console and the behaviour ends. Explore code in *exceptionOccured_End* function to see more.

See code below how the behaviour is ended in aa nice way. It is a tricky part and is not necessary if only one behaviour is implemented. See Figure 6.3 for safe ending of a single behaviour. Important is that *startBehaviour* is called at the first line of *Bull* method and *endBehaviour* is called on the last one.

```
static void startBehaviour() {
  endf = false;
  endconfirmed = new EventWaitHandle(false, EventResetMode.↩
     ManualReset);
}
static void endBehaviour() {
  //If you press an key, endf is set to true. ->
  //->All behaviour recursive functions look like: if(!endf)↩
     {..body..}else endconfirmed.Set();
  Console.WriteLine("Press enter to quit..");
  //Behaviours are usually infinite loops, they run and run.
  Console.ReadLine();
  ///<remarks>Blocks invoking next function</remarks>
  endf = true;
  ///Wait on EventWaitHandle until currently running function↩
     is finished.
  endconfirmed.WaitOne((int)(1000 * (toReset + toImg + to)));
  //releasing EventWaitHandle
  endconfirmed.Close();
}
```

6.3.2 *Bull's* internals

The diagram 6.4 shows the states of *Bull* behaviour. Three states are named after the functions, which control the robot. State "*GetStacked*" is the final state and it arises, if the e-Puck is surrounded with obstacles from the front and from behind. The *searchAround* procedure calls itself 4 times if no red obstacle is in front of e-Puck. It uses the struct *CountDown* to count the number of rotations.

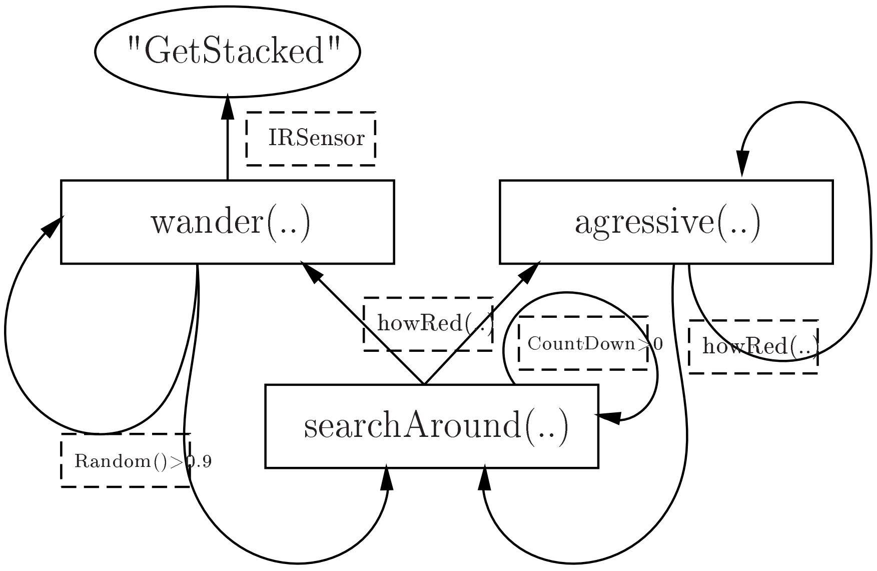


Figure 6.4: Bull behaviour diagram

Aggressive behaviour tries to bump into red obstacles. If it finds out an obstacle not red enough, it starts searching again. After four 90 degrees rotations in *searchAround* are made, the *wander* behaviour is switched on. From *wander* state the behaviour can switch back to *searchAround* or it ends stacked. Let us show how simple the *aggresive* function can be.

```
double redl = 0.2;
IAsyncResult ar = ada.BeginGetImage(toImg, null, null);
Bitmap a = ada.EndGetImage(ar);
howRed(a, out red, out dir);
if (red > redl) {
  ada.BeginMotors(0.4, 0.4, to, agressive, ada);
} else {
  ar = ada.BeginStop(to, null, null);
  ada.EndFtion(ar);
  ada.BeginGetImage(toImg, searchAround, new CountDown(ada, ↩
     3));
}
```

6.3.3 Billiard ball behaviour

The behaviour is invoked by *Billiard* function. The behaviour is rather simple. It goes to a wall and before the wall it stops. Then the robot tries to compute the orientation of the wall it turns around to go in a next direction and than it goes straight to the next obstacle. The behaviour runs without ending until the e-Puck is surrounded by obstacles.

The *rebound* function tries to compute the angle of rebound, but as well the simple *go2wall* function implements a restart of the behaviour. Let us focus on this part. We will show it on *go2wall* function in Figure 6.5

```
static void go2wall(IAsyncResult ar) {
  //value to decide if an obstacle is near enough
  int frontLimit = 1000;
  //does not throw ElibException
  Epuck ada = (Epuck)ar.AsyncState;
  try {
    // Doesn't create EventWaitHandle because the action has ↩
       already completed synchronously.
    // Can throw an TimeoutElibException
    int[] ir = ada.EndGetFtion(ar);
    if (ir[0] + ir[7] > frontLimit)
      ada.BeginStop(to, rebound, ada);
    else {
      //Does not use EndFtion, it saves the EventWaitHandle. ↩
         We suppose, that it succeeds now or in next rounds.
      ada.BeginMotors(0.2, 0.2, to, null, null);
      // The BeginGetIR command is enqueued in the same ↩
         momment as BeginMotors, therefore double timeout is ↩
         used.
      ada.BeginGetIR(2 * to, go2wall, ada);
    }
  } catch (ElibException e) {
    Console.WriteLine("Billiard restarted in go2wall, because↩
       of exception:\n" + e.Message);
    // Invokes go2wall function again. It needs to be invoked↩
       by BeginGetIR command, because it expects ar with IR ↩
       values.
    ada.BeginGetIR(to, go2wall, ada);
  }
}
```

Figure 6.5:

The exception can be throw on only in call of *EndGetFtion* at the beginning and no action in the body was done, so the function can be called again from the catch block without problems.

The problem of this implementation is the logic, not the implementation. The restart of the *go2wall* function does not solve the problem with slow delivery of IR sensors, it only tries and tries again until enter is pressed. The solution will be presented in next behaviours.

Next interesting feature of this example is not calling the *EndFtion* to wait until *BeginMotors* is confirmed. We suppose that if the commands of *BeginGetIR* sensors are delivered, than the *BeginMotors* are delivered too. Furthermore only one command delivered is enough to go to the next obstacle.

6.3.4 Exact movement and *GoAndTurn* behaviour

Exact movement is a very strong feature of e-Puck. E-Puck uses two stepper motors, which allows you to travel exactly 10 cm. The deviation is less than one millimeter. E-Puck has also encoders, which allow you to measure the distance of travel with the same resolution. On the other hand, communication over Bluetooth brings problems, which are typical for common electric motors. Usual motors have an inertia and they do not stop immediately and they do not reach the desired speed at once. The same problem is with *Elib* and Bluetooth communication. The commands have a delay. A common problem is also with a flat tyre, because one wheel has bigger perimeter than the other. Programmers of e-Puck do not have to solve these problems due to good design of e-Puck.

