Experiential Learning of Robotics

This book is designed for beginners to introduce them to the field of robotics. The content is based on the integration of electronics, computing, and mechanics.

dr. David Rihtaršič

Contents

Programing Exapmles

1 INTRODUCTION AND PREPARATION

Welcome to the educational robotics lecture using Arduino, Robduino module, and Fischertechnik parts! In this lectures, we will learn how to use these tools and materials to build and program simple robots for educational and recreational purposes.

First, we will introduce the Arduino controller and the Robduino module, and discuss their capabilities and limitations. We will also cover the basics of the Arduino programming language, including variables, functions, and control structures.

Next, we will discuss the Fischertechnik parts and how they can be used to construct robots with various shapes, sizes, and capabilities. We will cover the different types of parts that are available, such as beams, gears, motors, and sensors, and how they can be combined to create a wide range of structures and mechanisms.

We will then demonstrate how to use the Arduino controller and Robduino shield to program and control Fischertechnik robots. We will cover topics such as sensor input, actuator output, and feedback control.

Throughout this lecture, we will use hands-on activities and examples to illustrate the concepts and techniques that are covered. We will also discuss some of the challenges and considerations that are involved in building and programming robots with these tools and materials.

1.1 Introduction to embedded systems

Embedded systems are specialized computing devices engineered to perform dedicated functions within larger systems. Unlike general-purpose computers, these devices combine hardware—typically microprocessors or microcontrollers—with software (commonly called firmware) to execute specific tasks reliably and efficiently. Their design is tailored to meet the precise demands of applications ranging from consumer electronics to industrial control systems.

Embedded systems are specialized computing devices designed for dedicated functions, ubiquitous in our daily lives and production processes (Cai, 2023; Qian et al., 2009). These systems consist of hardware, including microprocessors or microcontrollers, and software, often referred to as firmware (Qian et al., 2009). Embedded systems are characterized by their application-specific nature, reliance

on computer technology, and customizable software and hardware (Cai, 2023). They are integral components of larger systems, such as robots, unmanned vehicles, and aerospace electronics (Sadiku et al., 2017). Embedded systems can be classified into two types: visible and invisible to users (Barkalov et al., 2019). The embedded systems market is rapidly growing, particularly in wireless and web applications (Qian et al., 2009). As technology advances, embedded systems continue to evolve, with emerging trends shaping their future development and wider adoption across various domains (Cai, 2023). Understanding embedded systems is crucial for leveraging their capabilities in an increasingly interconnected and technologically empowered future.

In the context of electronics projects, the term “embedded systems” is frequently mentioned because these systems are closely intertwined with the work of electronics. Many electronics projects inherently require the integration of both hardware circuitry and computational intelligence. For instance, when designing a sensor module or an actuator controller, engineers not only need to consider the physical electronic components but also how to process data and execute control algorithms. Embedded systems provide that bridge between raw electronics and smart functionality, making them an essential aspect of modern electronics projects. This overlap ensures that innovations in electronics are increasingly empowered by embedded computing, leading to more robust, efficient, and intelligent systems.

By including embedded systems in our discussion, we acknowledge their pivotal role in the evolution of electronics projects. Their inherent synergy fosters a collaborative approach where hardware design and software development merge, enabling the creation of sophisticated systems that address real-world challenges effectively.

1.2 Overview of robotics and its applications

Robotics is an interdisciplinary field that integrates various engineering disciplines to design, develop, and deploy machines that assist and enhance human capabilities. This field marries principles from mechanics, electronics, control systems, sensors, and artificial intelligence, resulting in systems that are not only efficient but also highly specialized for particular tasks (Minnich Alois, 2021; Yang & Hu, 2023).

1.2.1 Defining Robotics

At its core, robotics is concerned with creating machines that can perform tasks traditionally carried out by humans. These machines range from simple automated devices to complex, self-monitoring

robots capable of executing intricate algorithms. The fusion of hardware and software in robotics enables these systems to interact with and adapt to their environment, making them indispensable tools in modern engineering and industry.

1.2.2 Core Technologies in Robotics

Robotics draws upon several key technological areas:

- **Mechanics:** This involves the design and movement of physical components. Engineers focus on aspects like dynamics, kinematics, and material properties to ensure that robotic systems can perform tasks with precision and reliability.
- **Electronics:** The electronics in robotics include microprocessors, microcontrollers, and sensor circuits that are essential for data acquisition, processing, and actuation.
- **Control Systems:** These systems use feedback loops and algorithms to maintain and adjust the performance of the robot. They are critical in ensuring that a robot responds accurately to changes in its environment.
- **Sensors and Actuators:** Sensors provide real-time data about the environment, which the robot uses to make decisions. Actuators, in turn, convert the control signals into physical action.
- **Artificial Intelligence (AI):** AI enhances robotics by enabling adaptive behavior, learning, and decision-making, thereby pushing the boundaries of what automated systems can achieve.

Key robotics technologies such as gesture control, machine vision, voice recognition, and touch sensor technology further exemplify the integration of these disciplines, offering advanced methods for human-machine interaction and autonomous operation (Javaid et al., 2022).

1.2.3 Diverse Applications of Robotics

Robotics has penetrated virtually every sector of modern life. One of the most transformative applications is in healthcare. In this domain, robotics has revolutionized patient care by:

- **Enhancing Surgical Precision:** Robotic systems allow surgeons to perform minimally invasive procedures with heightened precision, reducing recovery times and improving outcomes.
- **Medicine Delivery and Hygiene:** Especially highlighted during the COVID-19 pandemic, robots have been deployed to deliver medications and maintain hygiene in environments where human contact needed to be minimized (Javaid et al., 2022).

Beyond healthcare, robotics is making significant inroads in areas such as:

- **Manufacturing and Industrial Automation:** Robots are used extensively on assembly lines, performing tasks that require consistency, speed, and accuracy.
- **Logistics and Supply Chain:** Autonomous robots facilitate warehousing, sorting, and transportation tasks, improving efficiency and reducing labor costs.
- **Service Industries:** From household cleaning robots to customer service kiosks, the applications are both diverse and rapidly evolving.
- **Agriculture and Environmental Monitoring:** Robotics supports precision farming, environmental sensing, and data collection, which are essential for sustainable practices.
- **Aerospace and Defense:** In aerospace, robotics contributes to both unmanned aerial vehicles and space exploration, where reliability and robustness are paramount.

The rapid evolution of robotics is also driven by the decreasing installation and maintenance costs, making these technologies accessible to a broader range of applications and industries (Javaid et al., 2022).

1.2.4 The Intersection with Electronics Projects

Robotics and electronics are deeply intertwined. The field of electronics forms the backbone of robotic systems, with embedded systems serving as a critical link between raw electronic components and the sophisticated control algorithms that drive intelligent behavior. Projects in electronics often extend naturally into robotics, as both fields share a common foundation in circuitry, sensor integration, and software-driven control. This overlap reinforces the importance of an interdisciplinary approach where innovations in electronics propel advancements in robotics and vice versa. Such integration ensures that modern systems are not only technically robust but also capable of meeting the increasingly complex demands of real-world applications.

1.2.5 Future Trends and Societal Impact

As robotics continues to evolve, its influence on society grows ever more profound. The development of self-monitoring robots—capable of executing complex algorithms and adapting autonomously—signals a future where robotic systems will play an integral role in everyday life. Advances in AI and machine learning are set to further enhance these systems, making them smarter, more intuitive, and more capable of complex decision-making (K. S & S. G, 2023).

The implications of these advancements are far-reaching:

- **Economic Impact:** Lower installation and maintenance costs will drive wider adoption across industries, potentially reshaping labor markets and production processes.
- **Social Integration:** As robots become more capable and versatile, they are poised to become regular collaborators in human tasks, from healthcare to household chores.
- **Ethical and Regulatory Considerations:** With greater autonomy comes the need for thoughtful regulation and ethical frameworks to guide the deployment of robotic systems in society.

1.2.6 Conclusion

In summary, robotics is a dynamic and interdisciplinary field that plays a pivotal role in the advancement of modern technology. Its integration of mechanics, electronics, control systems, sensors, and AI enables the creation of machines that significantly enhance human capabilities and transform various sectors, from healthcare and manufacturing to logistics and beyond. The inherent synergy between robotics and electronics not only drives technological innovation but also ensures that future developments in robotics will continue to shape and improve the fabric of our daily lives.

Understanding the broad scope and potential of robotics is essential for anyone engaged in the fields of engineering and technology, as it represents a cornerstone of modern and future innovations.

1.3 Basic concepts and terminology

Robotics is a rapidly growing field of technology that has the potential to revolutionize many areas of our lives. It involves the development of machines that can imitate or surpass human capabilities in performing a variety of tasks. Robotics is an interdisciplinary field of science, engineering, and technology that deals with the design, construction, operation, and application of robots.

Robotics is a complex field involving both hardware and software components. Hardware components include physical robotic parts such as motors, sensors, and actuators, while software components include algorithms and programming languages used to control the robot and its functions. Robotics also requires an understanding of various disciplines including mathematics, physics, mechanics, and computer science.

At its core, robotics is all about autonomy. Autonomy is the process of designing a robot to perform and complete specific tasks, such as carrying out a surgical procedure or assembling a car. Automation can help reduce costs, increase productivity, and improve the safety of both workers and products.

When it comes to terminology, there are a few key terms used in robotics. Robot is a machine that is capable of performing tasks on its own or under the control of a computer program. Robotics is the science and technology of robots and their design, construction, operation, and application. Sensors measure and detect environmental conditions, such as temperature, pressure, or light. Actuators convert electrical signals into mechanical motion. Computer vision is the ability of robots to interpret visual information from cameras. AI, or artificial intelligence, is used to give robots the ability to learn and think for themselves.

Robotics is an exciting field with many potential applications. Taking the time to become familiar with the basic concepts and terminology can help you better understand and apply robotics in practical situations.

1.3.1 What is robotics

- Science of robots. :)
- What is a robot?
- How does the robot works?
- How are robots constructed?
- What is intended task of the robot?
- How do we control a robot?

1.3.2 What is a robot?

- automated (coffee) machine
- …
- Printer
- 3D printer
- CNC machine
- …
- “Robot” Vacuum cleaner (a.k.a. Roomba)
- Industrial robot arm (YASKAWA)
- Humanoid robot

It is not defined by the definition… but we have to describe it.

1.3.3 International Organization for Standardization - ISO

- Standards are not excluding each other…
- `ISO 2806` - defining the CNC machines
  - describing the processing technology
- `ISO 8373` - defining the robots
  - describing machine autonomy

1.3.3.1 ISO 8373 - General Terms in Robotics

ROBOTICS science and practice of designing, manufacturing, and applying robots (2.6)

ROBOT actuated mechanism programmable in two or more axes (4.3) with a degree of autonomy (2.2), moving within its environment, to perform intended tasks

- Note 1 to entry: A robot includes the control system (2.7) and interface of the control system.
- Note 2 to entry: The classification of robot into industrial robot (2.9) or service robot (2.10) is done according to its intended application.

REPROGRAMMABLE designed so that the programmed motions or auxiliary functions can be changed without physical alteration (2.3)

AUTONOMY ability to perform intended tasks based on current state and sensing, without human intervention

MANIPULATOR machine in which the mechanism usually consists of a series of segments, jointed or sliding relative to one another, for the purpose of grasping and/or moving objects (pieces or tools) usually in several degrees of freedom (4.4)

- Note 1 to entry: A manipulator can be controlled by an operator (2.17), a programmable electronic controller, or any logic system (for example cam device, wired).
- Note 2 to entry: A manipulator does not include an end effector (3.11).

CONTROL SYSTEM set of logic control and power functions which allows monitoring and control of the mechanical structure of the robot (2.6) and communication with the environment (equipment and users)

ROBOTIC DEVICE actuated mechanism fulfilling the characteristics of an industrial robot (2.9) or a service robot (2.10), but lacking either the number of programmable axes (4.3) or the degree of autonomy (2.2) EXAMPLE:Power assist device; teleoperated device; two-axis industrial manipulator (2.1)

INDUSTRIAL ROBOT automatically controlled, reprogrammable (2.4), multipurpose (2.5)manipulator (2.1), programmable in three or more axes (4.3), which can be either fixed in place or mobile for use in industrial automation applications Note 1 to entry: The industrial robot includes: — the manipulator, including actuators (3.1); — the controller, including teach pendant (5.8) and any communication interface (hardware and software). Note 2 to entry: This includes any integrated additional axes.

SERVICE ROBOT robot (2.6) that performs useful tasks for humans or equipment excluding industrial automation applications Note 1 to entry: Industrial automation applications include, but are not limited to, manufacturing, inspection, packaging, and assembly. Note 2 to entry: While articulated robots (3.15.5) used in production lines are industrial robots (2.9), similar articulated robots used for serving food are service robots (2.10).

MOBILE ROBOT robot (2.6) able to travel under its own control Note 1 to entry: A mobile robot can be a mobile platform (3.18) with or without manipulators (2.1).

ROBOT COOPERATION information and action exchanges between multiple robots (2.6) to ensure that their motions work effectively together to accomplish the task

INTELLIGENT ROBOT robot (2.6) capable of performing tasks by sensing its environment and/or interacting with external sources and adapting its behaviour EXAMPLE:Industrial robot (2.9) with vision sensor to pick and place an object; mobile robot (2.13) with collision avoidance; legged robot (3.16.2) walking over uneven terrain.

1.2 History of robotics

Robotics technology has evolved rapidly in the last few decades, leading to a vast array of possibilities for what can be achieved. From manufacturing robots to autonomous vehicles and medical robots, robots are becoming increasingly advanced and capable of performing more complex tasks.

The potential applications for robotics technology are endless, and robotics is set to revolutionize the way we live and work in the future. From healthcare to transportation, robotics is transforming the way we interact with our environment and making life easier, safer, and more efficient.

With the advancement of robotics, we stand at the brink of a new era of technology, one that promises to completely revolutionize the way we live. The future of robotics is an exciting one, and it will be fascinating to see what the next few decades have in store.

1.3 Teaching robotics and robotics in teaching

Robotics in education is an exciting field that has the potential to revolutionize the way our children learn. By introducing robots into the classroom, educators can provide students with engaging, hands-on learning experiences that stimulate their curiosity, creativity, and problem-solving skills. Robotics offers a unique opportunity to develop 21st century skills such as collaboration, communication, critical thinking, and creativity. It allows students to learn in a safe environment with no risk of failure, and fosters an environment of experimentation and exploration.

Robotics can also be used to enhance subject-matter learning, enabling students to write code and program robots to solve problems. This opens up possibilities for developing skills such as design thinking, algorithmic thinking, and computational thinking. Robotics also has potential to promote STEM education, as students can learn about topics such as engineering and computer science through the use of robots.

In addition, robotics can help to develop social and emotional skills. Through the use of robots, students can learn to collaborate, work in teams, and develop leadership skills. Robotics also encourages students to develop empathy and to think critically about the world around them.

Overall, robotics in education is an important tool for preparing students for the future. By introducing robots into the classroom, educators can create engaging and interactive learning experiences that teach students valuable skills. Robotics can also be used to enhance subject-matter learning, promote STEM education, and develop social and emotional skills."

1.3.1 Robotics and Education

Robotics in education has been gaining a great deal of attention in recent years. This is due to its potential to create engaging learning experiences that help to facilitate deeper understanding of complex topics. Robotics provides an opportunity to engage in hands-on learning that encourages students to explore, tinker and construct their own learning. This approach aligns with both constructivism and constructionism, two educational theories that emphasize the need for students to build their own knowledge and understanding through exploration and collaboration.

In this context, robotics acts as a conduit for students to explore and understand the world around them. The work of Seymour Papert, a renowned MIT professor, has been influential in this field. Papert was an early advocate for the use of robotics in education, and his work led to the development of the popular children's robotic toy, the Logo Turtle. Papert recognized the potential of robotics to engage students and foster meaningful learning experiences.

Similarly, the work of Resnick at the MIT Media Lab was influential in the development of innovative robotic programming tools such as Scratch and LEGO Mindstorms. These tools have become popular

in teaching children robotics and programming. By providing children with the ability to control and program robots, these tools provide a powerful means for students to explore the possibilities of robotics and to develop a deeper understanding of its principles.

