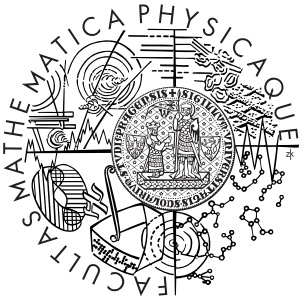
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Ondřej Staněk

Centralized multirobot system

The Department of Software Engineering

Supervisor: RNDr. David Obdržálek

Study programme: Computer Science

Prague 2012

I would like to thank my supervisor RNDr. David Obdržálek for his advice and comments that helped to shape this thesis. I am grateful for his support during the development of the PocketBot2 robot.

Very special thanks to Josef Vandělík, who designed and manufactured the wheelframe, a key part of the robot.

I would like to dedicate this work to my beloved grandfather Standa, who led my first steps in robotics.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, date signature

Název práce: Centralized multirobot system
Autor: Ondřej Staněk
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. David Obdržálek
e-mail vedoucího: David.Obdrzalek@mff.cuni.cz

Práce se zabývá návrhem a implementací centrálně řízeného multirobotického systému. Hostitelský počítač (či mobil) ovládá miniaturní mobilní roboty PocketBot2. Tito roboti mají celou řadu senzorů. Dokáží sledovat černou čáru a jsou vybaveni systémem pro detekci překážek a ostatních robotů. Ačkoliv roboti vznikli v rámci této práce, těžiště práce samotné je v řídícím software. V robotech PocketBot2 je implementován vestavěný řídící systém, který obsluhuje senzory robota a umožňuje vykonávání základních pohybových manévrů. K bezdrátovému přenosu dat mezi roboty a počítačem (mobilem) je využita technologie Bluetooth. Na straně počítače (mobilu) byla implementována multiplatformní řídící knihovna, která zprostředkovává přístup k senzorům jednotlivých robotů a umožňuje řízení jejich pohybu. Zajišťuje tak pohodlné rozhraní pro imlementaci centralizovaných multirobotických algoritmů.
Klíčová slova: robot, knihovna, PocketBot, Bluetooth, Java

Title: Centralized multirobot system
Author: Ondřej Staněk
Department: The Department of Software Engineering
Supervisor: RNDr. David Obdržálek
Supervisor's e-mail address: David.Obdrzalek@mff.cuni.cz

This thesis focuses on design and implementation of a centralized multi-robot system. A host computer (or cellphone) controls several tiny PocketBot2 mobile robots. These robots feature various sensors. They can perform line-following and they are equipped with a system for detecting obstacles and other robots. Although the PocketBot2 robots were designed and built within the frame of this thesis, the core of the thesis itself lies in the software. For the PocketBot2 hardware, an embedded control system was designed and implemented. It interfaces robot's sensors and carries out basic movement commands. Bluetooth technology is used for wireless data transfer between robots and the host. In the host, a multi-platform control library was implemented. It provides access to sensors of individual robots and controls their movement. The library ensures convenient interface for implementing centralized multi-robot algorithms.
Keywords: robot, library, PocketBot, Bluetooth, Java

Contents

Chapter 1

Introduction

Robots are created to help people in many diverse activities and procedures. In some industries, the automation reached a level where basically all manual workers can be replaced with robots, which are faster, more reliable and effective. Robotisation in the industry has happened quietly and almost without the notice of general public, but recently, robots infiltrate into our daily lives: Automated robotic vacuum cleaners became a common article in electro stores, as well as robotic toys. The state-of-the-art automobiles are equipped with auto-parking feature, which finds a parking spot and parks the vehicle automatically. It is sure that people will face automation more and more frequently in near future.

Robotics is applied in logistics and transportation as well. In factories and hospitals there are robots which transport material. Furthermore, robots have wide applications in military. In these cases, many robots operate in same place, so they might interact or even cooperate.

No matter if robots transport material in hospital or fight on a battlefield, they often have centralized control. The central control system organizes robots so that they fluently transport material throughout the building or that they keep a formation on the battlefield.

1.1 Goals

Goal of this thesis is to design and implement a software solution of a centralized multi-robot system. The system has two parts: a control library in the host and an embedded control system in individual robots. This thesis covers both the design and implementation of the library and the embedded system. The library will provide the application programmer with an interface for controlling robots from the host, so that he may im-

plement various centralized multi-robot algorithms. It is expected that a robot will be designed and built, however, the thesis does not describe this part. On this robot, the library and embedded control system will be tested.

1.2 Organization of the Thesis

We introduce mobile robots and robot control architectures in Chapter 2. In Chapter 3 we compare Centralized and Decentralized control systems. Chapter 4 gives a review of wireless technologies suitable for mobile robots.

Chapters 5 and 6 describe design guidelines for a Control Library and the Embedded Control System of a mobile robot. Short-distance communication system for mobile robots is explained in Chapter 7.

The Centralized Multi-Robot System will be tested on PocketBot2 robots, which are introduced in Chapter 8. Chapter 9 describes implementation of the Control Library in Java and focuses on robot representation and packet communication. Chapter 10 gives overview of Bluetooth Stacks and describes the porting of an open source BT stack to the PocketBot2 robot. The Embedded Control System of PocketBot2 robot is described in Chapter 11.

Appendix A contains example applications of the Control Library. The content of enclosed CD is described in Appendix B.

Part I

Analysis

Chapter 2

Mobile Robots

Mobile robots are automatic machines that can move in their environment. According to the environment where these robots operate, we divide them into several categories: *aerial* (flying) robots, *underwater* (swimming) robots and *land* robots. The last category can be roughly classified to *wheeled* robots and *legged* robots. Although legged robots are more difficult to design and control than their wheeled counterparts, they are very promising because of their universality and capability of movement in hardly accessible terrain.

Every mobile robot, no matter if it operates in air or under water, has some means for sensing and affecting its surroundings. These are called *sensors* and *actuators*.

2.1 Sensors

Sensors provide the robot with information about its environment. Thanks to data from sensors, the robot can perceive the world around it. Basically, sensors are for a robot the same what senses are for a human.

There are many types of sensors, from the simplest ones like contact switches for detecting collisions to more advanced sonars and laser scanners. The robot can be equipped with a camera, microphone, GPS, compass, accelerometer, gyroscope and dozens of others. In short, any measuring device can be a sensor for a robot [4].

2.2 Actuators

The robot has some means for affecting its surroundings as well. These are called actuators. The essential actuators of every mobile robot are

the mechanisms of its locomotion. For instance, the motors that propel robot's wheels, legs, propeller or whatever.

Further, the robot can be equipped with some kind of robotic arm for manipulating objects, or it can carry a vacuum cleaner or a gun. The possibilities are really wide, everything what affects robot's environment in some way is an actuator.

2.3 Robot Control

Robot control is a process that uses information from sensors to drive robot's actuators in order to reach a specific goal. In this section we will outline ways of robot control.

Mobile robots may or may not be autonomous. Bomb-defusing robots are usually at least partially remotely controlled by human operator. The operator evaluates data from sensors (in this case cameras) and drives robotic arms to defuse the bomb.

On the contrary, some robots need to be autonomous. For example Mars rover, the automated vehicle that collects data on the surface of Mars, cannot be teleoperated, because the round trip communication time from Earth to Mars ranges from 8 to 42 minutes. Additionally, the connection is available just few times a day [5]. The Mars rover (Figure 2.1) performs hi-level plans that are composed on Earth, thus the embedded control system in robot must be very complex.

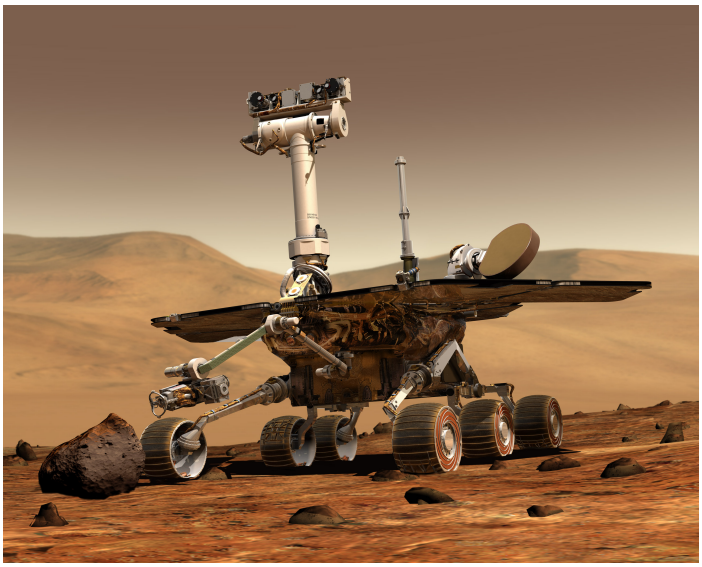


Figure 2.1: NASA Mars Rover

2.3.1 Fully Autonomous System

The architecture of an autonomous robot system similar to the one used in Mars rover is presented in Figure 2.2.

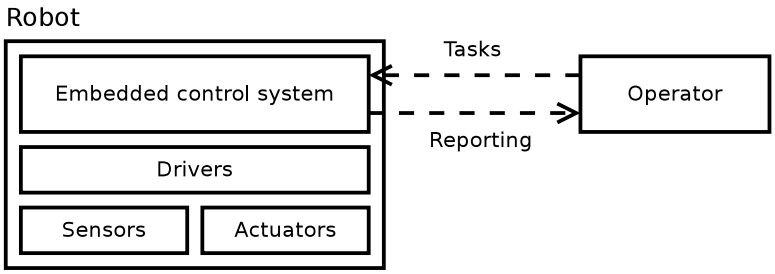


Figure 2.2: Fully autonomous system

The Embedded control system evaluates data from sensors and performs complicated tasks issued by the operator. The robot informs the operator about progress and results of its actions in return. This model of robot control is very robust and it is not so vulnerable to communication failures. However, it expects the robot has sufficient processing power to run a complex control system.

The most advanced autonomous robots feature *strong artificial intelligence.* Such robots are self-aware and they are no longer under direct control of men. Rather, they make decisions on their own. However, the research in strong AI is far from self-aware artificial beings known from sci-fi movies.

2.3.2 Detached Control System

Small mobile robots[1] are usually fitted with microcontrollers and their computational resources are often very limited. For more complicated tasks, such as image processing, localization or machine learning, the control system needs to run on detached host, which provides sufficient processing power. Such architecture is presented in Figure 2.3.

The robot does not contain any intelligence by itself, it just provides the host with access to all its sensors and actuators. The host retrieves data from sensors, does the processing and then sends back commands for actuators.

[1] In this thesis, the term *small robots* is used for robots that weight less than 1kg. Such are for example e-Puck or Khepera.

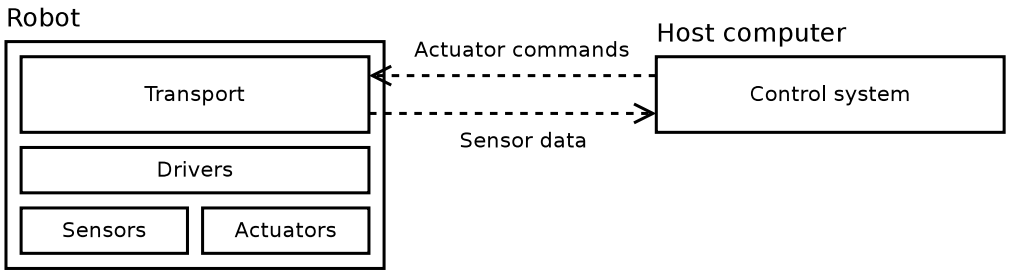


Figure 2.3: Detached Control System

This model of robot control was thoroughly investigated in the bachelor thesis of Ondřej Plátek [1], who implemented a control library for a small mobile robot e-Puck. In his work, Plátek presented further advantages of this architecture. He pointed out that the development cycle of a program gets shorter, because the programmer does not need to flash his program into the microcontroller of the robot. Moreover, as the control program runs in host computer, it can be easily debugged.

However, there are some downsides of this model. The most conspicuous one is the requirement of high-quality data transport between the robot and the host. If the connection drops, the robot stops immediately. The data transport shall offer high throughput since extensive data traffic is implied by this architecture. Last but not least, any motion control algorithms will yield suboptimal results by virtue of latency issues. Thus, this architecture is unacceptable if high performance is required.

An interesting implementation of detached control system was presented by Kurt Konolige in his paper Saphira Robot Control Architecture [6] as State Reflector.

Citation: *It is tedious for robot control programs to deal with the issues of packet communication. So, Saphira incorporates an internal state reflector to mirror the robot's state on the host computer. Essentially, the state reflector is an abstract view of the actual robot's internal state. There is information about the robot's movement and sensors, all conveniently packaged into data structures available to any micro-task or asynchronous user routine. Similarly, to control the robot, a routine sets the appropriate control variable in the state reflector, and the communication routines will send the appropriate command to the robot.*

Such architecture is presented in Figure 2.4.

This approach is very convenient for the application programmer. The programmer deals with a local representation of the robot, while a framework ensures data synchronization between the real robot and

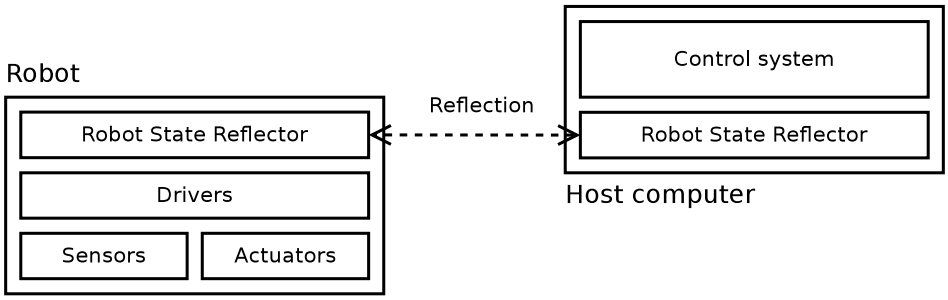


Figure 2.4: Detached Control System - Robot's State Reflection

its representation. As a result, all packet communication takes place in back-end and it is hidden from the application programmer. This also hides asynchronous programming issues, thoroughly investigated in [1].

However, this architecture still suffers from all drawbacks mentioned on the preceding page.

2.3.3 Combined Control System

To give a summary, the microcontroller of a small mobile robot cannot handle computationally intensive tasks like image processing, localization or machine learning. Such tasks has to be computed in detached host computer. However, motion control should be implemented in the microcontroller, otherwise the performance would decline due to latency in communication.

In this thesis, the design and implementation of Combined Control System is presented. See Figure 2.5.

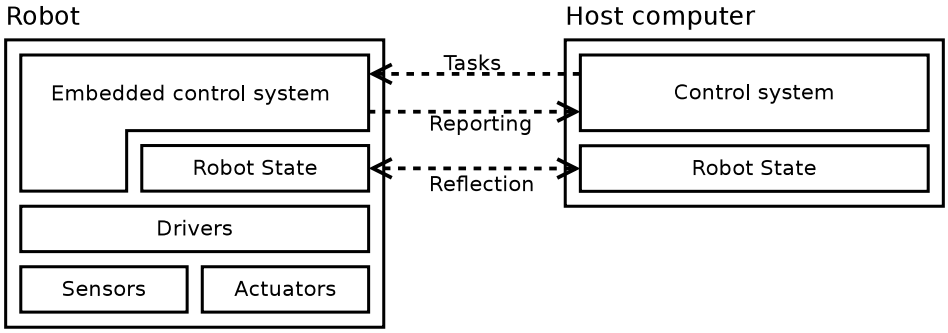


Figure 2.5: Combined Control System

Robot locomotion control is implemented in the robot itself, to gain best performance. Still, the host will have access to all sensors and

actuators of the robot. That provides the application programmer with a complete interface to all robot features. This architecture allows the programmer to use both the task-oriented control and the teleoperational approach, which might be eligible for some applications. In addition, accessing sensors and actuators remotely is great for debugging purposes.

Chapter 3

Multi-Robot Systems

Mobile robots are capable of traveling across their environment, therefore, the possibility of interacting with other robots in their surroundings arises. It is very exciting to study how multiple robots could cooperate towards reaching a common goal.

Multi-Robot systems have wide applications. Teams of robots can handle tasks that are difficult or even impossible for a single robot; or they may perform them in a faster or cheaper way. For instance, mapping and exploration are tasks that benefit from the multitude of cooperating robots.

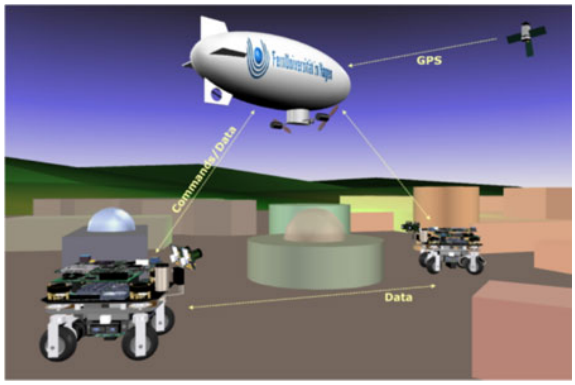


Figure 3.1: Heterogeneous multi-robot system

Another example is cooperative object manipulation, when the object is so heavy that individual robot cannot move it alone. It might be more appropriate to employ dozen of small robots than a single big one, because small robots can get to hardly reachable places. In addition, thanks to redundancy, the multi-robot system may be less vulnerable to failures of its units.

Multi-robot systems may be heterogeneous; they may consist of diverse robots with various capabilities. For example, on a battlefield there might operate aerial robots that help to coordinate land robots, locating possible targets and maintaining the formation of robots.

Self-reconfigurable robots are a very special case of multi-robot systems. They consist of homogeneous small robotic units that are tightly connected to each other, forming the body of the robot. Such robot can modify its structure by altering the arrangement of its units. It can adapt the shape of its body for different tasks, and it can even split itself into a number of independent robots and unify later in another place. Thanks to interchangeability of its units, the robot can recover from damages to its body. Self-reconfigurable robots are presented in Figure 3.2.