GoAndTurn is the only behaviour, which does not run in an infinite loop. It just goes around a perimeter of square with side of 15 cm. It is based on *goXmiliseconds* function, which lets the robot ride with given speed for a specified time. Functions *TurnAround* and *goXcm* are simply built on *goXmiliseconds*. The *square* function from *GoAndTurn* behaviour simply combines the functions. Let us introduce *goXmiliseconds* function.

```
static void goXmiliseconds(Epuck e, double L, double R, int ↩
    milisec, double addto) {
  int x = e.Working;
  if (x != 0)
    throw new ElibException("It would be extremely inaccurate↩
        , if commands are still waiting to be sent.");
  IAsyncResult ar = e.BeginMotors(L, R, addto, null, null);
  AsyncResultNoResult pom = (AsyncResultNoResult)ar;
  pom.Name += "goXms";
  e.EndFtion(ar);
  ar = e.BeginStop(addto, null, null);
  e.EndFtion(ar);
}
```

Figure 6.6: Function goXmiliseconds

The function needs to have no queued commands in *Epuck* instance, because it supposes that sending takes almost no time and any waiting makes it very inaccurate. The function sends a command to motors, then waits a given time and then stops. The idea behind the little heuristic is that every commands sending takes the same time and also the commands transferred to e-Puck and from e-Puck are equally fast.

More interesting feature presented here is extracting an *AsyncNoResult* class from *IAsyncResult* interface. For more information see Section 5.6.2. From user's point of view it is good to know, that *Name* property can be changed. *Name* property is used in logging. Its default value is the name of "Begin" function, which created the instance of *IAsyncResult*. The *Name* attribute is a feature of *Elib* and is not defined in *IAsyncResult*, therefore the cast is needed for accessing it.

6.3.5 Restarting *Go2light* behaviour

Robot following the light is a typical robotic task. *Go2Light* behaviour uses only one recursive function, which implements the behaviour. The following example also presents a guideline how to cope with unreliable connection in an infinite behaviour.

```

static void recGotoLight(IAsyncResult ar) {
  Epuck ada = (Epuck)ar.AsyncState;
  try {
    ar = ada.BeginGetLight(to, null, null);
    int[] light = ada.EndGetFtion(ar);
    //..omitted part of source code: Debugging printouts
    if (diff_fb > 0) {
      if (light[2] < light[5]) {
        Console.WriteLine("turn around right {0}", diff_lr);
        ar = ada.BeginMotors(speed, 0, to, recGotoLight, ada)↩
          ;
      } else {
      //..omitted part of source code: the lighst is on front↩
          left, back right,...
      }
    }
    //there is no need to repeat EndFtion in the branches
    ada.EndFtion(ar);
  } catch (TimeoutElibException) {
    exceptionOccured_Restart(ada);
  }
}
```

Figure 6.7: Function *recGotoLight* function

See the clear structure of *recGotoLight* function is in source code in Behaviours.cs file. Let us explore the interesting *exceptionOccured_Restart* function in Figure 6.8. The function ends the session with real e-Puck and closes a serial port by calling *Dispose* method. After the disposal, the commands can not be sent to e-Puck and we have to create a new connection. We use the same parameters. Optionally, here is the place to ask the user of behaviour to change a port name. The same approach is used in $KofGoXcm$, which will be presented below.

```
static void exceptionOccured_Restart(Epuck ada) {
  //Reconnect again to e-Puck
  ada.Dispose();
  ada = new Epuck(ada.Port, ada.Name);
  restarts--;
  if (restarts >= 0) {
    Console.WriteLine("Remaining " + restarts.ToString() + " ↩
       restart(s). Press enter to continue");
    ada.BeginStop(to, recGotoLight, ada);
  } else {
    Console.WriteLine("End of Go2Light, because all " + ↩
       restarts_startingValue.ToString() + "restarts have ↩
       been used.");
    Console.WriteLine("Behaviour has finished. Press enter to↩
        perform next actions");
  }
}
```

Figure 6.8: Function *exceptionOccured_Restart* function

6.3.6 Behaviour with return implemented via $Epuck's$ basic interface

The behaviour is invoked by $KofGoXcm$ function. The robot goes specified amount of centimeters. If the connection breaks during the behaviour, the behaviour ask the user to repair the connection and to keep e-Puck running. If the connection is repaired successfully then e-Puck goes to the destination, where it should have ended before the connection failure.

The behaviour uses $Epuck's$ basic interface, which has been introduced in Section 6.2.2. It has some design consequences. The example has two implementations, in fact two behaviours. The behaviour, which has a good connection at disposal, and the behaviour, which has to deal with a broken connection. We also keep the guideline from Section 6.3. It says, that the asynchronous call should be made only once at the end of a function. It is necessary to avoid parallelism and problems with ending of the behaviour.

In summary, the amount of functions grows rapidly, because we can not easily synchronously wait to answers from e-Puck and furthermore we have to implement two kinds of functions: Kof and Okf callbacks. The implementation of $okf(..)$ callbacks is straightforward. We will focus on a function, which is called to stop the e-Puck after it travelled the required distance.

Let us remind of the *Epuck* basic interface. In Figure 6.9 class *RobotAndTime* is used. As you can see, the structure of the code does not differ from *IAsyncResul*. First the necessary cast is performed in order to extract *RobotAndTime* class, then the logic is implemented and in the end the commands to e-Puck are sent. The *Kof* and *Okf* commands use the same state argument for passing data, e.g. *RobotAndTime* instance. The state argument together with timeout are obligatory and are located at the end of functions. The *Okf* callback for a sensor command has additional first argument. The first argument is used to return sensor values. For example the callback for *GetIR* command looks like *void OkGetIR(int[] values, object state)*;.

```
static void stopKof(object robotAndTime) {
  Console.WriteLine("stopKof was called.");
  RobotAndTime x = (RobotAndTime)robotAndTime;
  x.stopKof++;
  if (x.stopKof > 5) {
    if (!reconnect(x))
      return;
    else
      x.stopKof = 0;
  }
  double time = Stamp.Get() - x.StartTime;
  if (travelled(time, x.Speed) < (x.Cm + 0.05))
    x.E.Stop(stopOkf, stopKof, x, to);
  else {
    Console.WriteLine("We missed the destination spot, we ↩
       return back, try to repair the connection.");
    x.Cm = travelled(x.StartTime, x.Speed) - x.Cm;
    x.Speed = -x.Speed;
    x.StartTime = Stamp.Get();
    x.E.Motors(x.Speed, x.Speed, goOkf, goKof, x, to);
  }
}

class RobotAndTime {
  // Lot of parts omitted!!
  // The thread.Safe read and write are missing!!
  public double StartTime;
  double cm;
  double speed;
  public volatile int stopKof = 0;
  public volatile int goKof = 0;
}
```

Figure 6.9: *Kof* callback *stopKof*

The logic of this example is very interesting. After five unsuccessful attempts to stop, the *reconnect* function prompts the user to repair the connection. If the user refuse to continue, no other function is invoked, and the behaviour is ended.

Otherwise, the actions for going to the desired place are executed. If the robot does not drive away too far, the *Stop* function is called again. The *Stop* function calls *stopKof* function as its *Kof* callback. If the robot travels too long then it has to go in the opposite direction. The change of the direction is done in the second branch of the *if* command.

