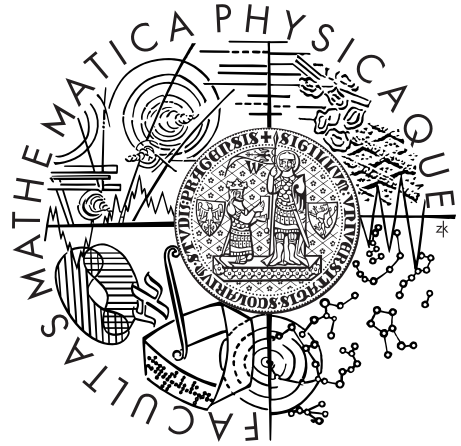
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Petrůšek

Prostředí pro vývoj modulárních řídících systémů v robotice

Environment for modular robot control system development

Department of Software Engineering

Supervisor: RNDr. David Obdržálek
Study Program: Computer science, Software systems

2010

I would like to thank to my supervisor David Obdržálek for his invaluable help, his patience and overall support. My thanks also goes to the MART team members for sharing ideas, encouragement, useful comments, and for building the robots. Last but not least, I would like to thank to my friends and family, especially parents, for supporting me during my studies.

I declare that I have written this master thesis on my own and listed all used sources. I agree with lending of the thesis.

Prague, 6th August 2010 Tomáš Petrůšek

Contents

Title: Environment for modular robot control system development
Author: Tomáš Petrůšek
Department: Department of Software Engineering
Supervisor: RNDr. David Obdržálek
Supervisor's e-mail address: David.Obdrzalek@mff.cuni.cz

Abstract: The subject of the thesis is the design and implementation of a modular control system environment, which could be used in robotics. Both autonomous and guided robots are supported. The higher-level software components like localization, steering, decision making, etc. are effectively separated from the underlying hardware devices and their communication protocols in the environment. Based on the layered design, hardware-independent algorithms can be implemented. These can run on different hardware platforms just by exchanging specific device drivers. Written in C++ using standard libraries, the final software is highly portable and extensible. Support for new platforms and hardware modules can be implemented easily. The whole system was tested on two robots and the particular instances of the systems built using this development environment are included in the solution and partially described in the thesis.

Keywords: modular control system, layered software architecture, robotic software, autonomous robot

Název práce: Prostředí pro vývoj modulárních řídících systémů v robotice
Autor: Tomáš Petrůšek
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. David Obdržálek
E-mail vedoucího: David.Obdrzalek@mff.cuni.cz

Abstrakt: Práca sa zaoberá návrhom a implementáciou modulárneho riadiaceho systému, vhodného pre použitie v robotike. Systém podporuje autonómnych ako aj ručne riadených robotov. V navrhovanom prostredí je kladený dôraz na oddelenie komponent vyšších vrstiev ako lokalizácia, riadenie, rozhodovanie atd., od komunikačných prostriedkov hardwarových zariadení. Vrstevnatý návrh umožňuje implementáciu algoritmov, ktoré sú aplikovateľné na rôznych fyzických platformách iba za výmeny použitých ovládačov zariadení. Finálny program je naprogramovaný v jazyku C++ s použitím štandardných knižníc, je dobre prenositeľný a rozšíriteľný. Do prostredia je možné jednoducho pridať podporu nových robotických platforiem a hardwarových modulov. Systém ako celok bol testovaný na dvoch robotoch. Konkrétne inštancie riadiacich systémov vytvorené v tomto programovom prostredí ako aj ich stručný popis sú súčasťou tejto práce.

Klíčová slova: modulární řídící systém, vrstvená softwarová architektura, robotický software, autonomní robot

Chapter 1

Introduction

Robots and robotic constructions occupy peoples' mind since the times. Robots in the previous century were mostly static, fulfilling just simple tasks. Nowadays, robots are more mobile, user friendly, smart, and "talkative". They stand much closer to humans. The rapid development of hardware calls for better and better software solution not only for the industry, but also for entertainment purposes.

There is no precise definition of a robot which satisfies everyone. Some of the definitions define it as automatically operated machine, that it looks and acts like a human being, it performs complex and repetitive tasks or it is even replacing humans. An electro-mechanical system is usually considered a robot, if it has several or possibly all of the following properties: it is able to move on its own; it can sense the environment and manipulate objects in it; it is programmable; it exhibits intelligent behavior.

> *"A robot is an automatically guided machine which is able to do tasks on its own, almost always due to electronically-programmed instructions. Another common characteristic is that by its appearance or movements, a robot often conveys a sense that it has intent or agency of its own."*
>
> *(Wikipedia)*

1.1 Motivation

Given the vast opportunities in this area (see e.g. the Handbook of Robotics [1]), there are several reasons, why it is good and interesting to develop an environment for modular robot control system development.

Most of the robots share some parts or ideas behind their design. We can think of moving on wheels, legs, decision making, localization, vision, etc. Using the similarities it could be possible to build a system which would be universal for all robots of the same kind. By extending this system it would become more and more universal, and in the end a generally applicable to almost any robot.

Localization is the key feature of every robot. The robot needs to know its position with sufficient precision and accuracy in order to perform useful actions. Otherwise, its actions are strongly limited. The goal of any localization algorithm is to determine the most probable position based on a given set of data supplied by many sensors of variable precision and accuracy. It should be independent of the purpose of the robot. The actual sensors, such as odometry, obstacle detectors, beacons and bumpers, should be the only variable component. Localization is a fairly well researched area, thus there are many possible algorithms to choose from (e.g. Kalman filter, Markov localization based on grid, or Monte-Carlo localization).

The choice of steering algorithms unlike localization is tightly related to the actual hardware design. There are multiple different types of steering: differential steering, Ackermann steering (car-like) or unidirectional steering. Steering algorithms can be further differentiated by the method of control. The first possible solution is to adjust the speed of the left and the right wheel separately. Another way is to control the speed and the direction, similar to the steering wheel and pedals in a car. Yet another option is to use a unidirectional joystick.

Vision algorithms, which are responsible for image recognition, are crucial too. Robots, which do not require computer vision may be autonomous, however not entirely universal. Examples include automatic welding machines found in car factories or robots producing printed circuit boards. Vision is critical for fully autonomous robots, as it provides much more useful information than dedicated sensors. One can use either one camera or even more cameras to provide stereo vision, which is not only able to identify an object's position, but also the distance from the robot.

Robotic hardware and software construction is a great topic for article publication and conference presentations. A broad space for research is available. There are plenty of different algorithms to implement, test, and compare.

1.2 Goals

The aim of this thesis is to design and implement a modular control system environment. By environment is meant the software environment with framework functions, where the modules are loaded based on the concrete instance of the implemented control system. It should provide some API to interact between these modules and to perform some robotic tasks. The modules are relatively small pieces of code, which are exchangeable (if they implement the same interface), and serves communication or computational purposes.

The software will be implemented in C++ with use of standard libraries to be easily portable across multiple hardware architectures and operating systems (with the exception of the low level parts of module implementations). Rationale for these decisions will be explained in Chapter 5.

The environment should be tested on real robots. New control system creation as also module creation will be documented and explained on sample implementation.

1.3 Thesis structure

The text is structured as follows.

Chapter 2 gives an overview of the related work and other existing solutions.
Chapter 3 proposes some key features of the software design.
Chapter 4 introduces the complete design of the system environment.
Chapter 5 lays out how the design was implemented, what technical means were used and how the code is structured.
Chapter 6 discusses the contribution of the thesis and its real life deployment. It also notices some future plans, that could improve the system.
Chapter 7 summarizes the achievements.
Appendices contain the user documentation (App. A) with installation guide, software core functionality description, and a new control system implementation guide. It also describes the hardware platforms and components (App. B) on which this software have been thoroughly tested. The CD contents listing is included as well (App. C).

Chapter 2

Related work

The author of this text is one of the leading members of the MART[1] team. Together with other team members he took part on building the robots, competing in the Eurobot contests [2] and other competitions, and also publications on international conferences.

Throughout the development of this system, three papers have been written. The first one describes the mapping between hardware and software in the design of the robots, the second one deals with robot localization in known environment, and the third is describing the overall (software and hardware) design of one robot. For more information, refer to [3], [4], [5].

This environment was also used as platform for implementation of the MOSYR[2] [6] software project. It was about implementing modules that form a control system as much universal as possible. The final goal was to share higher level modules and strategies between robots built on different hardware platforms with distinct hardware devices.

2.1 Existing solutions

This robot control system environment is not the only project in this area. In this chapter a brief description of several other solutions can be found. Each of the listed projects has some pros and cons but none of them meetings the defined requirements. However, some ideas seen in these projects also helped to find better solutions for some problems.

2.1.1 Microsoft Robotics Developer Studio

Microsoft Robotics Developer Studio [7] is a development, runtime, and simulation environment for robot control designated for academic and commercial application as well as for hobby use. It consists of two major parts, the Visual Programming Language and the Visual Simulation Environment.

The supported programming languages are C++, C#, Visual Basic .NET, IronPython or potentially any other .NET language. For users, who are not

[1] Mat-phys Robot Team

[2] Modulární systém řízení – Modular control system

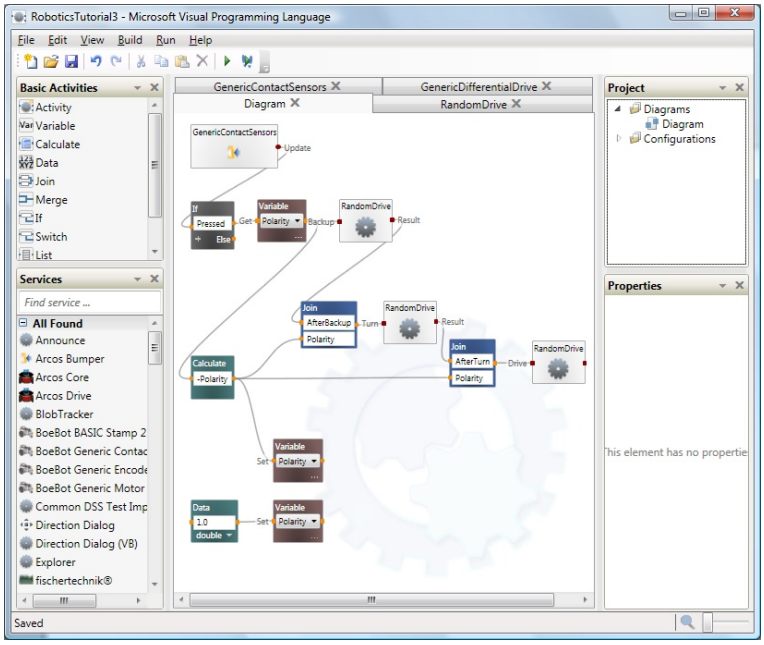


Figure 2.1: Microsoft Robotics Developer Studio – Visual Programming Language

able to create applications in any of the supported languages, a drag & drop style visual programming component called Visual Programming Language (see Figure 2.1) is installed with Microsoft Robotics Developer Studio.

Visual Simulation Environment (see Figure 2.2) is another important component of Microsoft Robotics Developer Studio. It supports physically accurate three-dimensional simulation of the robot's behavior in virtual environment.

Microsoft Robotics Developer Studio is based on Microsoft CCR (Concurrency & Coordination Runtime) and DSS Toolkit (Decentralized Software Services), to enable asynchronous, parallel, and distributed application execution. This framework is also suitable for many other areas outside robotics.

The final application can either be a classic Windows application or it can

Figure 2.2: Microsoft Robotics Developer Studio – Visual Simulation Environment

be a web service with a web-based user interface. The application can either control the robot over network via a connected PC (both wired and wireless connections by Bluetooth or Wi-Fi are supported), or it can run directly on a fully autonomous robot. A necessary requirement is the use of the Microsoft Windows platform.

The product is currently available in multiple editions: commercial Standard (approx. $500), Academic and Express edition, which is free but rather limited in terms of functionality and license.

2.1.2 Player Project

The Player Project [8] is composed of several independent components. The most important one is Player – a network server for robot control.

Player is a robot hardware abstraction layer (in the terminology used in this work, it consists of both the HAL and the HCL layer – see 4 for further information). It provides a TCP/IP based interface to sensors and actuators. On the other end of the connection, there is a so-called player-client, which is responsible for the higher-level robot control. Player-clients are not parts of the project. It is up to the programmer to create a corresponding application for the particular purpose of the robot. In spite of this fact, several pre-created solutions for interactive control are freely available. The key benefit of this philosophy is the freedom to choose any programming language according to the programmer's preferences, as long as it can access the TCP/IP networking stack. The Player Project itself is made available under the GPL license for Linux/Solaris/BSD.

Player controls the particular physical or simulated virtual hardware through drivers. These are provided for a given hardware set. It is also possible to create a custom driver for devices, which are not supported out of the box. A special category includes abstract drivers, which encapsulate a few useful algorithms, such as adaptable Monte Carlo Localization. Abstract drivers rely on other (ordinary) drivers.

The Player Project is tightly coupled with two other projects: Stage and

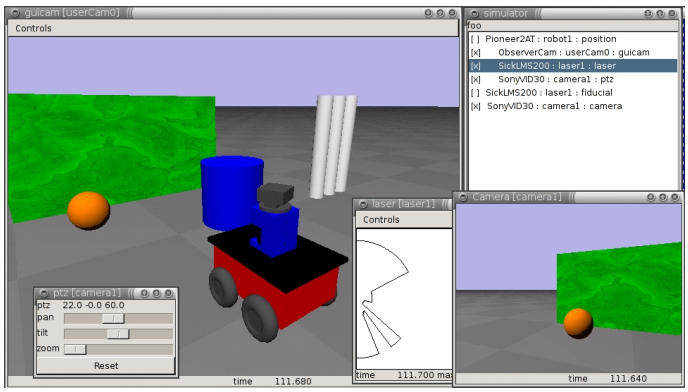


Figure 2.3: Player Project, Stage, and Gazebo

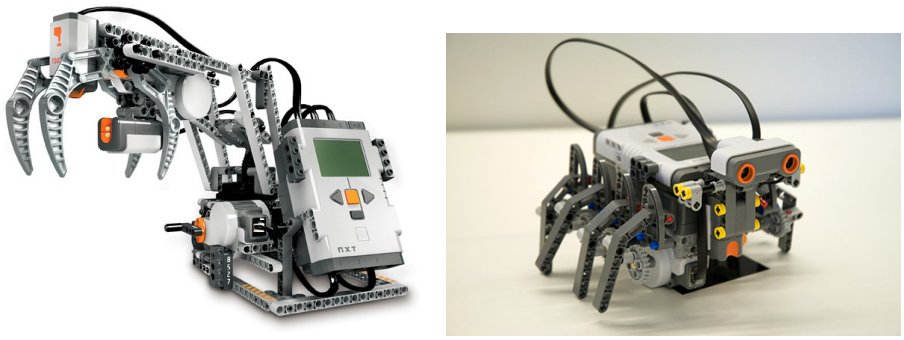


Figure 2.4: LEGO Mindstorms NXT – robotic claw arm and a small octobot

Gazebo simulators. Stage simulates a single or more typically a numerous population of robots in two-dimensional environment (2D bitmap). It is particularly aimed on quick simulation of autonomous multi-robots, without emphasis on the accuracy.