Figure 3.2: Self-reconfigurable Robots

Generally, control of a multi-robot system is a challenging issue. There are two approaches to this issue: Centralized control and Decentralized control [8]. This thesis focuses on Centralized Mutli-Robot Systems.

3.1 Centralized Multi-Robot System

In a Centralized multi-robot system, the global information about the state of the whole system is maintained. The system gathers information from all robots and keeps track of their position in environment. It may build a map on the basis of information received from robots. This system is either located in a stationary host, or in one robot that has the explicit role of a *master*. The master then organizes the team of robots to reach a common goal. It plans tasks for individual team members and supervises the whole process.

This architecture is straightforward to design, however, it is not robust to communication failures and unpredictable situations. Generally, centralized control is well suited for limited number of robots that operate in known and unchanging environment [8].

Centralized multi-robot systems have applications in autonomous logistics and traffic control. A great example of applied centralized multi-robot control is the transportation system in hospital Nemocnice Na Homolce [9]. See Figure 3.3. Mobile robots transport dishes and sheeting throughout the whole building. They follow paths marked on the floor and they can even use elevators.

Figure 3.3: Transportation robots in hospital Nemocnice Na Homolce

3.2 Decentralized Multi-Robot System

By contrast, decentralized systems do not involve any *master* who has a complete information about the state of the system and oversees the whole process. Instead, each robot is an autonomous unit that acts according the state of its surroundings. Of course, the robot is aware of presence of other robots and a local communication with them may take

place. A complex behavior of the group emerges from the interactions between the robots and environment. This architecture is very robust, it can perform well in unfriendly environments and it is scalable; potentially a huge number of homogeneous robots can cooperate towards reaching a common goal.

Decentralized multi-robot architecture is typical in *swarm robotics* [10]. Usually, robotic swarm consists of many simple robots that have primitive behavior. The robotic swarm is good at tasks such as dispersion in area, which finds its application in territory monitoring.

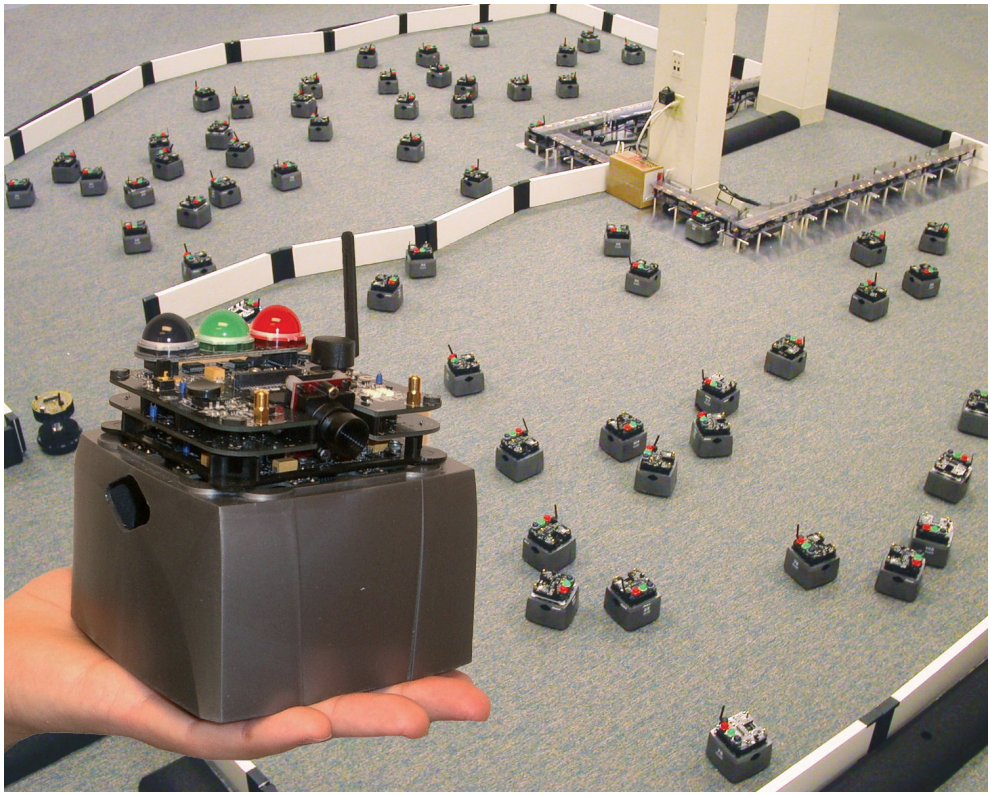


Figure 3.4: Swarm robots

Chapter 4

Communication

Robots are mobile, so the communication between robots and the host must be wireless. This chapter presents a background research on wireless communication technologies suitable for a small mobile robot.

The requirements on the data transport were outlined in Robot Control Section 2.3.

To recall:

- There will be an extensive data flow from the robot to host, because state of sensors will be sent periodically
- The data transport should be robust and reliable, so that the teleoperational mode could be used by the application programmer
- Communication links to several robots will be held simultaneously

Among the wide range of wireless technologies (Bluetooth, ZigBee, IrDA[1], Wi-Fi, proprietary wireless solutions..) we will discuss the former two, because they are the most suitable for our purpose.

IrDA is not suitable, as it uses infrared light, so there has to be a direct line of sight between the transceivers. That makes the technology unreliable for teleoperation purposes. Wi-Fi is a robust technology that definitely have its place in mobile robotics, however, for small mobile robots it is too expensive and complicated.

[1] set of protocols for wireless infrared communications, specified by Infrared Data Association

4.1 Bluetooth

Bluetooth is a wireless standard for interconnecting mobile devices. It is intended for forming ad-hoc Personal Area Networks (PAN). The effective range is approximately 10 meters for a regular (Class 2) Bluetooth device and 100 meters for Class 1 device.

The Bluetooth technology is designed with ease of use in mind. It provides mechanisms for convenient device and service discovery. Establishment and configuration of a Bluetooth connection is easy for the application user.

The data rate can be as high as 3 Mbit/s for the Bluetooth 2.0 + EDR[2] version. The Bluetooth technology supports various data transport modes, suitable for transferring both synchronous and asynchronous data. For packet communication, asynchronous L2CAP[3] data transport is of our interest. The L2CAP protocol ensures reliable connection, providing CRC[4] and packet retransmission. Moreover, Bluetooth allows broadcasting to devices connected to the channel.

For the sake of objectivity, it is important to mention the topology of a Bluetooth network (*piconet*), which brings in some limitations. The piconet is managed by a master, who defines a physical channel. Up to 7 active slaves can be connected to the channel. The communication can only take place between the master and a slave. That means slaves are not allowed to directly address each other. However, this is not an issue for a centralized multi-robot control system, since no communication between individual robots is expected.

Finally, the Bluetooth technology is supported basically in every cell phone and laptop, which is a great advantage.

4.2 ZigBee

ZigBee was designed as a wireless technology for connecting small sensors. This technology is employed in applications where use of Wi-Fi or Bluetooth is not suitable because of their complexity, cost or power consumption.

ZigBee incorporates wireless mesh networking. Two devices in a network can communicate with each other even though there is no direct link between them. The data is routed throughout the network, since in-

[2]Enhanced Data-Rate
[3]Logical Link Control and Adaptation Protocol
[4]Cyclic Redundancy Check

dividual nodes have routing capability. Mesh networking greatly extends the range of ZigBee technology.

ZigBee has throughput up to 250kbit/s. It is inferior in comparison to Bluetooth, however, ZigBee is a very promising technology for low-cost applications.

4.3 Conclusion

For a centralized multi-robot system, Bluetooth is a technology of our choice. This technology is widely supported in mobile devices (potential hosts). That gives the Bluetooth an advantage over ZigBee. The master-slave topology of a piconet reflects the needs of the centralized multi-robot system. Only the limitation of 7 active slaves on a Bluetooth channel is questionable. It actually means that only 7 devices can communicate with a master at a time. However, the Bluetooth architecture allows presence of *parked* (passive) slaves on the channel. Slaves in *parked mode* can only receive broadcast messages from the master, but cannot transmit any data. The master can *park* and *unpark* slaves, if it wishes to alter the set of active slaves, which could eventually allow communication with more than 7 slaves. Alternatively, the host may utilize more than one Bluetooth module to extend the number of connected slaves. This approach was carried out in [22] to control 21 devices using 3 Bluetooth modules.

Part II

Design

This part focuses on design of a control software, both in the host and in the robot. In the host, a Control Library will ensure the interface for the application programmer to control multiple robots. In the robot, the Embedded Control System will manage all robot's sensors and actuators. In addition, it will carry out locomotion of the robot. A mechanism of synchronization between the real robot and its representation in host will be described. Finally, possibilities of short distance communication between individual robots will be presented.

Chapter 5

Control Library

The Control Library provides an interface for accessing and controlling mobile robots from the host. The library should:

- Ensure convenient way for discovering robots and establishing links to them
- Handle simultaneous connections to many robots
- Provide an application programmer with access to all features of the robot (retrieving data from sensors, controlling actuators, issuing commands)
- Track the position of the robot (with the use of odometry)

Moreover, the control library should be universal by the means it does not depend on:

- Data transport used for communication with robots
- Operation system
- Hardware platform (It is expected the library will run both on PC and Android phone)

Generic parts of the Control Library will be designed and implemented to fit any mobile robot. Such parts are:

- Reflection mechanism between robot and its representation
- Packet communication
- Discovering and connecting robots

However, modules representing sensors and actuators make the implementation tightly connected to a particular robot platform. Nevertheless, the library will be designed in a modular manner, so it can be easily adapted for any mobile robot.

5.1 Robots and their Representation

A complete representation of the robot will be built in the host computer. The robot representation will mirror the state of all the sensors of the real robot. That means the programmer will not need to retrieve the state of sensors explicitly. Because the state of sensors changes only in the real robot, the sensor synchronization will be unidirectional, from the robot to host only.

In the same manner, there will be a unified method for accessing all robot's parameters and settings. Modifying the settings in the robot representation will yield to automatic update of settings in the real robot. Same as before, there will be no need for the application programmer to deal with packet communication. The communication issues will take place in the back-end of the Control Library.

The robot representation will also provide means for controlling robot's actuators. Every robot's actuator will have a corresponding representation in the host. The programmer will issue actuator commands on these representations and the command will be passed to the real robot.

Previous paragraphs described the *reflection mechanism* between the real robot and its representation in host, which is sufficient for the teleoperational control.

Beside this mechanism, the Control Library will also offer the task-oriented control. The robot can carry out advanced movement commands and fore-programmed behaviors. Those are basically such activities, whose processing in the host would yield suboptimal results due to latency in communication. Therefore, these tasks will be carried out in the Embedded Control System of the robot, and the Control Library will just initiate the actions.

5.2 Transport Layer Abstraction

The library will be designed in a manner that the link between real robot and its representation can be established on any transport layer. However, an assumption is made that the transport is reliable, i.e. it provides

mechanisms for retransmission of lost and corrupted data frames. This assumption holds for L2CAP Bluetooth transport layer, which will be implemented in the library.

None the less, stream data transports like RFCOMM[1] or RS-232[2] can be used as well, if a simple framing layer like SLIP[3] [11] is implemented. Analogously, unreliable data transport can be used if an underlying layer for retransmission of corrupted and lost packets is implemented.

5.3 Discovering Connectible Robots

On behalf of the application programmer, the Library should provide list of all connectible robots. Such requirement is inherent with the Bluetooth technology, which offers robust mechanism for discovering Bluetooth devices. For other wireless technologies, the same discovering functionality should be implemented.

However, if the transport technology does not support discovering devices, then the application programmer has to address robots the regular way, by providing an identifier (e.g. the unique media address of the robot).

5.4 Types of Communication Protocols

The communication protocol defines format of transfered data. Extensive data traffic is expected between the host and robots, so the protocol has to be designed carefully and with respect to high performance. Generally, there are two types of protocols: *text protocols* and *binary protocols.*

5.4.1 Text Protocols

Text protocols are oriented around text strings. The packets are readable for humans, and therefore they can be easily logged and debugged. The disadvantage of text protocols lies in overhead connected with translating native binary data to the textual form and back. Moreover, such translation increases the data length, which is unpleasant both for transferring as well as for logging the data.

[1] Radio Frequency Communication - a simple set of transport protocols, made on top of the L2CAP protocol, providing emulated RS-232 serial ports

[2] standard for serial communication, used in computer serial ports

[3] Serial Line Internet Protocol

5.4.2 Binary Protocols

In robotics, basically all data from sensors are retrieved in binary form. Because resources of microcontrollers are very limited, it is reasonable to use binary protocols. Binary protocols are oriented around data structures. Groups of consecutive bytes in a packet are coded as ordinary data types in their native binary representation (integers, floats, etc..). Very often, the data structures of the C language define the semantic of a binary packet.

Binary protocol will be implemented for the sake of best performance.

Chapter 6

Embedded Control System of a Robot

There will be a program in a robot that will provide access to robot's sensors and actuators. Moreover, this program will carry out locomotion commands issued from the host. Control program of a mobile robot is referred to as an Embedded Control System. Small mobile robots are considered in this thesis. It is expected that the Embedded Control System will run in a microcontroller, which has limited processing power.

This chapter describes design of an Embedded Control System for a small mobile robot with an 8-bit microcontroller. Programming 8-bit microcontrollers is very specific, because there is no operation system that guarantees fundamental features like threading, device drivers or file system. The programmer has to ensure required functionality on his own. Therefore, we point out guidelines for designing peripheral drivers and modules in a microcontroller, as well as some programming practices specific for microcontroller programming.

6.1 Microcontroller

Microcontroller (abbreviated MCU) is a single-chip device, that can run a computer program with a minimum of external components. It is a small computer fitted into an integrated circuit. Besides the processor core, the microcontroller usually contains volatile operation memory (typically SRAM[1]), non-volatile program memory (typically FLASH[2])

[1]Static Random Access Memory

[2]non-volatile memory that can be erased and reprogrammed in units of memory called blocks

and peripherals for interfacing other devices.

Microcontrollers are intended for use in embedded systems, such as consumer electronics, remote controls, automobile industry, toys and others.. For these applications, objectives like low power consumption, durability or cheap price are preferred over computational performance. With respect to low processing power, the programmer should design the software carefully.

6.2 Modules and Peripheral Drivers

The microcontroller is equipped with peripherals which allow interfacing sensors and actuators. To list a few of them, such peripherals include: General Purpose Input/Output (GPIO), Analog-to-Digital Converter (ADC), Universal Synchronous/Asynchronous Receiver/Transmitter (USART), I2C[3] serial bus, Timers with Output Compare feature for Pulse-Width Modulation (PWM) and many others..

All these peripherals need to be configured prior usage. For every sensor and actuator there must be a driver that configures appropriate peripherals and provides the interface for the particular sensor/actuator. The driver should carry out:

- Initialization of peripherals for interfacing sensor/actuator
- Configuration of the sensor/actuator (optional)
- Data retrieval from the sensor / Driving the actuator

Drivers provide basic interface to sensors and actuators. Modules make use of these interfaces to provide additional features and perform more complicated tasks.

In Figure 6.1 there is an example of relations between modules, drivers, peripherals and hardware. Module SpeedRegulator provides interface for setting wheel speed. This module uses two drivers: Encoders and Motors. The Encoder driver calculates the speed of rotation of a wheel and the Motor driver sets the power to motors (PWM duty cycle).

It is a good practice that every driver/module is in separate source file and provides unified interface for initialization - *`customDriver`*`_init()`. Optionally, if the driver/module requires executing operational routines on time regular basis, the *`customModule`*`_task()` is designated for this purpose [12]. Of course, the driver or module will introduce other specific

[3]Inter-Integrated Circuit

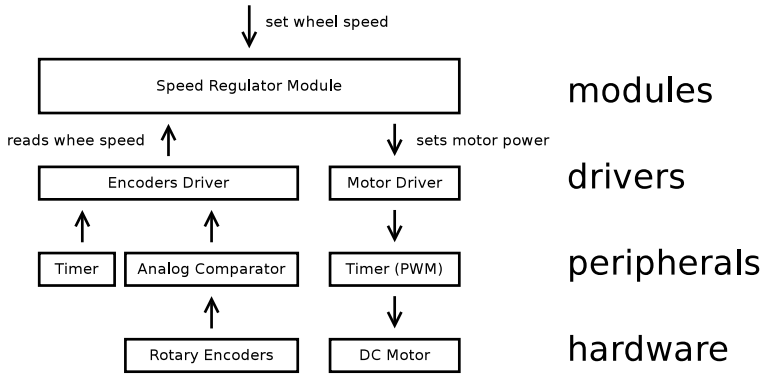


Figure 6.1: Example of Speed Regulator Module

functions for driving actuators and reading sensor values. It is crucial to implement these functions as non-blocking whenever possible. The use of blocking functions for sensor data retrieval significantly decreases the performance of the program.

More importantly, issuing movement commands through the use of blocking function is not possible at all! Imagine a scenario where programmer calls a `goStraightOn(50cm)` function which blocks until the movement is finished. Because there is only a single thread in the microcontroller, no other tasks can be performed at that time (except interrupt routines, of course).

6.2.1 Example of Non-blocking Module

We propose a solution how to avoid such blocking functions. The call of `goStraightOn(50cm)` function will only setup the Locomotion module to execute the task and then the function will return immediately. The movement command itself will be then carried out by the Locomotion module in function `locomotion_task()`, which will be called periodically and will oversee the progress of the movement command. The programmer might want to be informed when the movement is finished. For this purpose, a *callback function* may be passed to the `goStraightOn` function. In conclusion, because there is no operation system in microcontroller that would provide threading and context switching, the programmer has to mimic such functionality as shown above. For this reason, functions should be non-blocking.