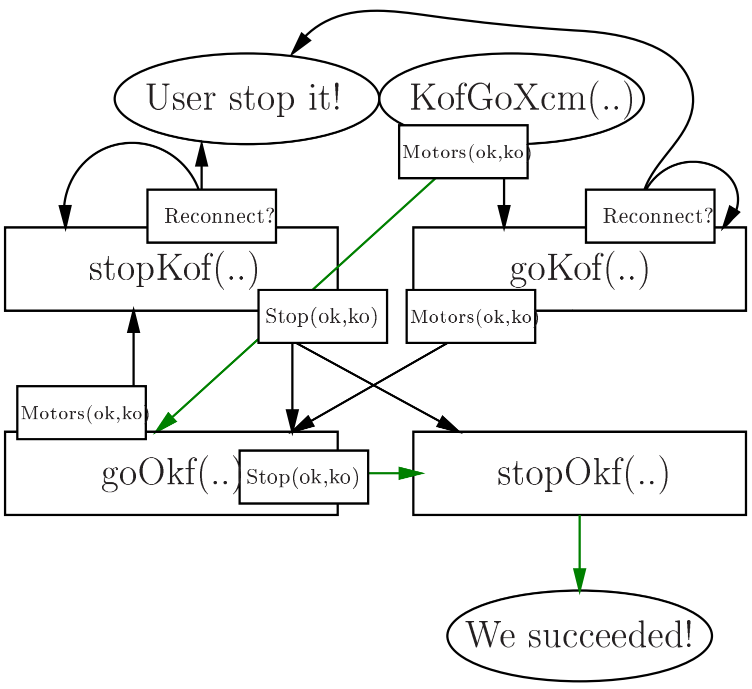


Figure 6.10: *KofGoXcm* basic *Epuck's* interface implementation

6.3.7 Logging of *Epuck* actions and image processing

In the file *TipsAndTricks.cs* there is located a simple example of image processing of a picture from e-Puck's camera. The example is invoked by the function *ShowProcessedImage* and it displays a window, where a black and white image is processed. The white is instead of red colour and black represents the other colours. A nice test of this function is to place the e-Puck's box with the e-Puck logo in front of the camera.

The function *LoggingExample* uses the function *ConsoleTestSensorsTimeout*, which has been presented at the beginning of this section in 6.2.2. It runs the function twice, each time with different timeout. The logging to file specified in *name* property is turned on. Try to run the function. Let us see the body of the function.

```
ada.LogStream = new StreamWriter(new FileStream(name, ↩
   FileMode.OpenOrCreate, FileAccess.Write));
ada.StartLogging();
for (i = 0; i < 2; ++i) {
  double to=1.0 / (i + 1);
  ada.WriteToLogStream("ConsoleTestSensors with timeout :"+to↩
     .ToString());
  ConsoleTestSensorsTimeout(ada, to);
}
ada.StopLogging();
```

Logging to a file is really simple. In needs only stream, where the log is written. After the start of logging all actions that send commands to e-Puck, performed on *ada* instance of *Epuck* are logged to file on path *name*. *WriteToLogStream* method inserts a commented line to the log file.

6.4 Elib tools (et)

Elib offers a standalone command line application, which parses the log from *Elib*. It is meant to be the base tool for a programmer, who wants to statistically analyse a log from *Epuck* class.

Et can be used only from the command line and parses the log row by row. It reads the log file, performs a chosen action and writes the result to the output file. If user does not specify the file it writes or read from, respectively, to command line. Elib tools supports three operation. Two of them process numeric data. The first operation counts an average of a selected column from a log file. The next operation sorts out rows, which have in specified column values from a given range. The last implemented operation separates rows, which have in specified column a value from a given collection of words. Elib tools application skips the rows beginning with '#', which is in *Elib* by default a comment in the log file. Let us look at the usage. The most important command is the second row, which invokes the help file. The help file introduces a complete usage of et command. Let us remark, that this invocation was performed in PowerShell on Windows. On Linux you need to add "mono" before every program, which was compiled for .Net or Mono.

```
>>et -s sourcelog.txt -d output.txt Avg 0
>>et -h
et (Elib Tools) HELP FILE
...here the help file continues
```

Design of *Elib Tools*

The purpose of *Elib Tools* is to be a simple tool for parsing a log file from *Elib*. The application is written in order to be as robust as possible. Wrong arguments do not throw any exception, but the application does no action.

The main contribution of *Elib Tools* application is its design, which can be easily extended. The application can be easily changed according to specific needs of a programmer. The programmer needs to modify *parseArguments* argument function. As you can see, the *parseArguments* function returns only paths of the source and the destination file and *Action* object.

```
Action action = parseArguments(args, out source, out destination);
```

Action object is an abstract class, which is introduced below in this section. It provides an interface for performing operation on every row of the source file. If a new functionality is added, it could be implemented only by deriving a class from *Action* and by overriding two functions. Let us describe an example of Avg command. Lets look first at *Action* class.

```
abstract class Action {
  const string Comment = "#";
  protected int column;
  protected TextReader r;
  public TextReader R { get { return r; } set { r = value; } }
  protected TextWriter w;
  public TextWriter W { get { return w; } set { w = value; } }
  protected char[] sep;
  public char[] Separators { get { return sep; } set { sep = value; } }
  public Action(int Column) {
    column = Column;
  }
  public void DoAction() {
    string line = null;
    while ((line = r.ReadLine()) != null) {
      if (!line.StartsWith(Comment))
        LineAction(line);
    }
    LastAction();
  }
  protected virtual void LastAction() { /*usually does nothing and is called after all lines are processed*/}
  protected abstract void LineAction(string line);
}
```

Figure 6.11: *Action* abstract class for performing row operation

The *Average* class, which inherits from *Action* class, computes the average by calling a method *DoAction*() inherited from *Action* class.

6.5 Purpose of *Elib* and its properties

In this chapter we have presented *Elib* and the guidelines, which are convenient to follow using *Elib*. This section follows up to contents of the previous sections and introduces a view from, which was the *Elib* created.

```
class Average : Action {
  long count;
  double avg;
  public Average(int Column)
  : base(Column) {
    count = 0;
    avg = 0;
  }
  protected override void LineAction(string line) {
    count++;
    avg *= (count - 1) / (double)count;
    try {
      avg += (Double.Parse((line.Split(sep)[column])) / count↩
          );
    } catch (FormatException e) {
      Console.WriteLine("Avg has to be done from Integer. ↩
          Error: " + e.Message);
      avg = double.NaN;
    }
    }
    protected override void LastAction() {
    w.WriteLine("{0:F8}", avg);
  }
}
```

Figure 6.12: *Average* class

Elib library was designed in order to help students of mobile robotics with controlling e-Puck by their programs. *Elib* extends possibilities of e-Puck processor, which can perform a very limited range of algorithms due too its low performance. Moreover *Elib* provides a user with numerous examples. The examples are commented and can be easily modified. On the other hand, a program, which use *Elib*, gives up of direct control over robot, which can be observed the best on *GoAndTurn* behaviour in Section 6.3.4.

We have presented a few of simple behaviours, which control e-Puck, but much more can be done with *Elib*. *Elib* can be used easily for controlling more than one e-Puck at the same time by creating more sessions using different instances of *Epuck* class. Programs, which need more than 8 KB of memory on e-Puck such as genetic programs or neural networks, can easily control e-Puck over Bluetooth.

The real challenge is processing a picture from e-Puck, because the dsPic processor of e-Puck is not sufficient and also *Elib* needs to wait quite a lot of time for a picture. On the other hand, *Bull* behaviour successfully use the camera to grab the picture and also a graphical application *Elib Joystick* shoots the pictures with the camera.