The second simulator Gazebo (see Figure 2.3) simulates one or more robots in three-dimensional world. It is more accurate, and thus naturally much more resource demanding than Stage. Gazebo is able to generate realistic virtual sensor values. Thanks to the built-in implementation of rigid body kinematics, it can simulate interactions between robots as well as with the virtual environment.

The theoretical portability between physical hardware in the real world and simulated hardware in virtual environment without changes to the player-client is one of the main advantages of the Player Project. However, the use of a virtual camera in Gazebo for image recognition tuning is disputable. The main practical difficulty of the Player Project is currently the awkward usability. Even the installation is being described as "tricky" in the official documentation.

2.1.3 LEGO Mindstorms NXT

LEGO Mindstorms NXT [9] is the most recent version of the robotic set from the producer of the well-known LEGO building set. All the required hardware, including a control unit NXT Brick (48 MHz ARM7, 64 KB RAM, three servomotors and several different sensors are included in the package.

LEGO Mindstorms is typically programmed in an intuitive drag & drop software on Windows or Mac OS X (see Figure 2.5). The compiled code is then transferred into the NXT Brick through USB or Bluetooth. Visual programming is however not the only way to create the control software for a LEGO robot. Both the runtime environment byte-code and the NXT Brick firmware are freely available under an open-source license. Therefore, many third-party solutions have emerged. It is hence possible to control the LEGO robot (see Figure 2.4) by programs written in a wide range of programming languages, including C and C++. Languages like Prolog and Python are

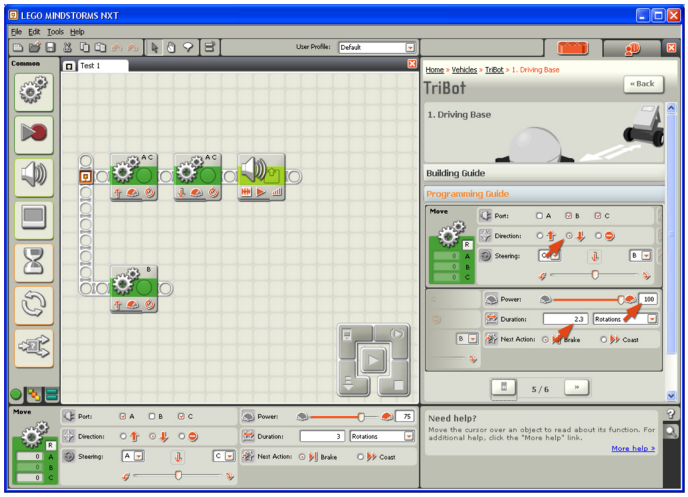


Figure 2.5: LEGO Mindstorms NXT – development software

partially supported too. Such applications must however run on a computer constantly connected to the NXT Brick over a wireless link.

Thanks to the open specification of the hardware peripherals and the connection protocol (IC or RS-485) there are many other third-party hardware components available (e.g. compass, gyroscope, accelerometer), apart from the standard set of sensors (contact, light, ultra-sound range finder and microphone).

It would not be fair to treat this product as just a toy. This platform is a great starting point into the world of robotics, especially for applications, where the described hardware is sufficient.

2.1.4 Others

As it is unfortunately not possible to mention every project in robotics concerning this topic, only couple of other interesting solutions are briefly described. Webots [10] is a professional three-dimensional robot simulator. It can simulate one or more robots, multi-robots, both mobile and walking robots, and even reconfigurable modular robots, which are hard to construct in practice. Webots run on Linux, Mac OS X, and Windows. It can accurately simulate collisions and rigid body kinematics. The control software for simulated robots can be written in a wide range of programming languages. The simulated world is described in a language similar to VRML. Webots is a commercial project, in the price range between hundreds and thousands of Swiss Francs, depending on the license. A free, but less advanced alternative to Webots is Simbad [11]. It is a three-dimensional autonomous robot simulator written in Java. It is designated primarily for scientific and academic use. Simbad performs simulation and three-dimensional visualization of one or more robots, including sensor simulation – cameras, range finders, and bumpers. It does not try to be a realistic physical simulation of the real world. Control programs for the simulated robots can only be written in Java (or Python).

Mobile Robot Programming Toolkit (MRPT) [12] is an extensive set of portable libraries written in C++, designated primarily for mobile robotics developers. Rather than complete robot architecture, it is only set of tools. MRPT covers a whole range of robotic solutions, such as simultaneous localization and mapping (SLAM), computer vision, route planning (obstacle avoidance), processing and visualization of extensive sets of measured data, compression and serialization, 3D (6D) geometry, probabilistic functions and Kalman filters.

Carnegie Mellon Robot Navigation Toolkit (CARMEN) [13], is a toolkit of programs to control mobile robots, their sensors, logging, localization, route planning and mapping. Communication between the programs is provided by the IPC package, which is distributed with CARMEN. Given the separation of the functionality into stand-alone programs, CARMEN is highly modular. It is written in C (not C++) and it only runs on Linux. Carmen's real usability is considerably questionable. The latest version (0.7.4-beta) has some unresolved issues, which have to be kept in mind when using this toolkit (i.e. the necessity to use a laser range finder). At the time of writing this text, it was not possible to try this environment out and it does not seem to be finished yet.

Description and images about some of these and the earlier described solutions can be found in [14].

2.2 Summary

Since several interesting software products in the area of robotics have been introduced, it is necessary to justify the decision to develop a new modular robot control system environment, which is considered more suitable for the specific needs.

The major drawback of Microsoft Robotics Developer Studio is the necessity to use the Windows platform, not only for development but also for the software running on the autonomous robot. Taking into account the choice of the available and commonly used robotic hardware platform, it was better to avoid such limitation. Considering the license costs of Webots from Cyberbotics, usage of this professional and complex tool was not possible. LEGO Mindstorms NXT is too tightly coupled with a particular hardware platform, and hence not suitable for this use. Simbad Project is only a simulator a visualization tool written in Java. It is not designated to control a real robot at all. MRPT has not been applied either, because at that time it was only for localization with Kalman filter and some basic computer vision. However, we have separately used some of the libraries used by MRPT (OpenCV [15]). CARMEN is considered to be the closest solution in terms of ideas. The key differences are their choice of the C language and only program-level modularity.

There was an attempted to use the Player Project with the Gazebo simulator, to test the high-level layer of this system. Instead of accessing the HAL layer directly, a TCP/IP emulation layer to the Player "HAL" had to be provided. Some simple simulations have been successfully set up and run,

but due to Gazebos' high demands for computational power and the lack of available documentation the more complex tests were unsuccessful.

Chapter 3

Problem analysis

The software design, as it will be described later in Chapter 4, was not made all of sudden. It was based on the specification and requirements for robotic platforms and real robots. This chapter is a brief description of the incremental design decisions and requirements.

Hardware independent algorithm implementation (1.1) means, that the algorithm will use devices of the same kind via the device interface. This divides the hardware drivers from higher level functions and leads to a *layered software design*. It is always a question, how many layers are enough.

Communication type and topology between the hardware modules and the mainboard is important in deciding how the lowest, the communication layer will look like. Some solutions (e.g. with RS-232 link) requires message merging and packet building, because if every module would be allowed to transmit its own message, the speed would not be adequate. Other solutions, which use bus topology with addressed messages do not need this, but they need to solve sending serialization. See Figure 3.1 to compare the bus and point-to-point topology. In case of using more communication channels, the device drivers has to be grouped together, to not wait for each others blocking I/O operations if not necessary. This requires *threads* in the communication layer.

There are always problems with speed. Some devices are fast (e.g. encoders) and some are too slow (e.g. range finder) in measuring or in communication. This triggers, that the slow devices' messages are rare and much more precious than the fast repeating ones. If the system would allow pushing the information to the next layer (the algorithm), the device speed would dictate the processing speed of the main computational unit, thus it would create an unwanted requirement for its computational power. The algorithm could be also implemented the way, that it drops some data coming from the fast device, but it is not easy to choose the right data to drop. Taking this in consideration, the *pull communication model* was approved for data-exchange between layers.

To distinct between hardware specific drivers and algorithms, it would be enough to have two layers, one for communication, and one for algorithms (later it will be called abstraction layer). But in case of autonomous robots there is also some logic, which drives the robot, asks for its position and

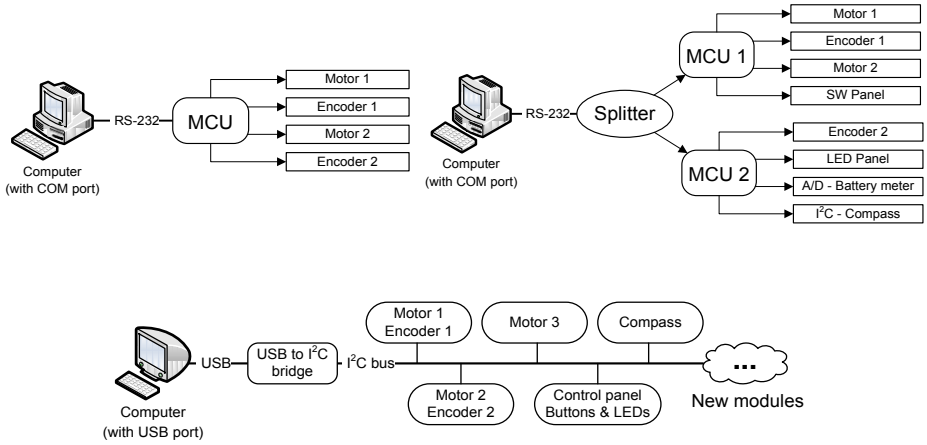


Figure 3.1: Comparing point-to-point (RS-232) to bus (I^2C) topology

achieves some defined goals. There are several ways how to implement this *logic*. It can be hardcoded, or finite *state machine* can be used, or even neural networks or fuzzy logic can be applied on this problem. Thinking about these possibilities, creating a separate layer for this task is a good idea.

Hence with the number of *three layers*, with *modules* in them, communicating using *pull model* and using *threads* for parallelism, the design is much more defined.

Chapter 4

Software design

Well-formed software design based on thorough analysis is essential in large-scale projects. Comprehensible design allows and simplifies upcoming implementation. Moreover, it makes the consecutive project expansion possible in any way.

Scalability is a very important feature of this project. Resulting software should be not only suitable for simple robots (for example those used for educational purposes), but also for complex autonomous robots built for outdoor tasks. Such use-cases differ in code size, amount of mounted actuators and sensors, which goes hand in hand with the amount of loaded driver modules and finally by computing power requirements.

The second most important feature of software in this domain is portability. In this case, portability in more general way is meant, not only between various operating systems but also between various hardware platforms. It is quite common for robotic systems not to run on typical desktop PCs but to use specialized hardware such as embedded devices or even microcontrollers only.

As it was already mentioned in Chapter 3, the design of this control system environment is based on the experience with design, implementation and testing in the previous years. In general, such approach leads to good-quality solutions. The only disadvantage of this approach is the time demand. In this case it means more than three years of testing, participating on various robot contests with the MART team and gradual upgrading of the control software because of requirements creep.

Nevertheless, it was worth it – this project's result is a system that satisfies all our requirements. Applications will not need adjustments in the software core – the only eventual changes would be small edits of existing modules or the creation of new specialized modules for devices that have not yet been used with the project.

In the beginning of this chapter, the control system environment is decomposed into individual layers. Then the focus is moved to the features, tasks, and inter-layer communication. At the end the threading model and one of the most important modules – the localization design is discussed, which is likely to appear in every implementation on a real robot.

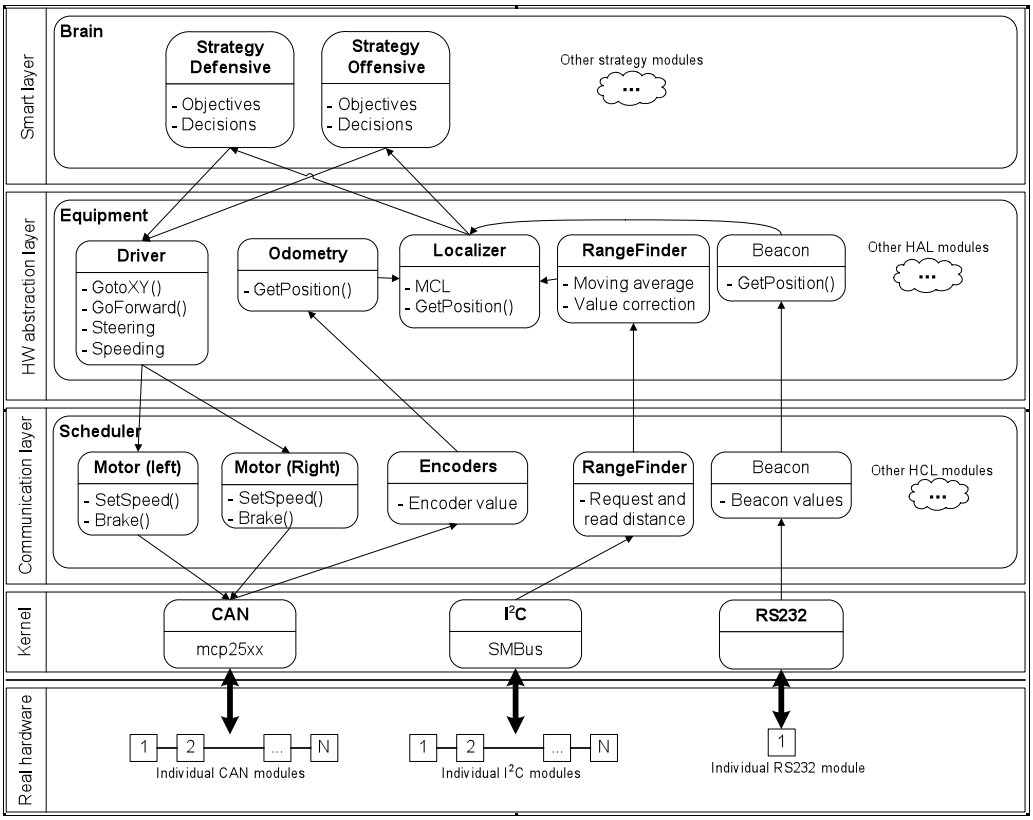


Figure 4.1: Layered software design

4.1 Layers

Layered architecture is very popular in many branches. One example could be client-server applications where layered approach helps to separate data, business logic, and the presentation layer. Another example is communication model design (such as ISO/OSI, TCP/IP, Bluetooth, CAN) where individual layers represent certain level of communication abstraction. Layered design approach brings many advantages. It helps to decompose a complex problem into smaller and simpler parts – layers. Then these parts should be solved in an isolated manner. It also increases modularity, for example, you can replace a particular layer with an alternate implementation if required.