6.2.2 Example of Non-blocking Peripheral Driver

Another good example is the implementation of a driver that transmits and receives data through the UART communication interface. The UART peripheral is only capable of sending and receiving the data on a single-byte basis. When sending a byte, the driver copies the byte to specific register of UART peripheral and initiates byte transfer. The byte transfer takes quite a long time, and when it is complete, the driver is notified. The programmer often needs to sent a sequence of bytes (text string or a data packet). A naive implementation of the UART driver would be a blocking `sendString(String)` function that sends individual bytes one after another and waits until the last byte was sent. But this is way too much ineffective, as UART transmission is orders of magnitudes slower than CPU! The non-blocking implementation of `sendString(String)` would rather copy the data to a *circular buffer*, from where the data is sent later, in an asynchronous manner, using the interrupt techniques. For receiving bytes, similar interrupt approach must be used, as no other way is reasonable in that case.

6.2.3 Debugging and Error Messages

It is a good practice to output textual debugging and error messages from modules, which will make the development and debugging easier. A unified debugging interface with adjustable verbosity levels will be used. This method was introduced in Dean's Camera LUFA project [12].

The `printf` function provides convenient way for printing strings and numbers in human readable form. First, it is necessary to setup the C `printf` function to work properly on a microcontroller. Obviously, there is no screen, so the output will be passed to a serial interface and displayed in PC. Any stream device can be used as output for the `printf` function. The programmer only need to set up `printf`'s TxByte hook accordingly.

Then, in a header file of a module, there will be a macro providing debugging output with adjustable verbosity levels. See Listing 6.1.

```
#define PROTOCOL_DEBUG(l, s, ...) do { \
    if (PROTOCOL_DEBUG_LEVEL >= l) \
    printf_P(PSTR("(PROTOCOL) " s "\r\n"), ##
       __VA_ARGS__); \
    } while (0)
#define PROTOCOL_DEBUG_LEVEL 1
```

Listing 6.1: Debugging macro definition

And the usage of macro will be as follows:

```
PB_PROTOCOL_DEBUG(1,"ProcessPacket: wrong
   checksum, discarding packet");
PB_PROTOCOL_DEBUG(2,"Received checksum: %d,
   Expected: %d", packet->checksum, sum);
```

Listing 6.2: Debugging macro - example of usage

This method of textual debugging is flexible and easy to manage. It guarantees visually pleasing and unified debugging output for diverse modules, and can be easily turned off in the release version of the software.

It is important to mention that `printf` might be an overkill in some usage scenarios. In addition, because `printf` function is time-consuming and non-reentrant, the debugging macros cannot be used in interrupt routines. We use the debugging macros in places that are not time critical, e.g. for informing about initialization of modules/drivers, or for signaling unrecoverable error states, where the use of time-consuming function does not matter any longer.

6.3 Robot Settings Reflection

There will be a lot of drivers and modules, and many of them will have adjustable settings. There will be a need to manage all these settings remotely from the Control Library.

Straightforward way would be to design a protocol that specifies many different packet types. For each module there would be one or more packets that update the settings of the particular module. So we would have packets for setting calibration values, then there would be another for setting parameters for PID[4] speed regulator, etc.. At the end, there would

[4]Proportional–Integral–Derivative controller, a generic loop feedback mechanism

be a lot of packet types, and that would make the protocol specification very clumsy and inconvenient to manage. Especially during development, it would be very unpleasant to modify the protocol specification each time the structure of module settings is changed.

Therefore, we introduce a model of globally managed settings. All settings are stored in one big data structure. This data structure will have its mirror in robot's representation in the host. We will implement a simple synchronization protocol that will ensure updates of robot's settings structure when its mirror is changed in the host.

Global settings structure brings further advantages. This approach allows to store and load all settings at once, or sent all the settings to the host at once. This is useful for initializing settings on startup of the robot, when the whole data structure is loaded from a non-volatile memory (EEPROM[5]).

The global settings structure contains sub-structures, which are defined in header files of individual modules. Each module can introduce a *customModule*`_settings {}` struct typedef in its header file. This struct will be part of the global settings structure and storage for its data will be in settings.o, not in the module itself. The module will access its settings by addressing the global settings structure, i.e.
`settings.`*`customModule.customParameter`*`.`

Moreover, a mechanism for updating the settings on a single-field basis is introduced in Section 9.6, which is the foremost advantage of this approach. It means that if only a single field is updated, only few bytes of data are transfered, not the whole settings structure.

The initial robot settings is stored in textual form in the settings.c file, from which the appropriate content of EEPROM memory is generated by compiler. Therefore, ensuring backward compatibility of settings structure during development is irrelevant. For the final version of the Library, the settings structure will not change anymore, so the application programmer may use persistent object techniques in Java for storing the settings in the host. Additionally, he may store the updated settings to the EEPROM of the robot.

[5]Electrically Erasable Programmable Read-Only Memory

6.4 Robot State Reflection

Previous section described the advantages of global settings management. In a similar manner, the state of all robot's sensors is managed globally. There is a global `robot_state` structure, which holds current state of all sensors. Each driver can introduce a *`customModule`*`_state {}` struct typedef in its header file. This will be part of the global `robot_state` structure. The driver will update the state in the `robot_state` structure, i.e. it will store new sensor state into
`robot_state.`*`customDriver.customSensorValue`*`.`

There will be a mechanism that synchronizes the state of all sensors with the robot's representation in host. By default, the whole `robot_state` structure will be transfered periodically to the host. This solution greatly simplifies the usage of the library, because the application programmer will not bother with asynchronous programming issues connected with data retrieval from the robot.

Of course, that implies big traffic on the data link. The application programmer might need only a subset of the `robot_state` structure, in that case this approach is inappropriate. Therefore, we implement methods for adjusting the behavior of the State Reflector. The application programmer may chose from several synchronization modes and he may change the update frequency.

If the application programmer wishes to synchronize only a subset of sensors, he may select a *reduced synchronization mode*. The *reduced synchronization mode* synchronizes only state of encoders, wheel speed and position of the line. Because the packet is much smaller, it can be sent at higher frequency to the host. This allows fine tuning of PID parameters, with a use of a chart for plotting the PID controller response. This is one of the few applications that require high frequency of updates. For other tasks, the *complete synchronization* is sufficient. If the programmer need to synchronize different subset of sensors at high frequency, he may easily implement another custom synchronization mode.

6.5 Movement Control

Movement control for a differentially driven robot with two propelled wheels is taken into consideration. Unlike Ackerman steering[6], differential drive is more flexible for a mobile robot, because it allows turning in place. Thus, trajectory planning is easier for differential driven robot

[6] car-like steering system

than for a robot with a car-like steering system. However, the Ackerman drive is more suitable for terrain or high-speed vehicles.

Differentially driven robot should be able to perform simple maneuvers. The most basic maneuvers are:

1. going straight on
2. turning in place

Both movements are parametrized with speed and distance or angle respectively. Combining these two types of commands into a sequence, the robot can drive on any polygonal line. That is sufficient for many trajectory planing algorithms.

The robot should move smoothly, i.e. it should gradually increase its speed until it reaches the desired speed. In the same manner, it should slow down when it is finishing the movement. Adhering these requirements, not only will be the robot's movement visually pleasant, but also it is much more friendly to the hardware, motors and robot's propulsion system in general. In addition, smooth movements prevents wheel slipping.

Locomotion control should be implemented with respect to limited processing power of a microcontroller. It is expected that there will be only one thread available. This significantly affects the design, because functions that issue movement commands must be non-blocking.

There should be a way to issue all movement commands of the designated polygonal trajectory at once. Therefore, the locomotion control should implement a queue of movement commands. Those commands will be performed one after another, until the queue is empty. The reason for this approach is, that in many usage scenarios the application programmer would need to issue one complicated maneuver at once. That could be avoiding an obstacle for instance. Without queuing the commands, the programmer would need to spend extra effort in setting callbacks for each movement, i.e. he would need to ensure the queue behavior on his own.

Chapter 7

Proximity Detection in a Multi-Robot System

Mobile robots move in the environment, so they might come across obstacles in their way. More interestingly, they might meet other robots. It would be very useful, if robot could make a difference between an obstacle and another robot. In particular, if robots could identify each other when they meet, the multi-robot system would gain much wider applicability. Thus, detecting that robots are close to each other is a desired feature of a multi-robot system.

I have designed and implemented a system for detecting obstacles and other robots. The system employs short-distance optical communication. It consists of two layers, the physical layer and the logical layer. In this thesis, the logical layer is of our interest. However, I will give a brief overview of the physical layer qualities and features, which is essential for understanding the system as a whole.

The proximity detection system is based on short-distance message transfer. It is desired that the range of physical channel is limited only to closest neighborhood of the robot, so that the messages can be only received by nearby robots, or reflected back from nearby obstacles. The physical layer should be able to adjust the transmit power, or the sensitivity of the receiver.

Two common transfer technologies satisfy these requirements; ultrasound and infrared (IR). The infrared transmitter can be directed, while directing ultrasonic transmitter is problematic. The Infrared physical channel was chosen, because IR transmitter and receiver components are cheaper and smaller in comparison to ultrasonic transceivers. However, combination of these two transfer technologies would gain much better results, because ultrasonic system can measure distance more accurately.

7.1 Physical Layer

Modulated infrared light is used for message transfer. The transmitters are infrared LEDs[1] and the receiver is a standard integrated circuit for remote control systems, employed widely in consumer electronics. The transmit power of the LEDs can be adjusted by changing the PWM duty cycle. The LEDs have narrow emitting angle ($\pm 15°$), thus they output directed signal. On the other hand, the receiver has wide receiving angle, so it can receive signal from all directions.

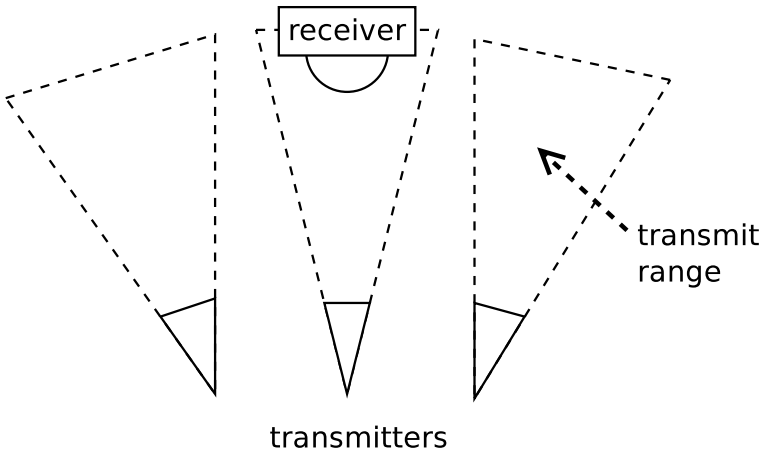


Figure 7.1: Transmitters on one robot, receiver on the other robot

This system can be used for detecting obstacles as well. Infrared light reflects back from the obstacle to the receiver, as shown in Figure 7.2.

[1] Light-Emitting Diodes

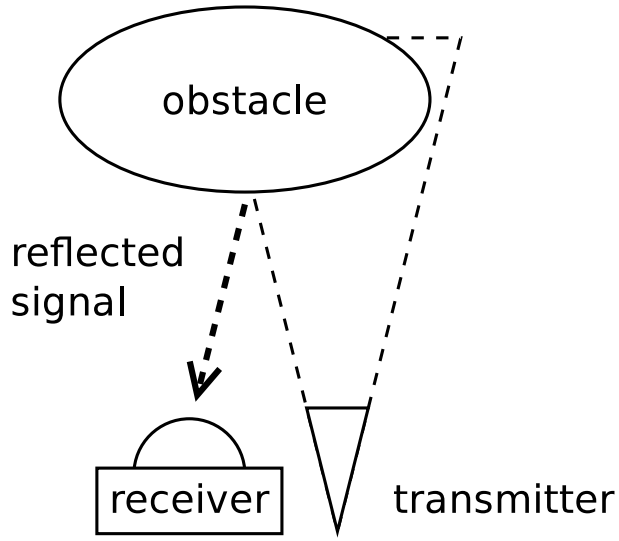


Figure 7.2: Obstacle detection

This method cannot be used for measuring obstacle distance, because different materials have different reflexivity. However, if the material of obstacle is known beforehand, the distance could be roughly estimated.

7.2 Logical Layer

The physical layer described in previous section provides means for controlling the transmit range and direction of transmission. It can handle detecting obstacles on its own.

I have designed a logical layer on the top of the physical layer, which gains more functionality. In particular, the robots will be able to identify each other by an ID[2] and they will also sense the orientation of the other robot and its distance.

The logical layer specifies format of infrared packets. The protocol is similar to well-known remote control protocols employed in consumer electronics. The packets contain following information:

1. The ID of a robot that transmits the packet
2. Which IR LED of the robot was used for the packet transmission
3. The transmitting power (range) used for packet transmission

[2]each robot has a unique identificator

Each robot has front and rear receiver. It can distinguish whether the other robot (the sender) is in front or in rear. Moreover, because the packet contains information which directed LED sent this packet, the receiving robot estimates the orientation of the other robot. And finally, the robot can roughly estimate the distance, because the transmit power is coded into the packet as well. Other data fields can be easily added to the protocol, to transmit arbitrary information.

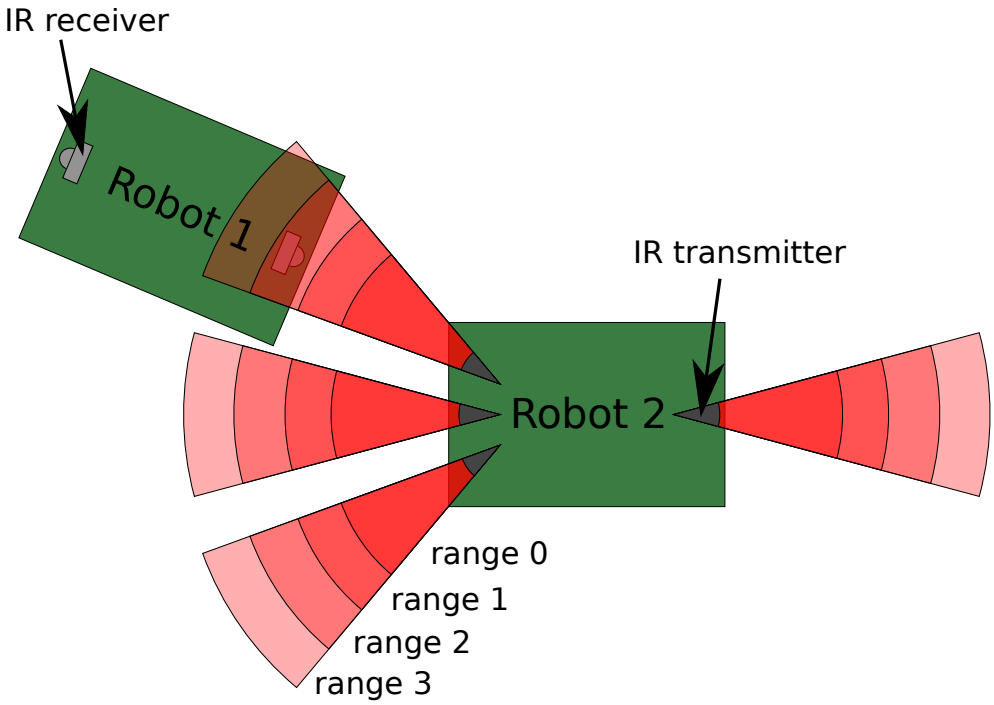


Figure 7.3: Detecting other robots using IR Proximity sensor

Part III

Implementation

In this part, we will describe the implementation of the Control Library in Java, using Bluetooth wireless technology for data transport. The implementation of robot's Embedded control system in an 8-bit micro-controller will be described as well.

Afterwards, we will concern on the implementation of specific control modules for sensors and actuators of an existing robot. The Control Library and Embedded Control System will be tested on real robots. This will prove that the presented concept of a Control Library and robot's Embedded control system is suitable for controlling multiple mobile robots. For this purpose, **I designed and built PocketBot2 robots.** For each sensor and actuator of the PocketBot2 robot, there will be a corresponding control module in the Embedded control system and in the Control library.

Chapter 8

PocketBot2

In this chapter I will present the key features of the PocketBot2 robot so that we get an idea what sensors and actuators we have to deal with in the Embedded System and in the Control Library. PocketBot2 is a tiny line-following robot designed for batch production. It is unique because of its size: The robot is so small that it fits into a matchbox (dimensions of robot: $48 \times 32 \times 12$ mm, see Figure 8.1). The tiny size makes PocketBot2 an ideal robot for testing multi-robot algorithms. First, one can run several PocketBot2 robots on a table or desk, whereas robots of regular size would need much more space. Second, if the algorithm goes wrong, the robots are very unlikely to make any damage to surroundings or themselves, because they weight only 20 grams. Last but not least, the programmer can easily manipulate robots without getting up from a chair.

Figure 8.1: PocketBot2, the matchbox sized robot (right box) with equipment (charger and programmer in left box)

8.1 Feature Overview

Although PocketBot2 is one of the smallest robots, it offers a wide range of functionality. In the terms of features, PocketBot2 can compete with many regular-sized mobile robots. We present a brief overview of PocketBot2 features in this section. Please refer to a 3D model of the robot in Figures 8.2 and 8.3.

8.1.1 Microcontroller

Robot's embedded control system runs in an 8-bit Atmel ATxmega128A3 microcontroller. The microcontroller operates at 32MHz and offers 8kB SRAM and 128kB of program memory (FLASH). The computation power is relatively small, therefore only the locomotion control is carried in the robot while other computationally demanding tasks have to run in the host computer.