Elib Joystick is a graphical application, which makes all sensors and actuators of e-Puck accessible in one window. It also supports capturing of an image, which is presented enlarged to the user. The graphical application runs in Single Thread Apartment on Windows, which means that the controls of the window can be accessed only from the main thread. In order to update the sensors *Elib Joystick* use either *EndFtion* or a *Dispatcher*. *Dispatcher* is a class, which allows to access the controls from a different thread. It is specific according to the technology, which is used to run the graphical part of the application. *Elib Joystick* uses Windows Presentation Foundation and uses a dispatcher to capturing image. The other operation, which has timeout 0.1 s, accesses the controls synchronously using either *EndFtion* or *EndGetFtion*. See Section 5.6.2 for information about these functions. *Elib Joystick* is introduced in appendix B.

Elib can access all sensors of e-Puck. See the Chapter 5.6 for more information about e-Puck's sensors and actuators. However, the camera is not used in the full resolution, because the e-Puck processor has no place to store the captured image. The *Elib* captures only the amplitude of sounds from e-Pucks microphones, because *ELib* library depends on *BTCom*. The frequency of a sound can be computed on e-Puck using Fast Fourier Transformation on e-Puck, but the *BTCom* does not implement it.

To conclude, *Elib* offers almost all sensors of e-Puck in full quality. It also controls all of e-Puck's actuators. Furthermore, it offers much more comfort than a programming e-Puck's processor directly. Read Section 6.4 about the advantages of remote control. Last but least, *Elib* can be used from all .Net languages including *C#*, *VisualBasic*, *F#*, which is a functional language of .Net based OCaml, or *IronPython*, which is a .Net implementation of Python. On Mono runtime, which runs both on Linux and Windows, the *C#* language can be used. The *Elib* is compatible with .Net 2.0 [16] and higher and with Mono [15] 2.0 and higher.

7. Conclusion

A remote control of e-Puck over Bluetooth, implemented in *Elib* library, has been tested on enclosed model program *TestElib*. Tests have shown, that communication used by *Elib* is fast enough to control e-Puck robot. *Elib* library offers easy to use interface for asynchronous programming, which allows programmers create more sophisticated application like *Epuck Joystick*. The goals of the thesis were accomplished.

The library is well documented and several model examples introduce in detail a usage of *Elib library* on easy to understand behaviours. *Epuck Joystick* clearly introduces all sensors and actuators of e-Puck in graphical application. It gives the user a possibility to explore e-Puck's capabilities without programming. In Addition, *Epuck Joystick* gives programmers a guide how to implement graphical application for a program, which uses asynchronous programming in *Elib*.

Enclosed *Elib Tools* application is designed for analysing programs, which uses *Elib*. It is a command line application, which can parse the log from commands sent to e-Puck and collect the information from the log. The source code of the application is also commented and the *Elib Tools* program is ready to be extended or modified according needs of programmers, who use *Elib*.

All programs are written in #. The choice of programming language allows *Elib* to implement help for intellisense, which is very popular among programmers using Microsoft Visual Studio IDE as well as Monodevelop IDE. *Elib* library can run on Linux due to Mono runtime as well on Windows due to .Net runtime. Last but not least, *C#* language is easy to learn and many of students already program in *C#*.

As a whole, the *Elib library* tries to offer programmer of e-Puck as much comfort as possible.

Future works

Most of imperfections of *Elib* results from the implementation of the *BTCom* program, which is the program deployed by default with e-Puck and which allows *Elib* to control the robot. A simple modification of this program would allow *Elib* acquire the frequency of the sound from microphones. Also a modification of *BTCom* would allow to use e-Puck's camera more intensively.

Although there are simulators for e-Puck available, there is a space for another simulator for e-Puck. To our knowledge only Commercial simulator Webots allows control real robot remotely from PC.

Hence, possible solution could be incorporation of *Elib* into an existing simulator.

A. Installation guide of *Elib* library and its tools

The following sections guide user on his first steps in e-Puck programming via *Elib*. First of all the requirements for running a sample *C#* program *TestElib* will be presented. Guide line how to explore and install the *TestElib* follows. The installation of *Elib* and *TestElib* differs according your operating system and runtime. The possibilities are Mono [15] and .Net runtime. Mono runs on Linux as well as on Windows. Microsoft runtime .Net runs only under Windows.

The installation does register neither *Elib* nor *TestElib* into an operating system. It enables a simple copy installation not needing of the administrator rights.

A.1 Requirements for running an e-Puck's first program

Let us suppose, that we have an e-Puck robot with *BTCom* 1.1.3 charged and equipped with default programmes. Let us start with turned off e-Puck. Turn e-Puck on. A green LED shines if the e-Puck is on and the battery is not empty. The low state of battery is signalled by a small red LED next to the green LED. If the red diode is blinking or shining take off the battery and let recharge it.

We need a Bluetooth device, because we want to send commands to e-Puck via Bluetooth. Install it and assure, that it is working. At last, we need the runtime, which is necessary to run a compiled *C#* programs. *Elib* library, *TestElib* and *Elib Tools* require .Net 2.0 or a higher version of .Net under Windows. To run the mentioned programs under Linux use at least version 2.0 of Mono runtime. A graphical application *Elib Joystick* and simple console application Simulator run only under Windows and *Elib Joystick* needs .Net 3.5 or higher.

All programs are aimed at programmers and therefore we recommend the following integrated development environments (IDE), which are serious benefits of *C#* programming. We published all programs in formats of a Microsoft Visual Studio 2008 solution or a MonoDevelop solution. Microsoft Visual Studio 2008 (MSVS) and MonoDevelop 2.4 IDEs can be obtained freely for educational purposes. If at least one of the above IDEs is installed, all prerequisities are fulfilled for a comfort exploration of presented examples. They substitute compiler, editor and debugger and save a lot of work. All following examples suppose, that MonoDevelop or MSVS is installed.

Last but not least we have to check it the e-Puck's selector is at the right position. E-puck is deployed with *BTCom* downloaded to its michrochip. The *BTCom* is saved under the second position of selector. Turn selector directly to e-Puck's microphone, shift it twice to the left in order to choose *BTCom* from default settings.

Prepare *Elib* library for an installation.

A.2 Copy installation

Let us suppose that all preconditions from Section A.1 are met. Let us describe the content of the *Elib* package.

Both .Net and Mono technologies use a solution and projects files to group a source codes of applications. We placed the project and solution files together with the relevant source files in the following folders:

1. Folder elib contains a solution with *Elib* project and *TestElib* project. It contains *Elib* library project itself too.
2. Folder testelib contains only the project console *TestElib*, which illustrates a typical applications of *Elib* library on sample programs.
3. Folder et contains solution and console project *Elib Tools*. It allows process log files generated by programs, which use *Elib*.
4. Folder joystick hides a solution of a graphical application *Elib Joystick*, which introduces e-Puck without programming. On the other hand, it shows how to build a Wpf application, which uses *Elib*, and its multithreaded asynchronous model.
5. Folder simulator contains a very simple windows console application Simulator and its solution. Simulator requires VSPE emulator of serial port. Simulator substitutes e-Puck by repeating constant answers. It allows to test your application without e-Puck.

The most significant parts are located in the first three folders. *Elib Joystick's* crucial programming techniques are described in Appendix B. The only other purpose of *Elib Joystick* is to provide a toy, which can access e-Puck's sensors and actuators without no knowledge of programming. The Simulator application is on the other hand a simple tool for programmers, who profile their application for e-Puck for a better performance. See its code for instructions how to use it and modify it.