Overall, robotics in education offers an exciting opportunity to foster meaningful and engaging learning experiences. Through robotics, students have the opportunity to explore the world around them, to tinker and construct their own learning, and to develop a deeper understanding of complex topics."

1.3.1.1 Definition of the robots in education

Slangen:
Definition of the robot must be based on the main operation that robot performs:

- zaznavanje (angl. Sensing),
- sklepanje (angl. Reasoning) &
- delovanje (angl. Acting).

This operation is constantly executing in a.k.a. S-R-A loop.

Slo. nat. curriculum:`Robotics in Engineering`
- almost exact interpretation of S-R-A loop Krmiljenje s povratnim delovanje (angl. feedback control regulation)

- including learning objective: . . . kjer učenci ugotovijo potrebe po **krmiljenju s povratnim delovanjem** in izpostavijo pomanjkljivosti, če takega krmiljenja ni.

(angl. where students identify the need for **feedback control** and point out shortcomings in the absence of such control)

- misconception: Playing with robots or using a robot is robotics.
- Robots are meant to be user friendly.

1.3.1.2 Robotics in Schools

- very popular in last decade

We can find robots in learning process as:

1. Robotics curses:

- Electronics
- Computer Science

- Engineering

2. motivation for learning other disciplines:

- Science
- Technology
- Engineering
- Math

1.3.1.3 Important educational impacts

1.3.1.3.1 LEARNING by DOING … learning as "BUILDING KNOWLEDGE STRUCTURES" through progressive internalization of actions… this HAPPENS especially felicitously in a context where the LEARNER is consciously engaged in CONSTRUCTING A PUBLIC ENTITY, whether it's a sand castle on the beach or a theory of the universe. (Papert, S. (1980). Mindstorms. Children, Computers and Powerful Ideas. New York: Basic books.)

1.3.1.3.2 PRACTICAL APPLICATIONS Applying knowledge and skills learned into a `public entity` make us proud of ourself. We have something to show to people that matters to us (friends, parents, classmates).

1.3.1.3.3 CREATIVITY There is not an only one solution to the problem. Kids can explore their ideas and put it to the test.

1.3.1.3.4 LEARNING from MISTAKES Kids are ALLOWED to LEARN from MISTAKES!?! In general, MISTAKES has very bad reputation in school sistem. To degree, that kids are often afraid to give an answer so as not to make a mistake (-> they stop trying). However, Robotics is so complicated field that mistakes can not be avoided. Thus, MISTAKES are very common thing in this learning proces of robotics.

1.3.1.3.5 CRITICAL THINKING Critical thinking is ability to do analysis of facts and form objective judgments based on reasonable arguments.

1.3.1.3.6 SELF-ASSESSMENT Kids are able to see if they fulfill the intended task or not. They can asses their own performance based on results of intended tasks.

1.4 Current state of the field

Educational robotics is a rapidly growing field that combines elements of education, technology, and robotics. It focuses on using robots as a tool to enhance learning and provide hands-on experiences for students. The field has gained significant attention in recent years due to the increasing interest in STEM education and the need to develop 21st-century skills.

One of the main objectives of educational robotics is to promote critical thinking, problem-solving, and collaboration among students. By engaging in robotics activities, students can develop a range of skills, including coding, engineering, creativity, and logical reasoning. Moreover, robotics can be integrated into various subjects, such as science, mathematics, and computer science, enabling interdisciplinary learning.

Educational robotics encompasses a wide range of approaches and technologies. For younger students, simple robots like Bee-Bots or Cubetto are often used to introduce basic programming concepts through hands-on activities. As students progress, more complex robots, like LEGO Mindstorms or VEX Robotics kits, offer opportunities for advanced programming and engineering challenges.

In addition to the physical robots, virtual robotics platforms have also gained popularity. These platforms allow students to simulate robot programming and control without the need for physical robots. Virtual robotics offers a cost-effective and accessible way to introduce robotics concepts in classrooms with limited resources.

Several educational robotics competitions and programs have emerged globally, encouraging students to apply their skills in real-world challenges. Examples include FIRST Robotics Competition, RoboCup, and VEX Robotics Competitions. These events provide a platform for students to showcase their robot designs, programming abilities, and teamwork.

The field of educational robotics is continuously evolving. Researchers and educators are exploring innovative ways to integrate robotics into curricula, create engaging learning environments, and develop effective pedagogical approaches. Factors such as artificial intelligence, machine learning, and human-robot interaction are also being explored to enhance the capabilities and functionalities of educational robots.

The future of educational robotics looks promising as it continues to inspire and engage students in STEM education. By providing hands-on experiences with robots, this field aims to prepare students for the digital age, where robotics and automation play an increasingly vital role.

1.5 Robotics Equipment

Fischertechnik and LEGO are both brands of construction toy systems that allow users to build and create a wide range of structures and mechanisms. Both systems use a modular approach, with a variety of interlocking parts that can be easily snapped together.

However, there are some key differences between Fischertechnik and LEGO parts:

Material: Fischertechnik parts are made of a durable, high-quality plastic called polycarbonate, which is known for its strength and resistance to wear and tear. LEGO parts are made of a softer plastic called acrylonitrile butadiene styrene (ABS), which is more flexible and less durable.

Precision: Fischertechnik parts are designed with high precision and tolerances, which allows for more accurate and stable constructions. LEGO parts have slightly looser tolerances, which can make them more prone to wobbling or sagging.

Size and shape: Fischertechnik parts are generally smaller and more compact than LEGO parts, which allows for more detailed and precise constructions. LEGO parts are larger and more blocky, which makes them more suitable for building larger structures.

Functionality: Fischertechnik parts are designed with a focus on mechanical and electrical functionality, and include a wide range of components such as gears, motors, and sensors. LEGO parts are more geared towards aesthetics and playability, and include elements such as minifigures and decorative elements.

Price: Fischertechnik parts tend to be more expensive than LEGO parts, due to their higher quality and greater functionality.

Overall, Fischertechnik and LEGO are both excellent construction toy systems, and the choice between them will depend on the specific needs and preferences of the user.

We can divide the equipment for robotics into three different groups: 1. Electronics, 2. Computer science, 3. Engineering.

In our couse we will be using next basic parts:

1.5.1 ELECTRONICS

- WIRES
  - 4x 15cm
  - 4x 10cm
- CONNECTORS

  - 8x 2.5mm FT
  - screw driver

- RESISTORS
  - 2x 330 Ohms
  - 2x 3.3k Ohms
  - 2x 33k Ohms
  - 2x 330k Ohms
  - 10k Ohms potenciometer (with wires)

- NON-LINEAR RESISTORS AND SENSORS
  - 1x phototranzistor FT & aperature
  - 1x reed switch
  - 1x push-button FT
  - IR distance sensor

- ACTUATORS
  - light bulb
  - 2x DC motor FT
  - 1x servo-motor
  - 1x servo attach
  - LCD (i2c)

1.5.2 COMPUTER SCIENCE

- Arduino UNO controller
- modul RobDuino-v2 (shield for robotics)
- Arduino UNO adapter -> FisherTechnik (3D print)
- USB kabel
- battery charger for 2x18650 LiIon battery
- 2x 18650 LiIon battery's
- 9V Power Supply

1.5.3 MECHANICAL ENGINEERING

1.5.3.1 CONSTRUCTION ELEMENTS

- 12x square block 15x15x30mm

- 6x square block 15x15x15mm
- 2x square block 7.5x15x30mm
- 5x square block 7.5x15x15mm
- 3x “L” profile 15x15x45mm
- 2x “L” profile 15x15x30mm
- 4x rim R1” fiksno
- 2x tire 11/90R1
- 4x square holder 15x15x15mm
- 2x angled block 60° 15x15mm
- 2x angled block 30° 15x15mm
- 1x pin rail 15mm
- 2x M4 nuts and bolts L=25mm

1.5.3.2 GEARING (GEARS and GEARBOX)

- 2x gearboxes with shafts
- 2x sliding bearing
- 1x axle/shaft 45mm
- 1x axle/shaft 90mm
- 2x mechanical pivot joint
- 2x sliding bearing
- 2x spojka osi 15mm (BCA)
- 1x objemka 5mm (RD)
- 1x worm gear with attachment nut
- 1x gear fi48mm Z30
- 1x os elise 30mm

1.5.4 OPTIONAL

- rubber bands
- black isolating tape

2 ARDUINO PLATFORM

Arduino is a popular open-source platform used for developing electronic projects. The platform consists of hardware and software components, including microcontrollers, shields, sensors, and an integrated development environment (IDE).

The Arduino IDE is the software used to program and upload code to the microcontrollers. It is available for Windows, Mac, and Linux operating systems and is free to download. The IDE includes a text editor for writing code, a compiler that turns the code into machine language, and a bootloader that allows the code to be uploaded to the board.

There are different types of Arduino boards available, each with its unique features and capabilities. The most common boards include the Arduino Uno, which is widely used for beginners, and the Arduino Mega, which has more input and output pins. Other notable boards include the Arduino Nano, which is small and compact, and the Arduino Due, which has a more powerful processor.

Overall, the Arduino platform is versatile and easy to use, making it a popular choice for hobbyists, students, and professionals alike. Its open-source nature allows for a vast community of users to develop and share projects and resources, making it an excellent starting point for anyone interested in electronics and programming.

2.1 Overview of the Arduino platform and its capabilities

Arduino originated from the Wiring project, which was developed at the Interaction Design Institute Ivrea in Italy. The Wiring project was an open-source electronics prototyping platform that was designed to provide a low-cost and easy-to-use environment for creating interactive physical computing applications. The project was led by Hernando Barragán, a professor at the Institute, and the platform was based on the open-source, programmable Atmel microcontroller. Arduino was derived from the Wiring project and was released in 2005.

Arduino is an open source hardware and software platform used for building interactive electronics projects. The Arduino platform was designed to facilitate creating digital projects for the physical world. It consists of a physical programmable circuit board (often called a microcontroller) as well as a set of software tools for writing code for the board.

The Arduino platform is based on the Atmel AVR microcontroller, so it is capable of running programs written in C or C++. The board itself is made up of a number of components, including a voltage regulator, a USB connection, an LED, and a set of analog and digital pins that allow you to connect external components to the board. The board also includes a reset button and a power switch, allowing you to reset and power the board on and off.

The Arduino platform has a huge amount of flexibility and can be used to create a range of projects from simple to complex. For example, you can use the Arduino platform to create a basic home automation system that turns lights on and off, or you can use it to create a complex interactive art installation. You can also use the Arduino platform to create robots and other self-controlled devices.

The Arduino platform has grown to become an incredibly popular choice for makers, hobbyists, and professionals alike. It is incredibly easy to use, and the large community of users provides a wealth of tutorials and information. Additionally, the open-source nature of the platform makes it easy to customize and expand upon existing projects. It is a great platform for anyone looking to get started with physical computing projects.

2.2 Arduinouno and electronics simulation

We can use several simulating programs to simulate robots. There are awesome platforms that allow simulations like: 3Dvisualizer or Webots … But since our robot will be based on the Arduino Uno controller probably the best option is:

- Thinkercad

You can sign in with your google account.

2.2.1 Task: Basic Blink project

1. Try to do some basic project (e.g. Blink) to turn on and off an LED.
2. Try to add your own LED on the different output pin and change the program like is shown on the fig. 2.1 to make it work (LED must blinking).

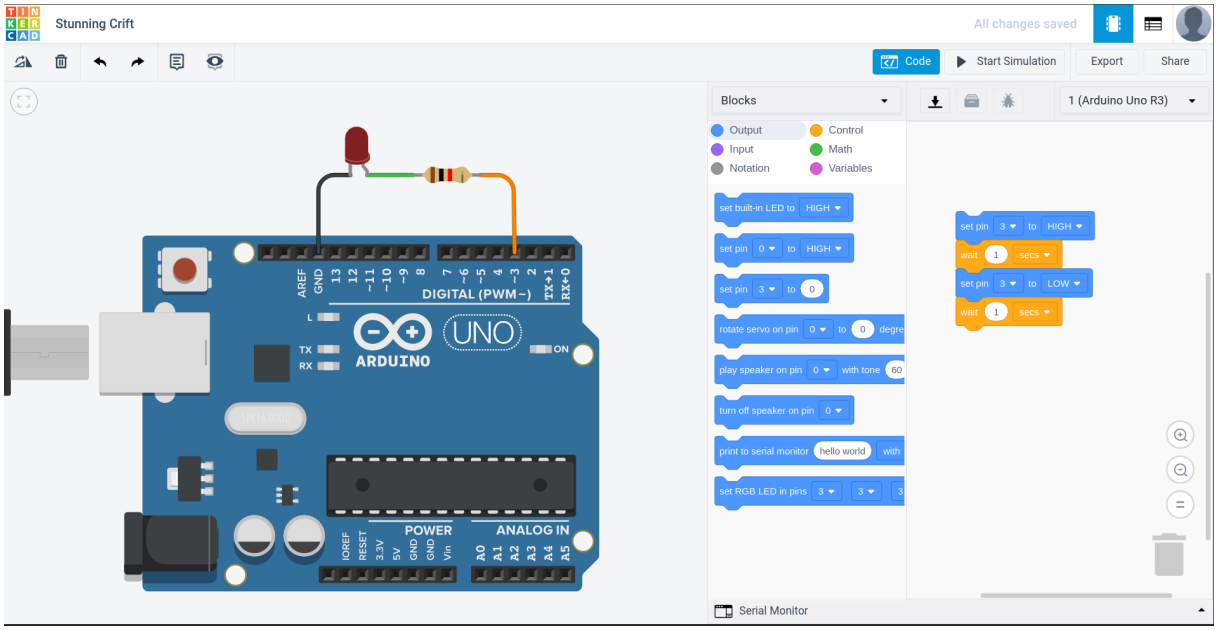

Figure 2.1: Blink example in Thinkercad.

2.2.2 Questions:

1. How can you control the output voltage potential?
2. In which direction will electric current flow?
3. What voltage is present on the resistor?

   - How can we determine the voltage on the resistor (describe 2 possibilities)?

4. What voltage is present on the LED?

2.3 Software installation

We will need softwate listed bellow:

1. Arduino IDE is basics “development environment”
2. RobDuino library for easier programming
3. Ardublockly is needed for introduction to programming

   - **Python** is needed for running Ardublockly

4. VSC in PlatformIO proper IDE include:

- auto-completion,
- error marking (e.g. forgotten ";"),
- auto-detect USB port,
- function information

2.3.1 Arduino IDE

1. Go to Arduino web page Arduino->Software->Download.
2. Download Arduino IDE 1.8.9 choose Windows Install…
3. … click JUST DOWNLOAD.
4. run arduino-1.8.9.exe and follow the instructions.
5. … don't forget to install also 3rd party drivers (for Chinese version of Arduino UNO controller)…
6. if you do forget… Try this Russian drivers from page.

2.3.1.1 Getting started

1. Run Arduino IDE
2. Connect Arduino Uno controller to USB port.
   Arduino Uno
3. Open simple basic program:
   files -> examples -> 01.basics -> blink

```
1  void setup() {
2    pinMode(LED_BUILTIN, OUTPUT);
3  }
4
5  void loop() {
6    digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the
       voltage level)
7    delay(1000);                       // wait for a second
8    digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the
       voltage LOW
9    delay(1000);                       // wait for a second
10 }
```

4. Make this settings in Tools menu ->
   1. Board: Arduino/Genuino Uno
   2. Port: COM3 *or similar*
5. Run :
   Upload to transfere the program to Arduino UNO controller.

6. If everything is OK you will get this message:

```
Done uploading.
Sketch uses 970 bytes (3%) of program storage space. Maximum is 32256
    bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes
    for local variables. Maximum is 2048 bytes.
```

9. Optional this preferences are suggested:
   File -> Preferences:

   1. Editor Language: English
   2. Editor font size: 20
   3. Show verbose output during: [ ]compiling [x] upload
   4. [x] Display linenumbers
   5. [x] Enable code folding

2.3.2 RobDuino

RobDuino is Arduino library which include some usefull functions for driving motors and on-board key usage…

2.3.2.1 RobDuino Library Installation

1. Download zip file:
   - RobDuino-master.zip
2. rename RobDuino-master.zip in:
   - **RobDuino.zip**
3. run Arduino IDE
4. choose:
   - Sketch -> Include Library -> Add .ZIP Library...
5. find
   - .../Download/RobDuino.zip
   - [OK]

2.3.3 Ardublockly

Ardublockly is graphical programming environment for programming Arduino controllers. A demo version of the program is also available on-line.