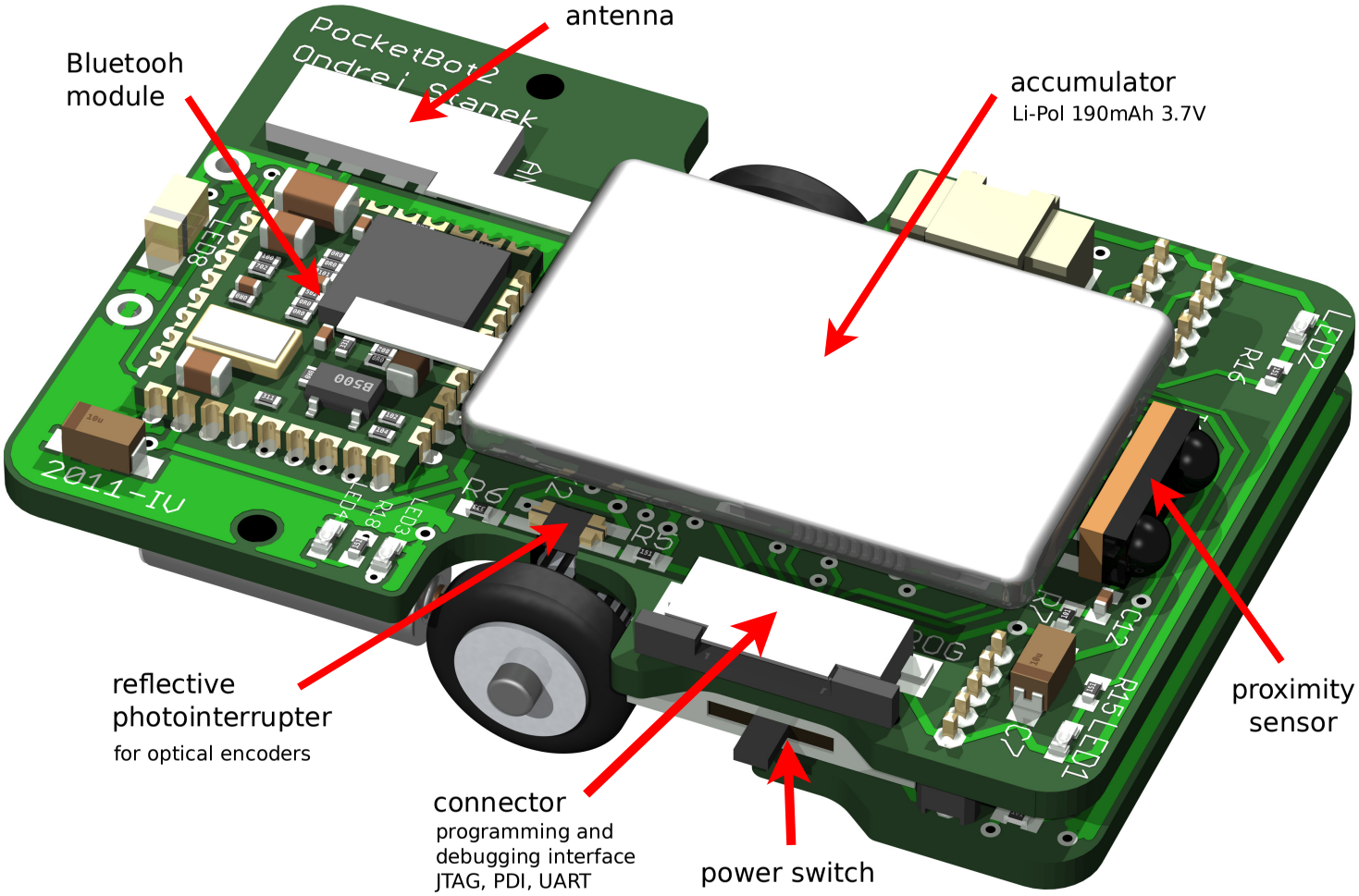


Figure 8.2: PocketBot2 - top view

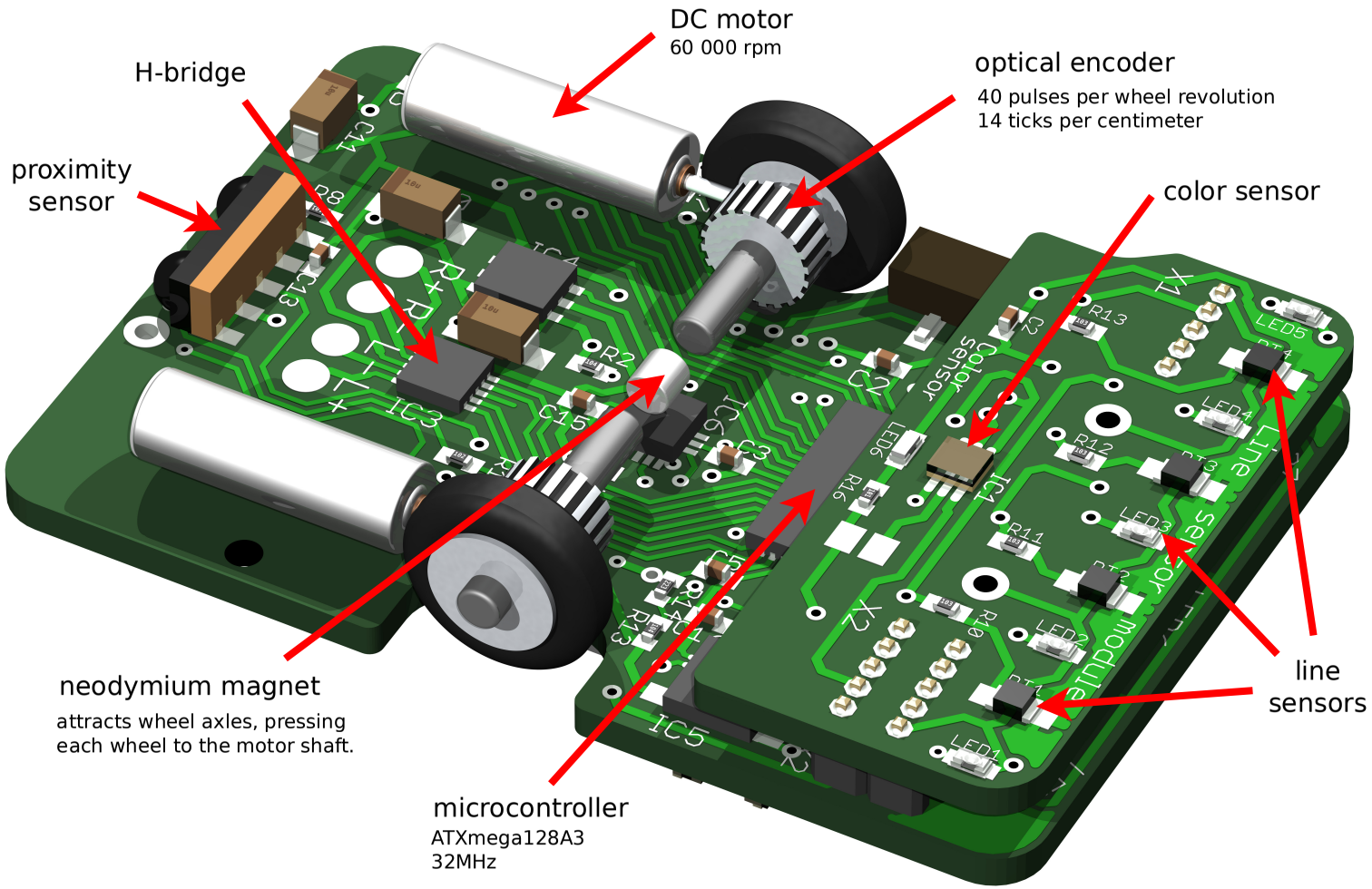


Figure 8.3: PocketBot2 - bottom view

8.1.2 Bluetooth

Bluetooth technology ensures wireless link between the robot and host (computer). The robot is equipped with an HCI[1] Bluetooth module. Unlike commonly used RFCOMM[2] Bluetooth modules, the HCI module can benefit from all features of the Bluetooth technology, such as broadcasting and forming scatternets.

8.1.3 Wheelframe

Two independently driven wheels (9mm diameter) provide differential steering. Powerful motors from mini helicopter are used, which guarantees high speed performance. The dimensions of the gear mechanism were crucial due to considerable space constraints. The wheelframe employs a friction gear system with magnetic pressure. A neodymium magnet in the central tube attracts wheel axles, pressing each wheel to the motor shaft.

Figure 8.4: Wheelframe of the PocketBot2 robot

8.1.4 Rotary Encoders

On each wheel, there is an optical sensor that measures wheel rotation. The sensor provides 40 pulses per wheel revolution, which gives approximately 14 pulses per 1 cm of robot's trajectory. This resolution is sufficient for locomotion control of the robot. Rotary encoders are essential for precise movement, speed regulation and localization.

[1] Host Controller Interface
[2] Radio Frequency Communication

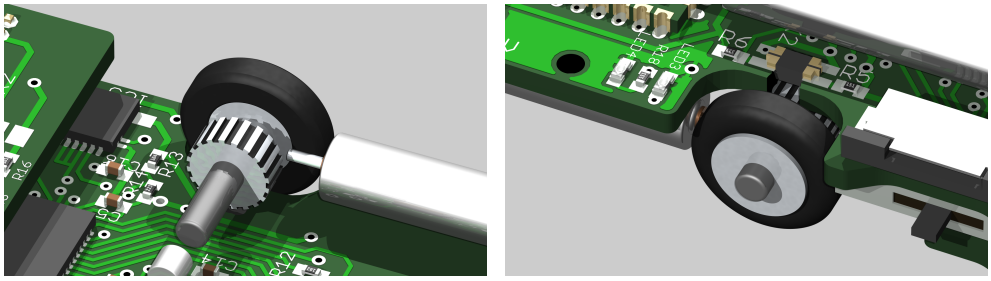


Figure 8.5: Rotary encoder

8.1.5 Proximity Sensors

Robot is equipped with front and rear proximity sensor. The proximity sensor is a complex unit that allows both obstacle detection and short-distance communication with other robots. It means that robots can identify each other when they meet, and they can make difference between a robot and an obstacle, as described in Chapter 7.

8.1.6 Line Sensor Module

Line sensor module allows line following. The compound sensor measures reflectivity of surface at 8 spots. Values from this sensor are processed and the estimated position of the black guiding line is calculated. This input is then used for the line-following algorithm, described in Section 11.2.

8.1.7 Other Sensors

Moreover, PocketBot2 contains a color sensor, 3-axis accelerometer and it has means for monitoring the battery voltage, current consumption and the temperature of the microcontroller core.

Chapter 9

Control Library

The Library for controlling robots is implemented in Java. Java was chosen because it is multi-platform and the same bytecode can run on Windows, Linux and Android OS. The Library ensures discovering robots in range, establishes a links to robots and performs synchronization between the robot and its representation.

The Library utilizes two open source projects; BlueCove and Javolution. The BlueCove library [15] is an implementation of JSR-82 Bluetooth specification [23]. It interfaces several Bluetooth stacks (BlueZ on Linux and Widcomm on Windows systems are fully supported), providing access to Bluetooth transport layers. The Javolution library, more particularly its Struct class, ensures interoperability between Java classes and C/C++ data structs.

The implementation of the Control Library is referred to as BTlib.

9.1 Package Overview

Package BTlib contains `PocketBot` class, whose instances represent connected robots. Next, there is the `Discoverer` interface and `PocketBotDevice` abstract class. These two ensure generalization and unified API[1] for different transport technologies. The `Discoverer` interface and `PocketBotDevice` abstract class are implemented for every requested transport technology (and in some cases even for individual platforms, because the system API may differ across platforms).

The Discoverer defines a unified interface for discovering robots. It carries out the discovery process and returns list of `PocketBotDevice` objects. `PocketBotDevice` represents connectible robots. It establishes

[1] Application Programming Interface

connection to a robot, instantiates the `PocketBot` object after the connection is established and then ensures packet transfer between the `PocketBot` object and the real robot. This abstraction allows to use different transport technologies and layers.

The BTLib.Bindings sub-package contains implementations of the `Discoverer` and `PocketBotDevice` for individual transport technologies and platforms. Currently, bindings for Bluetooth technology are implemented. Namely, there is an implementation for the JSR-82 Bluetooth API (BlueCove on Linux and Windows) and the Android API. Bluetooth L2CAP asynchronous data transport is used.

The BTLib.packets sub-package contains classes that represents various packets used for communication. All packet classes have a common ancestor, the `GenericPacket` class, which introduces attributes and methods common to all packets. Moreover, this design takes advantage of object polymorphism, which simplifies packet processing. The packet communication will be described further in Section 9.5.

Finally, the last package BTLib.Modules contains implementations of modules specific to a particular robot. Such are for example Proximity sensor, Line Sensor Module, Locomotion control and others. These module classes are instantiated within the `PocketBot` object.

9.2 PocketBotDevice Abstract Class

The `PocketBotDevice` class introduces abstract method `Connect`. The `Connect` method must be implemented by subclasses so that it establishes a link to the robot, starts a communication thread and returns the robot representation (instance of the `PocketBot` class). The communication thread listens for new packets and passes them to the `PocketBot` object. The `PocketBotDevice` declares `SendPacket` abstract method for transmitting packets. Finally, there is the `getName` method for obtaining user-friendly robot's name.

9.3 PocketBot Class

Robots are represented as instances of the `PocketBot` class. Robot's sensors and actuators are accessible through various Modules. The `PocketBot` class implements `ProcessPacket` method, which takes care of packet dispatching. The state of modules is updated every time the

`StatePacket` arrives. If the application programmer wishes to be informed about every update, he may register a `PocketBotEventListener`.

9.4 Modules

The Library Modules have 1:1 correspondence to the modules of the Embedded Control System. Modules may implement additional processing methods for the data they manage, such as unit conversions. Moreover, some new modules are introduced. Those are modules that process data from robot sensors, but are difficult to implement in the Embedded control system as they are computationally expensive. Great example is the `Odometry` module, which keeps track of robot relative position thanks to information from encoders.

9.5 Packet Communication

Data is transferred in binary format. The semantics of protocol packets is specified by C structures, so that the data in packets correspond to the robot's internal data representation. The protocol was designed this way for a purpose; the coding and decoding of packets in the robot is just a matter of structure *typecasting* in C language. Thus, implementing the protocol in C is very straightforward and effective. This chapter describes the implementation of such binary protocol in Java, which is far more complicated.

9.5.1 Structure of a Binary Packet

All packets have common header. The header contains packet type, checksum and payload length. The rest of the packet is the payload, i.e. arbitrary data. See Figure 9.1. For every packet type, the semantic of the payload is defined individually.

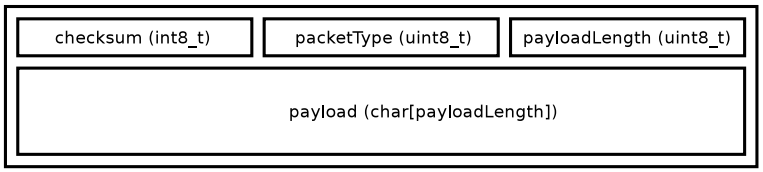


Figure 9.1: Structure of a binary packet

The `packetType` field defines how to interpret the payload. In Figure 9.2 there is an example of a packet that initiates a movement command. We will use this packet throughout this chapter to explain packet representation and handling.

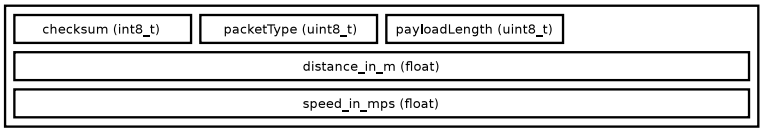


Figure 9.2: Packet example - StraigthMovement

In C language, the packet is represented as a structure:

```
typedef struct {
        int8_t checksum;
        uint8_t type;
        uint8_t payloadLength;
        struct payload {
                float distance_in_m;
                float speed_in_mps;
        };
} StraightMovement_t;
```

Listing 9.1: StraightMovement packet definition (C language)

However, this is not a sufficient definition of a protocol packet if we do not know some extra information about the compiler and platform. There are some pitfalls one may encounter. First, the *endianness* of data may differ across platforms. Second, the data structures may have internal padding, which is implied by the architecture. As a result, the packet is not defined only by a definition of a particular C structure, but also by the information about endianness and structure field alignment[2].

The Library is in Java. We face the problem how to interpret native C structures in Java. The memory handling differs fundamentally in these two programming languages. Unlike C/C++, the storage layout of Java objects is not determined by the compiler [14]. In addition, their data types are not compatible either. (Java does not have unsigned integer types, for instance.) Therefore it is necessary to ensure interoperability between C/C++ structs and Java classes. This is achieved by using Javolution `Struct` class.

[2]We force the C compiler to use *Packed structures* (without padding), which simplifies the issue

9.5.2 Javolution Library - Struct Class

Javolution Library provides the `Struct` class that allows to interpret C/C++ data structures in Java [14]. The `Struct` class has several inner classes, which represent C data types. Please see Listing 9.2 for a Java equivalent of the StraightMovement packet mentioned as an example:

```
public class StraightMovement_packet extends Struct {
  public Signed8 checksum = new Signed8();
  public Unsigned8 packetType = new Unsigned8();
  public Unsigned8 payloadLength = new Unsigned8();

  public class Payload extends Struct {
    public Float32 distance_in_m = new Float32();
    public Float32 speed_in_mps = new Float32();
  }
  public Payload payload = inner(new Payload());
}
```

Listing 9.2: StraightMovement packet representation (Javolution Struct)

All the data of the packet is stored in a `ByteBuffer`, which can be retrieved or set by calling appropriate methods on the `Struct` instance. Basically, the inner members only provide a *view* on the binary data stored in the underlying `ByteBuffer`. Therefore, by updating the `ByteBuffer`, all members of the structure are updated as well. And vice versa, if a member is set to a new value, the underlying `ByteBuffer` gets updated accordingly.

In conclusion, by altering the `ByteBuffer` of Javolution `Struct` we can achieve functionality similar to the C structure typecasting. For instance, sending a packet from Java would mean sending the content of underlying `ByteBuffer`, and decoding a packet would mean setting the received `ByteBuffer` as a base for the adequate Javolution `Struct`.

It is important to mention that Javolution `Struct` allows further configuration of byte order and structure packing. In this implementation, Little Endian Packed structures are used, as it conforms to the data representation in the C program of the robot.

9.5.3 GenericPacket Class

The `GenericPacket` is a superclass for each protocol packet. It contains the essential header data fields (`checksum`, `packetType` and `payloadLength`). `GenericPacket` class is designed both for receiving and

transmitting packets, it makes no difference. There is a simple way how to map incoming data to a packet class. Suppose the data of the packet is stored in a byte array. We wrap the array into a `ByteBuffer` and this `ByteBuffer` is set as an underlying data source for the `GenericPacket` or its subclasses.