The applications et, *Elib Joystick*, Simulator, Elib and *TestElib* can be installed to your computer only by copying. If you are using MonoDevelop, choose folders with suffix "*_monodev*". Copy the appropriate folder to the desired location. In order to see the source files, open the downloaded folder, select the solution file with suffix "*sln*" and open it with your IDE. Build the solution file in MSVS by pressing Ctrl + B and in MonoDevelop by pressing F7. In order to run the program use Ctrl + F5 in both IDEs. *Elib Joystick* and *TestElib* applications use a serial port to a communication with e-Puck. The serial port has to be configured before running the applications.

In order to configure the serial port turn your Bluetooth device and e-Puck on. The devices can be paired together using an operating system specific application for example Bluetooth Places on Windows and Bluetooth Manager on Linux. The applications create a virtual serial port, but before connecting to the port the devices has to be paired. The process of pairing requires a four digit number (PIN), which is the single number printed on e-Puck's body. After pairing the devices, connect to the serial port. The Bluetooth application assigns a serial port name to the e-Puck robot. On Windows the port name looks like 'COM3'. Linux port name has similar format like '/dev/rfcomm1'. The digits at the end of port names differs according to the situation. In order to achieve successful run of *Elib Joystick* or *TestElib* check whether the assigned port name match the port name used by *Elib Joystick* or *TestElib*. If the assigned port name differs from the port name used in *Elib Joystick* or in *TestElib* application, then replace the port name used in the applications by the new port name assigned by operating system.

We believe, that all conditions for a successful use of *Elib* has been presented. Let us note, that the most problems are caused by the hardware. Assure that you switch both e-Puck and Bluetooth on. Take in mind, that the red control for indicating e-Puck's low battery is not reliable, so we suggest fully charging the battery before the first use.

B. *Elib Joystick*

Elib Joystick is a graphical application for controlling e-Puck via Bluetooth. It accesses all sensors and actuators of e-Puck and allows user to control e-Puck interactively. It is inspired by Epuck Monitor [1], but it is written in *C#*, and uses asynchronous model of *Elib* to be more responsive than Epuck Monitor.

B.1 User Guide

Elib Joystick graphical user interface is very simple. The single window of *Elib Joystick* is divided into several areas. See Figure B.1.

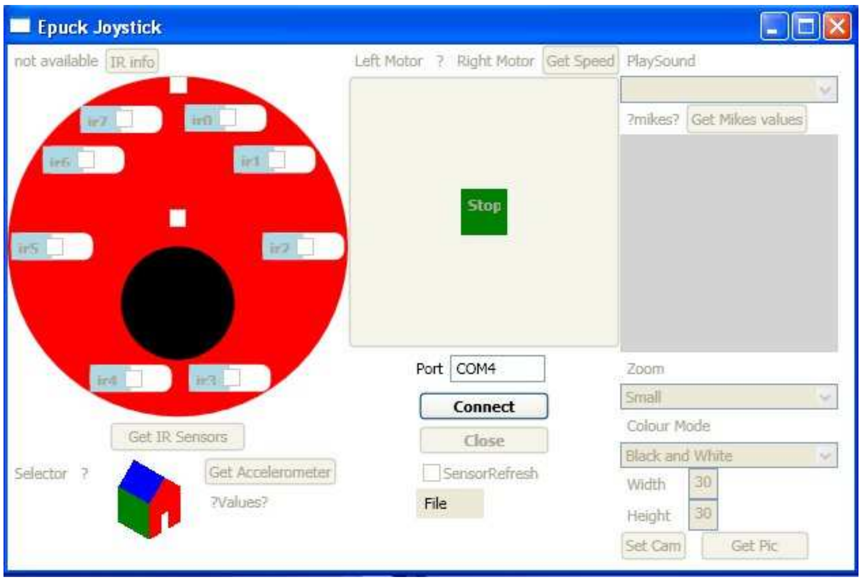


Figure B.1: Elib Joystick start screen

In the middle there is a connection panel. The connection panel is the only panel, which has activated controls after start up. To activate the rest of controls the following actions are necessary to perform. At first pair e-Puck with your computer. Then connect the assigned virtual port with e-Puck. Fill the assigned port's name in text box above the "Connect" button and press the "Connect" button. After pressing the "Connect" button *Elib Joystick* is connected to e-Puck. The rest of controls are immediately activated as soon as the *Elib Joystick* is connected to e-Puck. See A.2 guide for detail informations.

The single function available in a non connected state is Open File Dialogue depicted on B.2. In the Open File Dialogue can be specified the file, where are logged all commands sent to e-Puck after the connection to e-Puck is established.

In order to start using the application connect it to the right serial port and press the "Connect" button. Automatically the position of selector is retrieved and the session is established.

[1] Download at `http://www.gctronic.com/files/e-puckMonitor_2.0_code.rar`

At the beginning all sensors have uninitialised values, which is represented as a question marks in most cases. Press the relevant button for retrieving the sensors value. The values will be displayed in the nearest label. There are two exceptions. The first exception is the visualization of e-Puck's IR sensors. The values of the IR sensors are represented as the blue levels under check boxes on the perimeter of the virtual e-Puck. The second exception is the captured picture, which is displayed enlarged on the right side after its delivery.

If the connection is too slow or the connection is lost, the application is set to initial state. The e-Puck has to be reconnected in order to continue controlling the robot.

Let us remark that checking the Refresh check box can cause the application to be a little unresponsive. It is so, because the application waits on all values of sensors including an image synchronously. On the other hand, a capture and transfer of an image, which was invoked by pressing "Get Pic" button, is performed asynchronously. We have implemented both variants in order to compare the different approaches.

Figure B.3 shows the *MessageBox*, which tells the user to reconnect the application in order to continue.

The actuators of e-Puck can be controlled either by clicking on the appropriate button or check box. Another possibility is to choose a value from combo box. The last option is to specify the convenient values to text boxes and press relevant *set* button.

If you want to change the session press the *Disconnect* button, choose another

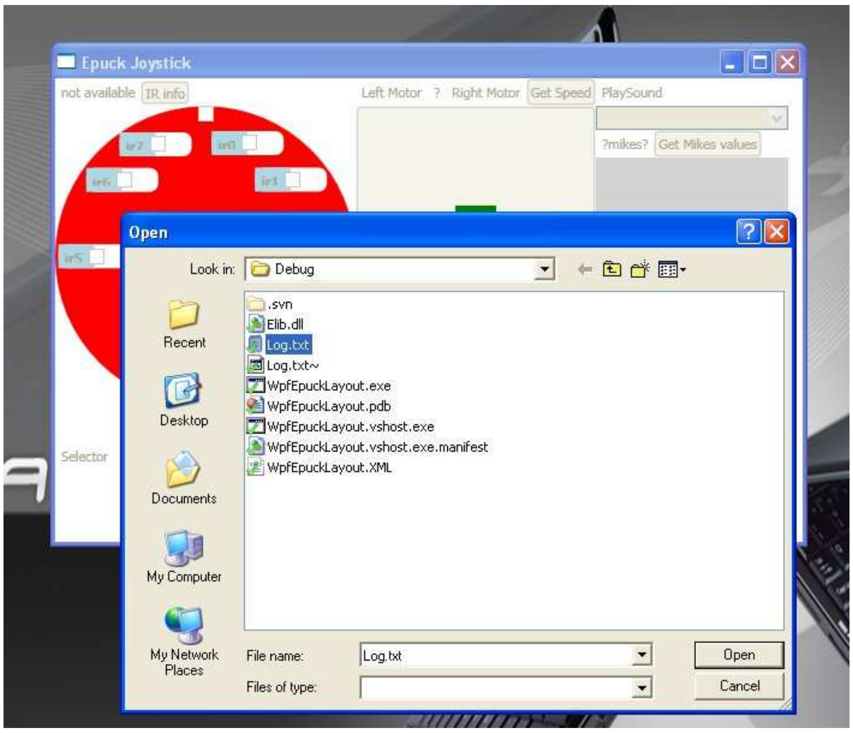


Figure B.2: Optional logging

port and press again "Connect" button or just quit the application and restart it.