Note: For actual programming you will need Arduino IDE installed.

Note: For running Ardublockly you will need to install Python program.

2.3.3.1 Python Installation

1. You will have to install Python 3.7 or grater. First Download the newest version of Python.
2. Run installation file and set this settings:
    1. `[x] Add Python to PATH` in
    2. choose `Clasic Instalation`

2.3.3.2 Ardublockly Installation

3. From github.com/.../ardublockly download **zip** file by clicking **Clone or download** and choosen Download ZIP file.
4. Extract `ardublockly-master.zip` to dirrectory of your choice e.g. `C:\\Program Files (x86)`
5. That is it! Installation is complete.

2.3.3.2.1 Running Ardublockly

6. Find this file `C:\\Program Files(x86)\\ardublockly-master` and double-click on `start.py`. Python program should run and you should see:
    1. terminal window with some code running...
    2. and a new window should apear in your Internet Browser. If this is will not happend try to run `start.py` with right mouse button and `Start program with` then choose `Python 3.7`.

2.3.3.3 Settings

7. Click `menu` and choose `Settings`:

1. `Compiler Location`: C:\Program Files (x86)\Arduino\arduino_debug.exe
2. `Arduino Board`: Uno
3. `Com port`: COM3 *or appropriate one*
4. Click [`RETURN`].

2.3.4 VSC in PlatformIO

Note: For programming Arduino controllers you will need Arduino IDE installed.

Download installation file:

1. run `VSCodeUserSetup-ia32-1.49.3.exe` installation file.
2. run VSC program and click `Extensions`
3. search for `PlatformIO IDE` and
4. run `Install`.
5. restart VSC or click `Reload now`.

2.3.4.1 Getting Started

Write basic program `Blink`:

1. plug in Arduino Uno.
2. open `PlatformIO - Home Page`:

- in left icon bar find `PlatformIO`

  - `QUICK ACCESS` -> `PIO Home` -> `Open`

3. choose + `New Project`
4. Setup:

- `Name`: ime_projekta
- `Board`: Arduino UNO
- `Framework`: Arduino Framework

5. click `Finish`
6. Find directory `src` (e.g. `source code`), where you can find main program code in file `main.cpp`
7. Copy-Paste this example:

```
#include <Arduino.h>
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13,HIGH);
  delay(500);
  digitalWrite(13,LOW);
  delay(500);
}
```

8. Run `Build` and `Upload`.

2.4 Starting with Arduino IDE

The Arduino platform is based on the Atmel AVR microcontroller family, and the Arduino Uno is based on the ATmega328 microcontroller. The Arduino Integrated Development Environment (IDE) is a software application that provides a way to write and upload code to the microcontroller. The Arduino IDE is available for Windows, macOS, and Linux, and it is open source.

Happy programming!

2.4.1 Board setup

1. Connect the Arduino Uno to PC with proper USB cable.
   `[Arduino Uno]`
2. Make shure that you will set the proper settings (see fig. 2.2). From the menu choose:
   `Tools`-

    1. `Board`: Arduino/Genuino Uno
    2. `Port`: COM3

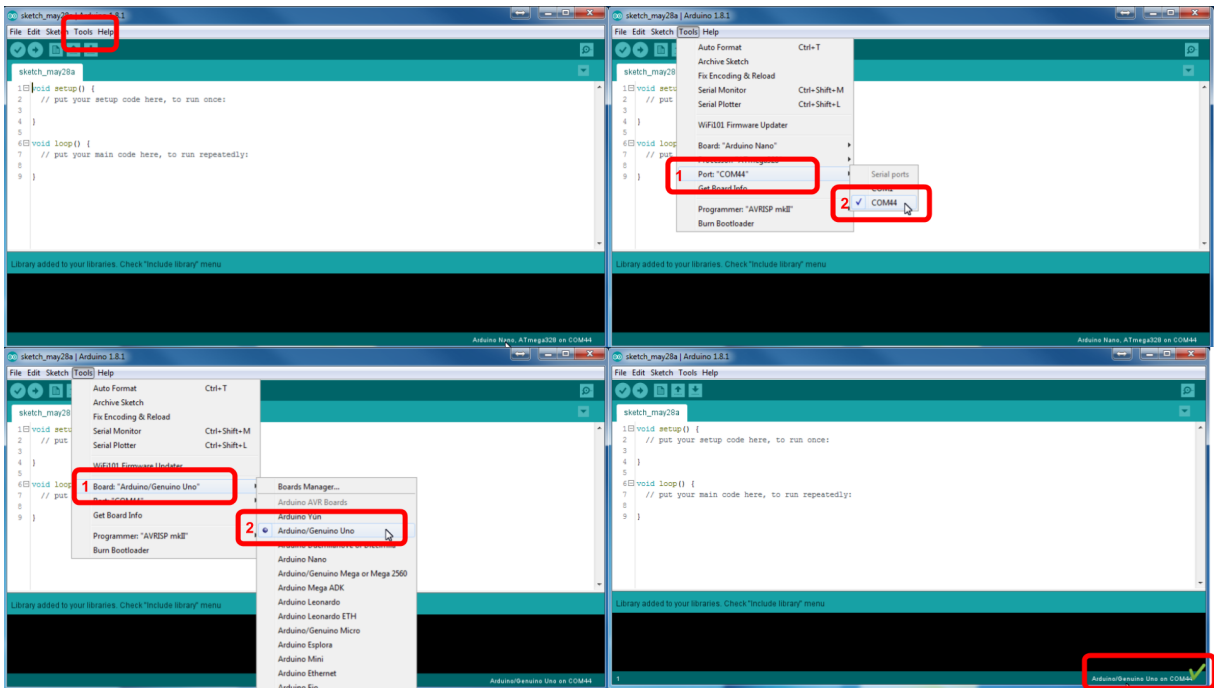

Figure 2.2: Arduino basic setup.

3. Open Arduino IDE program and open program with:
   `Files - Examples - 01. Basics - Blink.ino`
4. To upload the code you can click the icon `Upload`.
   If the uploading was successful you will be prompted with the text like:

```
Done uploading.
Sketch uses 970 bytes (3%) of program storage space. Maximum
is 32256 bytes. Global variables use 9 bytes (0%) of dynamic
memory, leaving 2039 bytes for local variables. Maximum is
2048 bytes.
```

2.4.2 Issues

2.4.2.1 LED_BUILTIN was not declared in this scope

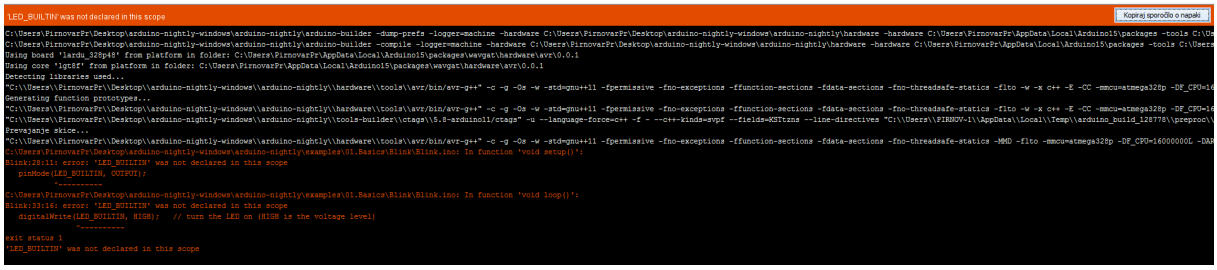

Figure 2.3: Error image.

Compiler ne ve kaj naj bi bilo “LED_BUILTIN” … na tem mesu naj bi bila številka priključka, ki ga želimo krmiliti. V tem primeru je to številka 13. Rešitvi sta lahko 2:

1. vse LED_BUILTIN zamenjaš s 13 ali

2. v vrstico pred “void setup()” dodaj `const int LED_BUILTIN = 13;`

Zadnja (druga) rešitev je boljša, ker pripomore k berljivosti programa… Spremenljivka LED_BUILTIN se imenuje “razlagalna spremenljivka” ker pomaga razlagati program. Tako postane tisti komentar “// turn the LED on (HIGH is the voltage level)” nepotreben, saj sama koda pove točno enako.

2.5 Setting up an Arduino development environment

2.6 Equipment testing

2.6.1 Basic testing in Arduino IDE

1. Connect the Arduino Uno to PC with proper USB cable.
   [Arduino Uno]
2. Open Arduino IDE program and open program with:
   Files - Examples - 01. Basics - Blink.ino
3. Make shure that you will set the proper settings (see fig. 2.4). From the menu choose:
   Tools-

   1. Board: Arduino/Genuino Uno

2. `Port`: COM3

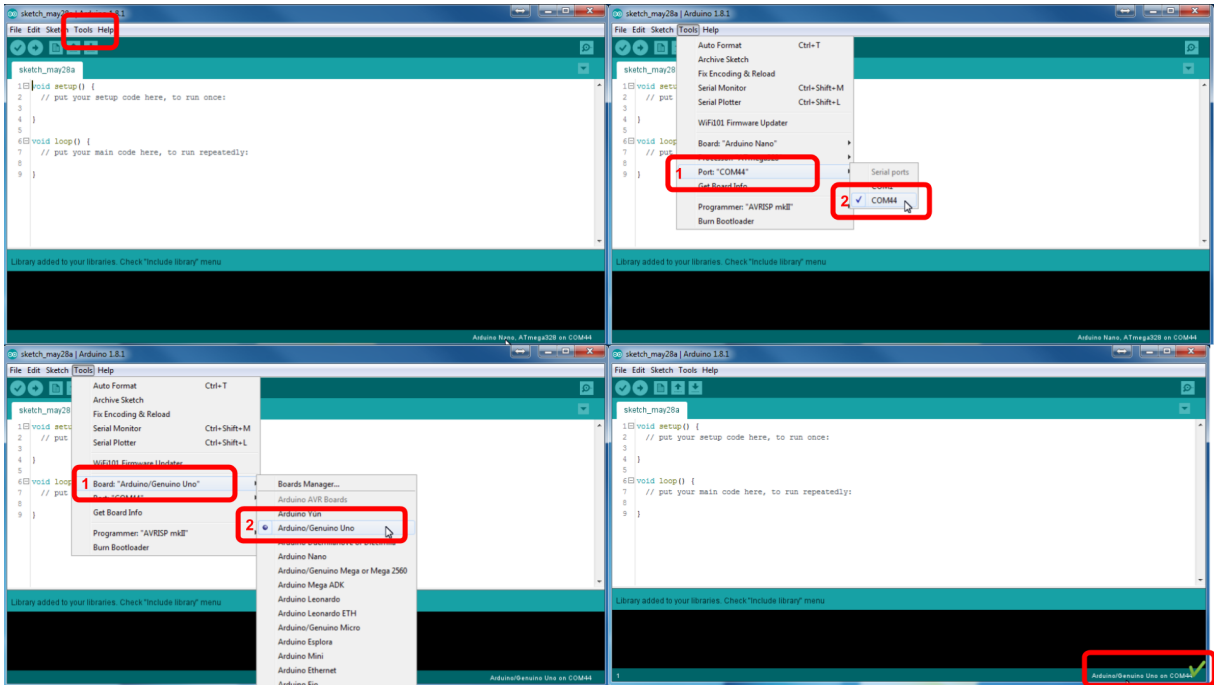

Figure 2.4: Arduino basic setup.

- To upload the code you can click the icon `Upload`.
  If the uploading was successful you will be prompted with the text like:

```
Done uploading.
Sketch uses 970 bytes (3%) of program storage space. Maximum
is 32256 bytes. Global variables use 9 bytes (0%) of dynamic
memory, leaving 2039 bytes for local variables. Maximum is
2048 bytes.
```

2.6.2 Issues

2.6.2.1 LED_BUILTIN was not declared in this scope

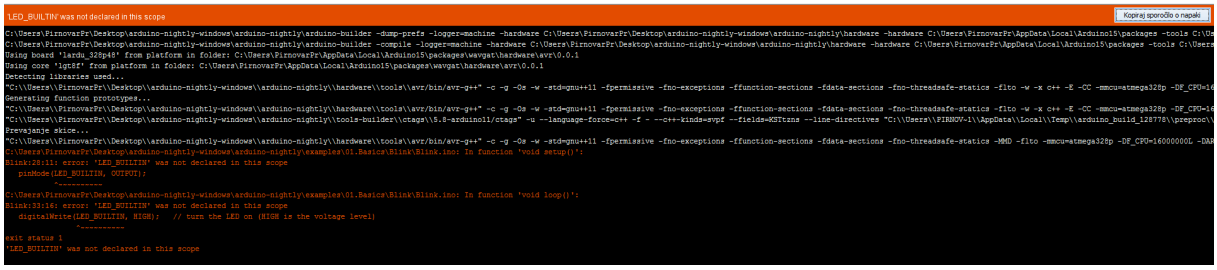

Figure 2.5: Error image.

Compiler ne ve kaj naj bi bilo “LED_BUILTIN” … na tem mesu naj bi bila številka priključka, ki ga želimo krmiliti. V tem primeru je to številka 13. Rešitvi sta lahko 2:
1. vse LED_BUILTIN zamenjaš s 13 ali
2. v vrstico pred “void setup()” dodaj `const int LED_BUILTIN = 13;`

Zadnja (druga) rešitev je boljša, ker pripomore k berljivosti programa… Spremenljivka LED_BUILTIN se imenuje “razlagalna spremenljivka” ker pomaga razlagati program. Tako postane tisti komentar “// turn the LED on (HIGH is the voltage level)” nepotreben, saj sama koda pove točno enako.