`GenericPacket` class implements checksum method for packet consistency verification. The checksum verification is independent of the type of a packet; the checksum method can be called from an instance of `GenericPacket`, without instantiating a particular packet subclass.

The `GenericPacket` greatly simplifies definition of arbitrary packets. Adhering our example, the `StraightMovement` packet based on `GenericPacket` class looks as follows:

```
public class StraightMovement extends
   GenericPacket {
    Float32 distance = new Float32();
    Float32 speed = new Float32();
    public StraightMovement(float distance_m,
       float speed_mps) {
        SetPacketType(0x40);
        this.distance.set(distance_m);
        this.speed.set(speed_mps);
        FinalizePacket(); //counts checksum
    }
}
```

Listing 9.3: StraightMovement packet based on GenericPacket class

Please note that the packet header fields are private members of the `GenericPacket`. Methods `SetPacketType` and `FinalizePacket` fills the header fields with necessary data.

Sending such packet to the robot is as easy as:

```
SendPacket(new StraightMovement(1.0, 0.3));
```

9.5.4 Processing Packets

Decoding a packet is straightforward as well thanks to the `GenericPacket` superclass. At first, we instantiate the `GenericPacket` and set the received data as its underlying `ByteBuffer`. `GenericPacket` implements checksum verification and retrieves packet type. According to the type of the packet, we instantiate the appropriate `GenericPacket` subclass.

Each Packet class implements a method `ProcessFor(PocketBot robot)`, which updates the state of the robot. That means the code for handling individual packets is distributed among the packet classes, which is good in terms of code maintainability.

```
GenericPacket genericPacket = new GenericPacket()
   ;
genericPacket.setByteBuffer(buffer, 0);
if (!genericPacket.verifyChecksum()) {
  throw new UnsupportedOperationException("Bad␣
     checksum");
}
int packetType = genericPacket.getPacketType();
switch (packetType) {
  case 0x01: {
    StatePacket packet = new StatePacket();
    packet.setByteBuffer(buffer, 0);
    packet.ProcessFor(robot); // updates the
       robot state
    break;
  }
  //... processing other packet in a similar
     manner
  default:
    throw new UnsupportedOperationException("
       Unknown␣packet␣type:" + packetType);
}
```

Listing 9.4: Processing packets in Java

Remark: The checksum verification is implemented only for the sake of soundness and was useful for debugging the protocol implementation. It might be also useful when implementing other transport layers. The underlying Bluetooth L2CAP channel is characterized as reliable according to the Bluetooth Specification [16]. It ensures data integrity checks and packet retransmission. In addition, the lower Bluetooth layers provide error correcting.

Citation from the Bluetooth Specification:

The baseband packet header uses forward error correcting (FEC) coding to allow error correction by the receiver and a header error check (HEC) to detect errors remaining after correction. (...)

On ACL[3] *logical transports the results of the error detection algorithm are used to drive a simple acknowledgement/repeat request (ARQ) protocol. This provides an enhanced reliability by re-transmitting packets that do not pass the receiver's error checking algorithm. (...)*

The L2CAP layer provides an additional level of error control that is designed to detect the occasional undetected errors in the baseband layer and request retransmission of the affected data. This provides the level of reliability required by typical Bluetooth applications.

However, the Bluetooth specification states that undetected errors may still occur:

Due to the error detection system used some residual (undetected) errors may still remain in the received data. For L2CAP channels the level of these is comparable to other communication systems.

We implemented data integrity check in the protocol. However, our checksum mechanism has not detected any residual errors, so we consider that mechanisms ensured by Bluetooth layers are sufficient.

9.6 Settings Reflection - UpdatableStruct Class

This section describes implementation of a settings synchronization mechanism. When the application programmer updates settings in robot representation, the change is automatically transfered to the real robot. The transfer is made on the single-field basis, which means only the necessary data is transfered, not the whole settings structure.

We implemented an `UpdatableStruct`, which is a subclass of the Javolution `Struct` class. The `UpdatableStruct` adds `update` method to inner data members, as well as to the `Struct` class itself. When the `update` method is called on any element of the `Struct` class, the particular data field or sub-structure is updated in the real robot.

There is an `UpdateSettings` packet dedicated for updating settings. (See Figure 9.3) This packet has variable length and contains `offset`, `dataLength` and `data` fields.

[3] *Asynchronous Connection-oriented [logical transport]*

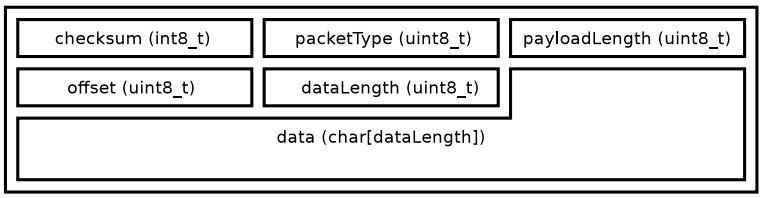


Figure 9.3: UpdateSettings packet

Fields `offset` and `dataLength` specifies a region of the settings memory space, which will be replaced by the `data` carried in the packet. The `UpdatePacket` class has constructors that accept `Struct` inner members as a parameter. The constructor assembles the `UpdateSettings` packet and fill it with the appropriate data, retrieved from the `Struct` inner member passed as an argument.

This solution hides communication issues from the application programmer, because the `SettingsUpdate` packet is assembled and sent automatically, in the back-end of the library.

9.7 State Reflection

Unlike the settings, state of robot is synchronized in opposite direction, from robot to host. There is a `StatePacket` that contains the whole robot_state structure. It is sent periodically from the robot. When this packet is received, it updates all modules in the robot representation and `PocketBotEventListener` is called.

Application programmer may configure the behavior of the State Reflector by calling the appropriate methods of `robot.stateReflector` inner class. He may set the period of updates (`setPeriod` method) and he may switch between synchronization modes.

There are three synchronization modes implemented: Full synchronization, Reduced synchronization[4] and No synchronization. The programmer may easily implement additional synchronization mode if it is required by the application.

[4]synchronizes only encoders, wheel speed and line position

9.8 Odometry

For every mobile robot, it is very useful to keep track of its position. Calculating the relative position of the robot using robot's geometric model and data from moving sensors is called odometry.

The odometry method is based on the Dead Reckoning principle [2], which calculates the current position from the previous position and estimated movement:

$$P_{t+1} = P_t + Movement$$

For a mobile robot on a plane we have:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \\ d_\theta \end{bmatrix}$$

We will now investigate how to calculate the translation $[d_x, d_y]$ and the change in rotation d_θ using the data from robot's wheel encoders. It is important to mention that the Dead Reckoning method suffers from incremental error because of its recursive character.

9.8.1 Robot's Model

We assume a differentially driven two-wheeled robot, which conforms to the PocketBot2 construction.

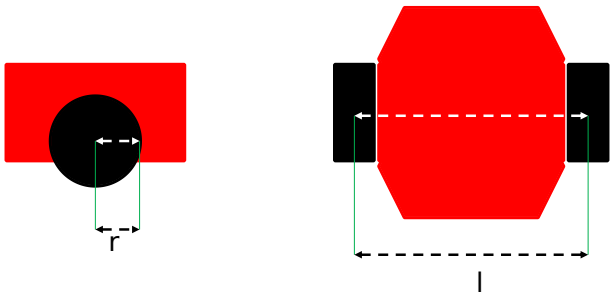


Figure 9.4: Model of a two-wheeled differentially driven robot

The physical dimensions of the robot are known: wheel radius r, circumference of the wheels $2\pi r$ and wheel gauge l. Further, in each iteration, we measure the distance traveled by the wheels (d_1 and d_2).

$$d_1 = 2\pi r \frac{t_1}{T} \qquad d_2 = 2\pi r \frac{t_2}{T}$$

Where t_1, t_2 is the count of encoder pulses on left and right wheel respectively, during a fixed time period of the iteration. Constant T is the number of pulses for one wheel rotation.

Then, the distance d traveled by the robot is given by the average of the distance traveled by its wheels.

$$d = \frac{d_1 + d_2}{2} = \frac{2\pi r \cdot (t_1 + t_2)}{2T}$$

Now, denoting θ the actual rotation, we have $d_x = cos(\theta) \cdot d$ and $d_y = sin(\theta) \cdot d$

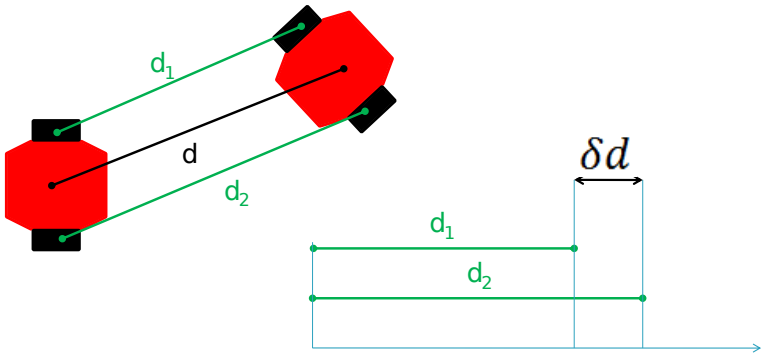


Figure 9.5: Translation of a robot in one iteration

Finally, we have to find the change in rotation d_θ. We need to look at the difference in distance traveled by the wheels: $\delta d = d_2 - d_1$. Our model makes following assumption: In every iteration, at first the robot moves straight on (both wheels are turning the same speed), until it travels the distance $min\{d_1, d_2\}$. Then, the robot starts to rotate. The rotation d_θ (in radians) is then given by δd and wheel gauge l:

$$d_\theta = \frac{\delta d}{l} = \frac{2\pi r \cdot (t_2 - t_1)}{l \cdot T}$$

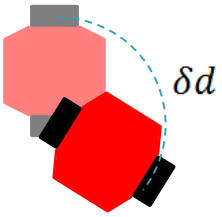


Figure 9.6: Rotation of a robot

Simon Coupland showed this method in his presentation [2]. However, the formula he proposes may bring rounding problems when it is implemented in floating point arithmetic in computer. According Simon Coupland, the angle θ is the sum of all changes in rotation, which are small real numbers:

$$\theta = \sum_i d_{\theta_i}$$

A better solution for calculating the heading angle θ exists.

One has to get insight that the robot's heading angle θ is invariant to partial movements, only the total distance each wheel traveled is of our interest. Then, the formula does not depend on partial distances t_1 and t_2 anymore, and it is as follows:

$$\theta = \frac{2\pi r}{l \cdot T}(N_2 - N_1)$$

Where N_1, N_2 is the total count of encoder pulses on each wheel since the very beginning. (Note that these are integer values.)

We present a simple proof that this formula is equivalent with the one Simon Coupland suggests:

$$\theta = \sum_i d_{\theta_i} = \sum_i \frac{2\pi r \cdot (t_{2_i} - t_{1_i})}{l \cdot T} = \frac{2\pi r}{l \cdot T} \sum_i (t_{2_i} - t_{1_i}) =$$

$$= \frac{2\pi r}{l \cdot T} \left(\sum_i t_{2_i} - \sum_i t_{1_i} \right) = \frac{2\pi r}{l \cdot T} (N_2 - N_1)$$

In conclusion, calculating the robot position from wheel encoders requires floating point arithmetics and trigonometric functions, therefore it is difficult to implement in the embedded control system, because the microcontroller of the robot has limited processing power. Odometry is a great example of a module that can be implemented better in the control library than in the robot itself.

Chapter 10

Bluetooth Stack

Bluetooth Stack is a software that implements various Bluetooth protocols in accordance with the Bluetooth specification [16]. The Bluetooth stack runs in the host (PC, embedded system) and interfaces Bluetooth Controller hardware. The Bluetooth Controller hardware consists of several functional blocks and layers. Please refer to Figure 10.1.

The Radio layer assures transmitting and receiving packets on the physical channel. Baseband layer manages the communication on the channel, Link Manager layer carries out connecting procedures, link control and other aspects of Bluetooth communication.

Every Bluetooth controller (USB Bluetooth dongles for PCs, HCI Bluetooth modules for embedded systems) offers a standardized Host Controller Interface (HCI). This interface defines how to access functionality of the Bluetooth Controller hardware from the host (Bluetooth Stack). The Bluetooth Stack then builds protocols on the top of resources provided by the Controller hardware. Such protocols provide features like reliable transport of data frames (L2CAP), service discovery (SDP), serial port protocol emulation (RFCOMM), binary object exchange (OBEX) and others. Moreover, the Bluetooth Stack allows configuring the Controller (device name, visibility) and provides interface for services that are implemented in the Controller (device discovery, pairing, link encryption, role switch..).

The communication between two hosts is presented in Figure 10.2. Each host has implemented a set of protocols (Bluetooth stack), that allows user data transmission. The Bluetooth stacks communicate with the Bluetooth Controller via HCI, and the Controller carries out the data transmission on the physical channel.

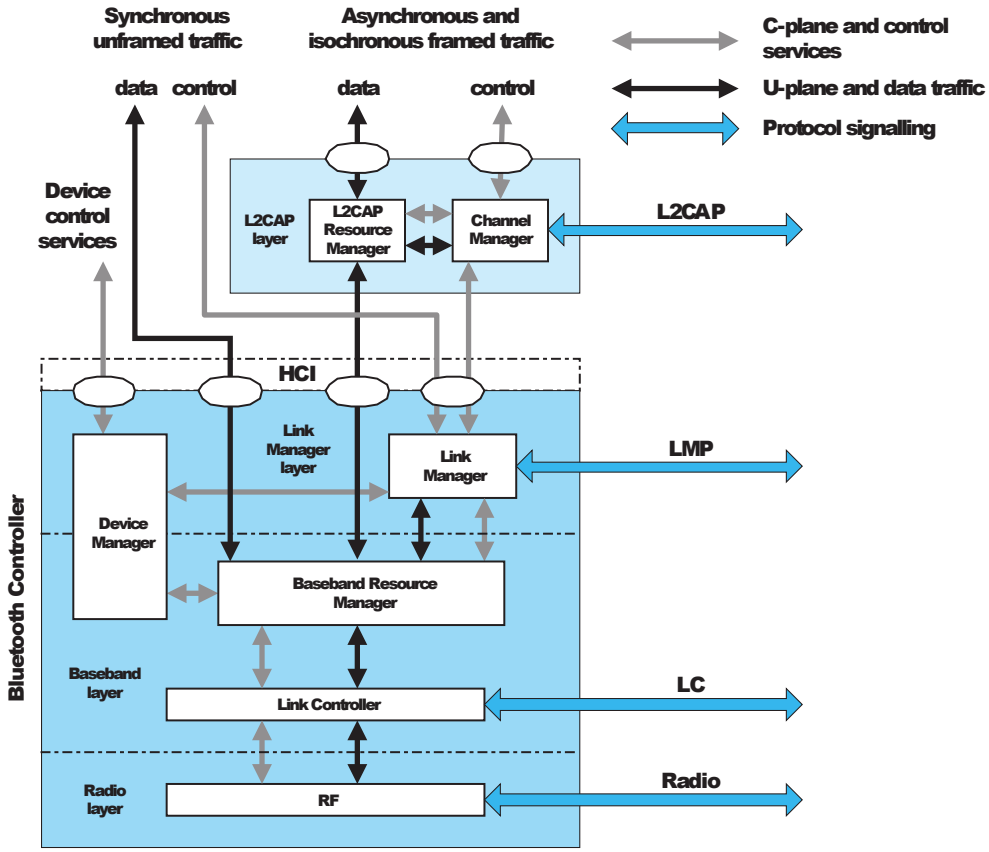


Figure 10.1: Bluetooth core system architecture

10.1 Host

Bluetooth stack is included in all modern operation systems. On Linux, there is a BlueZ stack that supports all core Bluetooth protocols. On Windows systems, several stacks are available. The Microsoft Bluetooth stack in WinXP (Winsock) unfortunately does not provide API for interfacing the L2CAP layer, which is necessary for framed data transport. Therefore, other stacks must be used. WIDCOMM and Toshiba stacks are some of the options. Some stacks have license limitations and they can be only used with a specific Bluetooth controller hardware, despite the fact that every Bluetooth hardware has standardized interface (HCI).

There is a need to interface any of these Bluetooth stacks from the Java Control Library. The Java APIs for Bluetooth (JSR-82) specifies an interface how to use Bluetooth technology from Java programs [23]. BlueCove is an open-source Java library that implements the JSR-82 speci-

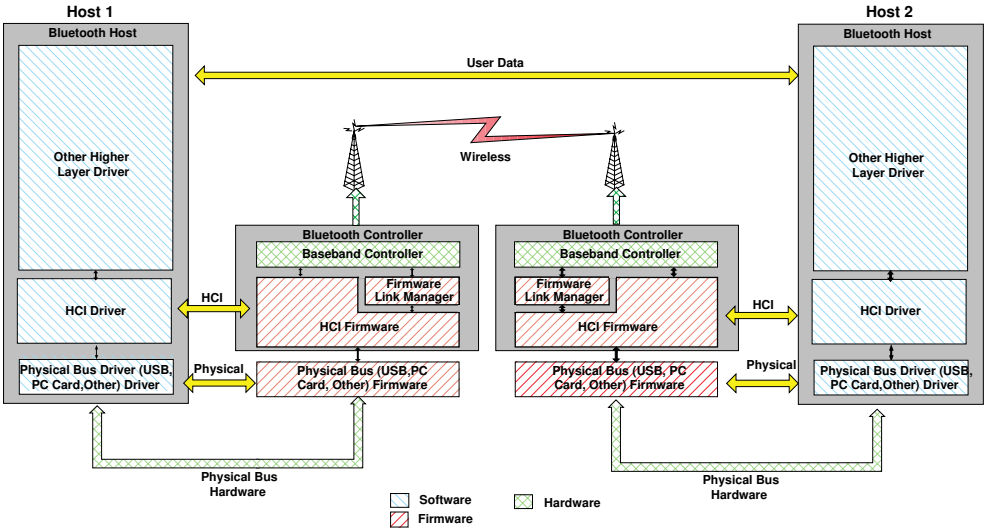


Figure 10.2: Bluetooth technology End to End overview

fication. So far, BlueCove can interface BlueZ, WIDCOMM, BlueSoleil, Mac OS X stack and Winsock (default stack in XP, Vista and Windows Mobile).