It is very handy to use layered architecture for large software projects as well as for this project. The problem of task decomposition is not always straightforward. The final number of used layers, their purpose, and their mutual connection is deduced from the analysis. This system's architecture consists of three layers: Smart Layer, Hardware abstraction layer (HAL) and Hardware communication layer (HCL). However, looking at problem of the robot's control two more layers can be seen: hardware itself and kernel modules – drivers (see Figure 4.1).

Inter-layer communication is only possible between neighboring layers and this rule is strict. A pull model is being used for communication purposes (with few exceptions). This means that particular modules pull information from the modules on the same or adjacent layers. This proved to be very useful for all sorts of components regardless of their speed. Setting the service call rate prevents modules from being flooded by unnecessary messages coming from fast devices. Otherwise, it would be necessary to implement message prioritizing and some intelligent message dropping system for modules that are unable to process messages on time.

Another kind of communication is inter-module communication used by modules belonging to the same layer. A good example is the robot driving module – Driver, which uses the localization module often – Localizer. Inter-module communication is being used solely on the Hardware abstraction layer.

4.1.1 Hardware abstraction layer

Hardware abstraction layer (HAL) consists of modules, which define certain abstraction above hardware components. These modules may be further combined and form logical blocks representing certain functionality. For example, instead of having distinct modules for left wheel encoder and right wheel encoder, there is logical block – odometry module that encapsulates these encoders.

This layer also consists of modules, which provide interface to access hardware components described in Section B.2. Furthermore, there are modules that combine a certain amount of modules from communication layer. For example, odometry, localization, and robot motion control modules. Odometry gets robot position according to the information from two encoders. Localization processes information from other modules such as the odometry module, rangefinder, beaconing system etc. Robot's motion control modules receive commands from the Brain and then motors perform its actions (in the case of differential driving, the left, and the right motor). As another example you can imagine a robotic hand abstraction on this layer. This would encapsulate several servomotors or other motors for movement and also multiple buttons as boundary position switches or touch sensors of a real hand.

4.1.2 Hardware communication layer

Hardware communication layer (HCL) is the lowest layer in this software. Under this layer, there are only the kernel drivers for the employed hardware or communication libraries and the hardware itself.

The main purpose of this layer is to map hardware devices to software objects. To describe what is understood under the term device mapping, let us consider this example. The range finder hardware component is attached to the communication bus as an independent device. To be able to use it in this system, it also has an object registered in the communication layer, which only exchanges data. There is no special function on this level, only the raw

device data – exactly the same noisy values as this rangefinder provides. To expose its full functionality and the meaning of the raw data, there is also a corresponding object on the abstraction layer. This HAL object can also filter the data to eliminate peaks, noise and non-sense values. This is described in detail in [3].

As mentioned earlier, not many calculations should be performed on this layer, only the communication with the devices. Instances of classes written to represent some kind of hardware drivers register to this layer. The purpose of this layer is to call these registered objects and let them communicate, to get new data from the hardware, send new instructions to the actuators or even setup the devices.

On this layer, there have been already implemented different communication protocols, e.g. RS-232, I^2C , IOCTL for USB devices, or CAN. Implementation and communication are done by the modules themselves, the essential infrastructure (libraries, system calls, files, etc.) is provided by the kernel of the operating system.

4.1.3 Smart layer

Smart layer is the topmost layer of this system. Its purpose is to control the robot. To do this, it uses the Hardware abstraction layer. It can perform autonomous decisions or procedures that could be triggered by human with a remote control (game pad, keyboard, buttons on robot control panel, or another way).

The Smart layer is referred as the Brain. The Brain is pluggable too, as the other layers of the system. The modules here are called strategies and their purpose is different as on the lower levels. Instead of the device communication (HCL) or the device functions (HAL), the strategy contains mission objectives, decision-making, state checking, and understanding of the modules on the abstraction layer. The knowledge of registered HAL objects is essential in this place, as this is the only way to control the robot from this layer.

Decision making in the strategy itself can be done by a finite state machine for instance. Its implementation is simple and the possible use still remains universal. Other decision-making models could be used as well. It is the user's choice. There are many references of successful use of Probability Fuzzy Cognitive Maps or Neural Networks for example.

The flow of consequent actions can also be re-planned asynchronously, by sensing some interruption to the normal flow with the state checking unit of the strategy. This is performed simultaneously with the use of threading (see Section 4.2).

There could be several strategies present in one robotic system. In case of robotic football, we can think about an offensive and a defensive strategy. These strategies can have similar states in their state machines, but the state-transition function will differ. In case of the Eurobot competition, we can find something similar. It is possible to implement a simple strategy for homolo-

gation where the main objective is to score one point with highest possible certainty and some different strategies for the game itself, where the robot is supposed to score as much as possible.

Similarly, as the states are changing in the finite state machine in one strategy, Brain can also change the whole strategy itself. This can be considered as a higher level of decision-making. This strategy exchange can happen on an extensive environment change, when the previous strategy is not suitable anymore.

The ability to have more strategies side-by-side gives the developers the opportunity for further testing and creating new strategies separately without changing the old ones. These strategies can be compared to each other to choose the better, faster, and more suitable approach for the given task. Which strategy is the best is usually not that obvious before conducting the tests. In case of competing with an opponent, the winning strategy can differ, as the opponents are manifold.

4.2 Threading

There are generally various hardware and software components in any robotic system. These components usually require different rate of service calls. After analyzing the character of these demands, it is necessary to assert their correct planning. For this purpose, there is a standard operating system mechanism – thread. It is essential to guarantee that these tasks do not slow down each other, and that they use system resources as efficiently as possible. For example acquiring the compass data (a slow, blocking I/O operation), should not limit robot's decision-making process (computational task on the CPU) on the game strategy level.

There are couple of threads on the communication layer. It is possible to assign a dedicated thread to every device and to define its service call rate. However, better approach is to divide the devices into groups by priority (for simplicity reasons) or by protocol they are using. These groups are defined in the strategy, so different strategies can prioritize different devices. It is also possible to create an individual thread for a device that has a blocking communication routine.

The abstraction layer modules proved to be fast enough to run in one thread. They are usually just computational functions enumerating some values. They do not block the whole layer on I/O operations as the communication layer modules do.

The Smart layer is divided into two threads named master and slave. Master thread is allowed to pause, temporarily or permanently reschedule, or even cancel the actual task running in the slave thread. Consider the following case – the robot has started following its planned route from one spot to another, all of a sudden, an obstacle appears on its route (another robot, human, game element etc.) at this moment it is necessary to react by rescheduling the actual task to the collision-avoiding maneuver. After successful collision avoidance, the robot can continue solving its previous task. On the contrary, permanent

rescheduling is an adequate reaction to unexpected changes in the environment or game conditions.

4.3 Localization

In this place, one possible (and tested) approach to the robot localization is presented as one of the modules of this modular system. This module is described, because it is one of the most important robotic subsystems. The decision-making in known or unknown environment and the driving is much harder to do without the knowledge of the exact position, especially in case of an autonomous robot or a robot remotely driven by human without eye contact. This is why this module appears in almost every robot.

There are several ways how to localize a robot. Some of the possible approaches are described in Section 1.1. The presented module implements Monte Carlo Localization (MCL) [16] as one of the most widely used probabilistic methods. This method seems as the most suitable way as it is well extensible, and the output of MCL is not only the accurate position but also the position as a probabilistic function. Thanks to this, the robot has information of the possibility that is completely lost or that its position is not well recognized in the environment. Similar output can also be provided by Markov localization, even faster. The speed-up is achieved by splitting the area into a grid, so the algorithm is less precise. Robotic platforms on which this system has been deployed and tested always had hardware support for floating point operations, thus it was possible to use the more precise MCL algorithm. On CPU architectures missing the floating-point unit (FPU), implementing Markov, Kalman or another type of localization algorithm is prefered.

As MCL supports different types of inputs, they can be divided into two main categories:

- Advancing inputs
- Checking inputs

The sample system contains two interfaces for these two types of inputs. The device or its abstraction in the Hardware Abstraction Layer implements the corresponding interface based on the type. Hence, the MCL core can use it as its input. The MCL core consults each device when it has new data, and the processing of the samples is being done by each device separately. This keeps the main code easier to read, simpler, and input independent. In addition, the device itself knows the best, how to interpret the raw data it measures.

The level of reliability can be specified for each input device. Then, the samples are adjusted by the devices with respect to their configured credibility. For example: two sets of odometry encoders, one pair on driven wheels and one pair on dedicated wheels, have different accuracy because the driven wheels may slip on the surface when too much power is used. Then, the credibility of the driven wheel encoders will be set lower than the credibility of the undriven wheel sensors. Data sampling frequency is also an important property

of the sensor and should be taken in consideration in the MCL calculations. For further information see [4].

4.3.1 Advancing inputs

Inputs of this type are used to move the samples. Such input could be the odometry for example (processing of wheel encoders). The information provided by these kind of inputs applied to samples is blurred by randomly generated noise, which simulates real sensor imperfection. After advancing the samples, boundary conditions are checked. As a result, the probability of samples representing impossible positions is decreased. There can be multiple advancing inputs. Result advancing information is then computed as a weighted average of all the inputs according to their reliabilities.

4.3.2 Checking inputs

Checking inputs are not affecting the position of the samples. Instead, they are just adjusting their probability (also called sample weights). The reason for this is that inputs of this type do not provide relative difference from the last measurement, but absolute position information. This also does not need to be one exact point, but an area or probabilistic function of position, which fits the Monte Carlo Localization algorithm perfectly. All checking inputs are processed separately; they are regulated only by setting their reliability levels.

As an example, the robot built for the Eurobot 2009 contest by MART team used these sensors:

encoders – combined in pairs to obtain odometry
compass – checks the direction of samples
beacons – checks the distance from stationary beacons
bumpers – checks collisions with the playing field borders and other objects
range finders(infrared, ultrasonic) – checks the distance to borders and obstacles

Chapter 5

Implementation

There are plenty of techniques to implement any program. Popular techniques of agile and extreme programming are well covered both on the Internet and in the book [17]. Contrary the fact that these techniques treat exhaustive design as unnecessary, before the start of the implementation using these techniques, the requirements have been analysed and the whole system was designed well (see 4). This applies to all the bigger subsystems (e.g. driving or localizing units), after testing the main idea with extreme programming (unclean but fast coding which brings results instantly), a well designed version takes place in the system.

Important decision of this project, concerning both design and implementation, was to choose the right programming language. Among the possibilities of Assembler, C, C++, C#, Java, Python, the winning combination was C and C++. The main arguments were – portability, optimization for non-standard platforms and low requirements on computing power. The reasons were the following: the platforms used in robotics are usually small, rather non-standard (almost obscure) devices with low power consumption, often redeemed with reduced computing power.

A project of greater scale, like this, would definitely have a good use for an object-oriented language. C++ object-oriented approach is much more advantageous (compared to options of pure C) there, and it is easier to manage the resulting code. Namespaces and class files helps to keep the code organized.

In this chapter, some implementation related decisions are described, how the layers are implemented and how the modules in these layers look like. Some of the used design patterns are shown and how an object-oriented wrapper is created on top of a C library. In the end of this chapter some larger module implementation notes are present, namely the robot-guiding module and the localizing unit.

5.1 Design patterns

Design patterns [18] are well-described, commonly used, and time-proved solutions to recurring problems concerning software design and implementation. Some of these patterns have been recognized in this system and the design

patterns helped to implement the solution right. In this place the used design patterns are described.

5.1.1 Singleton

Singleton is one of the most widely used creational design pattern. There are several different implementations in C++. In this control system environment the Meyers' singleton patter is used, which is supposed to be one of the safest implementations. The only disadvantage is that because of the disposal order, it is not possible to use lazy instantiation, when the singletons use each other. In this case, this does not cause any problems, because all the singleton objects are needed from the beginning. All the layers have one main object, which manages the registered objects in this layer. These are implemented as singleton objects. Here belongs the Brain for the Smart layer, Equipment for the abstraction layer and Scheduler for the communication layer. Another singleton object is the configuration holder for example.

5.1.2 Observer

As described in Section 3, the pull communication model is used between the layers. All the modules registered in the Hardware abstraction layer are implemented with use of the observer design pattern. They pull the required data from the modules registered in the Hardware communication layer.

5.1.3 Facade

Some of the HAL modules define a higher-level interface that makes some subsystem easier to use. The Driver provides a robot-guiding interface to motor subsystem. In the case of the localizing unit, there are several modules in the localizing subsystem. These are divided into two categories – the advancing and the position checking modules. The localizing unit itself creates a high-level interface to query the robot position, which is calculated from all the modules in this subsystem.

5.1.4 State

This design pattern helped us to implement the Smart layer. There are strategies, which change the behavior of the Brain. In closer look, there are also several states of the strategy, which change the behavior of the strategy itself.

5.2 The core and environment

For a huge, modular system, it is good to have some globally accessible objects helping and easing the code in modules. To avoid global variables the singleton design pattern is used. All these global objects together provide some

useful interface to the system, the layers, configuration, logging etc. and they together are called environment.

Code listing 1 Access singleton classes from the control system environment

```
Conf::inst();
Logger::inst();
Scheduler &hcl = Scheduler::inst();
Equipment &hal = Equipment::inst();
Brain &brain = Brain::inst();
```

For all modules, it is guaranteed, that this base of the system (Core) is instantiated. Hence, the modules can ask these objects for registration; to get references to other modules, they want to communicate with; or any other functions they provide. A part of the environment is also the static description of the playing field and the configuration. For more information, see Section 5.2.6.