2.6.3 Basic testing in Ardublockly

1. Connect the Arduino Uno to PC with proper USB cable.
   [Arduino Uno]
2. Run Ardublockly program. Which will be running as localhost and you will be using internet browser as IDE. The addres will be:
   http://localhost:8000/ardublockly/index.html
3. In the left corner of the program you can find [=] menu icon. From where you can choose (Slide 2 and 3)
   [] Settings:
   1. Compiler Location: C:\Program Files (x86)\Arduino\arduino_debug.exe
   2. Arduino Board: Uno
   3. Com port: COM3
   4. And press: [RETURN]
4. Finaly you can press button PLAY And if uploading was successful you will be prompted with the text (Slide 4):

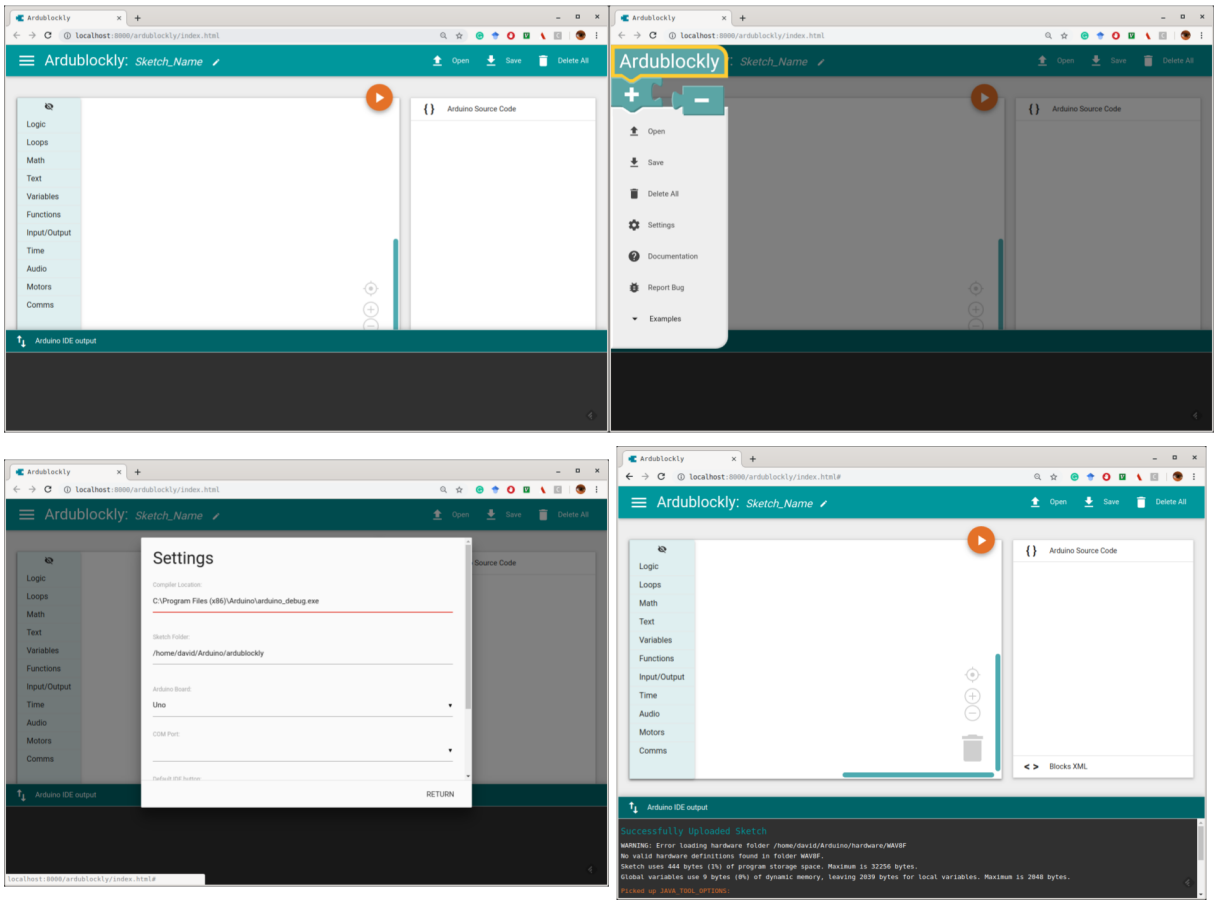

Figure 2.6: Ardublockly basic setup.

```
    Successfully Uploaded Sketch
    WARNING: Error loading hardware folder /home/david/Arduino/hardware/
       WAV8F.
    No valid hardware definitions found in folder WAV8F.
    Sketch uses 444 bytes (1%) of program storage space. Maximum is
    32256 bytes. Global variables use 9 bytes (0%) of dynamic memory,
    leaving 2039 bytes for local variables. Maximum is 2048 bytes.
```

2.6.4 Summary

Before uploading the programming code always check that the right board and serial port are set.

2.6.5 Issues

Ardublockly returns the Error id 55: Serial port Serial Port unavailable.
Try to re-connect the Arduino board. Wait a moment, check the settings and choose the COM port again then try again.

2.6.6 RobDuino module

1. Na krmilnik Arduino Uno priključite modul RobDuino in naložite naslednji program:

Listing 2.1: Equipment testing.

```
bool test_tipk = 1;
int l=1;

void setup() {
  for (int i = 0; i < 8; i++){
    pinMode(i, OUTPUT);
  }
  pinMode(A4, INPUT_PULLUP);
  pinMode(A5, INPUT_PULLUP);
  PORTD=1;
}

void loop() {
  char tipka_a4_is_pressed = !digitalRead(A4);
  char tipka_a5_is_pressed = !digitalRead(A5);
  if (tipka_a4_is_pressed) l = l >> 1;
  if (tipka_a5_is_pressed) l = l << 1;
  if (tipka_a4_is_pressed && tipka_a5_is_pressed) test_tipk = !test_tipk
      ;
  if (test_tipk){
    if (l < 1) l = 128;
    if (l > 255) l = 1;
    PORTD = l;
  }else{
    PORTD = analogRead(A0) >> 2;
  }
  delay(200);
}
```

2. Nato preverite delovanje obeh tipk (A4 in A5) na modulu in vrednosti izhodnih priključkov D0 .. D7.

2.6.7 Napajalni modul

Napajalni modul uporablja 2x Li-ion akumulatorja tipa 18650. Spodnje tiskano vezje je prikazano fig. 2.7.

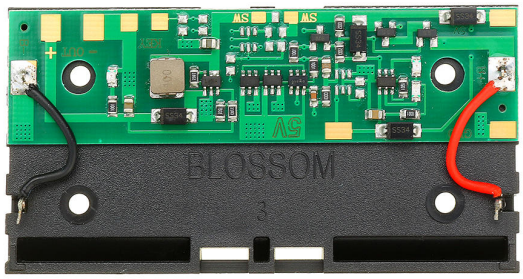

Figure 2.7: Napajalni modul.

Dodatno smo ga opremili z:

1. 2.5mm jack priključkom za napajanje,
2. 3-pinskim priključkom za napajanje,
3. preklopnim stikalom za izbiranje načina delovanja:
    1. ON - izhod za 9V je kaktiviran
    2. OFF - izključen izhod 9V napajanja in omogočeno je polnenje akumulatorjev preko 3-pinskega priključka (5V).

Pomembno: Pred prvo uporabo moramo ročno aktivirati napajalni modul tako, da povežemo GND na 3-pinskem priključku in NEGATIVNI terminal akumulatorjev.

2.6.8 Tipka

1. Priključite stikalo po shemi na fig. 2.8.

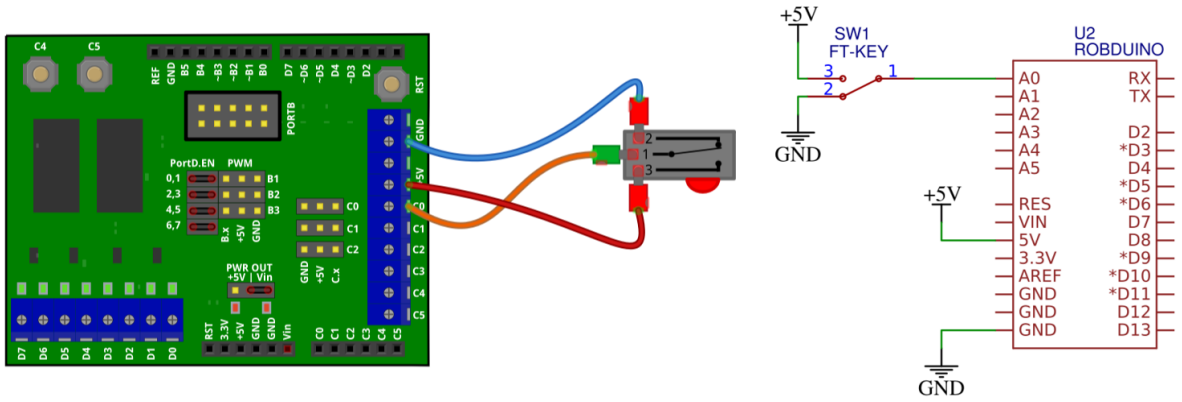

Figure 2.8: Priključitev tipke.

2. Nato naložite naslednji program.

```
void setup() {
    pinMode(A0, INPUT);
    pinMode(7, OUTPUT);
}

void loop() {
    char key_a0_is_pressed = digitalRead(A0);
    if (key_a0_is_pressed){
        digitalWrite(7, HIGH);
    } else{
        digitalWrite(7, LOW);
    }
    delay(100);
}
```

2.6.9 Svetlobni senzor

1. Priključite foto-tranzistor v delilnik napetosti z uporom, kot prikazuje fig. 2.9.

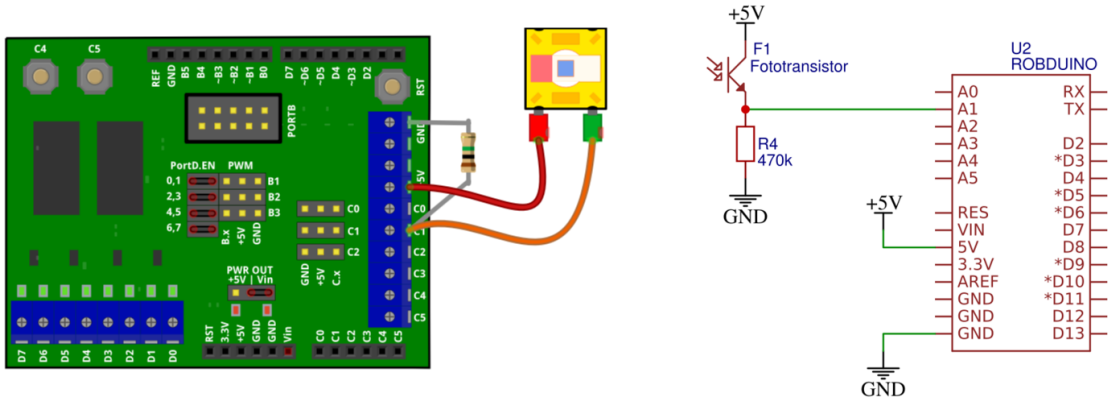

Figure 2.9: Priključitev foto-tranzistorja kot svetlobni senzor.

2. Nato naložite naslednji program in preverite odziv svetlobnega senzorja.

```
 void setup() {
  pinMode(A1, INPUT);
  Serial.begin(9600);
}

void loop() {
  int light_senzor_value = analogRead(A1);
  Serial.println(light_senzor_value);
  delay(100);
}
```

3. Odziv senzorja spremljajte v oknu serijske komunikacije.

2.6.10 IR senzor razdalje

1. IR senzor razdalje priključite na tri-pinski priključek kot je prikazano na fig. 2.10.

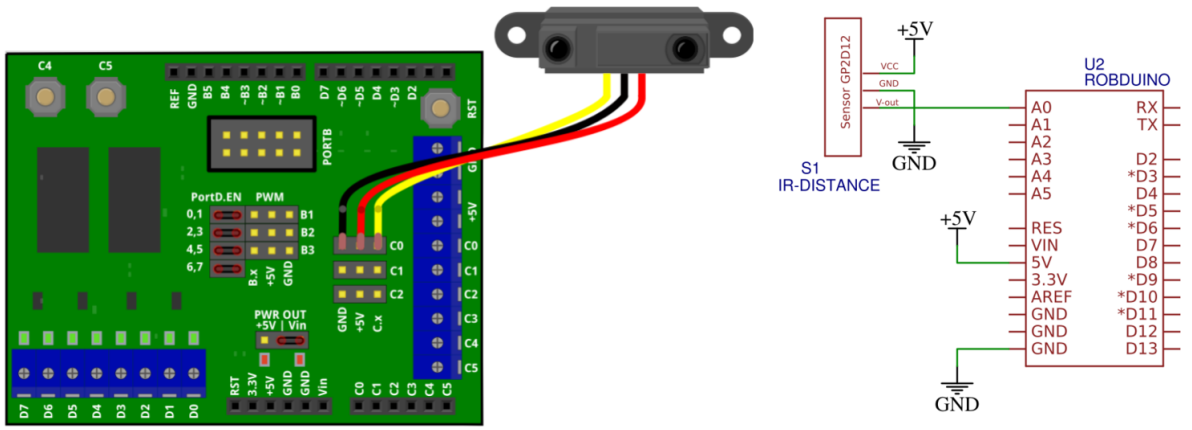

Figure 2.10: Priključitev IR senzorja razdalje.

2. Delovanje senzorja preskusite z naslednjim programom, njegov odziv pa spremljajte v oknu za serijsko komunikacijo.

```
void setup() {
  pinMode(A0, INPUT);
  Serial.begin(9600);
}

void loop() {
  int distance_senzor_value = analogRead(A0);
  Serial.println(distance_senzor_value);
  delay(100);
}
```

2.6.11 LCD (I2C)

1. Priključite LCD na I2C vodilo kot prikazuje

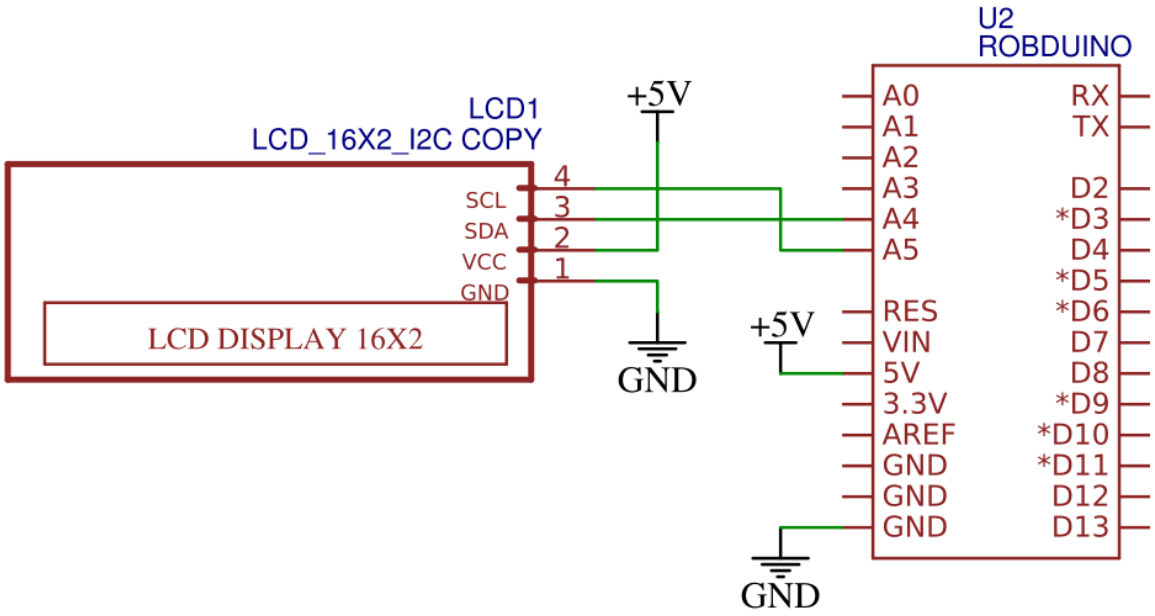

Figure 2.11: Povezava LCD na I2C vodilo krmilnika.

2. Priskrbite si knjižnico `LiquidCristal-I2C` iz naslova:
   https://www.arduino.cc/reference/en/libraries/liquidcrystal-i2c/

3. Knjižnico dodajte v Arduino IDE okolje tako, da dodate `ZIP` datoteko v :
   `Sketch >> Include Library >> Add .ZIP Library`

4. V VSC in PlatformIO vtičniku si lahko knjižnico naložite tako, da v terminalno okno vpišete ukaz
   `pio lib install "marcoschwartz/LiquidCrystal_I2C@^1.1.4"`

5. Nato preskusite naslednji program:

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C Lcd(0x27, 16, 2);

void setup() {
    Lcd.init();

    Lcd.clear();
    Lcd.backlight();

    Lcd.setCursor(3,0);
    Lcd.print("Hello");
    Lcd.setCursor(6,1);
    Lcd.print("World");
}

void loop() {
}
```

Če niste prepričani kateri i2c naslov uporablja naprava na LCD-ju le tega lahko preverite s programom `I2C scanner` (https://playground.arduino.cc/Main/I2cScanner/). Običajno I2C LCD-ji, ki jih naredijo kitajski proizvajalci uporabljajo I2C naslov `0x27` , `0x3F` ali manj pogosto `0x38`.

2.7 Programming Arduino UNO using C++

C++ is a powerful and efficient programming language widely used for embedded systems, including Arduino. It provides low-level control over hardware while still offering high-level abstractions, making it a great choice for programming microcontrollers like ATmega328.

One of the key reasons C++ is appropriate for hardware communication is its ability to manage memory efficiently and interact directly with registers and peripherals. Features like direct memory access, bitwise operations, and object-oriented programming allow developers to write both high-performance and modular code.

As Linus Torvalds, the creator of Linux, famously said:

> C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it…"

While Torvalds is critical of C++ in system programming and prefers using C, in embedded development, its mix of low-level access and high-level structure makes it an ideal choice for Arduino projects especially for starters.

2.8 Hello world in Arduino IDE

2.8.1 Task: Programming the microcontroller

1. Make a very simple program like setting the digital output bit D3 to logical state 1 or **HIGH**.

Listing 2.2: Hello World in ArduinoIDE.