B.2 *Elib* usage in *Elib Joystick*

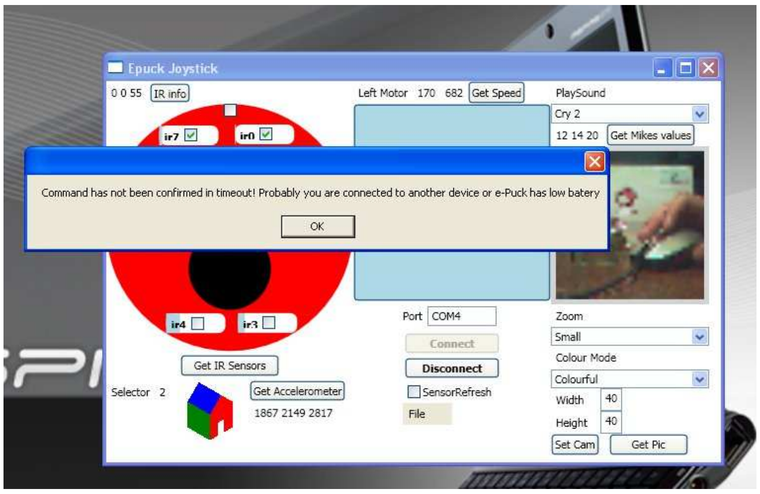
Elib Joystick is a graphical application and does not use any sophisticated algorithms. It also uses *Elib* really simply in most cases, but there is a crucial problem if the callbacks are used.

In *Elib Joystick* we use callbacks to retrieve an image, because it can last a long time. Problems in a graphical application arise if a graphical interface is accessed by a thread different from the main thread, where a message loop of the graphical application is running. The problem is solved, if all functions use the same thread in the so called Single Thread Apartment (STA).

Elib Joystick must solve this problem, because it presents a picture from e-Puck's camera, which is updated in *Label* of *Elib Joystick's* window. The callback, which updates the image, runs in a different thread. The problem is solved by using a *Dispatcher* class. *Dispatcher* accepts functions, which update the GUI from different threads and calls them in the main thread in order to maintain STA character of GUI.

The code snippets B.4 and B.5 present usage of the *Dispatcher* class and synchronous implementation of GUI update. Synchronous update does not require the *Dispatcher* class, because it is performed from main thread.

The *guid Dispatcher* used in B.4 snippet is obtained from the main thread and allows the passed functions to access objects as if they were called from the main thread.



Command has not been confirmed in timeout! Probably you are connected to another device or e-Puck has low batery

Figure B.3: Joystick after Bluetooth connection failure.

```
//Ep is instance of e-Puck,
Ep.BeginGetImage(imgto,
  //the callback(lambda function), which is called after an ↩
     image capture
  (ar) => {
    try {
    Bitmap bmp = Ep.EndGetImage(ar);
    //delegate, which wraps the lambda function that updates ↩
       the GUI
    updateUIDelegate d = delegate {
      //body of the callback, which updates the GUI
      pic.Source = Convert2BitmapSource(bmp,colorful);
    };
    //updating the GUI by passing the update GUI callback to ↩
       guid Dispatcher
    guid.Invoke(DispatcherPriority.Normal,d);
    } catch (ElibException) {
      notConfirmedCommand(this);
    }
  },
  null
);
```

Figure B.4: Update of an image using *Dispatcher*

```
IAsyncResult ar = Ep.BeginGetImage(imgto, null, null);
Bitmap bmp = Ep.EndGetImage(ar);
//pic is a Label
pic.Source = Convert2BitmapSource(bmp,colorful);
```

Figure B.5: Synchronous update of an image from the main thread.

C. *Epuck* class and exceptions reference documentation

Remark. The complete reference documentation can be found on CD, which is enclosed to this thesis.

C.1 Epuck Class Reference

Epuck class has two kind of methods. First kind is built on *IAsyncResult* interface and is much more comfortable. It uses functions starting with "Begin" or function starting with "End" prefix. Usage of "Begin/End" functions is described in Section **??**. "Begin/End" functions are based on functions which uses *Okf* and *Kof* callbacks. See Section 6.2.2 for explanation of how to use the functions with the basic *Okf*/*Kof* interface. We recommend to use *IAsyncResult* interface.

Public Member Functions

Auxiliary methods of *IAsyncResult* interface

"End" functions use *ar* instance to wait synchronously until the command of "Begin" function is confirmed or the timeout from "Begin" function elapses. If the timeout elapses, *TimeoutElibException* is raised. In addition the *EndGetFtion*, *EndGetImage* or *EndInfoFtion* return array of *integers*, a *Bitmap* or a *string*.

- void EndFtion (IAsyncResult ar)

  Returns void.

- int[ ] EndGetFtion (IAsyncResult ar)

  Returns an array of integers.

- Bitmap EndGetImage (IAsyncResult ar)

  Returns a System.Drawing.Bitmap.

- string EndInfoFtion (IAsyncResult ar)

  Returns a string .

Auxiliary methods od Basic *Epuck* interface

- delegate void OkfActuators (object data)

Functions, registered to this delegates, are called if the command for an actuator is successfully confirmed in timeout.

- delegate void OkfIntsSensors (int[ ] ans, object data)

  Functions, registered to this delegates, are called when a command requiring an array of `int` *is confirmed in timeout.*

- delegate void OkfKofCamSensor (Bitmap ans, object data)

  Functions, registered to this delegates, are called in both cases when a command requiring a `Bitmap` *is confirmed in timeout or the timeout elapses.*

- delegate void OkfStringSensors (string ans, object data)

  Functions, registered to this delegates, are called when a command requiring a `string` *is confirmed in timeout.*

- delegate void KofCallback (object data)

  Functions, registered to this delegates, are called when a command of any kind ***IS NOT confirmed*** *in timeout.*

Motors

- void Motors (double leftMotor, double rightMotor, OkfActuators okf, KofCallback kof, object state, double timeout)

  Sets Left and Right Motor speed. Acceptable values are from -1 to 1. Value 1 corresponds to 1 revolution per second. Wheels have perimeter of 12,88 mm.

- void GetSpeed (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  It gets the current speed of both wheels. Speed on a wheel is from -1 to 1. Value 1 corresponds to 1 revolution per second. Wheels have perimeter of 12,88 mm.

- void GetEncoders (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  It gets a current state of encoders. It is measured in steps. One forward revolution corresponds to +1000 steps. It is nulled if the e-Puck resets.