The disadvantage of the easy to use environment approach and the use of singletons is, that it makes unit testing harder.

5.2.1 Threading

As mentioned in Section 4.2, threading is used to ease the scheduling of module servicing in the layers according to the required frequency. There are several threading frameworks to use: OpenMP, MPI, pthread and some others. OpenMP is built into the C++ compiler, but it is optimized for calculation parallelization, and it is not very suitable for parallel run of different tasks (although it supports this in the new version). MPI has been created mainly to run in a heterogeneous environment to support distributed computations. In this case, it is a too strong tool and the effort to program in it would be to no avail. In this project the pthread library is used, which is the POSIX standard. It is also possible to use it on Windows via an open-source library. It has all the needed functions and the library calls are quite easy to understand. A small disadvantage of this library is that it has a C API. To make pthread fit better to an object oriented C++ code, some small classes wrapping the functionality of this library are implemented.

"*Thread*" class is the object-oriented approach to *pthread_thread_create* and related functions. Classes derived from the *Thread* class can have a function, which can run in a new thread. These classes are called thread holders. The added value to the pthread library is thread naming, thread chaining, and thread monitoring. It is also not necessary to have the bodies as static functions, the only thing to do, is to implement the pure virtual function *void loopBody()*. Thread chaining is some kind of responsibility model, which helps to start, stop, and join the threads properly (see Code listing 2).

Each thread holder can start its own "children" (*void registerChild(Thread *child)*). After receiving the signal to stop, it sends the signal to stop to all of its "children". After that, it joins them (*void joinChildren()*). When all the "children" threads finish, it joins the "parent".

Code listing 2 Thread management

```
bool start();
void die();
void* join();
```

There are two features for thread monitoring. Their main purpose is to help the developers understand what is happening in the "difficult parallel world". The first one is the dead lock detection mechanism implemented as loop counting (*bool isDeadLocked()*). The developer can easily ask the thread holder if its thread is deadlocked. The second feature is frequency measurement (*double getLps()*). This helps the developers to find out if the thread is running fast enough or to estimate the speed of information propagation between layers.

Mutex (mutual exclusion) is a common synchronization primitive. The *Mutex* class is a wrapper for *pthread_mutex_t*. It holds the mutex and it has an interface to lock (*void lock()*) and unlock (*void unlock()*) it. The pthread mutex initialization (*int pthread_mutex_init()*) and destruction (*int pthread_mutex_destroy()*) are done in the constructor and the destructor.

MutexLocker locks the mutex lock (given in constructor) on construction and unlocks it on destruction (e.g. stack unwinding). It should be used as a local variable (allocated on stack) to make the destructor do the work automatically at the end of the block (e.g. function). The member functions are defined as inline functions.

5.2.2 Brain and strategies

As mentioned in detail in Section 4.1.3, the robot control is designed to be as robust as possible. It was obvious that the robot has to react to situations, which can appear without any dependence on the state. Therefore, in addition to the strategy (state machine in this case), several triggers are created which are checked repeatedly by the *Brain*. Such problematic situations include, for example, a possible collision with the opponent's robot or being stuck after an unexpected bump to the border.

In case such asynchronous situation occurs, *Brain* switches from the actual state to the specific situation handler. Based on the situation character and its handler implementation, upon the handler completion, the previously executed state is re-started, or the state machine is set to the state specified by the handler. To implement this mechanism, simple threading was used. The main *Brain* thread controls the state switching and the special event checks as defined in the strategy. It only performs non-blocking and fast operations. Individual operating state actions are called in the *BrainSlave* thread. The main *Brain* thread may stop this slave at any time and if needed, it may replace it by a thread performing actions of a different state. In this section, the most important methods are described, which are being used in the implemented strategies.

The *Brain* is implemented as a singleton class that provides a basic frame-

work for all strategies, to get information about the environment and to control the robot. It encapsulates the actual strategy, phase, state, action, and their history. It provides logging that enables debug strategies to be implemented easily by calling *void Brain::debugMessage(string message)*.

The *BrainSlave* is an auxiliary static class designed to launch one long-running action from the *Brain*. For instance, to run the action implemented as function *void SampleAction()* it is enough to call the static method *BrainSlave:: PerformAction(ID, SampleAction)* and provide the pointer to this function and the identifier of the action as argument.

Calling this static method is safe, which means that:

- if the currently running action has the same ID, then nothing happens,
- if the currently running action has a different ID, then it is finished and the new action is launched.

Equally safe is to call the function *BrainSlave::TerminateAction()*, which kills the currently running action (if any). The *BrainSlave* only allows running one action at a time. Furthermore, there are many other methods to check the state of the currently running action. For instance, a method (*bool isAction-Running()*) checks if there is any action running at all or how long the action has been running (*double getActionTime()*). Also the ID of the currently running action also can be looked up (*int getActionID()*) and its progress between zero and one hundred percent (*int getActionProgress()*).

The strategy itself is implemented as a class with the interface of *Strategy* (see Code listing 3).

Code listing 3 Strategy interface

```
virtual string getStrategyName() = 0;
virtual string getStrategyDescription() = 0;
virtual void initialize() = 0;
virtual void execute() = 0;
virtual void finalize() = 0;
```

There are two methods to identify the strategy. Methods *initialize()* and *finalize()* are called at the beginning and at the end of the strategy. The most important method is *execute()*, which implements the actual strategy. This method is called in the main loop of the Smart layer thread held by the *Brain*.

5.2.3 Scheduler (HCL)

The *Scheduler* is the main class on the communication layer. As it was described earlier, it is implemented as singleton.

It provides the registration interface (*bool registerBoard(BoardId boardId, Board* board, LoopId loopId)*) for modules on this layer and it also is a globally accessible (*Scheduler::inst()*) entry point for getting references to these registered modules (*Board* getBoard(BoardId boardId)*). Moreover, it creates the threads for module communication groups based on priority and the communication protocol.

5.2.4 Equipment (HAL)

Every abstraction module in HAL layer, called robot's equipment, has to be registered to this singleton object. This singleton is also a thread holder and its function is to service its modules. The modules can be of two general types: *Device* and *Effector*. The difference is described in detail in Section 5.4.2. There are two overloaded registering functions, one for devices in general (*bool registerDevice(DeviceId deviceId, Device* device)*) and one for effectors (*bool registerDevice(DeviceId deviceId, Effector* effector)*).

All the active modules in the abstraction layer run in one thread. It proved to be fast enough, thus no other threads or even prioritizing were necessary. These modules are fast and computational only. They are not communicating with the hardware itself, so they do not lock on any blocking I/O operations. However on very slow platforms localization module can experience some performance troubles which can be solved by using smaller probability cloud (see Section 5.5).

5.2.5 Timers

Timer is a class providing time-related information, as countdown and stopwatch. It is used mainly for profiling – loop speed and frequency counting. It is also used by *Brain* (and strategies) for time-restricted competitions, to schedule the tasks better and more wisely. As mentioned in Section 5.2.1, it is used also for thread profiling (*double getLps()*).

For the loop frequency estimation a moving average has been implemented, to provide more precise information about the thread speed. For algorithm or code profiling in general, functions in Code listing 4 are used.

Code listing 4 Functions to profile algorithms and code in general.

```
void start();
double stop();
double meanTime();
```

5.2.6 Configuration management

Especially in projects in the field of robotics, there is a huge amount of configuration options. Lot of them are just describing the robot properties, others describe the environment. To process this configuration, a good decision is to create and parse configuration files instead of hard coding it. The configuration has to be globally accessible and stored in only one place.

To achieve this behavior, the singleton object – Conf was created. This object can load and save the configuration file (*bool readFromFile(std::string filename)*, *bool saveToFile(std::string filename)*), parse command line (in cooperation with getopt), and return the desired values (*Value get(std::string key)*).

As command line parser the getopt was choosen because of its simplicity. For Windows it is possible to use the open-source XGetopt [19].

5.3 CANBus

The CAN is an industrial standard for bus communication, used mainly in automotive industry [20]. It defines the communication on the physical, transfer, and object layers. For the application layer the widely used and free CANopen protocol [21] is used. As the CANopen stack implementation, a library called CAN-Festival [22] is used. Later it relieved not to be the best decision, as the lack of good documentation, pure C implementation, and the overcomplicated API made the CAN module development much harder and slower. As soon as these problems were recognized, the CANBus wrapper has been created (see 5.3.1).

The *CANBus* class is a library wrapper to provide general CANopen functions and not only to hide the C CAN-Festival library. It makes the use of CAN communication more straightforward and easier. It would also allow us to exchange the underlying library if needed.

CANopen as a protocol supports different states for all connected devices: stopped, preoperational, and operational. Functions to change the state are shown in the Code listing 5.

Code listing 5 Routines to change CANopen node state.

```
void setStopped(int nodeId);
void setOperational(int nodeId);
void setPreOperational(int nodeId);
```

There are two types of messages: SDO message and PDO message. All the message sending, requesting and receiving functions are in Code listing 6

Code listing 6 Routines to send CANopen messages.

```
bool sendSdoWriteMessage(int nodeId, int index, int subIndex, int data,
    int dataSize);
bool sendSdoReadMessage(int nodeId, int index, int subIndex, int* data);
bool sendPdoMessage(int cobId, const char* data, int length);
static void writeSdoCallback(CO_Data *md, UNS8 nodeId);
static void readSdoCallback(CO_Data *md, UNS8 nodeId);
void sendPdoRequest(int cobId);
```

To receive a PDO message, the module should register itself to the CANBus class with the function *void registerHclObject(...)*. The PDO messages from devices are sent on request and are received asynchronously. How to use this module is described in Section 5.4.1.

5.3.1 Changes to the CAN-Festival library

As mentioned earlier, some small changes to the sources of the CAN-Festival library were necessary. These changes involve resolution of some C/C++ compatibility issues and adding missing functionality (e.g. making correct *extern "C"* blocks around pure C code). To allow parsing the PDO messages in HCL modules, the *struct_CO_Data* structure was extended with a function pointer *void (* PDO_Receive_Callback)(CO_Data *, Message *)*, which points to a callback function to be called after receiving a PDO message. The original behavior was parsing the message and pasting its contents into global variables, all automatically generated as C code by a Python tool, without any possibility of dynamic change or programmer interaction. This approach would not allow adding, removing, or reconfiguring any HCL module with CAN without recompiling the whole project. After the mentioned changes, the CAN-IDs of hardware modules are stored in the configuration file and it is possible to change them easily, and to start or stop using particular modules without the need of recompiling the application.

5.4 Modules

For every big project that tends to be universal, modular, scalable, or highly extendable, easy module creation and well-defined interfaces are one of the main aspects of the robustness and overall quality. The aim was also on simplicity, flexibility, and module management. The implementation allows easy module creation, configuration, and dynamic registration of modules in layers.

In the implemented robot control system, there are many different modules, for almost everything, on each layer and even within the layers (think about localization). Proper terminology was needed to distinguish between these types of modules. The Abstraction layer modules are called "devices" – as a functional unit; and the Communication layer modules are called "boards" – like a circuit board.

5.4.1 HCL module

The Communication layer modules are dedicated purely to communication. These modules register into the *Scheduler* object (see Section 5.2.3), which calls their service routines to communicate periodically. Due to the slow communication with hardware, no data interpretation is supposed to be performed in these modules. The only work done in the service routine is the communication via some communication protocol (CAN, I^2C), message parsing, packing, and unpacking, and some A/D conversions (e.g. voltage level to distance). The member variables in these modules should mimic the hardware boards' firmware variables, with which they exchange data. In this form, the board modules in the Communication layer can be considered "drivers" for the real hardware.

The modules in the Communication layer inherit from the "Board" class.

This interface is similar to the Device class for modules in the Abstraction layer. The communication routine *service()* is being called periodically by the Scheduler thread. The *halt()* function is used to prepare the device for power-down. It can, for example, stop the motors or move the arm down slowly to prevent any damage to the hardware. The *getLPS()* and *getTimeResolution()* functions provide communication speed profiling support.

Code listing 7 Board interface on HCL

```
class Board {
public:
    Board(std::string name);
    virtual ~Board() {}

    virtual bool init() = 0;
    virtual void service() = 0;
    virtual void halt() = 0;

    std::string getName();
    double getLPS();
    double getTimeResolution();

    ...
};
```

The drivers for the boards communicating via the CANopen protocol have some special requirements opposed to the I^2C boards. They even need to register to the CANBus object (see Section 5.3) to be allowed to send and receive CAN messages corresponding to their COBID. The important part of the interface interacting with the CANBus class is in Code listing 8.

Code listing 8 CANBoard interface on HCL

```
class CANBoard {
public:
    CANBoard(int nodeId);
    virtual ~CANBoard();

    virtual bool receivePdo(int cobId, byte* data, int length) = 0;
    virtual void requestPdo();

    int getNodeId();
    void setNodeId(int nodeId);

    ...
};
```

5.4.2 HAL module

Modules in the Hardware abstraction layer are registered in the *Equipment* object (see Section 5.2.4). As the name Equipment shows, the purpose of these

modules is to expose some functionality of the hardware devices mounted on the robot. The word device is meant for a complete device, like a hand, odometry, localization, or range finder. It is sometimes a combination of smaller components that are useless alone, like small motors, servos, or buttons on different hardware boards.

All modules in this layer inherit from the "Device" class. It guarantees that all the modules (devices) have a name assigned and it also assures that they implement the functions of the interface. The *init()* function is called to initialize the module, e.g. getting references to other modules, reading configuration, etc. It is called just after all the modules have been registered in the layer (to resolve module dependencies). The *halt()* function is to prepare the device for shut down, it stops sending requests or even cancels the last requests (to the Communication layer modules). The *refresh()* function is called periodically after registering the module in the layer, and its function depends on the module itself.

Code listing 9 Device on HAL

```
class Device {
public:
    Device(std::string name);
    virtual ~Device();

    virtual bool init() = 0;
    virtual void refresh() = 0;
    virtual void halt() = 0;

    std::string getName();

    ...
};
```

Devices that can move some parts of the robot may need some kind of deactivation routine. As an example, it can be used for safety reasons, or according to the Eurobot rules, each participating robot has to turn all the effectors off after the match time limit (90 seconds) expires. For this purpose, there is an Effector class, which adds the *deactivate()* function to the Device interface.

Code listing 10 Effector on HAL

```
class Effector : public Device {
public:
    Effector(std::string name);
    virtual ~Effector();

    virtual void deactivate() = 0;
};
```

5.5 Localization

As described in Section 4.3, the localization module is considered one of the biggest modules in the sample robot control system. The implementation possibilities are the widest possible of any subsystem. This module itself is modular and allows other devices to register in it, thus it is called subsystem. In this chapter, some interfaces are described, the algorithm of Monte Carlo Localization [16], and how information from the beacons contributes to positioning.