```
1   void setup() {
2    // put your setup code here, to run once:
3    pinMode(3, OUTPUT);
4    digitalWrite(3, HIGH);
5   }
6
7   void loop() {
8    // put your main code here, to run repeatedly:
9
10  }
```

2. Send the program t o controller Arduino UNO .

2.8.2 Questions:

1. Explain the purpose of next programming characters in presented example:
    1. ;
    2. { }
    3. `pinMode(3, OUTPUT);`
    4. `digitalWrite(3, HIGH);`
    5. `//put your ...`
    6. `void setup()`
    7. `void loop()`

2.8.3 Summary:

2.8.3.1 Using curly braces - { and }

Using curly braces in C++ is important part of writing the programming code. Imagine that you want to merge several members of programming code to a single pile. As we would separate pencils into one pile and markers to another - to be more organized. In real life we would do by elastic bundle or rope. If you have to choose single character from the keyboard to indicate that several members are combined to the same pile - which character would you choose? Probably curly braces {} are the best choice.

2.8.3.2 Function Name

Function name should be stacked together from 2 - 5 short words that uniquely describing the functionality of the function. The first word should start with lower case and all the others words following should start with upper case. Some examples should be:

```
badname();
goodFunctionName();
```

2.8.3.3 Function Declaration

```
int measre_Temperature_Avg(int temperatureSensor);
```

2.8.3.4 Function Definition

```
void loop() {
    //some programming
    //code goes here...
}
```

2.8.3.5 Function Call

```
    digitalWrite(3, HIGH);
```

2.8.4 Issues:

2.8.4.1 *Error: expected ';' before 'something'*

Probably you forgot to put ; (semicolon) at the end of the command. Find the row starting with "**something**" and look the row above... probably missing "**;**".

2.8.4.2 *Light at the digital output D3 is not ON.*

Check if the enable switch fot the digital outputs is at the right position (ENABLE).

2.9 Communicating with sensors and actuators

2.10 Implementing control algorithms

2.11 Debugging and testing

2.12 Getting Started with the RobDuino Module in Early Robotics

Robotics is a fantastic way to introduce students to electronics, programming, and system design. The **RobDuino module** was developed specifically for early robotics education using the Arduino UNO. It simplifies connections between the microcontroller and common electronic components such as sensors, DC motors, and LCDs. This chapter explains the hardware features, provides step-by-step setup instructions, and offers sample projects and exercises to deepen your understanding.

2.12.1 Introduction

The RobDuino module is designed to make it easier to connect sensors, actuators, and other components to an Arduino UNO. By integrating an H-bridge driver on the digital pins and providing accessible screw terminals, it allows for quick prototyping and experimentation. In this chapter, you will learn:

- **Key Hardware Features:** How the module's pins are arranged and their specific functions.
- **Wiring Techniques:** How to correctly connect sensors, motors, and other devices.
- **Programming Basics:** How to use Arduino functions like `digitalWrite()`, `analogWrite()`, `digitalRead()`, and `analogRead()` with the module.
- **Sample Projects:** Practical examples such as blinking an LED and controlling a DC motor.
- **Self-Assessment:** Questions to test your understanding.

2.12.2 Hardware Overview

Before diving into the projects, it is essential to understand the RobDuino module's layout and functionality.

2.12.2.1 Digital Output Pins (D0–D7)

- **H-Bridge Driver:** Each digital pin (D0 to D7) is equipped with an H-bridge driver. These pins are intended to be used as **digital outputs only**.
- **Usage:** Use `digitalWrite(pin, HIGH)` or `digitalWrite(pin, LOW)` to set the voltage level.
- **PWM Capability:** On pins D6, D5, and D3, you can also use the `analogWrite(pin, value)` function to output a Pulse Width Modulated (PWM) signal. This is useful for controlling the speed of motors or the brightness of LEDs.

2.12.2.2 Analog and Digital Input Pins (A0–A5)

- **Versatile Use:** The analog pins can function as either digital or analog inputs.
- **Digital Read:** Use `digitalRead(pin)` to determine if the voltage is HIGH (+5V) or LOW (0V).
- **Analog Read:** Use `analogRead(pin)` to obtain the actual voltage level (a value between 0 and 1023).
- **Special Note:** Pins A4 and A5 are also equipped with jumpers connected to GND. This configuration can be beneficial for certain sensor setups and helps to stabilize the readings.

2.12.2.3 Power and Control Terminals

- **+5V and GND Terminals:** These screw terminals are used to power sensors and other modules.
- **Reset Button:** Resets the Arduino UNO, restarting the currently loaded program.
- **ENABLE Switch:** This switch can disable all outputs on pins D0–D7, which is especially useful for safely disconnecting actuators (e.g., DC motors) during testing or when not in use.

2.13 Wiring and Setup

2.13.1 Connecting the RobDuino Module

1. **Digital Connections:**
   - Connect your output devices (LEDs, motors, etc.) to the screw terminals labeled D0–D7.
   - For PWM control, make sure your device is connected to D3, D5, or D6.
2. **Analog Connections:**

- Attach sensors (such as light sensors, potentiometers, etc.) to the screw terminals labeled A0–A5.
- Remember: if using A4 or A5, the built-in jumper to GND might affect your circuit, so consult the sensor's datasheet if necessary.

3. **Powering Devices:**

   - Use the +5V and GND terminals on the RobDuino module to supply power to sensors or other low-voltage devices.

4. **Control Elements:**

   - Familiarize yourself with the **Reset** button for restarting your program.
   - Use the **ENABLE** switch to disable outputs when you need to prevent accidental activation of actuators.

2.13.2 Safety Tips

- **Double-check connections** before powering the system.
- **Use current-limiting resistors** when connecting LEDs to avoid damaging components.
- **Ensure proper grounding** to prevent erratic sensor readings.

2.14 Programming Examples

Programming the RobDuino module is very similar to programming a standard Arduino UNO. Here are a few example sketches to illustrate its use.

2.14.1 Example 1: Blinking an LED on a Digital Output Pin

```
// Connect an LED (with a current-limiting resistor) to the D0 terminal.

void setup() {
  // Set digital pin 0 as an output
  pinMode(7, OUTPUT);
}

void loop() {
  digitalWrite(7, HIGH);   // Turn the LED on
  delay(1000);             // Wait for 1 second
  digitalWrite(7, LOW);    // Turn the LED off
  delay(1000);             // Wait for 1 second
}
```

2.14.2 Example 2: Controlling LED Brightness with PWM

```
// Connect an LED to the D6 terminal which supports PWM

void setup() {
  // Set digital pin 6 as an output
  pinMode(6, OUTPUT);
}

void loop() {
  // Increase brightness
  for (int brightness = 0; brightness <= 255; brightness++) {
    analogWrite(6, brightness);
    delay(10);
  }

  // Decrease brightness
  for (int brightness = 255; brightness >= 0; brightness--) {
    analogWrite(6, brightness);
    delay(10);
  }
}
```

2.14.3 Example 3: Reading an Analog Sensor

```
// Connect a potentiometer or any analog sensor to the A0 terminal

void setup() {
  Serial.begin(9600);  // Initialize serial communication for debugging
}

void loop() {
  int sensorValue = analogRead(A0);  // Read the analog value from
     sensor
  Serial.println(sensorValue);       // Print the sensor value to the
     Serial Monitor
  delay(500);                        // Wait half a second before the
     next reading
}
```

2.14.4 Example 4: Controlling a DC Motor Using the H-Bridge

```
// Assume the DC motor is connected via an H-bridge on digital pins D7 (direction) and D6 (PWM speed control)

void setup() {
  pinMode(7, OUTPUT); // Motor direction control
  pinMode(6, OUTPUT); // PWM speed control
}

void loop() {
  // Set motor direction to forward
  digitalWrite(7, HIGH);

  // Ramp up the motor speed
  for (int speed = 0; speed <= 255; speed++) {
    analogWrite(6, speed);
    delay(10);
  }

  // Run at full speed for 2 seconds
  delay(2000);

  // Ramp down the motor speed
  for (int speed = 255; speed >= 0; speed--) {
    analogWrite(6, speed);
    delay(10);
  }

  // Brief pause before reversing the motor
  delay(500);

  // Change motor direction to reverse
  digitalWrite(7, LOW);

  // Repeat the speed ramping for reverse motion
  for (int speed = 0; speed <= 255; speed++) {
    analogWrite(6, speed);
    delay(10);
  }

  delay(2000);

  for (int speed = 255; speed >= 0; speed--) {
    analogWrite(6, speed);
    delay(10);
  }

  delay(500);
}
```

2.15 Sample Projects

2.15.1 Project 1: LED Blinker

- **Objective:** Understand basic digital output.
- **Components:** LED, resistor, wires.
- **Procedure:** Connect the LED to terminal D0. Upload the blinking LED sketch (Example 1). Observe the LED turning on and off at one-second intervals.

2.15.2 Project 2: Motor Speed Controller

- **Objective:** Learn how to control a motor's speed and direction.
- **Components:** DC motor, H-bridge (integrated on the module), external power supply (if required), wires.
- **Procedure:** Connect the motor as indicated in Example 4. Experiment with changing the PWM values and delay times to observe variations in motor speed and behavior.

2.15.3 Project 3: Analog Sensor Monitor

- **Objective:** Acquire and monitor analog sensor data.
- **Components:** Potentiometer or light sensor, wires.
- **Procedure:** Connect the sensor to terminal A0. Use Example 3 to read sensor values and view the data on the Serial Monitor.

2.16 1.6 Self-Assessment Questions

1. **Digital Outputs:**
    - What is the primary function of the digital pins D0–D7 on the RobDuino module?
      (Hint: Consider their connection to the H-bridge drivers.)
2. **PWM Capability:**

- Which digital pins support PWM, and why is PWM useful in robotics?

3. **Analog Input:**
   - How do you obtain a numerical value from an analog sensor connected to the RobDuino module?
     (Which function is used, and what is the typical range of values?)
4. **Enable Switch:**
   - What is the purpose of the ENABLE switch on the module, and how does it affect connected actuators?
5. **Ground Jumpers on Analog Pins:**
   - Why might pins A4 and A5 have jumpers connected to GND, and how can this affect sensor operation?

2.17 Chapter Summary

- **Module Overview:** The RobDuino module is a powerful tool designed for early robotics education, integrating an H-bridge on digital pins and providing accessible screw terminals.
- **Digital vs. Analog:** Digital pins (D0–D7) are intended as outputs (with PWM available on D3, D5, and D6), while analog pins (A0–A5) are versatile and can be used for digital or analog inputs.
- **Power and Control:** The module includes dedicated +5V and GND terminals, a reset button for rebooting the system, and an ENABLE switch for safely managing actuator outputs.
- **Programming:** Standard Arduino functions (`digitalWrite()`, `analogWrite()`, `digitalRead()`, and `analogRead()`) are used to interface with the module, making it accessible to beginners.
- **Practical Projects:** Through projects like LED blinking, motor control, and sensor monitoring, students can gain hands-on experience in electronics and robotics.

By mastering these basics, you lay a solid foundation for exploring more complex robotics projects in the future. Use the provided examples and self-assessment questions to test your knowledge and reinforce your learning.

Happy Robotics!

3 ROBOTICS HARDWARE

When we think of robots, the first image that often comes to mind is the robotic arm. This is no coincidence—robotic arms dominate industrial automation, performing tasks such as welding, assembly, and material handling with precision and efficiency. According to the International Federation of Robotics (IFR), industrial robots (primarily robotic arms) accounted for over 70% of all robot installations worldwide in recent years, highlighting their importance in manufacturing and production lines.

However, mobile robots are rapidly gaining ground. From autonomous warehouse robots to self-driving vehicles and planetary rovers, mobile robotics is becoming a key player in logistics, healthcare, and exploration. With advancements in artificial intelligence and sensor technology, these robots are no longer confined to structured environments but can adapt to dynamic, real-world conditions.

As we explore robotic hardware, we will see how both stationary and mobile robotic systems rely on a combination of mechanics, electronics, and control systems to interact with their environment.

3.1 Overview of different types of robots

Robotics is a diverse field that encompasses a wide range of machines designed to perform tasks autonomously or semi-autonomously. Robots can be classified based on their structure, application, mobility, and control mechanisms. Below is a detailed breakdown of the different types of robots commonly used in industry, research, and everyday applications.

3.1.1 Classification by Mobility

One of the most fundamental ways to categorize robots is based on how they move and interact with their environment.

3.1.1.1 Stationary Robots (Fixed Robots):

These robots are anchored in place and perform tasks within a limited workspace. They are primarily used in industrial settings and rely on precise control systems.

- **Robotic Arms** – The most common type of stationary robot, used in manufacturing for welding, painting, assembly, and material handling. Examples include the **ABB IRB series** and **Fanuc robotic arms**.
- **SCARA Robots (Selective Compliance Articulated Robot Arm)** – Specialized for high-speed pick-and-place tasks, common in electronics manufacturing.
- **Delta Robots (Parallel Robots)** – Used for high-speed sorting and packaging in food and pharmaceutical industries.

3.1.1.2 Mobile Robots:

Unlike stationary robots, mobile robots can move freely in their environment. They are equipped with wheels, tracks, legs, or even flight capabilities.

Wheeled Robots: Wheeled robots are among the most common mobile platforms due to their efficiency and simplicity.

- **Differential Drive Robots** – Use two independently controlled wheels for movement, like the **TurtleBot** or **Roomba** vacuum cleaner.
- **Omnidirectional Robots** – Equipped with mecanum or omnidirectional wheels, allowing movement in any direction without turning, used in logistics and soccer robots.
- **Self-Balancing Robots** – Utilize gyroscopes and accelerometers for balance, such as the **Segway** or the **NASA's Robonaut**.

Tracked Robots

- **Tank-like robots with caterpillar tracks** – Used for rough terrain, such as bomb disposal robots (*PackBot* by iRobot) and planetary rovers (*Curiosity* and *Perseverance*).

Legged Robots: Designed for navigating complex environments where wheels are ineffective.

- **Bipedal Robots** – Humanoid robots like **Boston Dynamics' Atlas** or **Honda's ASIMO**.
- **Quadrupedal Robots** – Four-legged robots used for search and rescue, such as **Spot by Boston Dynamics**.

- **Hexapods and Multi-legged Robots** – Used for stability and adaptability in hazardous environments.

Aerial Robots (Drones, UAVs): Flying robots designed for surveillance, mapping, and delivery services.

- **Quadcopters and Multirotors** – Used for aerial photography, military reconnaissance, and package delivery (*DJI Phantom*, *Amazon Prime Air*).
- **Fixed-Wing Drones** – Longer flight times for mapping and agriculture (*Parrot Disco*, *NASA's UAVs*).
- **Hybrid VTOL Drones** – Combine the advantages of quadcopters and fixed-wing aircraft.

Underwater Robots (ROVs and AUVs): Submersible robots for underwater exploration, maintenance, and research.

- **Remotely Operated Vehicles (ROVs)** – Controlled via tethered cables, used for underwater inspections (*Deep Discoverer*).
- **Autonomous Underwater Vehicles (AUVs)** – Capable of autonomous navigation for ocean mapping (*Bluefin-21* used in MH370 search).

3.1.2 Classification by Application

Different industries require specialized robots optimized for their specific tasks.

3.1.2.1 Industrial Robots

Used in manufacturing and automation.

- **Assembly Line Robots** – Perform repetitive tasks with high precision (*KUKA, Fanuc, ABB robotic arms*).
- **CNC and 3D Printing Robots** – Convert digital designs into physical objects using subtractive or additive manufacturing.

3.1.2.2 Service Robots

Designed to assist humans in daily tasks.

- **Household Robots** – Vacuum cleaners (*Roomba*), lawn mowers, personal assistants (*Amazon Astro*).
- **Medical Robots** – Used in surgery (*Da Vinci surgical robot*), rehabilitation, and diagnostics.

3.1.2.3 Military and Defense Robots

Used for surveillance, reconnaissance, and combat.

- **Unmanned Ground Vehicles (UGVs)** – Explosive ordnance disposal (*Talon, PackBot*).
- **Autonomous Combat Robots** – Armed drones (*MQ-9 Reaper*).

3.1.2.4 Space Exploration Robots

Robots designed for planetary exploration and maintenance.

- **Rovers** – Used for planetary exploration (*Curiosity, Perseverance*).
- **Autonomous Satellites** – Repair and maintenance (*DARPA's Robotic Servicing Program*).

3.1.2.5 Agricultural and Environmental Robots

Designed for precision farming and environmental monitoring.

- **Autonomous Tractors** – Used in precision agriculture (*John Deere's autonomous farming*).
- **Pollination Robots** – Artificial bee drones for pollination (*Harvard's RoboBee*).

3.1.3 Classification by Intelligence and Autonomy

Robots can also be categorized by their level of intelligence and autonomy.

3.1.3.1 Pre-Programmed Robots

Follow predefined instructions with minimal real-time adaptation (*CNC machines*).

3.1.3.2 Teleoperated Robots

Controlled remotely by humans (*surgical robots, bomb disposal robots*).

3.1.3.3 Autonomous Robots

Use AI and sensors to make decisions (*self-driving cars, warehouse robots*).

3.1.3.4 Collaborative Robots (Cobots)

Designed to work safely alongside humans (*Universal Robots' cobots*).

3.1.4 Conclusion

Robots come in many forms, each optimized for different environments and applications. While industrial robotic arms dominate automation, mobile robots are rapidly expanding into logistics, exploration, and personal assistance. The future of robotics will likely see even more integration of AI, making robots more adaptive and capable in complex environments.

3.2 Motor as main actuator

A DC motor (Direct Current motor) is an electrical machine that converts electrical energy into mechanical energy. It works by using electromagnetic principles to generate rotary motion.

Here is how a DC motor works in more detail:

- The DC motor has two main parts: the stator and the rotor. The stator is the stationary part of the motor, and the rotor is the rotating part.
- The stator consists of a coil of wire that is wound around a core. When an electric current flows through the coil, it creates a magnetic field around the core.

- The rotor consists of a permanent magnet or a coil of wire that is mounted on a shaft. When the rotor is placed inside the stator, the magnetic fields of the stator and the rotor interact with each other.
- If the stator's coil is energized with a DC current, the magnetic field it creates will rotate around the core. This causes the rotor to rotate as well, since it is attracted to the moving magnetic field.
- The speed and direction of the rotor's rotation can be controlled by adjusting the strength and polarity of the current flowing through the stator's coil. This is typically done using an H-bridge circuit, which allows the current to be reversed and the motor to run in both directions.

3.2.1 Task: Wire a DC motor to a battery

1. Connect the DC motor to the battery and make it run.
2. You can try different combinations to connect the terminals of the motor like:
    - + and -
    - - and +
    - - and -
    - + and +.

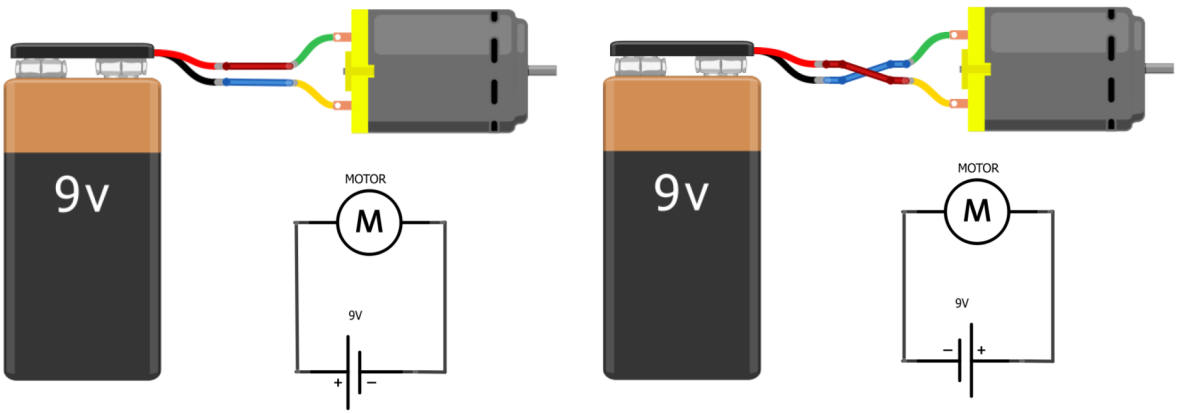

Figure 3.1: DC motor connection.

3.2.2 Questions:

1. In which direction the motor's shaft spins in different situations?
2. In which direction the electric current flow?
3. Why does motor is not spinning when both connectors are connected to + terminal of the battery?

3.2.3 Summary

The rotation of the DC motor depends on the direction of electric current.

3.2.4 Issues

3.2.4.1 *When I connect the DC motor to + and - terminals of the battery the motor's shaft does not spin.*

Check the voltage of the battery… battery may be discharged.
Check the connectors of the motor… may be bad.

3.3 DC motor control with digital outputs

3.3.1 Task: Control the DC motor with controller

1. Connect the DC motor to Digital Output D7 and D6.
2. Write the program and check all the combinations of digital outputs; 00, 01, 10 and 11. First combination is shown in prog. example 3.1

Listing 3.1: DC Motor Control with Digital Outputs.