Android phones are based on Linux and they incorporate BlueZ stack. In the Library implementation, we used the native Bluetooth API provided by the Android platform. However, there are some indications that BlueCove library can run on Android systems as well [17].

10.2 Microcontroller

There are two ways how to implement Bluetooth technology in a robot. The Bluetooth stack is located either in the Bluetooth module itself, or it is implemented in the microcontroller of the robot. The first option is commonly used in robotics. Implementing a Bluetooth stack in a microcontroller is a *very* challenging task, so many authors rather choose single-purpose Bluetooth modules that have protocol stack integrated on chip. Such modules are known as Bluetooth RFCOMM modules and they are meant as a cable replacement for the Serial Port (RS232). When the RFCOMM module is linked with a computer, the operation system creates a virtual serial (COM) port. The data stream sent to the virtual port is transparently carried over Bluetooth channel to the serial interface of robot's microcontroller. This allows to build up a wireless connection to the robot with minimal effort. However, the capabilities

of integrated Bluetooth stack are very limited; basically only a single point-to-point RFCOMM connection initiated by the host (computer or phone) is possible. The configuration of such module is limited as well.

The second option is to use a generic HCI Bluetooth device and implement the Bluetooth stack in the microcontroller. This approach is free of any limitations; the programmer may implement any feature of the Bluetooth technology, such as forming piconets and scatternets, broadcasting, role switching, device discovery, encryption, HID[1] protocols, QoS[2] control, asynchronous or synchronous data links and others. We chose the HCI Bluetooth module, because it greatly extends the robot's communication possibilities. In future, a decentralized robot control system might be implemented for PocketBot2 robots thanks to this design decision. Moreover, HCI modules are cheaper and smaller than their RFCOMM counterparts.

However, as stated before, implementing a complete Bluetooth stack is a very difficult task. Therefore, we ported an existing Bluetooth stack to the Atmel XMega microcontroller. There are many Bluetooth stacks for embedded systems, even more than for computers [18]. Unfortunately, the stacks are often proprietary and have strict licensing policies. Two open source solutions suitable for our purpose exist: lwBT and LUFA BT stack.

10.2.1 lwBT Stack

The lwBT [13] is a lightweight Bluetooth stack for embedded systems. It is a part of bigger project, lwIP[3]. The stack supports BCSP[4] and H4[5] HCI transport layers[6]. This stack was not suitable for porting, because it involves "old school" C programming techniques that makes the code difficult to read.

[1] Human Interface Device

[2] Quality of Service

[3] lightweight TCP/IP stack for embedded systems

[4] BlueCore Serial Protocol, proprietary solution of CSR

[5] H4 is the standard way of transmitting Bluetooth data over a UART as defined in the Bluetooth specifications

[6] although HCI is a universal interface for Bluetooth modules, several HCI packet transport layers exist. The HCI can use both USB and UART data transports

10.2.2 LUFA BT Stack

LUFA[7] is an open source project that implements USB interface for Atmel AVR microcontrollers [12]. It contains many examples how to interface various USB devices. One of these devices is a USB Bluetooth dongle and the example includes Bluetooth stack that supports L2CAP, SDP and RFCOMM protocols. The code is well documented and self-explanatory. Stack is optimized[8] for AVR microcontrollers, which suits our purpose. However, it is necessary to replace the USB transport layer with UART transport, i.e. the stack has to be ported to different physical interface.

10.2.3 Porting LUFA BT Stack

Before porting the LUFA BT stack to Xmega's UART interface, I decided to test the stack first on PC, which provides better debugging possibilities. Therefore, I first made a port of the stack to Linux, using Serial (COM) port for interfacing the HCI module. Afterwards, I ported the stack to XMega microcontrollers.

The LUFA BT stack is a monolithic solution that tightly connects the HCI USB interface code with the stack code (in particular, with HCI commands layer and ACL layer of the stack). This design was probably chosen because it decreases overhead connected with introducing separated generic HCI layer, although doing so would greatly simplify porting the stack to other data transport technologies. So, considerable changes has to be done to the HCI and ACL layer of the LUFA BT stack, in order to implement UART physical transport. Such patches to these fundamental layers of the Bluetooth stack would made very difficult to update the stack in future, when new version is released. Therefore, I decided to implement a thin layer that mimic USB interface on the top of the UART HCI interface. This will make minimal interventions to the LUFA BT stack, while it does not decreases performance, as the packets of H4 HCI UART protocol can be simply mapped to USB pipes and endpoints.

The porting of a stack has two phases. First, the physical interface has to be ported from USB to UART, second, architecture specific code constructions has to be replaced. (However, this only considered the PC port. It was not necessary for the Xmega port.) The BluetoothACLPackets.c/h and BluetoothHCICommands.c/h were modified so that they did

[7] Lightweight USB Framework for AVRs

[8] uses AVR-specific *pgmspace* functions that reduce memory overhead when handling strings

not include the original USB handling routines, but they included the /hci_phy_iface/interface.h instead, which mimic the USB functionalities on UART interface.

In conclusion, the LUFA BT stack was ported in a way that it is easily updatable in future, when a new version of the stack will be released. In addition, adhering the same method, I ported the LUFA BT stack to a 32bit ARM microcontroller as well.

Chapter 11

Embedded Control System

Robot's Embedded Control System is implemented in C and runs in the ATXmega128A3 8-bit microcontroller. It consist of drivers and modules that interface various sensors and robot's motors. Moreover, it implements the communication protocol and global management of robot's settings and state. Bluetooth stack is included as well.

In Figure 11.1, the complete schema of all implemented modules and drivers is presented. The block diagram shows relations between modules, drivers, peripherals and hardware of the PocketBot2 robot. In this thesis, we will describe implementation of interesting units of the Embedded control system. Documentation to all implemented modules and drivers can be found on enclosed CD.

11.1 Scheduler

A simple Scheduler is implemented. The Scheduler helps to carry out asynchronous tasks. Other modules and drivers can register a callback function which will be called asynchronously by the Scheduler at specified time. The Scheduler is based on a Timer peripheral with four *compare channels* (CCx). The Scheduler design is kept as simple as possible. There is no priority queue for events. The Scheduler provides four channels, and only one event can be scheduled on a channel at a time. One scheduler channel is used for timing the main loop, which initiates sensors measurements and carries out periodic tasks of modules. The frequency is adjustable, and a warning is given when tasks cannot be made in the specified period of time. Another scheduler channel is occupied by the Line Sensor driver, and last two scheduler channels are used by the Proximity Sensor driver.

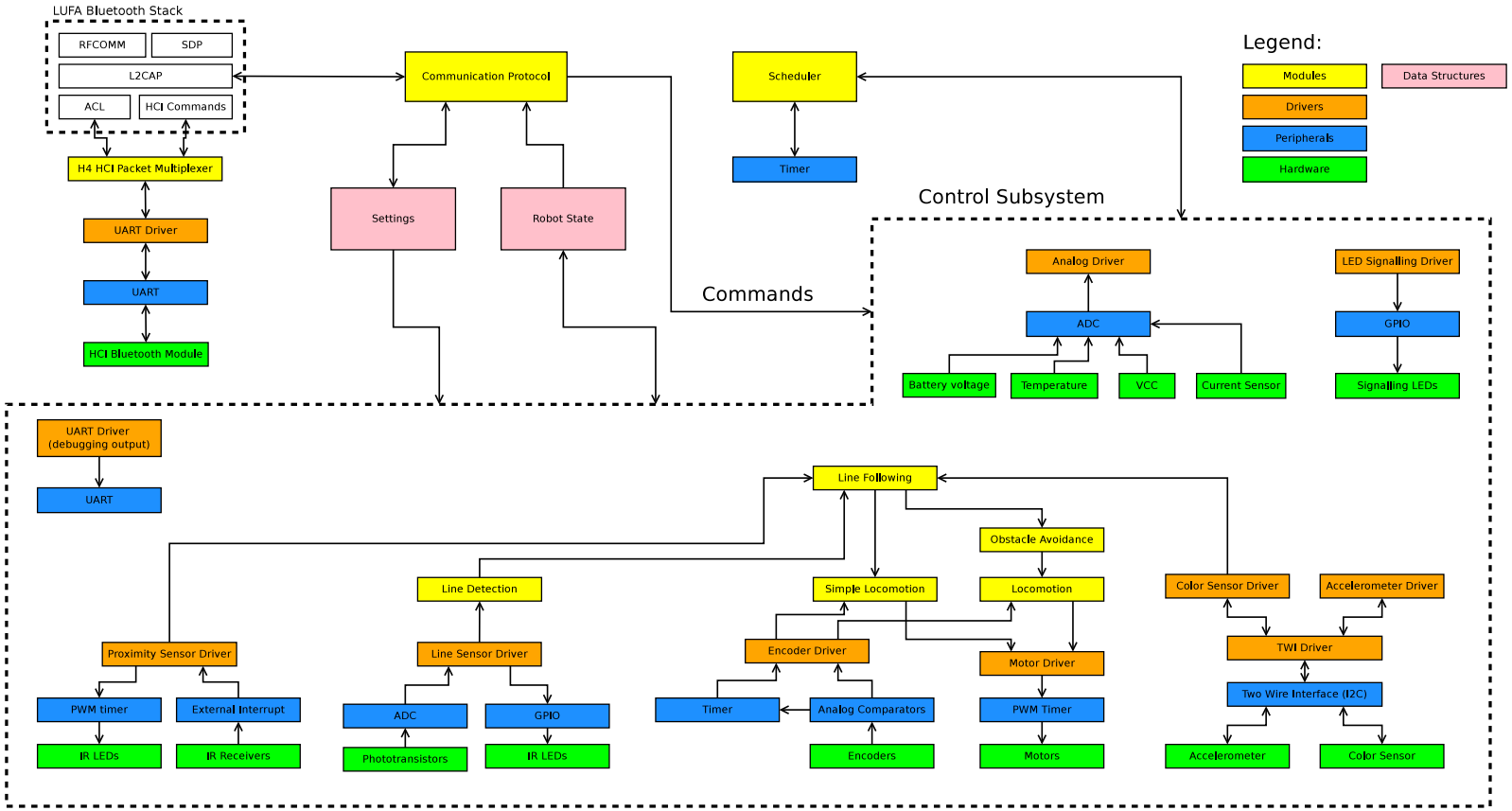


Figure 11.1: Embedded Control System block diagram

The callback functions are called from an interrupt routine, therefore they should return very quickly. Moreover, there are other significant limitations. For instance, it is not possible to call `malloc` within interrupt routines, because this function is not reentrant. However, the Scheduler only calls functions that initiates simple actions, thus, the simplified implementation of a scheduler is not limiting in any way.

11.2 Line Following

The PocketBot2 robot is capable of line following. It follows a black line marked on white surface. The line can be forked or interrupted, there could be an obstacle and finally, several robots may operate on the same guiding line. That is a source of interesting situations and challenging problems related to multi-robot control. For example, algorithms for traffic control can be tested on PocketBot2 line-following robots.

The line-following algorithm is a simple control loop feedback mechanism [19]. Optical sensors measure the light reflectivity of the surface and acquired data is processed by the line detection algorithm. The algorithm is designed in such a manner that line width does not matter. The line detection algorithm outputs signed integer value that states the actual deflection of a guideline (error term $e(t)$ in time t). Values close to zero mean that the line is located accurately in the middle of the sensor module, positive values state how much does the line deflects to the right and negative values state the deflection to the left. This output is then used for controlling the line tracking.

11.2.1 PID controller

This section describes the proportional-integral-derivate (PID) controller [20], which is a universal closed loop feedback algorithm. The PID controller is used in several modules of the Embedded Control System. We will describe how the PID controller is utilized to control line tracking.

The PID controller adjusts the wheels' speed according to the actual line deflection and previous states. Defining $u(t)$ as the controller output (difference between speed of left and right wheel), the PID algorithm is as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t)$$

where K_p (Proportional gain), K_i (Integral gain) and K_d (Derivative

gain) are parameters of the PID regulator. See PID controller overview in Figure 11.2.

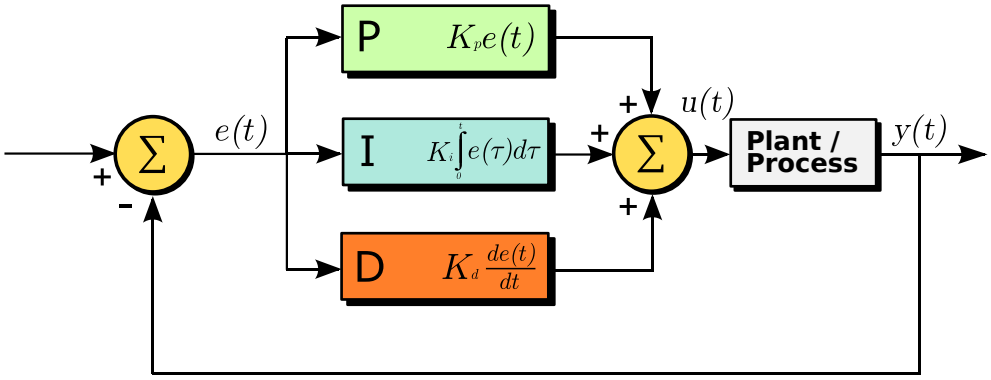


Figure 11.2: PID controller overview

The Proportional term K_p affect wheel speed in a way that the robot turns towards the guiding line. However, at higher speeds, the Proportional regulation is not sufficient, because the system starts to overshoot. Therefore, we set the Derivative term K_d which affects $u(t)$ when the rate of change of error $e(t)$ is considerable. That suppresses overshooting. The Integral part of regulator is not used for line following, so the K_i is set to 0.

In the microcontroller, a discrete version of PID regulator is implemented [21]. We approximate the derivative and integral terms:

$$\frac{d}{dt}e(t) \approx \frac{e(t) - e(t-h)}{h}$$

and

$$\int_0^t e(\tau)d\tau \approx h\sum_{i=0}^{t} e(i)$$

Where $h = 1$ is the time period between discrete samples of error term $e(t)$. The discrete version of PID controller is as follows:

$$u(t) = K_p e(t) + K_i \sum_{i=0}^{t} e(i) + K_d \left(e(t) - e(t-1)\right)$$

Please note that the algorithm does not directly set PWM for the motors, it just calculate appropriate wheel speed. The wheel speed is

then maintained by *another* PID regulator for each wheel separately. See Figure 11.3. This approach brings better line following performance. On the top of it, the performance is not affected with battery state.

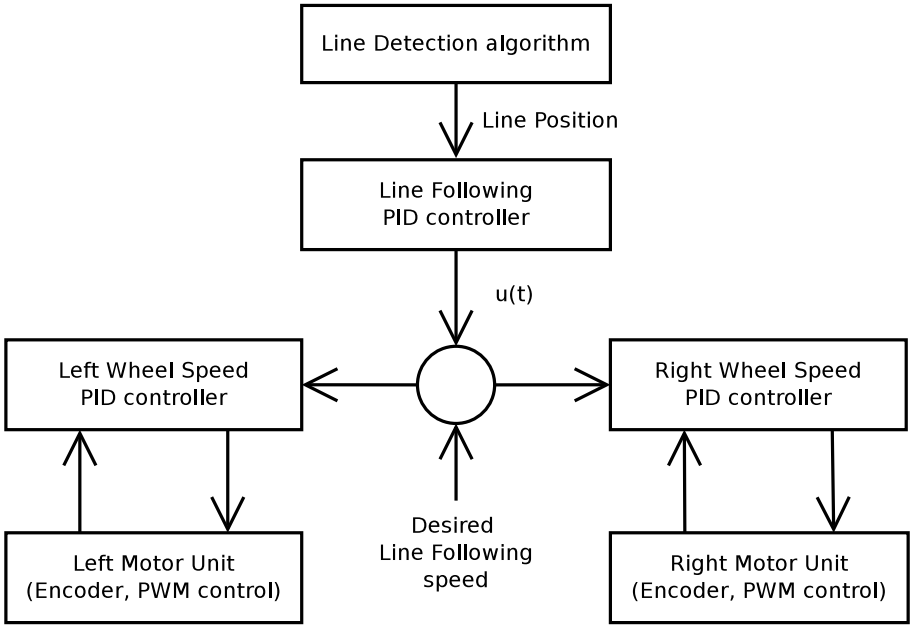


Figure 11.3: Line Following controller overview

To conclude, the PID controller drives the robot so that the line is always centered to the middle of the sensor module, so that the robot performs smooth line following.

11.3 Line Sensors

Line sensors measure the reflexivity of the surface. This information is then evaluated to calculate the line position. The sensor module consists of 4 detectors and 5 emitters. The emitters (infra-red LEDs) are interleaved with detectors (phototransistors), so that each detector is surrounded with two emitters. Thanks to this design it is possible to measure the surface reflectivity on eight spots under the sensor module, using only four phototransistors and five IR LEDs. See Figure 11.4. Generally, this approach reduces the number of components and ADC inputs required for a line sensor module, which is desired with respect to dimension constraints.

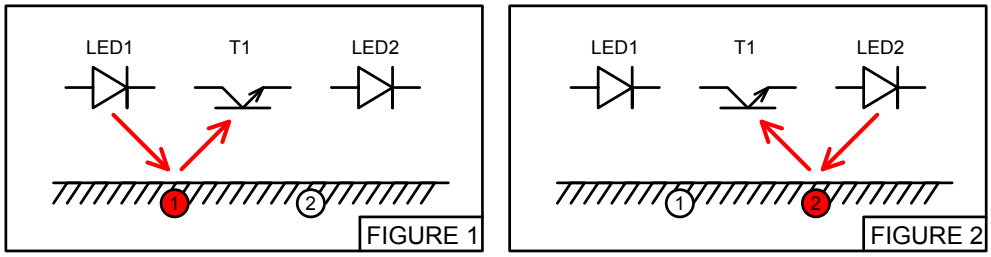


Figure 11.4: Line Sensors - measuring two values using one detector (T1)

11.3.1 Ambient Light Suppression

Light conditions often vary according to time and place, so it is necessary to use ambient light suppression algorithm for the line sensors to work properly. The method is simple: Every sensor does two measurements. At first, it scans for the amount of ambient light. Then, it turns its infrared LED on and measures the value again. Subtracting these two values, the bias of ambient light is suppressed sufficiently for the line following task.