5.5.1 Monte Carlo Localization algorithm

MCL maintains a list of robot's possible states or positions. Each state is weighted by its probability of correspondence to the actual state of the robot. In the most common imprementation, the state represents the coordinates in 2D Cartesian space and the heading direction of the robot. It may of course be easily extended to 3D space and/or contain more information depicting the robot's state. All these possible states compose the so-called probability cloud.

MCL algorithm consists of three phases: prediction, measurement, and re-sampling phase.

During the prediction phase, a new value for each item of the cloud is computed, resulting in a new probability cloud. To simulate various inaccuracies that appear in real hardware, random noise is added to each position in the prediction phase. This is very useful. For example, if the wheels were slipping and no random noise was added, the probability cloud would travel faster than the real hardware.

During the measurement phase, data from real sensors are processed to adjust the probability of the positions in the cloud. The probability of samples with lesser likelihood (according to the sensors) is lowered and vice versa. For example, when the sensors show the robot orientation is north-ways, weights for samples representing other orientations are lowered.

The last phase – re-sampling, manages size and correctness of the cloud. Positions with probability lower than a given threshold are removed from the cloud. To keep the number of positions constant, new positions are added. These new positions are derived from and placed around the existing positions with high probability.

As you can see, MCL as particle filter applied to the localization has a very robust algorithm allowing great separation of localizing devices, which leads to its modular implementation. The probability cloud size denotes its computational complexity, and thus the CPU requirements. Many old robots do not use MCL or algorithms combining different inputs. They run on microcontrollers, and there is not enough power to implement and run such demanding algorithms. Nowadays, with more and more powerful embedded platforms, Monte Carlo Localization has become very popular in robotics.

5.5.2 Point and Position

To localize the robot or any other playing element in the environment, some structures describing their position are used. There are more possibilities how to describe this position. If it is sufficient to know just the coordinates of the object, the Point structure is used. To have more information, the Position structure can be used, which contains, apart from the coordinates, the orientation of the object and its probability. As measuring units millimeters are used for the coordinates and radian degrees for the angle. The probability is set between 0.0 and 1.0 in a double.

```
struct Point {
    double x;
    double y;
};

struct Position: public Point {
    double angle;
    double probability;
};
```

To simplify the work with positions (and there is plenty of them), the C++ operator overloading features are used. With this, it is as easy to work with points and positions as with simple types. It allows adding and subtracting relative movement in some direction quickly, getting the distance and viewing the angle between points or positions, work with points as with vectors, and even convert them to other representations (e.g. OpenCV points).

```
Point Point::operator+(const Point point) const;
Point Point::operator-(const Point point) const;
Point operator*(double r, const Point point);
Point operator*(int n, const Point point);
Position Position::afterMove(double distance) const;
double Position::getAngleTo(const Point point);
operator CvPoint() const;
operator CvPoint2D32f() const;
```

5.5.3 Localization interfaces

There are many ways, how to implement the proposed design described in Section 4.3, and many ways how to implement the MCL algorithm explained in 5.5.1. The interaction possibilities are manifold and choosing the right interface is very important. In this section some of the public functions – interfaces in the tested localization subsystem are shown.

The *RobotLocalizer* class, which is a HAL module, has a *MonteCarloLocalization* object. It provides wrapper functions to the whole localization concerning our robot. Other playing elements, obstacles, and enemy robots are

localized with other classes. *RobotLocalizer* initializes the *MonteCarloLocalization* object and ensures that it is being refreshed when possible, while the *MonteCarloLocalization* (see Code listing 11) is purely the implementation of the algorithm itself.

Code listing 11 Monte Carlo Localization implementation

```
class MonteCarloLocalization {
public:
    Position getPosition();

    void addSampleChecker(MCLSampleChecker* sampleChecker, double weight);
    void addSampleAffector(MCLSampleAffector* sampleAffector);

    void refreshCheckers();
    void refreshAffectors();

    ...
};
```

A form of a Sample checker input with definitive weight is *RangeManager*. It is the description of the playing field with all information about the possible and impossible robot positions. All the samples, and what is more important, the final position estimation are checked in *RangeManager* not to return incorrect result to the *Brain* or other module requesting the actual position from *RobotLocalizer*.

To allow dynamic registration of the advancing and checking inputs to MCL, an interface describing each input module was specified. The advancing inputs have to implement the *SampleAffector* interface and the checking inputs the *SampleChecker* interface (see Code listing 12). Both contain a function, which is periodically called by MCL to check or to move the samples.

Code listing 12 MCLSampleAffector and MCLSampleChecker interfaces

```
class MCLSampleAffector {
public:
    virtual void affectSample(int count, Positions& positionChanges) = 0;
};

class MCLSampleChecker {
public:
    virtual std::list<double> checkSample(Positions& positions) = 0;
};
```

5.5.4 Beacons in MCL

The beaconing system consists of three transmitting and one receiving beacon (see Figure 5.1). Information is being passed from the beaconing system to the main computing unit via messages containing the beacon ID (i.e. transmitter identification) and the time difference between the infrared and ultrasonic

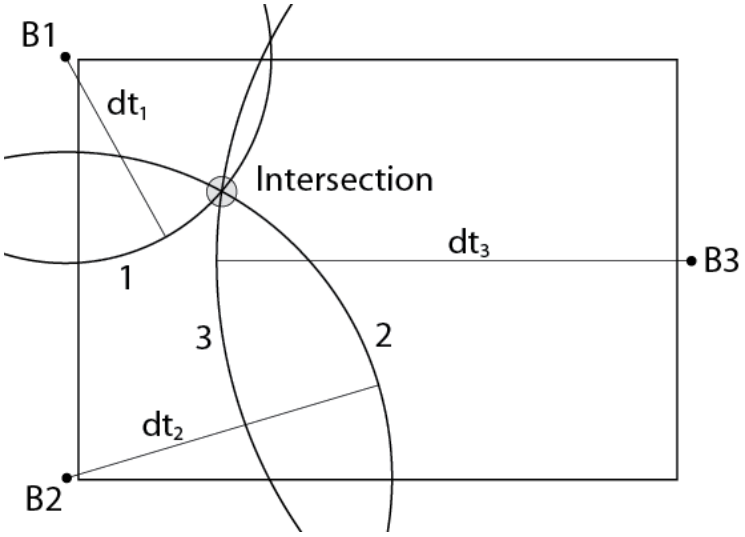


Figure 5.1: Beaconing system – transmitting beacons on the sides marked with B1, B2, and B3 receiving beacon in the Intersection

transmissions. The hardware itself and the physical principle of this absolute positioning subsystem are described in detail in [4].

There are two reasons why each message contains the time difference (delta) instead of the calculated distance: computational power of the micro controller and the degree of robustness. The main computing unit is much more powerful than the receiving beacon, so we let the beacon do less work and we even benefit from this decision. We considered deltas to be the perfect raw data for our purpose – distance measurement. The computation is done in the main computing unit, which controls all the other devices and it is highly configurable. It means that all the parameters of the equation for the distance calculation can be changed easily without the need of adjusting the beacons' hardware or the firmware of the devices. It even allows us to calculate or adjust the parameters on the fly, if distance information is provided based on external measurement.

The configuration of the main computing unit contains not only the important constants for the equation, but also the positions of the transmitting beacons. As we know the distance and the beacon ID, we can increase the weights of the MCL samples in the circular belt formed by these two values and the range constant. MCL samples far from the belt are penalized (see Figure 5.2). This approach is much better and more robust than just waiting for intersections and then computing the robot position using simple triangulation. These intersections may not happen very often because of the time gap between individual beacon transmissions (especially when the robot is moving fast). At the same time, it is good to implement different weighting for the samples on a belt, near an intersection of two belts and near the intersection of all three belts.

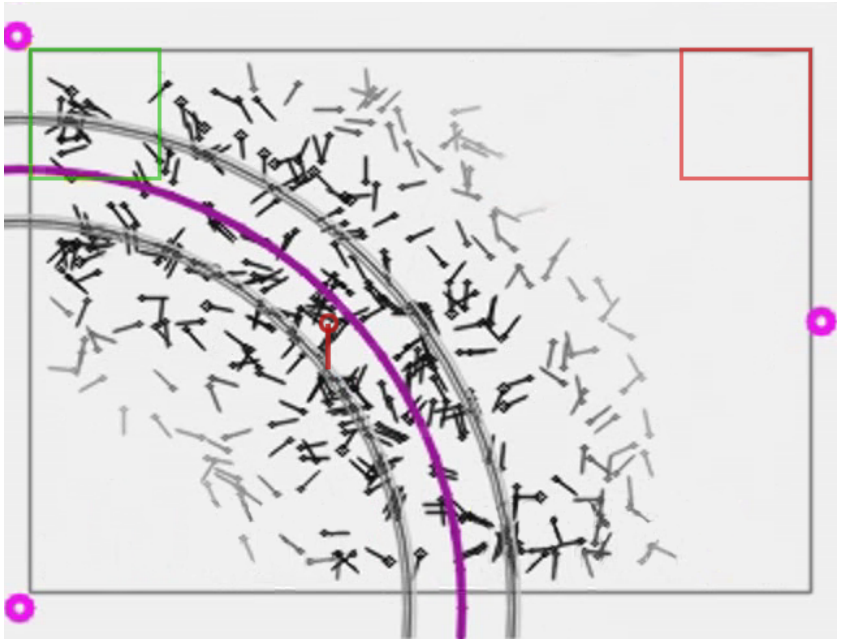


Figure 5.2: MCL after processing one beacon input. The circular belt marks the input from the bottom left beacon. The "pins" represent oriented MCL samples; sample probability is proportional to their darkness.

5.5.5 Entirely lost

MCL can also determine the robot position from scratch if it has some absolute sensors. The only change to the algorithm itself is the re-initialization of the sample cloud. At the beginning of the localization (when the robot is lost) samples are spread uniformly all over the playing field. The sensors providing absolute positioning information lower the weight of misplaced samples and new samples are placed in regions with higher probability (see Figure 5.2). This is repeated until a sufficiently reliable position estimation of the robot is reached.

5.6 Driver

Another important module is the "Driver" module. It is responsible for the robot movement and will be present in every mobile robot. This section describes some possibilities and also tested solutions, how to implement mobile robot driving in this control system environment. A complex API is demonstrated which can be used by the Brain.

There are several ways how to guide the robot and this depends mainly on the hardware construction. Differential steering by two powered wheels with attached encoders providing odometry information is one of the most seen practice. This system with this driving module has been developed and tested on this kind of robots. The bodies of the robots were always supported by one unpowered and uncontrolled caster.

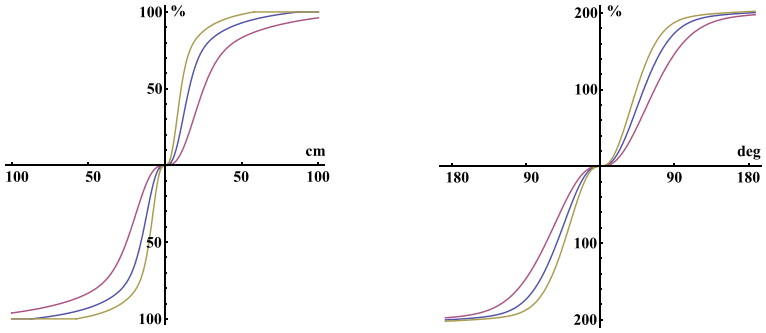


Figure 5.3: Autopilot control curves for different robot weights: Speed to Distance (left), Motors speed difference to Angular deviation (right)

Two driving methods are created, which use absolute and relative movement, respectively.

5.6.1 Absolute movement

The absolute method drives the robot to a certain place (for example, on the playing field); until this place is entered, using absolute coordinates. For successful navigation with this movement method, at least estimative position knowledge is required (provided by the Localization module). The command for this method is *void gotoXYA(Position pos)*. Calling this function should cause robot movement to the desired coordinates *"XY"* and turning to the desired angle *"A"*.

The structure *Position* contains this information and provides a few helper functions like position *operator+ (const Point& point)* for advancing the position or *double getAngleTo(Point point)* for angle estimation from the old to a new position. This structure also contains the information about probability (variable *double probability* should be in range from 0.0 to 1.0). This is estimated by the localization module and it can carry information whether the robot is entirely lost or the exact position is more or less known. In a probabilistic localization model like MCL, this is set according to the definition. In another models it can be found out with additional mechanisms and can only be set to 0.0 (lost) or 1.0 (position is known). The actual movement was controlled by "Autopilot" in this mode, with the help of two curves (see Figure 5.3). One curve shows the dependency between the speed and the distance to the goal, the second curve shows the dependency between the difference of motors speeds and the angular deviation of the actual robot heading to the desired heading. For further information see [5].

The Autopilot control functions are designed so that the robot slows down when approaching the goal position and thus can arrive to the required position more precisely (this applies to both the x-y placement as well as the angular rotation of the robot). Before the movement was implemented in this way, the

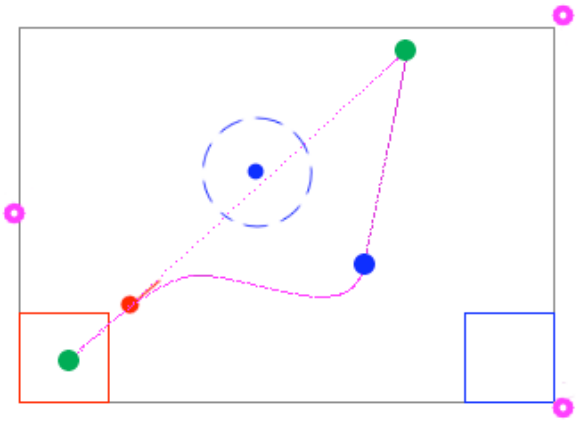


Figure 5.4: Route re-planning after the opponent has been detected. The robot travels from bottom left to top right position, its original route shown as dotted line, new route as solid line (screenshot)

robot could exhibit oscillation around the desired position because of its mass and the traction characteristics.

Both curves are defined by two parameters which should be set according to the robot body construction and should be recalibrated when the wheel-frame is modified or when the robot weight changes considerably.