```
1   void setup()
2   {
3     pinMode(7, OUTPUT);
4     pinMode(6, OUTPUT);
5     // D7=0, D6=0 Že če je to ok je vse ok
6     digitalWrite(7, LOW);
7     digitalWrite(0, LOW);
8     delay(3000);
9     // Write other combinations here...
10
11  }
12  void loop()
13  {
14
15  }
```

3. For each combination of digital outputs mark the state of the motor (fulfill the tbl. 3.1).

Table 3.1: All combinations of the states of motor's connectors.

D7	D6	Motor rotation
0	0	
0	1	
1	0	
1	1	

3.3.2 Questions:

2. Try to stop the shaft of the DC motor for a short time and try to remember how difficult it is?
3. Why does motors' shaft not spinning if the digital output state are 1 and 1.

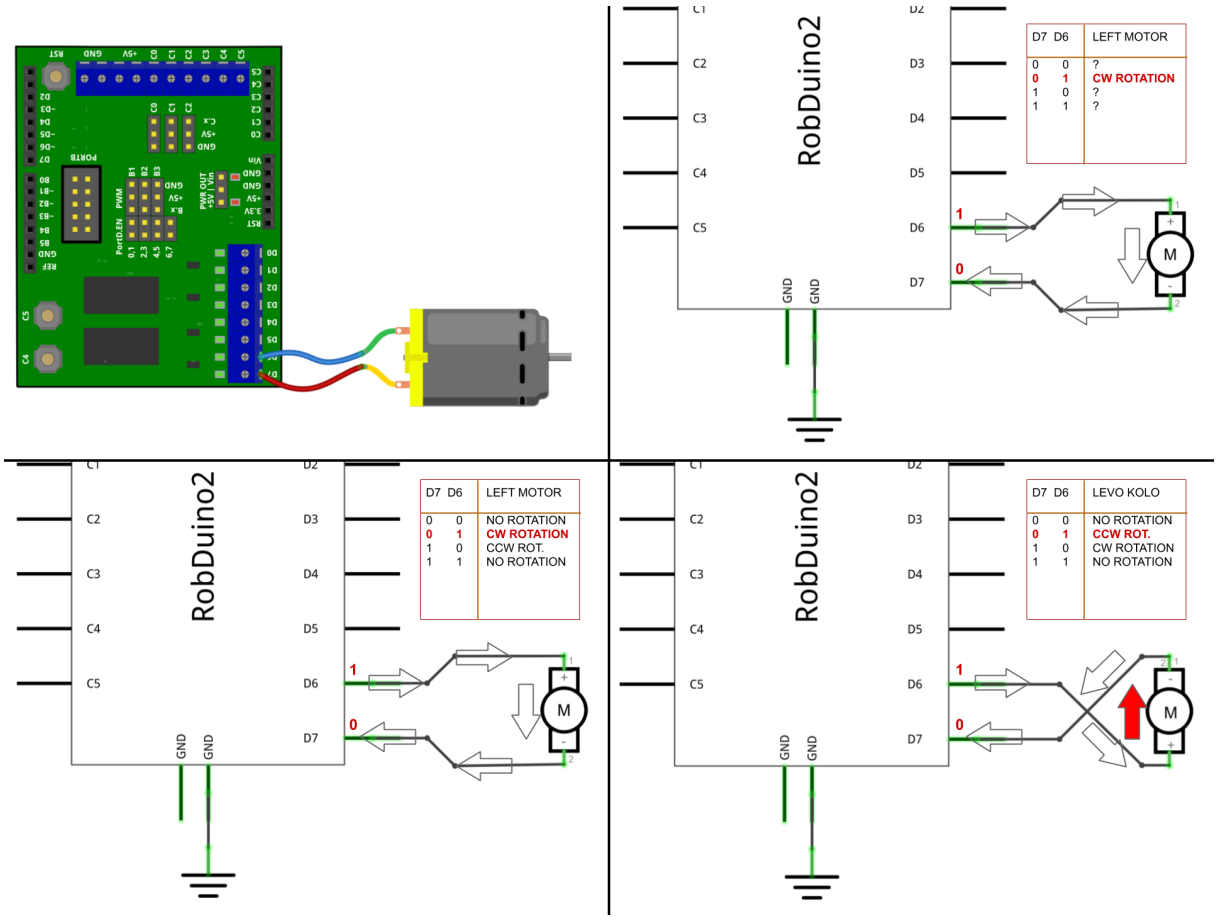

Figure 3.2: Wireing the DC motor to controller.

3.3.3 Summary

The motor's shaft is spinning according to the direction of the electric current trough the motor. The torque is weak.

3.3.4 Issues

3.4 Gear reducer

Gear reduction is the process of using a set of gears to reduce the speed of a mechanical system while increasing the torque (rotational force). It is commonly used in robotics and other applications where it is necessary to trade speed for power.

There are several ways to achieve gear reduction, but the most common method is to use a gear train, which is a series of interconnected gears that transmit motion from one gear to another. By using gears with different sizes and ratios, it is possible to reduce the speed of the output gear while increasing the torque.

For example, consider a simple gear train with two gears: a larger driving gear (Gear A) and a smaller driven gear (Gear B). If the driving gear has 10 teeth and the driven gear has 20 teeth, the gear reduction ratio will be 2:1 (Gear B will rotate at half the speed of Gear A, but with twice the torque).

Here is the eq. 3.1 for calculating the gear reduction ratio:

$$R = \frac{N_1}{N_2} \tag{3.1}$$

where:

- R is gear ratio or often called `mechanical advantage`,
- N_1 is number of teeth on driving gear and
- N_2 is number of teeth on driven gear.

3.4.1 Task: Use reductor on a DC motor shaft

1. Add geared reductor to DC motor.
2. Try to stop the shaft of the geared reductor and compare your fillings with the stopping the motor shaft.

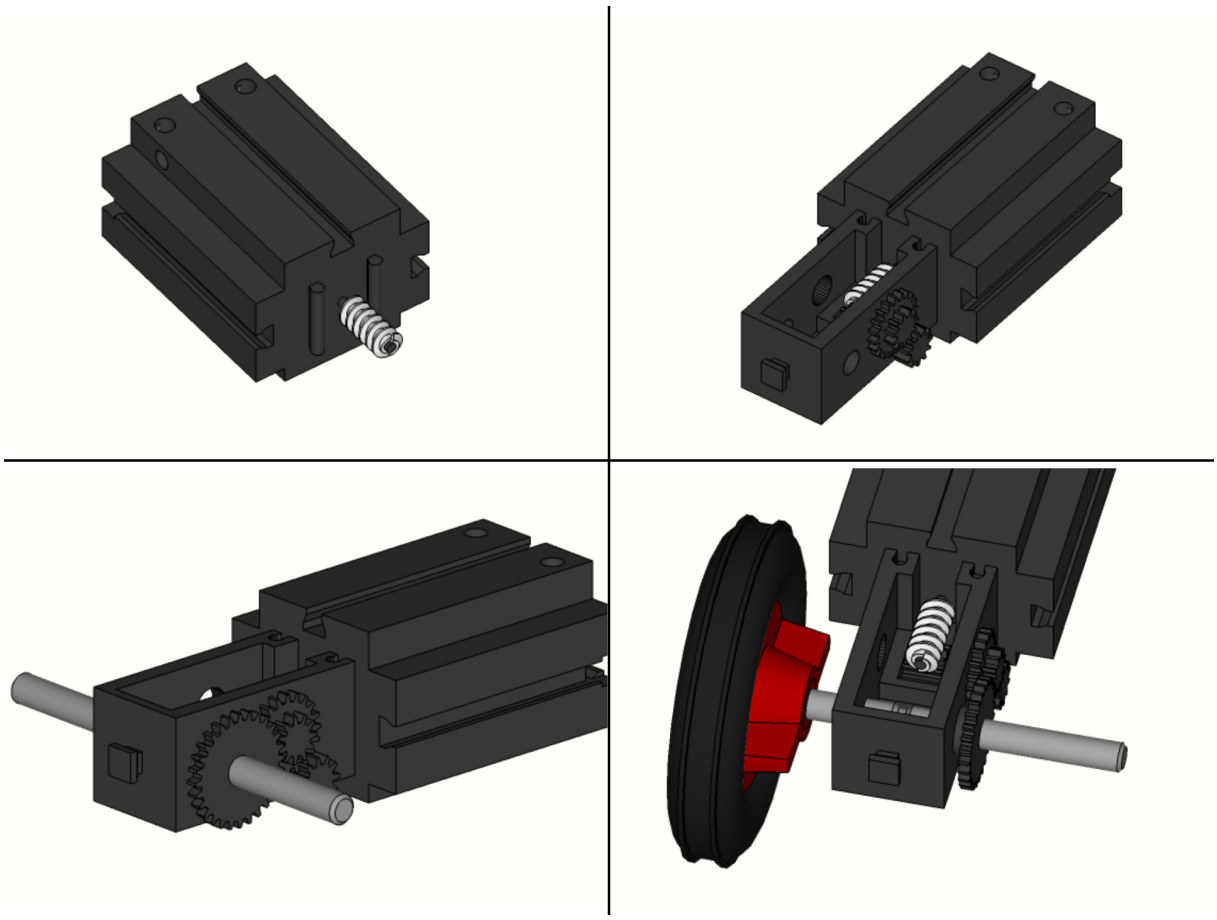

Figure 3.3: Adding the reductor to the motor.

3.4.2 Questions:

1. How difficult is to stop the shaft of the reductor in comparison to motor shaft?
2. How fast the shaft of the reductor is spinning in comparison to the shaft of the motor?
3. Are you able to freely rotate the shaft of the reductor by hand?
4. What happened with the produced mechanical power?
5. Try to calculate the geared ratio of the reductor.

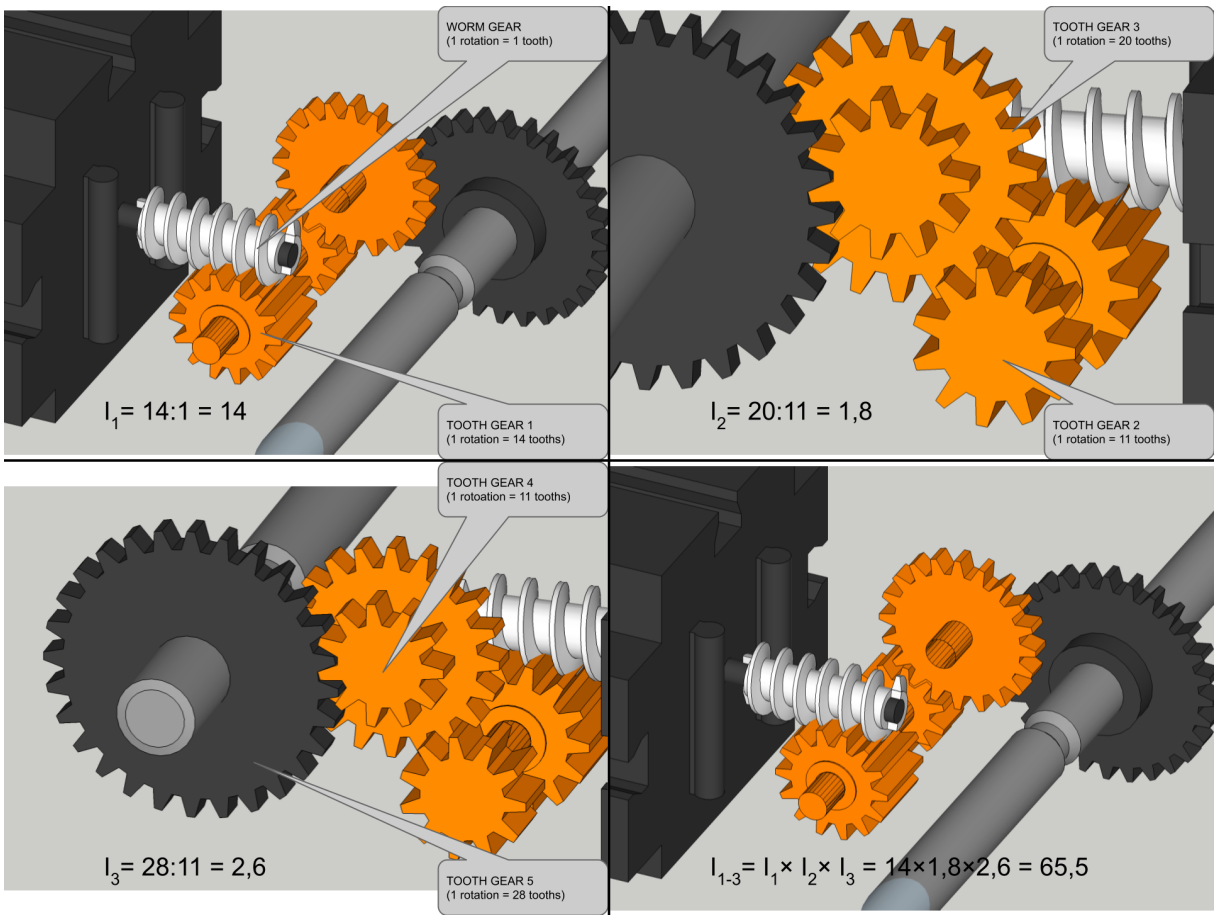

Figure 3.4: Gear ration calculation.

3.4.3 Summary

3.4.3.1 Gear ratio

The gear ratio describing the ratio between the angular velocity of input gear G1 and angular velocity of output gear G2 (see eq. 3.2).

$$i = \frac{\omega_1}{\omega_2}. \tag{3.2}$$

Because each gear moves tooth per tooth and if two touching gears have different numbers of teeths their's angular velocity will be different. In fact the anguar velocity will be inversely proportional as eq. 3.3 suggests:

$$\frac{\omega_1}{\omega_2} = \frac{N_2}{N_1} = i. \tag{3.3}$$

3.4.4 Issues

3.4.4.1 *The reductor's shaft is not spinning although the DC motor is working properly.*

Check if the reductor is attached all the way to the motor. Check if the worm gear of the motor is in contact with first gear of the roductor.

3.5 Robot construction

Building a functional robot is a crucial step in understanding robotics hardware. In this learning unit, we will focus on assembling a mobile robot using **Fischertechnik** components, a modular construction system widely used in robotics education.

Why Fischertechnik?

Fischertechnik is a **high-quality, engineering-focused construction system** designed to teach mechanical and technical principles. Unlike simpler building kits, such as LEGO, Fischertechnik provides **greater mechanical accuracy and modularity**, making it ideal for robotics prototyping.