11.4 Locomotion Control

This section describes implementation of a locomotion system for a differential driven mobile robot. The system implements features listed in section 6.5. To recall, the robot should be capable of straight and rotation movements. Moreover, its movement should be smooth, i.e. the speed of the robot should change fluently.

The Locomotion system is designed as two chained PID controllers, as shown in Figure 11.5.

There is a Robot Speed PID Controller, which calculate the PWM duty for motors according to the desired and actual speed of the robot. The calculated PWM duty is then proportionally distributed to left and right motor, based on the output of the Wheel Balance Controller. This controller adjust the ratio of left and right motor power, so that a constant trajectory ratio between left and right wheel is maintained. Obviously, for a straight movement, the desired trajectory ratio is 1:1. For a rotation movement, the ratio is -1:1. And finally, an arbitrary ratio of wheel trajectory yields circular movement of the robot.

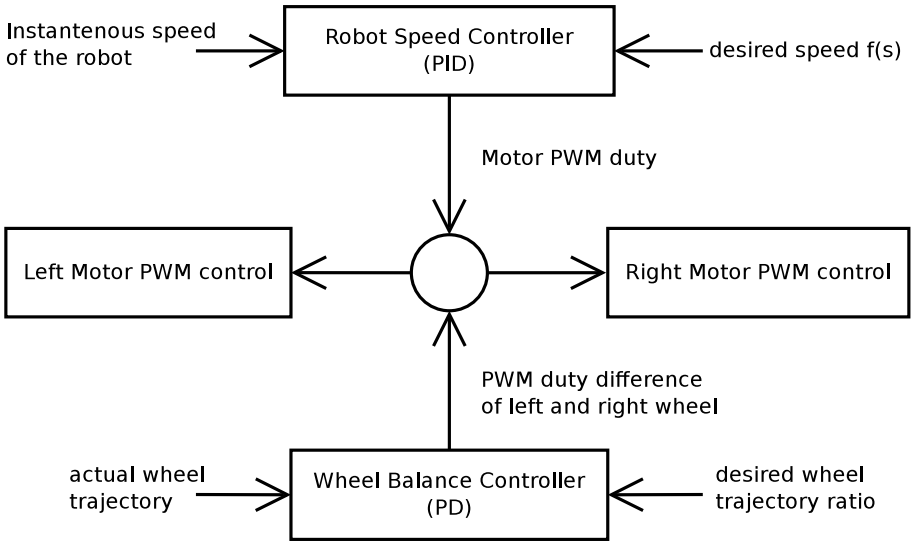


Figure 11.5: Locomotoion system overview

It is important to ensure smooth take-off at the beginning of the movement and smooth breaking. We define a function f of the distance moved and the desired speed at that given point. The f can be any continuous function (yet reasonable for motion control). Regarding the resource limitations of a microcontroller, we implemented the f as a union of three linear functions, see Figure 11.6.

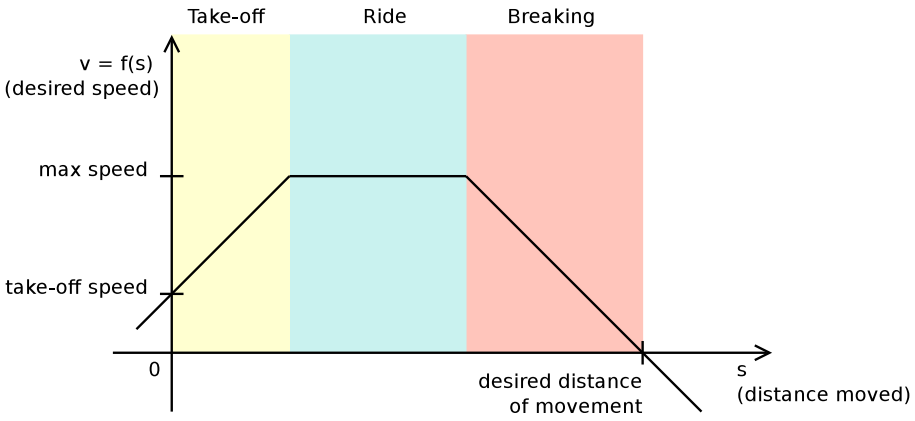


Figure 11.6: The f function

11.5 Speed Measuring

In order to control the speed of motion, the robot must have some means to measure its actual (instantaneous) speed, which is then used as input for the PID regulator. In this section, the basic theory is explained and then confronted with the possibilities of the robot hardware. The common method for measuring speed (The Fixed delta-t Approach) was not suitable for PocketBot2 robot, therefore, dual method (The Fixed delta-s Approach) was used, which is more difficult to implement. The implementation of such method in microcontroller is described as well at the end of this section.

The instantaneous speed v in time T_0 is given by equation:

$$v(T_0) = \lim_{\Delta t \to 0} \frac{s(T_0 + \Delta t) - s(T_0)}{\Delta t} = \frac{ds(T_0)}{dt}$$

Where $s(t)$ is the length of path traveled until time t. In the language of calculus, speed is the first derivative of distance with respect to time.

This relation, however, assumes that both the time and distance are continuous. That is certainly true in real life, but unfortunately, it is not how a robot perceives the world. In fact, the robot can only sense and operate with discrete values. That means both for distance and time, there is some smallest unit the robot can recognize. The minimal unit of distance is inferred from the resolution of wheel encoders, while the minimal time interval depends on the frequency of the microcontroller.

For the PocketBot2 robot, the minimal measurable distance unit $\hat{s}$ is:

$$\hat{s} = \frac{wheelPerimeter}{\#encodersTickPerWheelRevolution} \doteq \frac{29mm}{40} \doteq 0.725mm$$

And the minimal time interval $\widehat{t}$:

$$\hat{t} = \frac{1}{CPUfrequency} = \frac{1}{32MHz} = 31.25ns$$

Facing these facts, it is obvious we cannot accurately measure the instantaneous speed $v(T_0)$, at least in favor of what physicists mean by this term. But we can do a good approximation of the instantaneous speed, and that is what we will discuss bellow.

11.5.1 The Fixed delta-t Approach

By definition, speed is the ratio between length of a path and time interval; $v = \frac{\Delta s}{\Delta t}$. The most frequent implementation in robotics is very

straightforward: To calculate speed, we just count encoders ticks (Δs) during a fixed time period (Δt). However, this approach rises a question that is not easy to answer: How small Δt should I take?

Generally, the smaller the Δt is, the better is the approximation of instantaneous speed. However, for very short Δt the unpleasant discrete characteristic of encoder pulses ($\hat{s}$) manifests itself into the calculated speed, so that the speed turns to show discrete characteristic as well. And that is not a good input to a PID speed regulator.

This problem gains in importance only when the resolution of the wheel encoder is poor, i.e. when the $\hat{s}$ is fairly big and/or when the speed is low. Unfortunately, that is the case of the PocketBot2 robot, so the standard "fixed delta-t" approach does not work well here. The goal would be to find the optimal Δt parameter, so that the calculated speed is a good approximation and still it is sufficiently smooth. My assumptions are that the optimal Δt value does not depend solely on the fixed $\hat{s}$ and $\hat{t}$ parameters, but also on the speed itself, which is of course variable.

Nevertheless, majority of robots have sufficient encoder resolution, so their authors do not have to bother with this question, as they just pick the Δt equal to the period of the PID speed regulator loop.

But for me, this challenging problem of finding optimal Δt drove me to the dual solution of the speed measurement issue.

11.5.2 The Fixed delta-s Approach

Due to problems described above, a different technique of speed calculation is used in PocketBot2. Although the encoder resolution $\hat{s} = 0.725mm$ seems to be comparable to other robots, one has to realize that the tiny size of the PocketBot2 implies lower operating speeds. Consequently, if the "delta-t" approach is used, the Δt parameter need to be chosen very carefully.

The "delta-s" approach is the second side of the same coin. It takes the advantage of the fact that the time resolution $\hat{t} = 31.25ns$ is sufficiently smooth, thus it allows to measure the period of a single encoder pulse ($\hat{s}$) precisely. It means that the instantaneous speed is updated after every encoder pulse, which provides the best approximation of actual speed that the hardware configuration can offer. Interesting to mention, that the same method of speed measurement is used in a bicycle computer. However, it is quite unusual in robotics, as it is more difficult to implement.

There is one drawback of this approach. When the wheel stops, there are no more encoder pulses, thus no more updates of the speed register occur, so the register retains speed value that is no longer valid. This can be easily resolved by setting a timeout that will zero the register automatically. The timeout will only restrict the minimal speed that can be detected.

11.5.3 Implementation

The "fixed delta-s" approach requires measuring the time period of each encoder pulse very precisely. Fortunately, microcontrollers are well equipped for such tasks. The AVR Xmega microcontroller features several 16-bit timers with input capture function.

The essential part of each timer is a special counter register (CNT) that increments by one with each clock pulse. The timer clock is derived from the CPU clock, it is regular, thus the CNT register provides precise time reference. When an event occurs (such as a pulse from the wheel encoder) the input capture function immediately stores the actual value of the counter (CNT) to a corresponding capture register (CCx). This is done by the hardware, without any intervention of the CPU. The CCx register holds the time of event capture, until it is read and processed by an interrupt routine. The interrupt request is generated at the same time the CCx register is written, but generally it is not certain when the interrupt handler code will be executed, as there might be other interrupts pending or interrupts might be even temporarily disabled for a while. That is the reason why to store the CNT value to a temporary CCx register; because the interrupt routine would not manage to read the CNT register in time.

Wheel encoders are not wired directly to the input capture pins of the MCU, because they produce analog signal that has to be processed with an integrated Analog Comparator beforehand. The configurable Event System of AVR Xmega then delivers the event from the Analog Comparator to the timer.

Now, when we can measure the period of encoder pulses precisely, it is easy to calculate instantaneous speed.

The chart in Figure 11.7 shows instantaneous speed calculated after each encoder pulse. During the tests, the wheel was free-runing, and a fixed PWM duty was supplied to the motor.

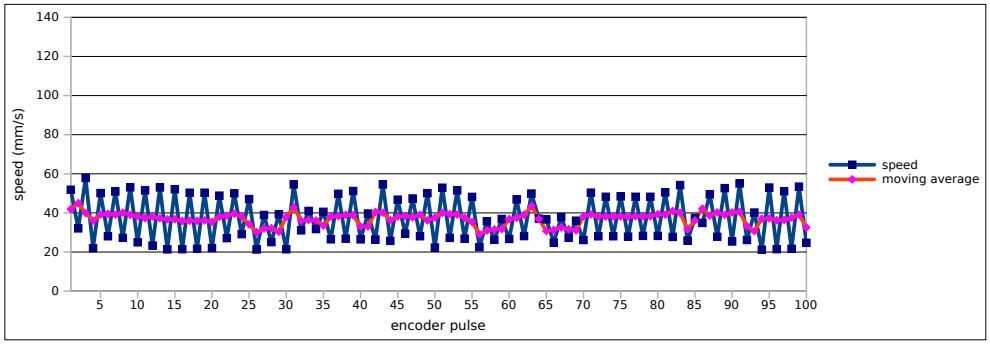


Figure 11.7: Measured speed of PocketBot2 wheel - approx. 40mm/s

From the chart one can clearly see that the calculated speed differs for odd and even pulses. This is caused by the physical construction of the optical encoder. The black stripes of the encoder are narrower than the white ones, which explains the regular changes in calculated speed. See the illustration photo in Figure 11.8. The problem was easily fixed by averaging the latest two measurements. This moving average is plotted in orange.

Figure 11.8: PocketBot2 wheel encoders

Although the rotary encoders in PocketBot2 do not provide very good output in comparison to industrial rotary encoders, thanks to sound implementation of the encoder driver and locomotion control, the PocketBot2 robot shows satisfying performance.

Text Summary

This work described analysis, design and implementation of a software solution for centralized multirobot system.

In the Analysis part, mobile robots were introduced through the concept of sensors and actuators. Various control system architectures were analyzed. A conclusion was made that for a small mobile robot, locomotion control should be carried out in the Embedded Control System, while other computationally extensive tasks should be performed in the host. Then, centralized multi-robot systems were characterized and compared to decentralized robotic systems. Finally, among contemporary wireless technologies, Bluetooth was chosen as an appropriate technology for a centralized multi-robot system.

The Design part drew up guidelines for designing both the Control Library for mobile robots and the robot's Embedded Control System itself. The idea of modules that conform to robot's sensors and actuators was introduced. Moreover, a mechanism of data synchronization between the real robot and its representation in the Control Library was described. This part suggested requirements for a robot locomotion system and described design of the Proximity detection system, which allows recognizing other robots in vicinity.

The Implementation part introduced PocketBot2 robots, which were designed and built by the author of this thesis. Implementation of generic synchronization mechanism between robot and its representation in the Library was described. Issues connected with implementing binary packet communication were investigated and a solution was presented. Afterwards, implementation of several modules specific for a particular robot was shown. An open source Bluetooth stack was ported to the PocketBot2 robot hardware.

Conclusion

In this work, a Centralized Multi-Robot System was developed. It consists of a multi-platform Control Library and an Embedded Control System for PocketBot2 robots. PocketBot2, a tiny robot suitable for testing multi-robot algorithms, was designed and built. Documentation of the robot hardware can be found online at [24].

The Library provides convenient interface for accessing all features of the PocketBot2 robots. It was tested on Windows, Linux and Android. Several example applications were implemented, which demonstrate wide possibilities of the Control Library and the PocketBot Multi-Robot System as a whole. These applications are described in the Appendix of this thesis and demonstration videos are found on the attached CD.

This thesis covers only a part of a broader project carried out by the author. The author has developed the whole PocketBot2 robotic platform from scratch. Therefore, the shape of the robotic platform was entirely liable to author's design decisions. There is a development diary [25] that thoroughly maps the project workflow.

The PocketBot project was awarded several prizes in international competitions: First place in RobotChallenge[1] 2012 (Freestyle Exhibition), Gold medal in InfoMatrix[2] 2011 (Hardware Control Category). The first version of the robot won RobotChallange 2010 and Infomatrix 2010 in the same categories.

[1] international championship for mobile robots, Vienna, Austria

[2] international project competition, Bucharest, Romania

Future Works

Centralized control was implemented and tested on three PocketBot2 robots. My future plan is to design decentralized algorithms, so that robots could perform complex tasks without the supervision of the master. A solid Embedded Control System was implemented in the robot. The Proximity System for short-distance communication and universal Bluetooth Stack provide necessary communication channels for decentralized algorithms.

Decentralized algorithms will require higher processing power, therefore, the 8-bit microcontroller in the PocketBot2 robot will be replaced with a 32-bit ARM microcontroller in the next version of the robot.

Appendix A

Example applications

A.1 Hello robot!

We demonstrate a basic example how to find and connect robot, and how to read sensor data and issue commands. The HelloRobot example application searches for a robot, establishes a connection and starts "square walk" of the robot. While the robot is moving, the application prints its current coordinates and heading angle to standard output. The usage of the library is simple and straightforward, and the example program is self-explanatory, please see comments in code.

```
import cz.ostan.PocketBot.BTlib.*;
import cz.ostan.PocketBot.BTlib.Bindings.BlueCove
   .*;
import cz.ostan.PocketBot.BTlib.packets.*;
import java.io.IOException;
import java.util.List;

class HelloRobot {

    public static void main(String args[]) throws
        IOException {
        Discoverer discoverer = new
           BlueCoveDiscoverer();

        // find all connectable robots
        List<PocketBotDevice> robots = discoverer
           .findRobots();
```

```
if (robots.isEmpty()) {
    System.out.println("No robots found, exiting");
    System.exit(1);
} else {
    System.out.println("Some robots found, taking the first one");
}

// establishes connection to the robot, starts robot state synchronization
PocketBot robot = robots.get(0).Connect();

System.out.println("Robot "+robot.getName()+" connected");

// register eventListener
robot.setEventListener(new PocketBotEventListener() {
    // called everytime the robot state is updated
    public void onNewState(PocketBot robot) {
        // print the robot position (xy coordinates, angle)
        System.out.println(robot.odometry.getPosition());
    }
});

// issues movement commands (square walk)
final double speed = 0.2;
final double length = 0.1;
final double angle = 90;
for (int n = 0; n < 4; n++) {
    robot.locomotion.EnqueMovement(new StraightMovement(length, speed));
    robot.locomotion.EnqueMovement(new RotationMovement(angle, speed));
```

```
        }

        // wait until all movements are finished
        while (!robot.locomotion.
           MovementsFinished());

        // close connection and exit
        robot.getConnection().Close();
    }
}
```

Listing A.1: HelloRobot.java - example of library usage

When compiling the code, path to the library files have to be specified in the *classpath*. Invoking the Java Compiler from command line looks as follows:

```
$ javac -cp "PocketBotBTlib.jar:javolution-5.5.1.
   jar" HelloRobot.java
```

The compiler creates the HelloRobot.class file, which can be then executed:

```
$ java -cp ".:PocketBotBTlib.jar:javolution
   -5.5.1.jar:bluecove-2.1.0.jar:bluecove-gpl
   -2.1.0.jar" HelloRobot
```

Please note that when executing the code, both the library jar files and the directory containing HelloRobot.class has to be specified in the *classpath*.

A.2 Cooperative Object Manipulation

Robots follow a line. An obstacle may appear on their track. One robot does not have sufficient power to push the obstacle away. However, when two robots cooperate, they are able to eject the obstacle out from their track.