5.6.2 Relative movement

The second driving method is a relative movement. It is used for precise positioning of the robot. For example, this method is used to adjust the robot position precisely if needed, when the robot is already in the "right place". Another use for this method was to recover it from blockage at the border. In such case, it is quite likely that the robot does not know the exact position. Otherwise, it would not drive itself so that it collides with the border. Therefore, the relative movement is the only right way. Two typical commands for this driving method are *void goStraight(double distance)* and *double rotateBy(double angle)*.

5.6.3 Advanced planning

The combination of Autopilot functions allows driving in non-trivial curves. Using purely those functions, the motion would be limited to a simple point-to-point navigation. However, because of the action planning, it is possible to create a list of successive sub-goals. It would be nice to be able to create a curve that represents the optimal path through all the sub-goals to the final position. In case of the Eurobot 2008 contest, that could be used to collect more balls on the table during one continuous motion and bring them to the destination container. As we show in Figure 5.4, it can also be used for collision avoidance by simply inserting a new sub-goal into the goal list.

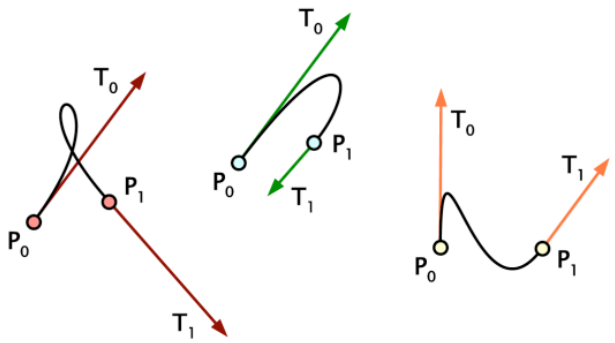


Figure 5.5: Examples of Hermite curves (P0, P1: control points, T0, T1: tangents)

Simple planning of the motion from one point to the next one is not feasible as it does not guarantee smooth joints and the robot cannot physically make instant movement changes. Therefore, more advanced algorithms had to be found, than just simple joining of the curves and to create one segmented curve with "nice behavior", at least to provide continuous and smooth joins. Hermite curves[1] were chosen, because these curves meet the expectations, they are easily calculable, and on the top of it, they pass through their control points (see Figure 5.5). To calculate one curve segment only the positions and the tangents of its two endpoints need to be known. If more segments are combined to create a joint curve and the tangents for the join points (the "checkpoints") are the same, the resulting curve meets the expectations of smooth joints. For this purposes, it was even possible to make the tangents optional, which lets the input to be minimal.

The tangent vectors can be sensed as analogous to the direction and speed of the curve at that point. Reasonably good compound curve can be acquired, when the tangents always point to the next checkpoint, while minimizing inputs for the calculations. The interface allows adding, removing and changing the curve checkpoints to form a queue. After the next checkpoint in the queue is reached , it becomes the starting point. Figure 5.6 shows an example of a planned trajectory to collect all balls detected on the playing field surface during the so-called "harvest" phase. The ball positions are used as checkpoints, and as such, they define the curve segments. If a new target is detected along the robot journey, it can be easily added to the already existing list as a new checkpoint.

Unfortunately, this extension to the *Driver* module (called *Maneuver*) cannot be declared as finished. It was implemented and tested in simulations, but it has not been tested on real robots yet.

[1]Cubic Hermite spline (also called cspline), is a third-degree spline with each polynomial of the spline in Hermite form. The Hermite form consists of two control points and two control tangents for each polynomial.

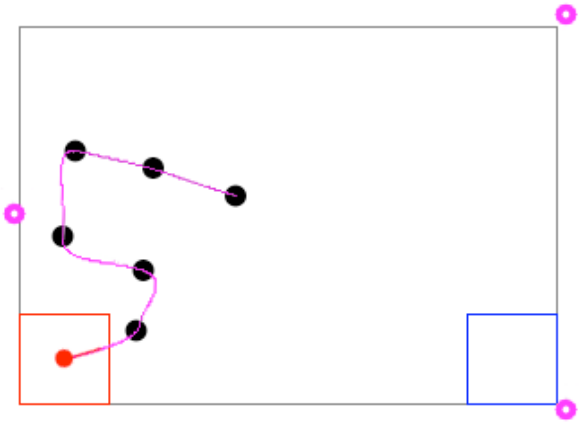


Figure 5.6: Example of Hermite curves used for harvesting balls (screenshot)

5.7 Joystick

Joystick is considered a special device in our system. It is special because its function is not to measure or move some parts, but mainly to control the robot, drive it, and to reconfigure some other devices. It can be considered as "brain functionality", where the impulse to act is coming from the joystick held by a human. To keep as much great ideas as possible, from the design in the implementation, the code for the joystick is split to all three layers of our controlling system based on the functionality of the code.

Joystick is still a hardware device connected to the control system, so it has an object registered in the Communication layer and in the Abstraction layer. The Communication layer object polls the joystick file (created by the Linux kernel) and gets the user actions, button presses, and joystick axis movements. The Abstraction layer object is a pure abstraction for the joystick functionality. It provides interface for the upper layer to register callbacks actions on the joystick events. These events do not need to be strictly to guide the robot. They can trigger some other handling routines, or work like hints for the Brain and the loaded strategy.

The controlling functions of the joystick are defined by the joysticks strategy object. This strategy can be loaded into the *Brain*, to be executed and all the functions will be registered as callbacks to the user action. In this layer, it is normal to use other, strategy specific devices, so it is not necessary to change the lower-layer code for a different robot, where other components are connected to the main unit. This implementation decision, to follow the design decision also in this place, supported the idea of code separation by its functionality.

Chapter 6

Evaluation and future work

6.1 Real life deployment

This robot control system environment, with some additional modules (partially described earlier), was developed and tested on two robots built by the MART team.

Both robots have two traction wheels and use differential steering. Their bodies are made of aluminum profiles and Plexiglas. The robots were designed and used for Eurobot autonomous robot contest [2]. These robots have almost rectangular ground plan with maximal dimensions allowed by Eurobot contest rules. Among others, they carry all electronics (including the main computational unit) and lead-acid accumulators.

Eurobot 2007 and 2008

The first robot (Figure 6.1) called Logion has a simple "harvester" manipulator designed to fulfill the objectives of Eurobot 2008 contest – collect, transport, and deploy balls. This robot already took part in the Eurobot 2007 contest and it has been refactored later. At first for the participation in the "outdoor" Robotour 2007 contest, robot Logion was equipped with an alternative bogie with bigger wheels. This robot has been presented on the Eurobot conference and in the North Star competition in St. Petersburg (Russia) in 2008 too. Its parameters are shown in Table 6.1.

Eurobot 2009 and 2010

The newer robot (Figure 6.2) was designed for the Eurobot 2009 contest and it had a hand-manipulator with three degrees of freedom to gripe and hold wooden columns and a claw to take wooden columns out of the dispenser. Later it was rebuilt to met the Eurobot 2010 contest rules. The mechanical solution is more robust, electronic boards are fixed by communication bus connectors (CANON-9) to the body of robot. In addition, several professional hardware components, e.g. Maxon motors with gearboxes or EPOS industrial control units, are used there. Its parameters are shown in Table 6.2.

Main computational unit:	VIA EPIA – IA-32 (x86)
Communication bus:	I^2C (via USB-I^2C converter) RS-232 (serial link) USB
Hardware modules:	HBmotor boards (I^2C device) commutator motors, encoders, switches MCP23016 board (I^2C device) buttons and LEDs SRF02 (I^2C device) ultrasonic distance measuring sensor CMPS03 (I^2C device) compass module Localization beacons (RS-232 device) Webcams (USB)
Contests:	Eurobot 2007 Robotour 2007 Eurobot 2008 North Star 2008

Table 6.1: The robot constructed for Eurobot 2008

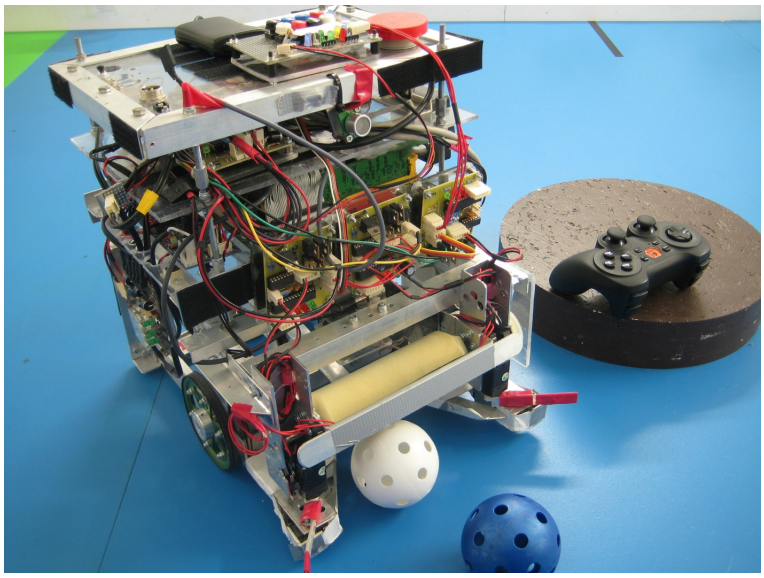


Figure 6.1: The robot constructed for Eurobot 2008

Main computational unit:	Beagle Board – ARM (with expansion board)
Communication bus:	CAN-bus (via SPI) RS-232 (serial link) USB
Hardware modules:	Motor board (CAN device) commutator motor, encoder Stepper board (CAN device) stepper motors, switches Servo board (CAN device) modeler servos, SHARPs, switches EPOS (CAN device) Maxon motor, encoder Human interface board (CAN device) character-LCD, keypad Localization beacons (RS-232 device) Webcams (USB)
Contests:	Eurobot 2009 Eurobot 2010

Table 6.2: The robot constructed for Eurobot 2009

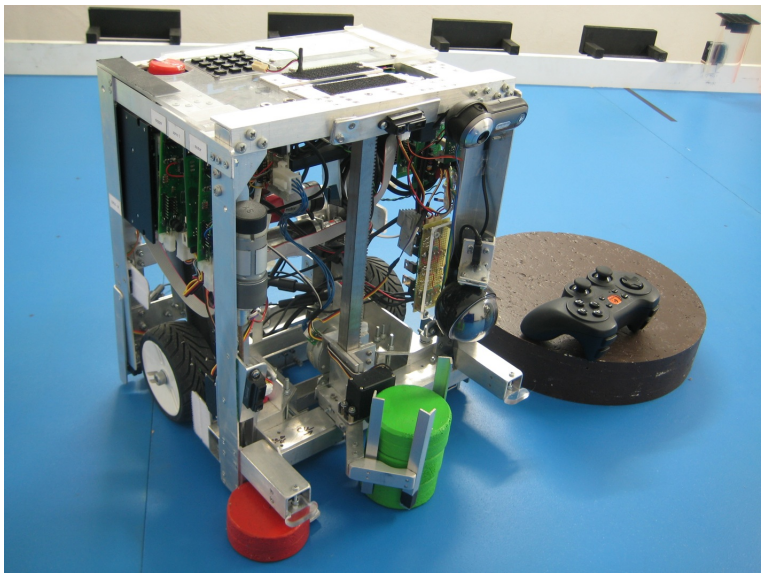


Figure 6.2: The robot constructed for Eurobot 2009

6.2 Future work

The completion of a project like modular robot control system development environment is an opportunity to think about the future direction of its development. Although the intended targets were mostly met, there is a wide range of other options to extend this system. The most important of them are introduced in this chapter.

6.2.1 Portability

Because of using some platform dependent libraries, the robot control system development environment in the current version can only run on Linux systems. A possible improvement would be to adjust the system to be platform independent, which would increase the possibilities of usage and also development. This involves finding or developing a proper replacement for these libraries (CANopen stack). The source code would have to be modified too, all direct calls to platform dependent functions would have to be replaced by their platform independent equivalents.

6.2.2 Real-time environment

The developed control system environment is not a real-time system. By "real-time" in the usual sense is not understood a system necessarily fast, but a system with guaranteed response time, under all circumstances. Due to the intended use of the control systems with this environment, it should not be considered as a drawback. The analysis and development of a real-time system would be far more complex and the system would probably require a more powerful hardware to run. None of the similar solutions described in Section 2.1 is a real-time system.

Considering the costs, the extension of the existing system to meet the requirements for a real-time system, is not being planned. It is necessary to realize that the control system built with this environment should not be used in applications where it could threaten lives, health, or significant material value, fully autonomous car driving for example.

6.2.3 Distributed environment

Control systems based on this control system environment are designed to run on one computer, with all the modules connected to it. The only possibility to achieve some kind of distribution is to make smarter hardware modules. This still might not be satisfactory, if the used mainboard platform does not support the peripheral interface of the designed hardware modules. This could be achieved by making the environment transparently distributed on more computing units.

From other point of view, with distributed system it would be possible to use more, but smaller platforms as computational unit. These could have much smaller power consumption than one powerful unit. Also the modules in

this modular system could be load balanced and migrated based on the needs, e.g. memory size, processing speed, FPU[1], computer vision instructions etc.

6.2.4 Runtime module loading

The control system environment loads the necessary modules to all layers based on the configuration in the loading strategy. This is satisfactory when the strategy is simple and all the actuators and sensors are known while loading and the necessary modules are compiled with the project. A more flexible approach would be dynamic - runtime loading of all modules, which could make module changing, compiling and testing much easier and possibly event driven.

6.2.5 Movement modules and Hermite curves

The current module for movement is able to drive robots with differential steering. It is a popular design. The construction of these robots is relatively simple and such robots are quite skillful. Other robot designs exist however too, especially robots using Ackermann steering and unidirectional robots. To control these robots we would need to add appropriate modules to the Hardware abstraction level.

The current route planning is simple and reliable, but as described in [5], there is a more elegant solution, for the time being, waiting for its implementation:

"Simple planning of the motion from one point to the next one is not feasible as it does not guarantee smooth joints and the robot physically cannot make instant movement changes. Therefore, we had to seek more advanced algorithms than simple joining of the curves and to create one segmented curve with "nice behavior", at least to provide continuous and smooth joins. We decided to use Hermite curves, because these curves meet the expectations, are easily calculable and on the top of it, they pass through their control points. To calculate one curve segment we need only to know the positions and the tangents of its two endpoints. If more segments are combined to create a joint curve and the tangents for the join points (the "check-points") are the same, the resulting curve meets our expectations. For our purposes, it was even possible to make the tangents optional, which lets the input to be really minimal."