- **Mechanical Stability** – The system uses interlocking parts that provide excellent structural integrity, ensuring that robots remain durable even in moving applications.

- **Precision and Functionality** – Fischertechnik components include gears, axles, sensors, and motors, allowing students to construct robots that function similarly to real-world industrial machines.

- **Realistic Engineering Design** – Unlike snap-together toys, Fischertechnik encourages **engineering-oriented thinking**, requiring students to assemble parts in ways that mimic professional robotics design.

- **Integration with Electronics** – Fischertechnik models can incorporate **microcontrollers (e.g., RobDuino, Arduino)** and sensors, bridging the gap between mechanical assembly and programming.

- **Flexibility for Modifications** – The modular nature of the system allows for easy redesigns and upgrades, making it perfect for experimentation.

By using Fischertechnik, students **develop hands-on problem-solving skills**, reinforce their understanding of mechanics, and gain practical experience in constructing **stable, functional robotic systems**.

In the next steps, you will **follow a structured assembly guide** to construct your first mobile robot, install its power source, and integrate the **RobDuino controller**, which will later be programmed to perform autonomous tasks.

3.5.1 Task: Assemble robot construction

1. Construct the mobile robot according to this sequences on the fig. 3.5.

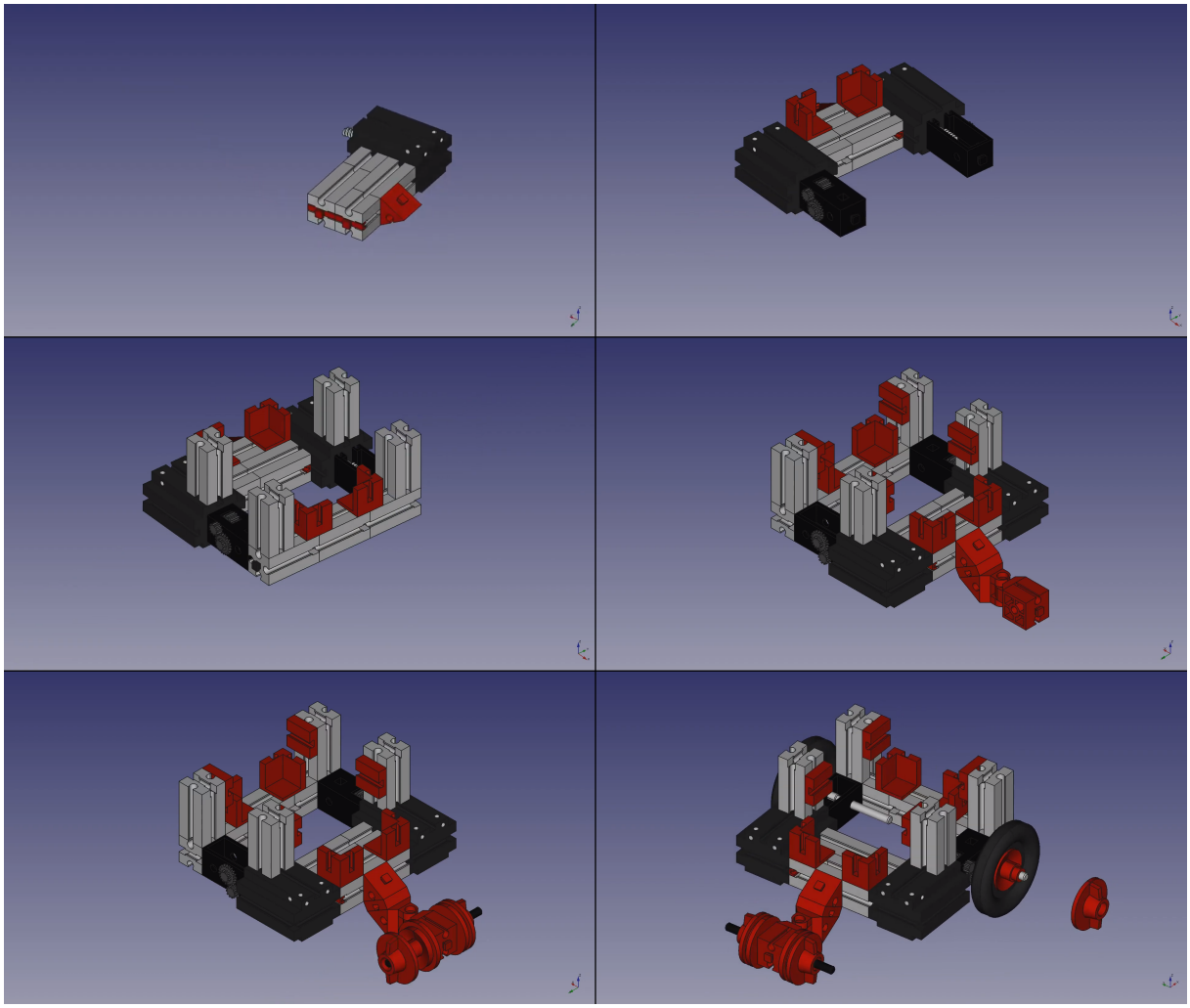

Figure 3.5: Construction sequences.

Or you can follow the video instructions

2. Add the battery between the red cornered bricks. The connector shuld be pointing to the back of the robot.
3. Add also the RobDuino controller. Clip the controller between the grey upstanding bricks.

3.5.2 Questions:

1. Where do you think is th front side of the robot?
2. Are you able to rotate the wheels freely by hand?

3.5.3 Summary:

<++>

3.5.4 Issues:

3.5.5 Issues:

Incorrect Height Adjustment of the Rear Support Wheel
- Ensure that the **rear support wheel is positioned at the correct height** so that its **rotation axis remains perfectly vertical**.
- An incorrect height may cause **unwanted tilting**, leading to unstable movement.

Misalignment of Wheels and Axles
- Check that the **driving wheels** are **securely attached** and properly aligned with the chassis.
- A misaligned axle can cause uneven movement or excessive friction, reducing efficiency.

Loose or Improperly Fastened Components
- Ensure all structural components, including the **RobDuino controller and battery**, are **firmly clipped into place**.
- Loose parts may cause mechanical instability, electrical disconnections, or unexpected behavior.

Battery Orientation and Connector Placement
- The **battery should be correctly positioned** between the red cornered bricks, with the **connector facing the back of the robot**.
- Incorrect placement may make wiring difficult or even prevent proper power delivery.

Wheel Rotation and Friction Issues
- Before finalizing assembly, **manually rotate the wheels** to ensure they spin **freely** without obstruc-

tion.
- If the wheels do not rotate smoothly, check for excessive tension in the axle or misaligned parts.

By carefully addressing these points during assembly, you will **improve the robot's stability, movement, and overall performance**, ensuring a successful build.

3.6 Understanding basic robot movement

3.6.1 Tasks: Make robot move

1. Connest both DC motors to RobDuino controller according to tbl. 3.2:

Table 3.2: Motors connections to RobDuino Output pins.

MOTOR	RobDuino Output pins
Left DC Motor - con. A	D7
Left DC Motor - con. B	D6
Right DC Motor - con. A	D5
Right DC Motor - con. B	D4

2. Write simple programming instructions to move the robot forward. Make right sequence of programming instructions (e.g. `digitalWrite()` and `delay(time_in_ms)` functions) to achive:

    1. move the robot forward,
    2. do it for 3000 ms and
    3. stop the robot.

3.6.2 Questions:

You probably ended up with something like prog. example 3.2:

Listing 3.2: First moves.