- void SetEncoders (int leftMotor, int rightMotor, OkfActuators okf, KofCallback kof, object state, double timeout)

  Sets encoders values. One revolution corresponds to 1000 steps.

- IAsyncResult BeginMotors (double leftMotor, double rightMotor, double timeout, AsyncCallback callback, Object state)

  Sets Left and Right Motor speed. Acceptable values are from -1 to 1. Value 1 corresponds to 1 revolution per second. Wheels have perimeter of 12,88 mm.

- IAsyncResult BeginGetSpeed (double timeout, AsyncCallback callback, object state)

  It ask for the current speed of both wheels. Speed on a wheel is from -1 to 1. Value 1 corresponds to 1 revolution per second. Wheels have perimeter of 12,88 mm.

- IAsyncResult BeginGetEncoders (double timeout, AsyncCallback callback, object state)

  It ask for a current state of encoders. It is measured in steps. One forward revolution corresponds to +1000 steps. It is nulled if the e-Puck resets.

- void BeginSetEncoders (int leftTicks, int rightTicks double timeout, AsyncCallback callback, object state)

  Sets encoders values. One revolution corresponds to 1000 steps.

IRSensors

- void GetIR (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  It gets the proximity from IR sensors. Obstacle can be recognized up to 4 cm.

- void GetLight (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  Returns a command to get the array of integers from IR sensors. The more ambient light, the lower the values. Usual values are above 3000. Maximal value is 5000.

- void CalibrateIRSensors (OkfActuators okf, KofCallback kof, object state, double timeout)

  It starts calibration of the proximity IR sensors. It makes IR sensors more accurate for measuring proximity. The calibration adapts sensor for different reflection of IR light in the current environment.

- IAsyncResult BeginGetIR (double timeout, AsyncCallback callback, object state)

  It ask for the proximity from IR sensors. Obstacle can be recognized up to 4 cm.

- IAsyncResult BeginGetLight (double timeout, AsyncCallback callback, object state)

  Ask for the array of integers from IR sensors. The more ambient light, the lower the values. Usual values are above 3000. Maximal value is 5000.

- IAsyncResult BeginCalibrateIRSensors (double timeout, AsyncCallback callback, Object state)

It starts calibration of the proximity IR sensors. It makes IR sensors more accurate for measuring proximity. The calibration adapts sensor for different reflection of IR light in the current environment.

LED diodes

- void LightX (int num, Turn how, OkfActuators okf, KofCallback kof, object state, double timeout)

  Sets a LED with number n on,off or into inverse state. Acceptable values are 0..7(resp. 8). Value 8 represents all diodes at once.

- void BodyLight (Turn how, OkfActuators okf, KofCallback kof, object state, double timeout)

  Sets Body led on, off or into an inverse state.

- void FrontLight (Turn how, OkfActuators okf, KofCallback kof, object state, double timeout)

  Sets Front led on, off or into an inverse state. It can produce enough light for capturing close obstacles with e-Puck's camera.

- IAsyncResult BeginLightX (int num, Turn how, double timeout, AsyncCallback callback, Object state)

  Sets a LED with number num on,off or into inverse state. Acceptable values are 0..7(resp. 8). Value 8 represents all diodes at once.

- IAsyncResult BeginBodyLight (Turn how, double timeout, AsyncCallback callback, Object state)

  Sets Body led on, off or into an inverse state.

- IAsyncResult BeginFrontLight (Turn how, double timeout, AsyncCallback callback, Object state)

  Sets Front led on, off or into an inverse state. It can produce enough light for capturing close obstacles with e-Puck's camera.

Camera

- void GetImage (OkfKofCamSensor okf, OkfKofCamSensor kof, object state, double timeout)

  *It gets a picture. It can take a long time. E.g. picture 40*40 in colour takes more than 0.4 sec under good light conditions and with battery fully charged.*

- void SetCam (int width, int height, Zoom zoom, CamMode mode, OkfActuators okf, KofCallback kof, object state, double timeout)

It sets the parameters of a camera. Maximum size of a picture can be 3200 bytes. The picture size S = width∗height bytes; for black and white mode S = width∗height∗2 bytes; for colour mode.

- void GetCamParams (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  It gets current camera settings. The picture size S = width∗height, black or white mode and zoom.

- IAsyncResult BeginGetImage (double timeout, AsyncCallback callback, object state)

  It ask for a picture. It can take a long time. E.g. picture 40∗40 in colour takes more than 0.4 sec under good light conditions and with battery fully charged.

- IAsyncResult BeginSetCam (int width, int height, Zoom zoom, CamMode mode, double timeout, AsyncCallback callback, Object state)

  It sets the parameters of a camera. Maximum size of a picture can be 3200 bytes. The picture size S = width∗height bytes; for black and white mode S = width∗height∗2 bytes; for colourful mode.

- IAsyncResult BeginGetCamParams (double timeout, AsyncCallback callback, object state)

  It ask for current camera settings. The picture size S = width∗height, black or white mode and zoom.

Sound

- void PlaySound (int SoundNum, OkfActuators okf, KofCallback kof, object state, double timeout)

  It begins to play sound. Values 0-5 are for different sounds. 6 turns speaker off.

- void GetMikes (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  It gets the current amplitude of sound in array 3 integers from e-Puck's 3 speakers.

- IAsyncResult BeginPlaySound (int SoundNum, double timeout, AsyncCallback callback, Object state)

  It begins to play sound. Values 0-5 are for different sounds. 6 turns speaker off.

- IAsyncResult BeginGetMikes (double timeout, AsyncCallback callback, object state)

It ask for an array of 3 integers. The integers represent the current amplitude of sound from e-Puck's 3 speakers.

Accelerometer

- void GetAccelerometer (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  It returns vector of values, which indicates the slant of e-Puck.

- IAsyncResult BeginGetAccelerometer (double timeout, AsyncCallback callback, object state)

  It ask for a vector of values, which indicates the slant of e-Puck.

Stop, reset

- void Stop (OkfActuators okf, KofCallback kof, object state, double timeout)

  It stops e-Puck and turn off leds.

- void Reset (OkfActuators okf, KofCallback kof, object state, double timeout)

  It restarts e-Puck. It takes around 1.5 sec. LEDs are turned off and motors are stopped. The IR sensors are calibrated.

- IAsyncResult BeginStop (double timeout, AsyncCallback callback, Object state)

  It stops e-Puck and turn off leds.

- IAsyncResult BeginReset (double timeout, AsyncCallback callback, Object state)

  It restarts e-Puck. It takes around 1.5 sec. LEDs are turned off and motors are stopped. The IR sensors are calibrated.

Textual info

- void GetVersionInfo (OkfStringSensors okf, KofCallback kof, object state, double timeout)

  It gets the BTCom version.

- void GetHelpInfo (OkfStringSensors okf, KofCallback kof, object state, double timeout)

  It gets for Epuck's help sent from e-Puck.

- IAsyncResult BeginGetInfoVersion (double timeout, AsyncCallback callback, Object state)

  It asks for the BTCom version from e-Puck.

- IAsyncResult BeginGetInfoHelp (double timeout, AsyncCallback callback, Object state)

  It asks for Epuck's help sent from e-Puck.

Other commands

- void GetSelector (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  It returns a selector position (array of integers of length 1).