6.2.6 New hardware components

Although hardware is not an essential part of this work and although most of the hardware components have been designed, there is some space for further work in this area. It is worth to think about the development of a programmable CAN-I^{2}C converter, which would enable use of sensors with I^{2}C interface (CMPS03 compass, SRF02 rangefinder, etc.). These were used on

[1] Floating-point unit

the previous robot with I^2C bus. Another potential improvement could be a new hardware component for battery voltage monitoring, current and charge status, or an expansion of current Motor board that will be able to process encoder data autonomously and take over some responsibilities in robot localization.

6.2.7 CANopen stack

As already mentioned in Section 5.3, the experience with the CAN Festival open-source library used in this project has not been completely positive. This forced us to build our own abstraction over this library. For the final list of topics for further work, it is worth to consider the development of our own CANopen stack, which would directly meet all the requirements of the modules and also the control system environment.

Chapter 7

Conclusion

The goal of this work was to design and implement an environment for modular robot control system development. The proposed environment features many advantages over single purpose, dedicated systems designed for particular robotic hardware. Systems built for this environment are structured, the hardware dependent code is separated from the implemented algorithms or decision making, thus these systems are easily reusable on other robots. New system and module creation is well documented and working examples are supplied.

In this text the advantages of modular control system environment have been described (1.1). Several other solutions have been consulted and their unsuitability was shown (2.1). After the problem analysis the software design and implementation is discussed (5). The layered design is explained (4.1), Monte Carlo Localization algorithm implementation is proposed (5.5), and also the usefulness of modular implementation is shown (5.4). Different approaches to steering and robot guiding are described (5.6 and 5.7).

The software developed as part of this work was written in C++ using object-oriented design principles including the use of several well-known design patterns (5.1). It was tested on two different robots (6.1). These robots were successfully used on the Eurobot autonomous robot contests and also other competitions. In the appendix the User documentation takes place with the installation guide. The supported platforms are introduced as well as the communication protocols used by the robots.

Appendix A

User documentation

A.1 Installation guide

Hardware requirements

The control system environment does not have any special hardware requirements. Generally, any hardware platform, which is supported by the Linux kernel and the particular Linux distribution, should be suitable. However, a CPU with built-in FPU (floating-point unit) is strongly recommended as the software heavily depends on floating-point operations. The software has been developed on and is thus guaranteed to work on the following architectures, which are described in detail in Section B:

- VIA EPIA – IA-32 (x86)
- Beagle Board – ARM
- High performance PC – AMD64 (x86-64)

Any recent processor should be powerful enough to run Linux without graphics interface smoothly. However, advanced computer vision algorithms tend to be very demanding on resources. In this case, the hardware platform must be chosen carefully.

The system was tested on two robots, as described in 6.1. These robots use the following buses/protocols:

- RS-232 (serial link)
- I^2C
- CAN-bus
- USB

In order to use modules, which depends on a particular hardware component, the robot computer must provide a suitable bus to connect the device (e.g. USB for joystick, webcam). To allow convenient package installation and software development and deployment, it should support some kind of remote connectivity, such as Ethernet, Wi-Fi or Bluetooth.

Software requirements

The control system environment can be currently compiled and deployed on any UNIX compatible operating system, if the required development tools are present and the dependencies are satisfied. In this document, we present a guide how to configure a suitable environment to successfully compile and run our software system. GNU Linux is the particular UNIX-like system of choice, as it is widespread, freely available, well documented, and it offers a wide range of software packages. We do not assume any particular Linux distribution.

Although an effort has been made to keep the software as portable as possible, the Windows operating system is not officially supported yet. Hardware dependent parts of the system (I^2C & CAN-bus interface) are particularly problematic and will have to be implemented differently. We assume that the software will be compiled directly on the robot. Other ways, such as cross-compilation are possible too, but it is not described in this text.

General dependencies

The following packages will be present in any packaging system employed by the particular Linux distribution. In case a required tool or library has to be installed, documentation for the specific packaging system should be consulted, how to install new software on the system.

GNU tool chain

The GNU tool chain is our choice for software development in the C/C++ programming language. We would particularly recommend Gentoo Linux, which contains all the necessary development tools out of the box. The following packages are required:

- GNU Compiler Collection (GCC)
- GNU make

pkg-config

pkg-config is a handy tool, which provides unified interface for querying installed libraries. This allows to avoid hard-coded include and library paths in project's Makefile-s.

Miscellaneous utilities

A suitable file decompression program, able to extract the .tar.gzip and .tar.bz2 archives with the project source code and libraries is necessary.

Specific dependencies

The following software packages may not be available on the particular Linux distribution. Therefore, detailed installation instructions are being provided.

The installation can either be performed by a non-root user (the compiled binaries, libraries, and header files will remain in the user's home directory), or the root can install the packages into the operating system directly.

These packages are needed only in case, the sample implementations of control system are compiled. Namely, with the Eurobot prefix (see A.2).

OpenCV

The computer vision algorithms implemented on MART robots depend on the OpenCV library (`http://opencv.willowgarage.com/`). The OpenCV 2 version has been used.

In order to be able to install and use OpenCV, Video4Linux support must be enabled in the kernel configuration, together with a correct device driver for the particular camera, connected to the actual computer. As web-cams have become very popular, any reasonably recent Linux distribution should ship with a kernel that supports video capture. The GTK+ windowing toolkit to display camera input for debugging purposes was used, so the GTK+ library may have to be installed separately.

CanFestival

Controller Area Network (CAN) support must be enabled in the kernel configuration, in order to use CAN modules. The Linux kernel supports CAN since the 2.6.25 version. As in the previous case, correct device drivers for the particular hardware must be present. The Linux kernel development headers must be installed on the system, version 2.6.25 at least. Older versions do not contain the necessary CAN headers.

The CAN-bus wrapper class (see 5.3) is built around the CanFestival library (`http://www.canfestival.org/`), which provides a high-level interface and a complete implementation of the CANopen stack. To achieve the desired functionality and to avoid compilation errors, resulting from poor design, the source code of the CanFestival library was modified slightly (see 5.3.1). Therefore, it is necessary to install a custom version of CanFestival:

1. Get the source code: *CanFestival-3.tar.gz* can be found on the CD.
2. Extract the archive:

   ```
   tar {xzf %CD%/code/CanFestival-3.tar.gz
   cd CanFestival-3
   ```

3. Configure the package:

   ```
   ./configure --prefix=${HOME} --can=socket
   ```

4. Compile and install the package:

   ```
   make && make install
   ```

5. Add the package information to pkg-config:

   ```
   export PKG_CONFIG_PATH=‘pwd‘:${PKG_CONFIG_PATH}
   ```

6. Add the local library path to the dynamic linker, if necessary:

   ```
   export LD_LIBRARY_PATH=${HOME}/lib:${LD_LIBRARY_PATH}
   ```

Project compilation

The Makefile of the control system environment placed on the CD is by default set to compile with the skeleton modules, pretending to be a control system. These example modules (see Section A.2) are not using any additional libraries (like OpenCV or CAN), so they can be compiled and run with no dependencies either on software nor hardware.

1. Extract the source code from the CD:

   ```
   tar {xzf %CD%/code/logion.tar.gz
   cd mosyr
   ```

2. Compile the project:

   ```
   make
   ```

3. Run the application:

   ```
   ./robot
   ```

Installation script

In order to simplify the installation process as much as possible, the presented steps are assembled into a comprehensive installation script, which can be found on the CD (*code/install.sh*). The contents of the “code” directory have to be copied to a local user-writeable directory before executing the script. If all the requirements stated in the General dependencies section have been satisfied, all the specific libraries as well as the project itself should compile automatically. In case any problems occur, consult the previous steps to isolate the problem. The installation script installs the CanFestival library and builds the control system environments source code.

A.2 Robot implementation guide

In this chapter it is explained, how to develop a new robot control system to this system environment. More precisely said, how to extend it with new modules to be used on a new robot. Unfortunately, to build a new robot a lot of hardware work needs to be done and specifications for all the hardware devices and protocols has to be read, so this guide tries to explain the functionality on fictive robot. The advantage of this is that you can see how the system works without the need of any special devices, all you need is just an ordinary computer.

As an example robot implementation, we have created a module skeleton with a bunch of example modules for all layers, example configuration file, example *StrategyLoader* etc. All the functions and modules are explained in this chapter and the files can be found in the "*/modules/skelet/*" directory in the source code.

The source code structure is very simple. The core (or common) files are in the root directory and the modules for different robots are in the modules directory in their own folders. The doc directory contains the doxyfile, the getopt directory the windows implementation of getopt and the OD directory the code for CAN-Festival object dictionary. Object files are generated to the obj directory and the logs are store to logs directory. In each robots module directory the layers have separate folders, HCL, HAL, and SL.

Creating a new control system

By creating a new control system is meant to create a new directory in */modules/*. Its name will identify the control system and its namespace should be called the same name. The makefile placed in the new system directory can call the main make file (see Code listing 13).

Code listing 13 Example Makefile (*/modules/skelet/Makefile*).

```
# make calls the main makefile
all clean run start showdep:
  $(MAKE) -C ../../ -e VERSION=$(shell basename `pwd`) \
    USE_OPENCV=no USE_CAN=no USE_CMP_VISION=no $@

.PHONY: all clean run start showdep
```

HCL module

The modules in the Communication layer inherit from the "*Board*" class (see Code listing 7). To create a functional HCL module it is necessary to implement all the abstract functions from this interface. Example implementation file is */modules/skelet/HCL/ExampleBoard.cpp*.

Construction

The constructor of your module should have at least one parameter, the name of the module. It is possible to hardcode this value, but all instances of this module will share the same name, which can cause troubles while debugging communication, hardware, or software issues.

Initialization

In your constructor, you should have only minimal member initialization, allocation of containers and in general code that will never be called again for that instance. All the initialization should be placed to the *init()* function. Error checking is strongly advised. The *init()* function returns a boolean value whether the initialization was successful or not. It is executed while registering the module in the *Scheduler*. If initialization fails, the module is not added.

Servicing

After the Scheduler's thread is started, all the registered modules are called consequently (depending on the loop they registered to). The *service()* routine should contain all the communication (data transfers in both directions) and data parsing. No calls to upper layers are allowed here, not even in the same layer. Communication modules have to work separately, with their hardware counterpart only.

Stopping

The *halt()* function is to prepare the device for power-down. It can, for example, stop the motors or to move the arm down slowly to prevent any damage to the hardware.

Destruction

Destructor is called on the object disposal, which is done after unregistering the module from Scheduler. If you do not unregister the module manually, it is done automatically when the Scheduler's thread is stopped.

CAN modules

The drivers for the boards communicating via the CANopen protocol have some special requirements opposed to the I^2C boards. In order to communicate via the CAN bus, they need to implement the *CANBoard* interface and they need to register to the *CANBus* object. After this registration is done, they can send and receive CAN messages corresponding to their *cobId*. The important part of the interface to interact with our *CANBus* class is in the Code listing 14.

The *CANBoard* interface (Code listing 8) to be implemented has only two member functions, *receivePdo()* and *requestPdo()*. The *receivePdo()* function

Code listing 14 CANBus class API.

```
class CANBus {
public:
    void registerHclObject(const CANBoard &hclObject, int cobId);
    void unRegisterHclObject(int cobId);

    bool sendSdoWriteMessage(int nodeId, int index, int subIndex, int data,
        int dataSize);
    bool sendSdoReadMessage(int nodeId, int index, int subIndex, int* data);
    bool sendPdoMessage( int cobId, const char* data, int length );
};
```

is abstract and it is called after receiving a PDO message on the CAN bus with the *cobId* of the module (depending on the registration in the *CANBus* object). The second, *requestPdo()* does not have to be abstract, its default is to just request PDO transfer (RTR – Request To Receive) to 0x180 + *nodeId*. This is the CANopen default[1].

HAL module

Modules in the Hardware abstraction layer are registered in the *Equipment* object. As the name *Equipment* suggests, the purpose of these modules is to expose some functionality of the hardware devices mounted on the robot. The word device is meant for a complete device, like a hand, odometry, localization, and range finder. It is sometimes a combination of smaller components that are useless alone, like small motors, servos, or buttons on different hardware boards. All modules in this layer inherit from the "*Device*" class (see Code listing 9). It guarantees that all the modules (devices) have a name assigned and it assures that they implement the functions of the interface. The *init()* function is called to initialize the module, e.g. getting references to other modules, reading configuration, etc. It is called just after all the modules have been registered in the layer (to resolve module dependencies). The *halt()* function is to prepare the device for shut-down, it stops sending requests or even cancels the last requests (to the Communication layer modules). The *refresh()* function is called periodically after registering the module in the layer, and its function depends on the module itself. Implementation of these three pure virtual functions is necessary to build a HAL module (device). Example implementation file is */modules/skelet/HAL/ExampleDevice.cpp*.

Devices that can move some parts of the robot may need some kind of deactivation routine. As an example, it can be used for safety reasons, or according to the Eurobot rules, each participating robot has to turn all the effectors off after the match time limit (90 seconds) expires. For this purpose, there is an *Effector* class (see Code listing 10), which adds the *deactivate()* function to the Device interface.

[1] This works for all our modules based on the CiA (CAN in Automation) specification. Some devices (like EPOS controllers) use different *cobId* for RTR, so the virtual function *requestPdo()* cannot be used in the default implementation in those modules.

Construction

The constructor of your module should have at least one parameter, the name of the module. It is possible to hardcode this value, but all instances of this module will share the same name, which can cause troubles while debugging.

Initialization

In your constructor, you should have only minimal member initialization, allocation of containers and in general code that will never be called again for that instance. All the initialization should be placed to the *init()* function. Error checking is strongly advised. The *init()* function return a boolean value whether the initialization was successful or not. It is executed after all the modules are registered in the *Equipment*, because many modules are dependent on the existence of other HAL modules. If the initialization fails (*init()* returns false), the module is removed from the *Equipment*.

Refreshing

After the Equipment's thread is started, all the registered modules are called consequently in a loop. The *refresh()* routine should contain (or call) all the data exchange between modules (HCL and HAL) and data processing. No calls to the upper layer are allowed, but cooperation inside the layer may be necessary for some modules.

Stopping

The *halt()* function is to prepare the device for power-down. It can, for example, stop the motors or to move the arm down slowly to prevent any damage to the hardware. The *deactivate()* function is only to stop the effectors, freeze the robot.

Destruction

Destructor is called on the object disposal, which is done after unregistering the module from Equipment. If you do not unregister the module manually, it is done automatically when Equipment's thread is stopped.