This example shows how to coordinate movements of robots. Robots perform line following, until one of them reaches the obstacle. Then, the robot stops and wait for help. When second robot comes to the obstacle, the control program in host issues synchronized movement commands, so that both robots move simultaneously and push the obstacle away from track. Afterwards, robots continue in line following, avoiding collisions with each other.

Please see demonstration video located on the enclosed CD. Slides from the video are presented in Figure A.1 on the next page.

A.3 Multiple Robots on a Track

Three PocketBot2 robots ride on a round track marked with a black tape. Robots avoid collisions with each other. When two robots meet, they identify each other and exchange a message. One robot carries a message token. When it meets another robot, it passes the message to it. The robot which is currently carrying the message token has orange light on. As result, the message token (indicated by orange light) travels around the track. If there is an obstacle in way, robot shall drive around.

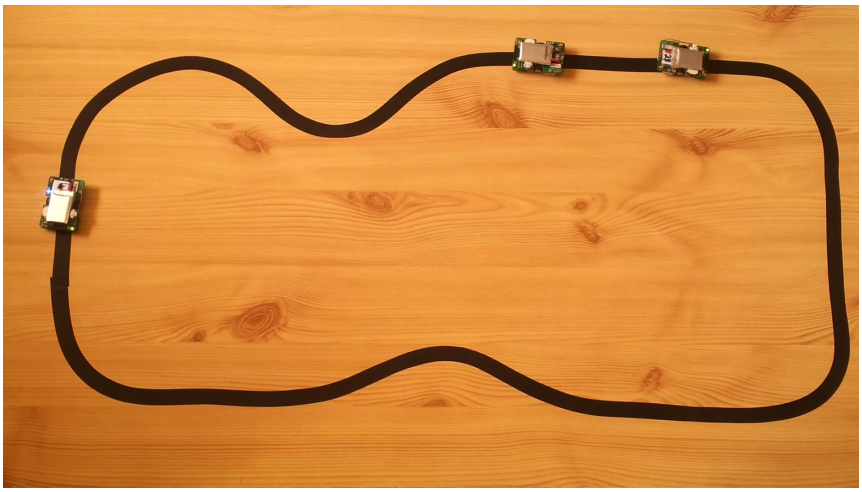


Figure A.2: Multiple Robots on a Track

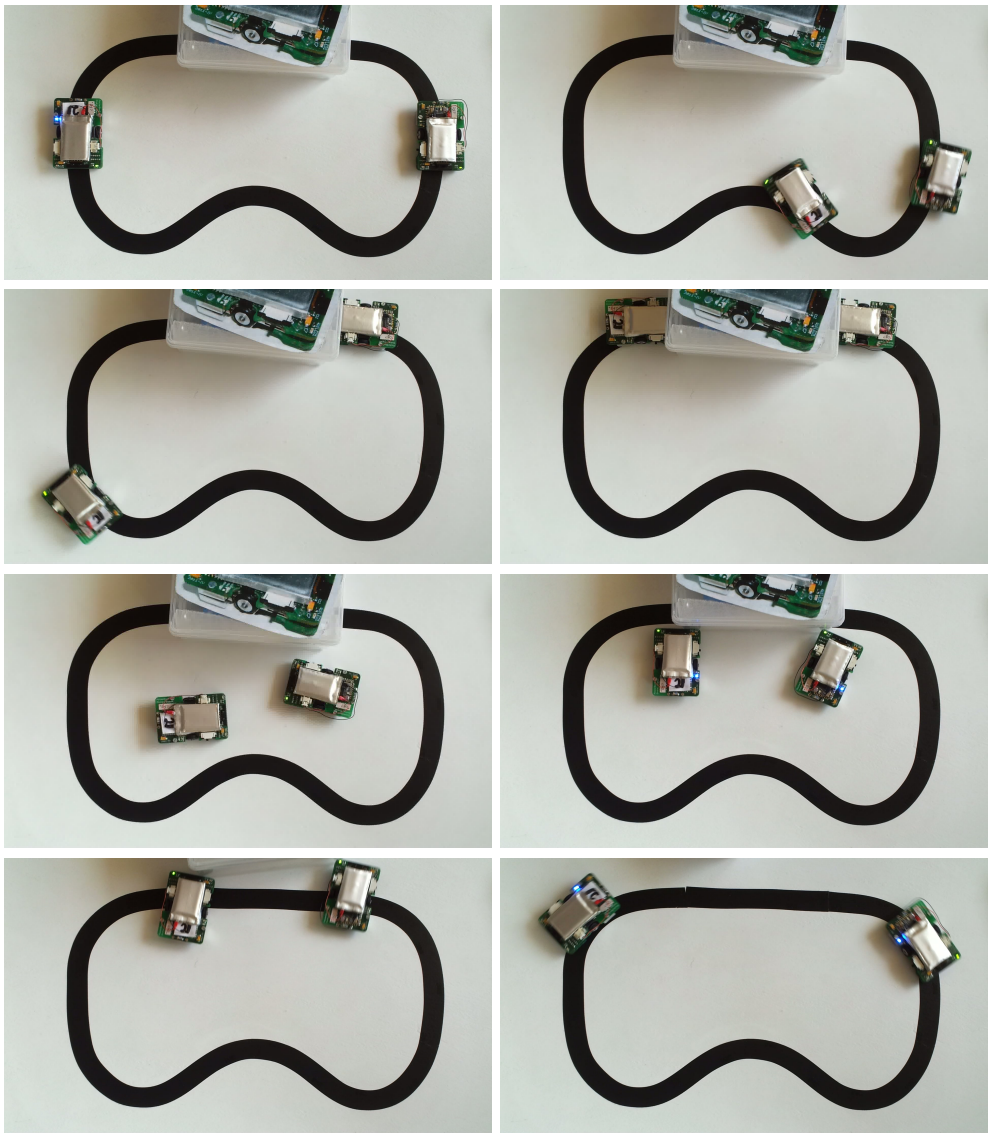


Figure A.1: Cooperative object manipulation

This behavior is implemented in the Embedded Control System of robots, and therefore it is an example of decentralized control. The role of the Control Library is to provide control over the system; the central host can start and stop robots, reverse their direction and adjust the line following speed. However, the system is fully functional without the centralized control. You can find a demonstration video on attached CD.

A.4 Robot Control Panel

PocketBotBTpanel is a general purpose GUI application, which demonstrates all features of the PocketBot2 robot. It displays current state of all robots sensors, controls motors and allows issuing movement commands. Moreover, it is used for adjusting the settings of the robot and tuning PID parameters.

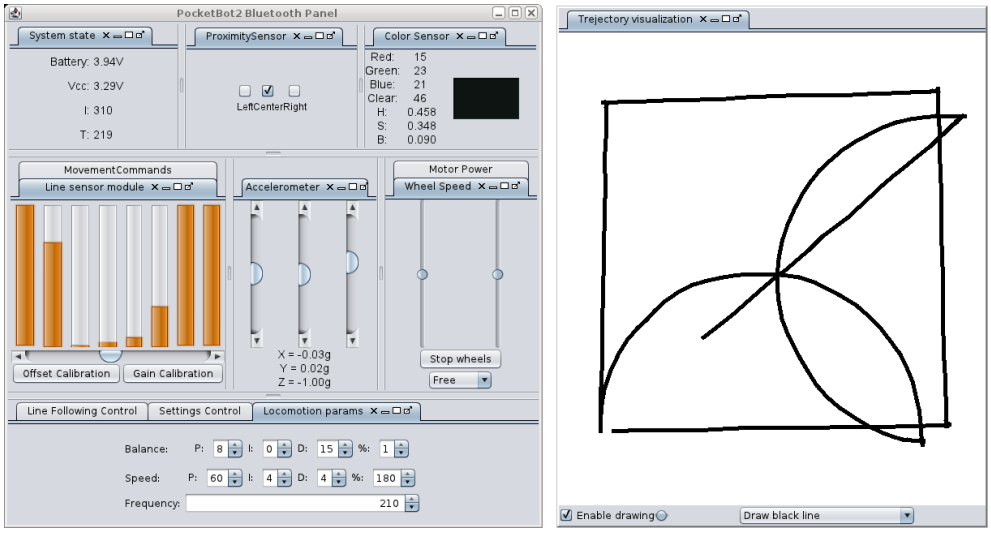


Figure A.3: PocketBotBTpanel - the toolbox for diagnosing robots

The application visualizes the trajectory of the robot. When robot performs movement commands or follows a guiding line, the shape of the track is drawn on screen.

A.5 Control Application for Android Phone

The PocketBot2BTControl application provides well arranged overview of the robot's fundamental features. It shows the state of line sensors and allows sensor module calibration. User may control speed of line following. A colored line following mode is supported. Difficult segments of the track can be marked in red, so the robot knows it had better to slow down. In the same manner, the straight segments can be marked with a blue tape, indicating that the speed can be increased safely.

The touchscreen provides very convenient way how to control robot movement; the application implements a joystick that controls speed of robot's wheels.

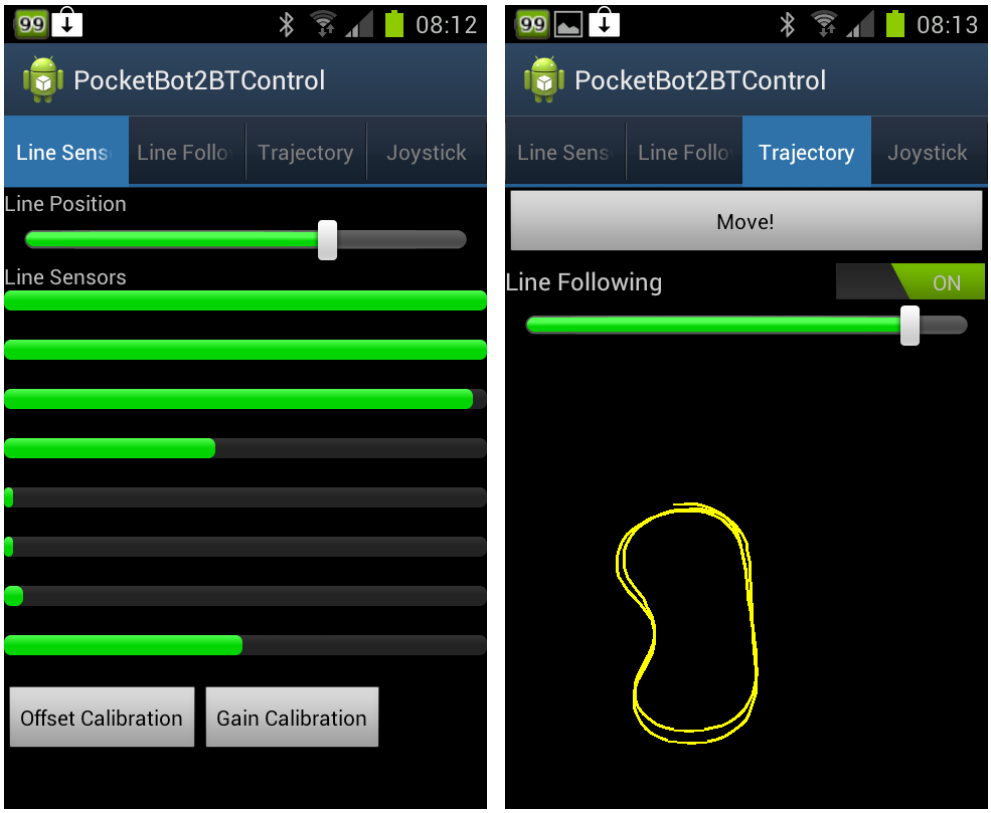


Figure A.4: Control application for Android phones
The screenshot on the left shows visualization of robot's sensor module. Calculated line position is shown as well. In the second picture, there is a trajectory visualization of a robot that follows a line.

Appendix B

Content of CD

The enclosed CD contains documentation, source codes, demonstration videos, development diary, PocketBot2 project web page, program binaries, external libraries and electronic version of this thesis. The CD is organized as follows:

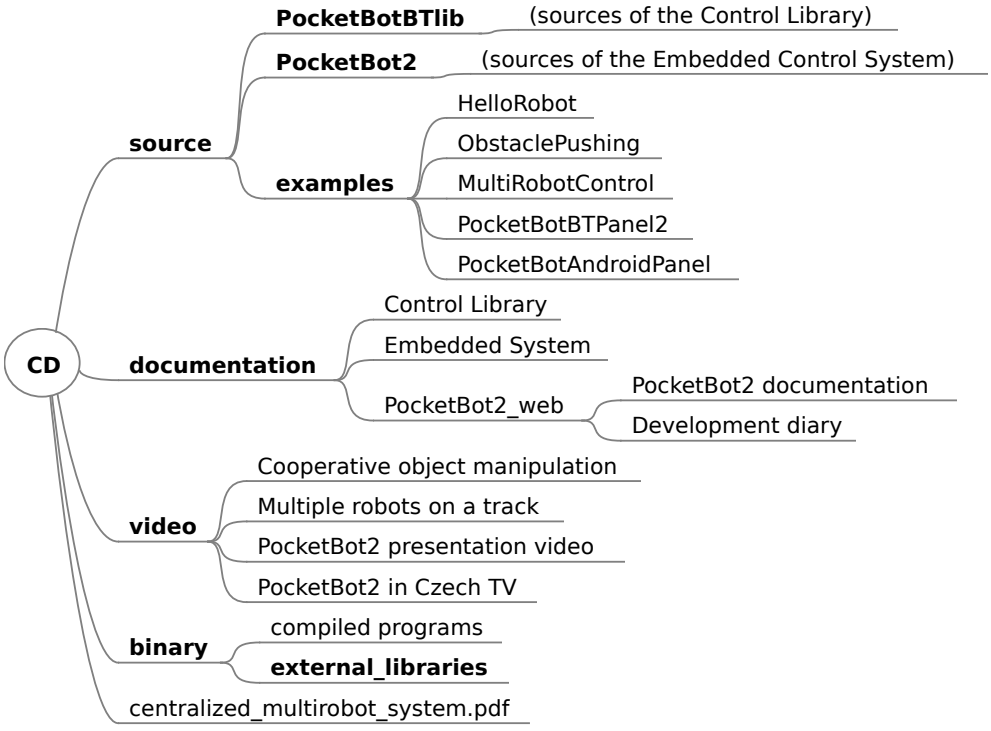


Figure B.1: Content of the enclosed CD

Bibliography

[1] Plátek Ondřej. *Object Oriented Library for Controlling an e-Puck Robot* [online]. 2011
http://code.google.com/p/epuck-lib/

[2] Coupland Simon. *Beginning Navigation* [online]. RobotChallenge, 2011
http://www.robotchallenge.org/fileadmin/user_upload/_temp_/RobotChallenge/Tutorial/BeginningNavigation.pdf

[3] *Steering Techniques* [online]. BEAM Wiki.
http://www.beam-wiki.org/wiki/Steering_Techniques

[4] *List of measuring devices* [online]. Wikipedia.
http://en.wikipedia.org/wiki/List_of_measuring_devices

[5] *Comparison of embedded computer systems on board the Mars rovers* [online]. Wikipedia.
http://en.wikipedia.org/wiki/Comparison_of_embedded_computer_systems_on_board_the_Mars_rovers

[6] Kurt Konolige. *Saphira Robot Control Architecture* [online]. 2002
http://www.cs.jhu.edu/~hager/Public/ICRAtutorial/Konolige-Salphira/saphira.pdf

[7] Toshiyuki Yasuda. *Multi-Robot Systems, Trends and Development* [online]. 2011
http://www.intechopen.com/books/multi-robot-systems-trends-and-development

[8] Xu Ke. *Integrating centralized and decentralized approaches for multi-robot coordination* [online]. 2010
http://mss3.libraries.rutgers.edu/dlr/showfed.php?pid=rutgers-lib:30534

[9] Nohýnková Štěpánka. *Za roboty Na Homolku.* RobotRevue 2010/11. RCR. ISSN 1804-056X

[10] McLurkin James. *Swarm Robots* [online].
http://people.csail.mit.edu/jamesm/swarm.php

[11] Romkey J. *A Nonstandard For Transmission of IP Datagrams over Serial Lines: SLIP* [online]. RFC 1055. 1988.
http://tools.ietf.org/html/rfc1055

[12] Camera Dean. *LUFA (2012)* [online].
http://www.fourwalledcubicle.com/LUFA.php

[13] Dunkels A. *lwIP - a lightweight TCP/IP stack* [online].
http://read.pudn.com/downloads141/sourcecode/windows/610246/lwbt/doc/lwbt-doc.pdf

[14] Javolution
http://javolution.org/target/site/apidocs/javolution/io/Struct.html

[15] BlueCove
http://bluecove.org/

[16] *Specification of the Bluetooth System* [online]. Bluetooth SIG. Nov 4 2004.
https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=40560

[17] Shokry Mina. Bluecove-developers discussion group. Aug 14 2009.
http://groups.google.com/group/bluecove-developers/msg/847f714034b34126

[18] *Bluetooth stack* [online]. Wikipedia.
http://en.wikipedia.org/wiki/Bluetooth_stack

[19] Nise S. N. *Control Systems Engineering.* Willey. Fourth Edition. 2004 ISBN 978-0-471-44577-7

[20] *PID controller* [online]. Wikipedia.
http://en.wikipedia.org/wiki/PID_controller

[21] Tham M. *Discretised PID Controllers* [online].
http://lorien.ncl.ac.uk/ming/digicont/digimath/dpid1.htm

[22] Bukovan Michal. *Řízení modelů lokomotiv přes rozhraní Bluetooth* [online]. 2007.
https://dip.felk.cvut.cz/browse/pdfcache/bukovm1_2007bach.pdf

[23] *JSR-000082 Java APIs for Bluetooth* [online]. Motorola. April 5, 2002
http://jcp.org/aboutJava/communityprocess/final/jsr082/

[24] Staněk Ondřej. *PocketBot2: a matchbox-sized robot* [online]. 2011.
http://ostan.cz/PocketBot2/

[25] Staněk Ondřej. *PocketBot 2 - deníček vývojáře* [online]. 2010
http://ostan.cz/PocketBot2/development_diary/index.html