- void GetIRData (OkfIntsSensors okf, KofCallback kof, object state, double timeout)

  It gets the IR data in in array of 3 integers converted from hex number with following meaning. IR check : 0xx, address : 0xx, data : 0xx.

- IAsyncResult BeginGetSelector (double timeout, AsyncCallback callback, object state)

  It ask for a selector position(array of integers of length 1).

- IAsyncResult BeginGetIRData (double timeout, AsyncCallback callback, Object state)

  It gets the IR data in in array of 6 integers with following meaning IR check : 0xx, address : 0xx, data : 0xx.

Public Attributes

- const double WheelDiameter = 4.1

  units: [cm]

- const double Perch = 5.3

  Distance between wheels of an e-Puck in cm.

- const double MaxSpeed = 12.88

  12.88 cm/s is a maximum speed of e-Puck. In Elib 13cm/sec corresponds to 1.00. From -1.00 to 1.00 is the speed linearly growing.

- static readonly int[] IRSensorsDegrees = new int[8] { 10, 30, 90, 170, 190, 270, 330, 350 }

Eight Infra Red sensors are placed on the perimeter of e-Puck, which can be obtained on the instance e of e-Puck by e.BeginGetIRSensors(..) method or by `e.GetIRSensors(..)` *method. IrSensorsDegrees describes the degrees measured from front(There is a cam.) As you can see most of the sensors are on the front side of e-Puck.*

Properties

- static string BTComHelp `[get]`

  It gets the BTCom help.

- int Working `[get]`

  Gets the number of unconfirmed commands in the Epuck instance.

- int NotSent `[get]`

  Gets the number of waiting commands in the notSent queue = commands waiting to be sent via Serial Port(Bluetooth).

- string Name `[get]`

  Gets the name specified in a constructor.

- string Port `[get]`

  Gets the port specified in the constructor.

- bool Log `[get]`

  Return a `bool` *flag, which indicates whether logging is on.*

- TextWriter LogStream `[get, set]`

  Enables sets or get TextWriter of e-Puck, where all actions of e-Puck are logged if logging is turned on.

Exceptions

Elib wraps exception, which can be thrown during its usage, in *ElibException* class. *ElibExceptions* thrown during using of *Elib* can be caused by other exception. For example, an *System.TimeoutException* is thrown if a program tries to connect to port, which is already owned by another process. In *Elib* such situation can happens and *Elib* wraps this exception with *SerialPortException*, which is inherited from *ElibException.* The original *System.TimeoutException* can be retrieved from *InnerException* property. See Figure C.1.

In the example there is shown how every single exception is wrapped and thrown again in *Elib.* There is also depicted a way how to extract the original exception. The code, which catch exceptions from *Elib* would print following output:

```
My model exception
```

The following snippet illustrates the structure of *Elib's* exceptions and introduces all inherited subclasses. The subclasses serves to differentiate the *Elib's* exceptions.

```
// The ElibException is thrown, if an unusual situation happens in Elib.
// It wraps all other exceptions, which are thrown from Elib
public class ElibException : Exception {
  //Only constructors are implemented
}
// The TimeoutElibException is thrown if the "End" function implementing IAsyncResult was called
// and indicates that the answer to command has not been delivered in time.
public class TimeoutElibException : ElibException {
  //Only constructors are implemented
}
// If SerialPort throws any exception, then this exception wraps the original exception.
// After that the SerialPortException is thrown.
public class SerialPortException : ElibException {
  //Only constructors are implemented
}
// ArgsException is thrown if wrong arguments are passed to a
```

```
//in Elib all exception are caught like this exemplary exception
try {
  throw new ApplicationException("My exemplary exception");
} catch (ApplicationException e){
  throw new ElibException("Just an example", e);
}
//...................................................................
//retrieving original exception after catching the ElibException
try {
  //...some functions which use Elib and
  // which throws new ApplicationException("My model exception")
} catch (ElibException e) {
  Console.WriteLine(e.InnerException.Message);
}
```

Figure C.1: How to retrieve the original exception?

```
    function in Elib.
public class ArgsException : ElibException {
  //Only constructors are implemented
}
// Thrown if command to e-Puck has nonsense values.
public class CommandArgsException : ArgsException {
  //Only constructors are implemented
}
// Thrown if session with e-Puck has not started or has ↩
   already ended.
public class UnconnectedException : ElibException {
  //Only constructors are implemented
}
```

The bibliography

[1] C. Chiculita. *Tiny Bootloader.*
`http://www.etc.ugal.ro/cchiculita/software/picbootloader.htm`, 2010.
[Online; accessed 3.8.2010].

[2] M. Corporation. *Lambda Expressions (C# Programming Guide).*
`http://msdn.microsoft.com/en-us/library/bb397687.aspx`, 2010.
[Online; accessed 3.8.2010].

[3] M. Corporation. *The Microsoft Robotics Developer Studio 2008.*
`http://www.microsoft.com/robotics/`, 2010.
[Online; accessed 3.8.2010].

[4] M. Corporation. *Visual Studio Professional 2008.*
`http://msdn.microsoft.com/en-us/vstudio/aa700830.aspx`, 2010.
[Online; accessed 3.8.2010].

[5] Cyberbotics. *Webots 6.*
`http://www.cyberbotics.com/products/webots/index.html`, 2008.
[Online; accessed 3.8.2010].

[6] M. Czardybon. *Delegates (C# Programming Guide).*
`http://msdn.microsoft.com/en-us/library/ms173171(VS.80).aspx`, 2009.
[Online; accessed 3.8.2010].

[7] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, St´ephane Magnenat, Jean-Christophe Zufferey, Dario Floreano, Alcherio Martinoli.
The e-puck, a Robot Designed for Education in Engineering.
`http://infoscience.epfl.ch/record/135236`, 2009.
[Online; accessed 3.8.2010].

[8] http://pyrorobotics.org. *Welcome to Pyro!* .
`http://pyrorobotics.org/`, 2010.
[Online; accessed 3.8.2010].

[9] IFR. *WORLD ROBOTICS 2009*, EXECUTIVE SUMMARY of
1. World Robotics 2009 Industrial Robots
2. World Robotics 2009 Service Robots .
`http://www.worldrobotics.org/downloads/2009_executive_summary.pdf`, 2009.
[Online; accessed 3.8.2010].

[10] S. Magnenat. *Enki* the fast 2d robot simulator.
`http://home.gna.org/enki/`, 2007.
[Online; accessed 3.8.2010].

[11] J. Richter. *Implementing the CLR Asynchronous Programming Model*
`http://msdn.microsoft.com/en-us/magazine/cc163467.aspx`, 2007.
[Online; accessed 3.8.2010].

[12] F. Mondada. *SOUND SENSORS* .
`http://www.e-puck.org/index.php?option=com_content&task=view&id=34&Itemid=5`
2006.
[Online; accessed 3.8.2010].

[13] T. P. Project. *The Player Project* .
`http://playerstage.sourceforge.net/`, 2010.
[Online; accessed 3.8.2010].

[14] M. team. *Monodevelop.*
`http://monodevelop.com/`, 2010.
[Online; accessed 3.8.2010].

[15] Wikipedia. *Mono (software).*
`http://en.wikipedia.org/wiki/Mono_(software)`, 2010.
[Online; accessed 3.8.2010].

[16] Wikipedia. *.NET Framework.*
`http://en.wikipedia.org/wiki/.NET_Framework`, 2010.
[Online; accessed 3.8.2010].