Smart Layer module

Smart Layer modules are of two types: strategies and actions. Actions are small classes run as Brain Slaves and strategies are a definition of action execution order, decision-making (usually a state machine) and the HCL and HAL module instantiation (as the strategy knows the best, what modules it needs for operation).

Strategy loader

To have the correct strategy loaded it is necessary to create a factory method (*StrategyLoader::getStrategy()*) returning the right strategy instance. The configuration option defining the actual strategy is called "STRATEGY". The factory method is called from the Brain during its initialization with the parameter retrieved from the configuration. After getting the instance of the right strategy, it is initialized, all modules are loaded, and all the layers start their threads. Example strategy loader is */modules/skelet/StrategyLoader.h*.

Strategy

Strategy in general defines the goals, how the robot should behave, and what it should do to achieve these goals. It also contains what hardware devices and software modules it needs to load for its operation. Strategy can be implemented as a finite state machine, what we consider the fastest, easiest, thus the best solution of this problem.

New strategies can be created by inheriting and implementing the abstract functions of the "*StateMachineStrategy*" and "*Strategy*" classes. The functions are to name and describe the strategy, to register all the modules, to initialize and finalize the strategy and at last but not least the execute function which is called from the Brains main loop and should contain all the decision-making. An example strategy implementation is */modules/skelet/SL/ExampleStrategy.cpp*.

Action

Action is a very small class implementing only one function from its abstract parent (interface). This function called *body()* should contain all the necessary operations to achieve the goal of the action. Example actions might be: find something, collect something, drive somewhere, use some manipulator etc.

The purpose of this class and all the implemented actions based on this interface is to have a bunch of actions that can be used in different strategies. This approach helps the reusability and the speed of development. Strategies can share actions as in fact the different state machines may have common states.

The parent class called *Action* makes all its subclasses compatible with the BrainSlaves, so all actions can be started as a *BrainSlave* in a separate thread. The constructor of the action classes should contain all the pointers to the HAL modules the action uses. This will not only save time getting all the references, but also allows using the action from other strategies without hiding any action dependency. Example action implementation is */modules/skelet/SL/ExampleAction.cpp*.

Configuration

For configuration management we use the singleton *Conf* class. All the variables in the configuration file, the command line, and even those set from the

running system are always accessible via this singleton. Configuration files are parsed before the command line, to allow the command line options to overwrite parameters. First is the configuration file in the root directory of the project and after that the configuration file from the module group (e.g. *modules/skelet/*). The configuration files are named *config.cfg*. The default value for different types is 0, 0.0, "" (empty string) or false. The default log file with comments if in Code listing 15.

Code listing 15 Example configuration file (*/modules/skelet/config.cfg*).

```
# Example configuration file

# logging
LOG_SCREEN                  true
LOG_FILE                    true
LOG_COMMUNICATION           false
LOG_NETWORK                 false

# logging verbosity
LOG_FROM_LEVEL              0

# if hide by default is set to true, log filter will be applied
# to show logs, otherwise the log will be empty (if no filter is set)
LOG_HIDE_BY_DEFAULT         false

# how often to print the statistics (loops per second of each layer)
# defined in brain thread loops (0 means no statistics)
STATISTICS_PRINT_FREQUENCY  300


# SL setting
STRATEGY                    ExampleStrategy

# HCL modules
HwExampleBoard              1

# HAL modules
HalExampleDevice            101
```

Appendix B

Platform support

This appendix is a brief description of used hardware platforms, hardware components, and communication protocols. As they are not directly concerning the control system environment, just the control systems based on it, this description is present only for completeness and to introduce the testing platforms.

B.1 Hardware platforms

VIA EPIA – IA-32 (x86)

VIA EPIA (Figure B.1) is an embedded computing platform from VIA Technologies Inc. A wide range of different form factors are being offered, including Mini-ITX, Nano-ITX, Pico-ITX. VIA EPIA motherboards feature an integrated IA-32 (x86) compatible VIA CPU, either passively (heat sink only) or actively cooled (heat sink + fan). The only components, which have to be provided by the user, are the system memory and power source. Small size and low power consumption are the key features, making this platform suitable for embedded applications such as firewalls, HTPCs (home theater PC), small file servers, robotics, etc. However, there has been some criticism that VIA EPIA is too expensive for the processing power it provides. The board for testing included a 1.5 GHz VIA C3 CPU, and 512 MB of DDR II RAM.

Beagle Board – ARM

Beagle Board (Figure B.2) is a low-power, low-cost, fan-less single board computer produced by Texas Instruments, collaborating with a group of volunteers, with vast open-source development possibilities in mind. It features Texas Instruments OMAP3530 system-on-a-chip. A 600 MHz ARM Cortex-A8 CPU in soldered on the board. The CPU is based on the latest 7th generation of the ARM architecture. It also includes an embedded 430 MHz digital signal processor (DSP) and a 2D/3D graphics accelerator with OpenGL ES 2.0 support from Imagination Technologies. Beagle Board is further equipped with 256 MB of flash memory, 256 MB of RAM (we have an older model which

only has 128 MB of RAM), SD/MMC card slot, RS-232 serial port, JTAG interface, USB (not functional in our revision), HDMI video output and audio input/output. The whole board should not consume more than 2W of power. The project's website states that Beagle Board is suitable for a plenty of embedded applications, for example low-cost Linux PCs, "kitchen" computers, thin client terminals, set-top boxes, network sniffers & routers, robotics, etc.

Figure B.1: VIA EPIA

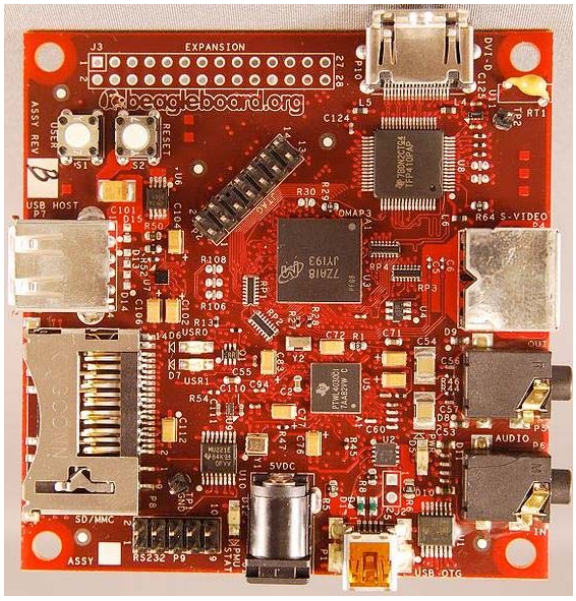


Figure B.2: Beagle Board

High performance PC – AMD64 (x86-64)

There are several different 64-bit and/or multi-core processor architectures available today. The CPU architecture of our choice was AMD64, in an Intel Core 2 Duo processor. It is the most widely used and most affordable 64-bit platform available in the market. The biggest disadvantages compared to embedded platforms (including VIA EPIA and Beagle Board) are the heat dissipation (efficient passive cooling is almost impossible) and power consumption, which are much higher, especially in dual or more core processors. Nevertheless, the latest VIA Nano processor seems very promising. It is fully compatible with AMD64 and dual-core versions are expected in the future.

64-bit architecture and especially multiple cores are expected to be used exclusively in the future. Except for significantly larger amounts of virtual and physical memory accessible, applications taking advantage of the 64-bit instruction set tend to be much faster. One of the most important improvements of the x86-64 extensions is the doubled amount of general purpose registers, which should help especially computationally intensive tasks such as image preprocessing in computer vision. Symmetric multithreading enabled by multi-core CPUs (multiple physical processors are possible too) allows threads to be executed truly in parallel, leading to lower latency and better utilization. Synchronization issues may emerge, however.

B.2 Hardware components

The hardware components are designed to work with I^2C , CAN-bus or RS-232, which is their only dependency (for example, they are not dependent on the architecture of the particular robot, etc.). This makes them universal, hence they can be reused in robots not driven by the presented control system environment.

Source code of the communication modules for these hardware components can be found in */logion/modules/Eurobot*/HCL/*[1] directories in the projects source code. They are not further described, as they are not directly the aim of this work. For further information see [5] or [6].

B.3 Communication

RS-232

RS-232 is an industrial standard for serial asynchronous full-duplex communication between two devices. Although it was designed in 1969, it used to be a typical interface for personal computers, until it was replaced by USB. RS-232 is not a bus. Hence, it is not possible to use it to interconnect more than two devices without additional components, such as splitters and multiplexors.

[1] the * can be replaced by the year of the robot (2008, 2009, or 2010)

The ATmega MCUs used on robots hardware components 6.1 used for testing have an integrated hardware peripheral with hardware-based RS-232 support.

USB

Universal Serial Bus (USB) is a specification to establish communication between devices and a host controller. An USB system has an asymmetric design, consisting of a host, a multitude of downstream USB ports, and multiple peripheral devices connected in a tiered-star topology. USB devices are linked in series through hubs. Additional USB hubs may be included in the tiers, allowing branching into a tree structure with up to five tier levels. USB device communication is based on pipes (logical channels). For more information see [23].

On the tested robots the USB was used for connecting joystick, cameras, and USB-to-I^2C converter.

I^2C

I^2C (Inter-Integrated Circuit) is an industry standard, multi-master serial bus, developed by the Philips Company (1992) for the purposes of short distance, low-speed communication among multiple devices. It is typically used to interconnect various components of the same device, usually within the same PCB (e.g. interconnection of EEPROMs, LCDs and button panels inside a video recorder). In 2006, the I^2C technology was relieved of licensing fees and made free for non-commercial use (nevertheless, slave addresses allocated by NXP are still paid). TWI (Two Wire Interface) is another name for the I^2C bus. This is because the I^2C bus was relieved of patent restrictions and licensing fees gradually. Therefore, some vendors (e.g. Atmel) used the TWI name for the I^2C bus to prevent trademark licensing issues.

The most significant features of the I^2C bus are simplicity, existence of integrated hardware peripheral with hardware-based I^2C support in ATmega MCUs and the fact that it is a very common solution (e.g., many sensors used in robotics have an I^2C interface). Moreover, of course, it is a bus, and thus it allows multiple devices to be daisy chained or connected to one point. Limitations experienced during testing: low speed and reliability issues when using IC for longer distances.

CAN-bus

CAN stands for Controller Area Network, a bus designed by the Bosch Company primarily for automotive industry. CAN, thanks to its features, has also found application in other branches. It is a very reliable, high speed and robust solution with electrical noise tolerance and a long possible distance of connected nodes.

Devices are connected to the CAN-bus using two mutually inverted signal wires. Eventual conflicts that may occur when multiple nodes transmit at

the same time are solved by CAN in a principally same way as I^2C does. Thus, the CAN-bus does not suffer from throughput collapse in case of bus overload, contrary to the Ethernet protocol (which uses a random wait time in case of collision). CAN-bus is a serial bus with data transmitted in frames, usually consisting of eleven bites of an object identifier, data field up to 8 bytes, and a CRC control field. Bit stuffing is used for CAN frames having five consecutive bits of the same polarity. CAN-bus is a professional solution with high reliability and wide spectrum of applications. Its remarkable features are compensated with a significantly more complex protocol and end node computational requirements, compared to simpler solutions. However, this is an adequate price for the CAN-bus benefits.

Appendix C

CD contents

./code Thesis (software) source code, additional third party libraries, installation script (see Appendix A for detailed description).

./text Thesis text in PDF.

./video Various videos demonstrating the software in action on MART robots.

Bibliography

[1] B. Siciliano, O. Khatib: *Springer Handbook of Robotics*, ISBN: 978-3-540-30301-5, Springer, 2008.

[2] *Eurobot Autonomous robot contest*, `http://www.eurobot.org`.

[3] Jusko P., Obdržálek D., Petrůšek T.: *Software-Hardware Mapping in a Robot Design*, in A. Gottscheber et al. (Eds.): Research and Education in Robotics - EUROBOT 2008, Springer CCIS 33, pp. 19–28, 2009.

[4] Obdržálek D., Basovník S., Jusko P., Petrůšek T., Tuláček M.: *Robot Localisation in Known Environment Using Monte Carlo Localisation*, in Proceedings of the International Conference on Research and Education in Robotics - EUROBOT 2009, Springer CCIS 82, pp. 100-111, 2009

[5] Basovník, S., Dekár, M., Jusko, P., Mikulík, A., Obdržálek, D., Pechal, R., Petrůšek, T., Piták, R.: *Logion – A Robot Which Collects Rocks*, in Proceedings of the International Conference on Research and Education in Robotics - EUROBOT 2008, pp. 276-287, May 2008

[6] Petrůšek T., Mikulík A., Dekar M., Skalka M., Burda M.: *Modular robot control system*, `http://www.ksi.mff.cuni.cz/sw/mosyr.html`.

[7] *Microsoft Robotics*, `http://msdn.microsoft.com/en-us/robotics/default.aspx`.

[8] *The Player Project*, `http://playerstage.sourceforge.net`.

[9] *LEGO Mindstorms NXT*, `http://mindstorms.lego.com`.

[10] *Webots*, `http://www.cyberbotics.com`.

[11] *Simbad Project*, `http://simbad.sourceforge.net`.

[12] *The Mobile Robot Programming Toolkit (MRPT)*, `http://babel.isa.uma.es/mrpt/`.

[13] *CARMEN robot navigation toolkit*,
http://carmen.sourceforge.net.

[14] Michael Somby: *A review of robotics software platforms*,
http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/A-review-of-robotics-software-platforms/.

[15] *Intel OpenCV*,
http://opencv.willowgarage.com.

[16] Dellaert, F., Fox, D., Burgard, W., Thrun, S.: *Monte carlo localization for mobile robots*, in Proceedings of the IEEE International Conference on Robotics & Automation (ICRA99), 1998.

[17] Abrahamsson P., Salo O., Ronkainen J., Warsta J.: *Agile Software Development Methods: Review and Analysis. s.l.*: VTT Publications 478, 2002.

[18] Andrei Alexandrescu: *Modern C++ design: generic programming and design patterns applied*, ISBN: 978-0201704310, Addison-Wesley, 2001.

[19] *XGetopt* – A Unix-compatible getopt() for MFC and Win32,
http://www.codeproject.com/KB/cpp/xgetopt.aspx.

[20] *Advanced CAN solutions for hardware, software, consulting and education*,
http://www.kvaser.com/can/protocol/.

[21] *CAN in Automation (CiA)*,
http://www.can-cia.org.

[22] *CanFestival Open Source CANopen library*,
http://www.canfestival.org.

[23] Wikipedia, the free encyclopedia,
http://wikipedia.org.