```
  void setup()
  {
    pinMode(4, OUTPUT);
    pinMode(5, OUTPUT);
    pinMode(6, OUTPUT);
    pinMode(7, OUTPUT);

    digitalWrite(7, HIGH);
    digitalWrite(6, LOW);
    digitalWrite(5, HIGH);
    digitalWrite(4, LOW);

    delay(3000);

    digitalWrite(7, LOW);
    digitalWrite(6, LOW);
    digitalWrite(5, LOW);
    digitalWrite(4, LOW);
  }

  void loop()
  {

  }
```

1. Is this code “easy readable”?
2. Why is readable code important?

3.6.3 Summary:

3.6.3.1 <++>

3.6.4 Issues:

3.6.4.1 <++>

3.7 Sensors and actuators

4 ELECTRONICS FUNDAMENTALS

Whether you're a curious hobbyist or aspiring engineer, learning the fundamentals of electronics is a crucial step towards understanding and building robots. Electronics is the backbone of robotic systems, providing the necessary control and communication between various components.

In this introduction, we'll explore the basic concepts of electronics that are essential for robotics. We'll cover topics such as circuits, components, sensors, actuators, and microcontrollers. By the end of this guide, you'll have a solid foundation to dive deeper into the world of robotics.

Components: Electronic components are the building blocks of circuits. Resistors control the flow of current, capacitors store electrical charge, and diodes allow current to flow in only one direction. Other components, like transistors and integrated circuits (ICs), provide amplification and complex functionalities. Familiarizing yourself with these components will enable you to construct and manipulate electronic circuits.

Circuits: At the heart of electronics lies the concept of circuits. A circuit is a path through which electric current flows. It consists of various components, such as resistors, capacitors, and diodes, connected by conductive wires. Understanding how circuits work is vital to designing and troubleshooting robotic systems. Central to understanding and designing these systems are the basic principles of electricity and electronics. In next chapters we will dive these principles, focusing on:

1. Ohm's Law,
2. Kirchhoff's Current Rule and
3. Kirchhoff's Voltage Rule,

and illustrates each with practical examples relevant to robotic device.

Reading sensor's values: Sensors are essential for robots to perceive their environment. They convert physical quantities, such as temperature, light, sound, or distance, into electrical signals. Common types of sensors include proximity sensors, temperature sensors, accelerometers, and cameras. By integrating sensors into your robot, you can gather valuable data to make informed decisions and enable autonomous behavior.

Controlling Actuators: Actuators are responsible for physical movement in robots. They convert electrical energy into mechanical motion. Examples of actuators include motors, servos, solenoids, and pneumatics. Actuators allow robots to perform tasks such as locomotion, gripping objects, or

manipulating their environment. Understanding how to control and interface with actuators is crucial for creating dynamic and interactive robots.

Microcontrollers: Microcontrollers are the brains of many robotic systems. They are small, programmable devices that provide computing power and control to robots. Microcontrollers can read sensor inputs, process data, and send commands to actuators. Arduino and Raspberry Pi are popular microcontroller platforms used in robotics. Learning to program microcontrollers will unlock endless possibilities for your robotic creations.

As you embark on your journey into robotics, keep in mind that electronics is a vast and evolving field. It requires a combination of theoretical knowledge and hands-on experience. Experimentation and continuous learning will be your allies in mastering electronics fundamentals in robotics.

Now that you have a glimpse into the foundational aspects of electronics for robotics, you're ready to dive deeper into each topic. Explore tutorials, online resources, and hands-on projects to further expand your knowledge. The more you learn and practice, the more you'll be able to bring your robotic ideas to life.

Remember, robotics is an exciting and interdisciplinary field that combines electronics, mechanics, programming, and more. So, have fun, stay curious, and let your creativity guide you as you explore the world of robotics!

4.1 Basic circuit components

4.1.1 Resistors

4.1.2 Diodes

4.1.3 Power source

When it comes to powering an Arduino UNO controller for robotics projects, there are several options available depending on the specific requirements of your project. Here are some common power supply options:

1. USB Cable: The simplest and most common way to power an Arduino UNO is through a USB cable connected to a computer or a USB power source, such as a wall adapter or power bank. This is convenient for testing and prototyping, but it may not be suitable for mobile or standalone robot applications.
2. External Power Supply: The Arduino UNO can also be powered by an external power supply connected to its power jack. The board accepts a voltage range of 7 to 12 volts. You can use a

DC power adapter or a battery pack with the appropriate voltage rating. Make sure the power supply can provide enough current to meet the requirements of your project.

3. 9V Battery: Another option is to power the Arduino UNO using a 9V battery. You can connect the battery to the power jack or use a battery clip to connect it to the Vin (voltage input) and GND (ground) pins on the Arduino board. Keep in mind that a 9V battery may not provide sufficient power for more demanding robotic applications.

4. LiPo Battery: For mobile or portable robot projects, lithium polymer (LiPo) batteries are a popular choice. LiPo batteries provide higher energy density and can deliver the necessary current for driving motors and other power-hungry components. However, you will need additional circuitry, such as voltage regulators and protection circuits, to ensure proper voltage levels and prevent overcharging or over-discharging of the battery.

When choosing a power supply, consider the voltage and current requirements of your Arduino UNO and the peripherals connected to it, such as motors, sensors, and other components. Ensure that the power supply can provide enough current and voltage stability for your specific project needs.

Always prioritize safety when working with power supplies. Use appropriate connectors, check polarity, and follow proper wiring practices to prevent short circuits or damage to your Arduino UNO and other components.

4.1.3.1 Battery UPS power supply

We utilize Uninterruptible Power Supply (UPS) power supply units such as the one available on AliExpress (see fig. 4.1). These UPS units are specifically employed for providing power to simple mobile robots. They offer a cost-effective solution, allowing us to ensure uninterrupted power supply to the robots' systems. The chosen UPS units from AliExpress are reliable and affordable, making them an ideal choice for our requirements.

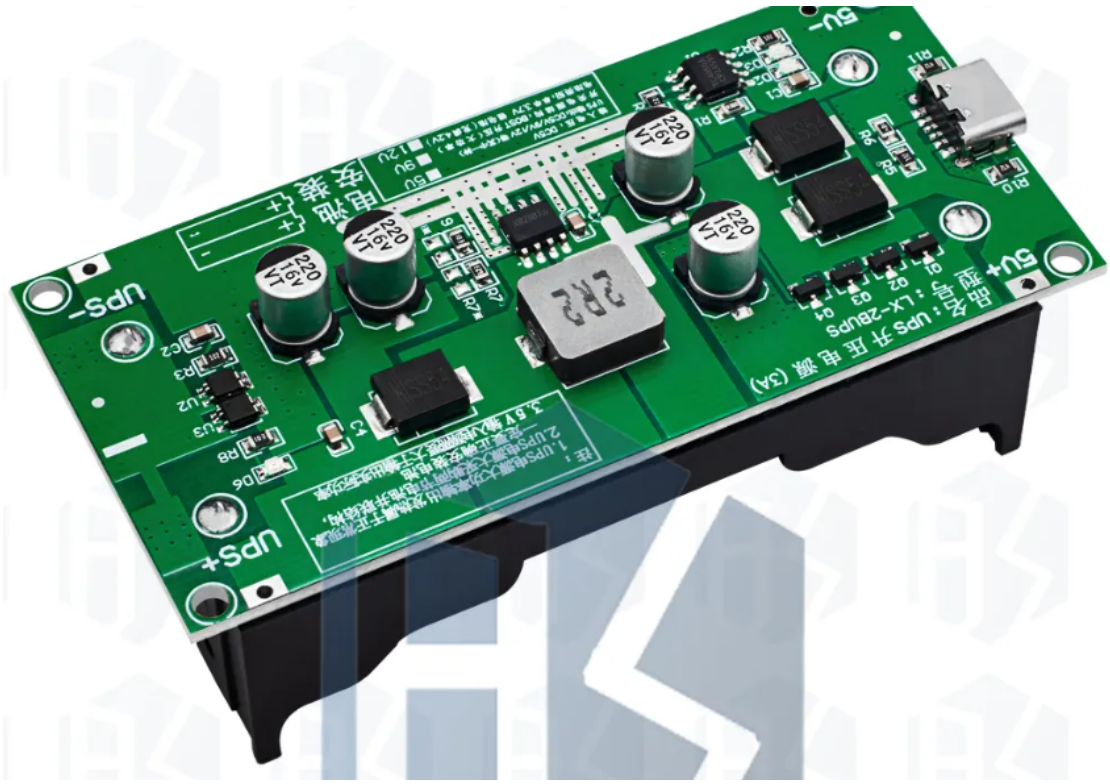

Figure 4.1: UPS power supply[1].

The UPS described in the provided schema (fig. 4.2) is designed in such a way that the output voltage is controlled by a resistor divider circuit consisting of resistors R7 and R9.

[1]Source: https://www.aliexpress.com/item/1005005452676689.html?spm=a2g0o.productlist.main.19.455b3926DHH1L4&algo_pvid=de392f56-63b1-4837-96de-e710e8a0eb9a&aem_p4p_detail=202311030140378048689945398110001719497&search_p4p_id=20231103014037804868

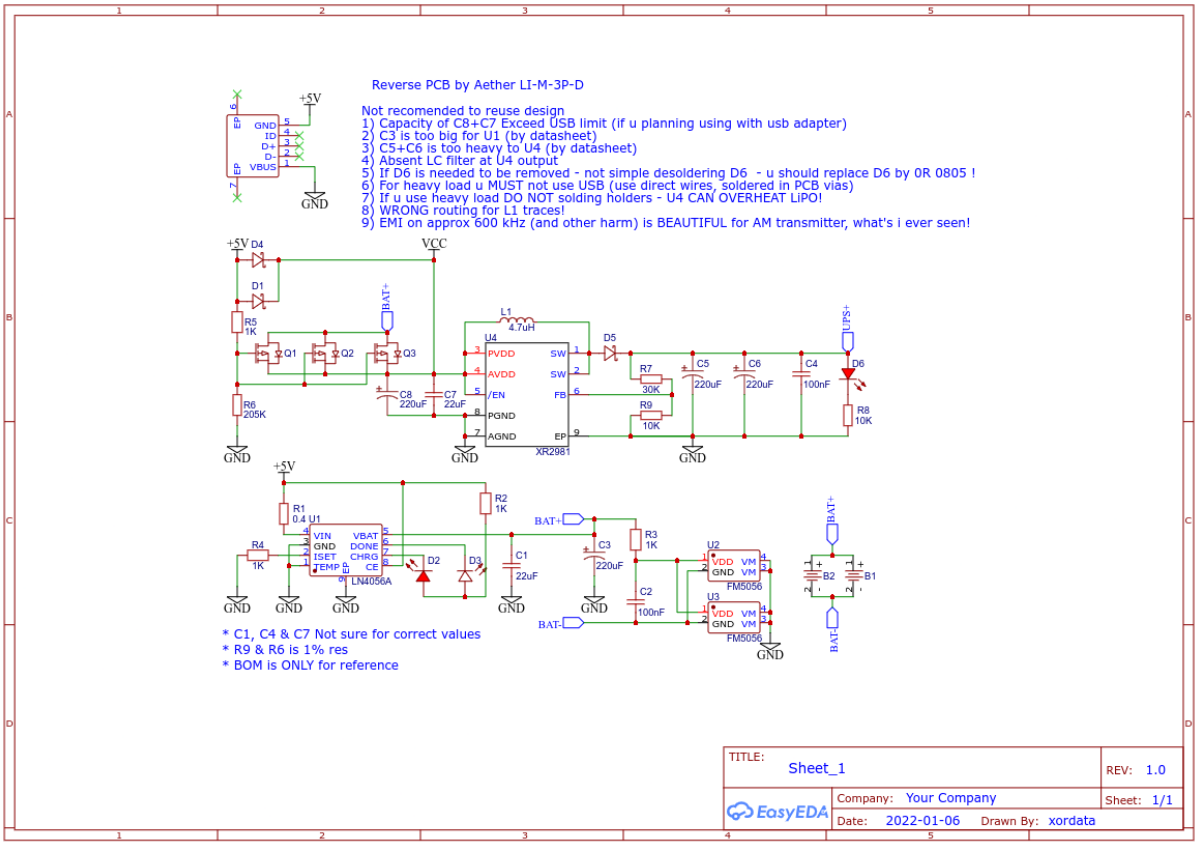

Figure 4.2: Schematic of UPS[2].

In this UPS schema, the resistors R7 and R9 are chosen in a way that their ratio divides the voltage proportionally to achieve the desired output voltage. By adjusting the values of these resistors, the output voltage can be regulated.

In addition to the regular setup, we incorporated an extra switch between the BAT+ (battery positive) and R3 resistor in power supply system. This switch serves the purpose of powering off the UPS (uninterruptible power supply). This feature provides convenience as it allows us to easily turn off the mobile robot and put the UPS into charging mode. By using this switch, we can efficiently manage the power supply to the robot and ensure that the UPS is charged when not in use.

4.2 Ohm's Law

Ohm's Law is a foundational principle in the field of electronics, stating the relationship between voltage, current, and resistance in an electrical circuit. It is succinctly expressed as eq. 4.1:

[2]Source: https://oshwlab.com/xordata/aether-li-m-3p-d

$$I = \frac{R}{V}, \tag{4.1}$$

where:

- I is the current flowing through the circuit (in amperes), and
- V is the voltage across the circuit (in volts),
- R is the resistance (in ohms).

Practical Example in Robotics:

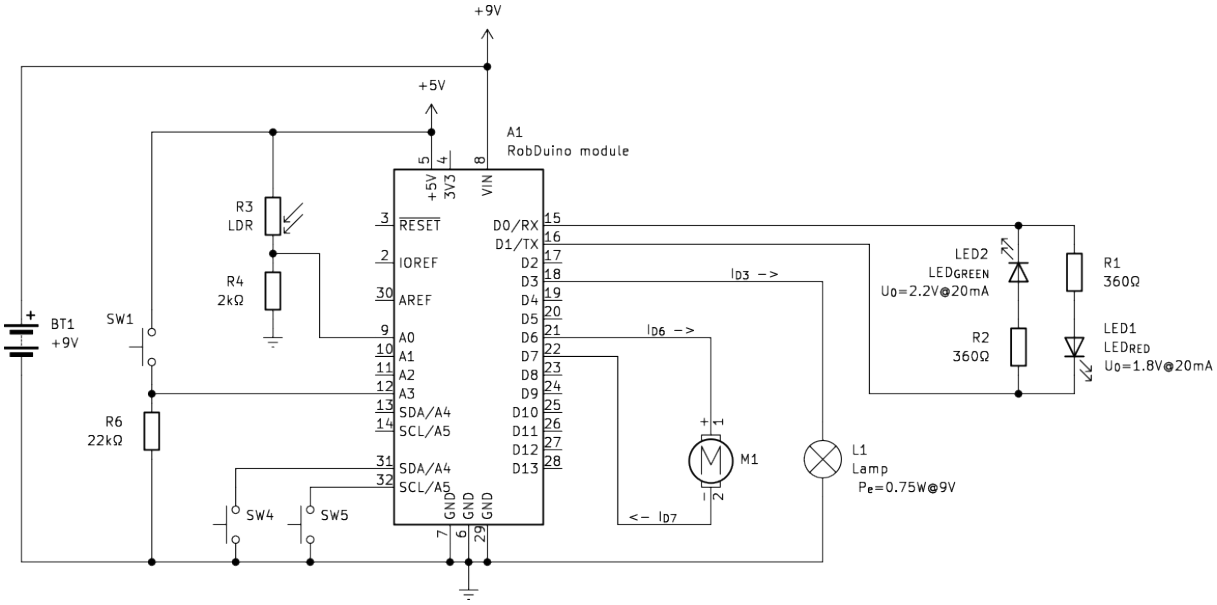

Figure 4.3: Example of tipical electrical scheme of robotic device with one degree of freedom.

1. Consider a simple robotic arm that uses a DC motor for movement (fig. 4.3). If the motor has a resistance of 20Ω and is connected to a 9V power supply, Ohm's Law can determine the current flowing through the motor:

$$I_{motor} = \frac{9V}{10\Omega} = 450mA. \tag{4.2}$$

Understanding this helps in selecting the right power source and ensuring that the motor and control electronics are compatible, preventing overheating and damage.

2. To apply Ohm's Law in calculating the current flowing through a light lamp with a power rating of 0.75W at a supply voltage of 9V, and connected to a digital output (D3), we start by understanding the relationship between power, voltage, and current. Ohm's Law is traditionally stated as eq. 4.1, but we can also express electrical power (Pe) in terms of voltage and current as eq. 4.3:

$$P_e = VI. \tag{4.3}$$

Since we are again interested in electrical current trough lamp we can fill in the data:

$$I_{D3} = \frac{P_e}{V} = \frac{0.75W}{9V} = 83mA. \tag{4.4}$$

4.2.1 Questions

1. Calculate electrical current trough resistor R_1 if the voltage across it is $U_{R_1} = 7.2V$!
2. Calculate the current trough resistor R_4 if measured voltage potential on A_0 pin is $V_{A_0} = 2V$!

4.3 Kirchhoff's Current Rule

Kirchhoff's Current Rule, also known as the first Kirchhoff ~~law~~ rule, states that the total current entering a junction in a circuit equals the total current leaving the junction. This law is based on the principle of conservation of charge and is expressed with eq. 4.5:

$$I_{x_1} + I_{x_2} + ... = I_{y_1} + I_{y_2} + I_{y_3} + ... \tag{4.5}$$

where:

- electrical currents with index I_x are entering currents and
- currents with index I_y are leaving junction currents.

We will explain the Kirchhoff's current rule on the same example shown in fig. 4.4

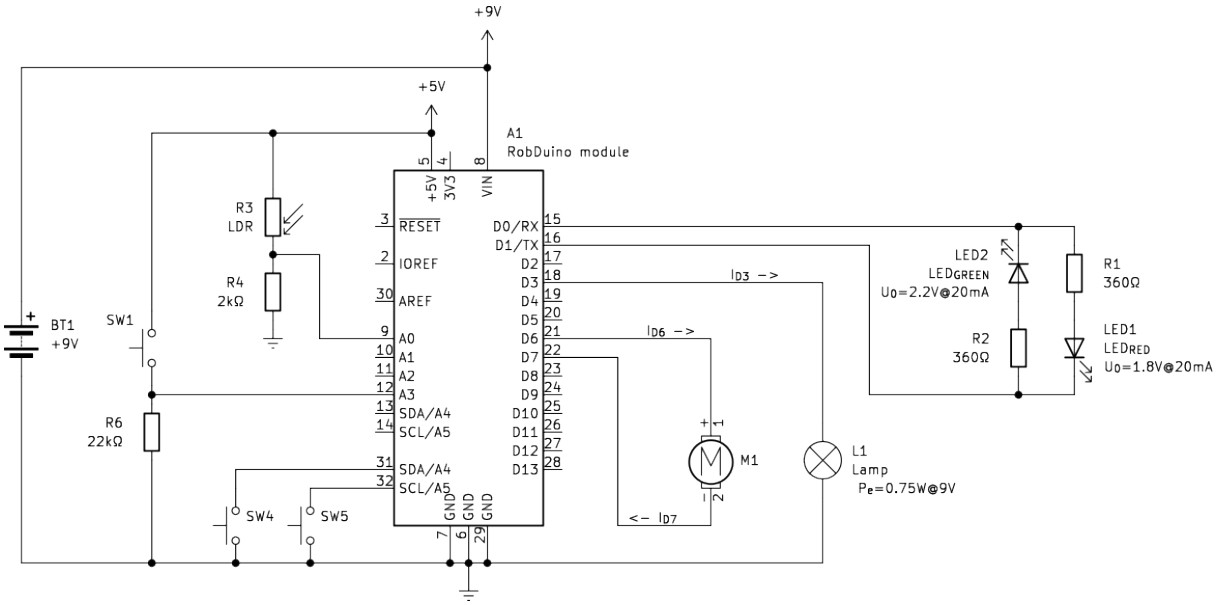

Figure 4.4: Electrical sheme of robotic device.

Practical Example in Robotics:

Imagine a robotic hand with multiple sensors (e.g., touch sensor and light sensor) connected to a single microcontroller. If the sensors draw 0.23 mA (when SW_1 is closed) and 1.0 mA, and they are all connected to the same power supply junction, the total current entering the junction is:

$$I_{tot} = I_{tch} + I_{light} = 0.23mA + 1.0mA = 1.23mA \tag{4.6}$$

This information is critical for designing the power distribution network of the robot, ensuring that the power supply can handle the total current draw.

4.3.1 Questions

1. What is the total current of actuators (motor, light bulb, LEDs) when they are all on?
2. Current into input pin A_0 is approximately $I_{A_0} = 20nA$. Compare this current to other two currents at the middle junction in the light sensor. Can it be ignored?

4.4 Kirchhoff's Voltage Rule

Kirchhoff's Voltage Rule (KVR), or the second Kirchhoff rule, states that the sum of all electrical potential differences around any closed network (or loop) is zero. This law is grounded in the conservation of

energy principle and is expressed wiht eq. 4.7

$$+U_1 - U_2 + ... + U_n = 0 \tag{4.7}$$

where:

- voltage is positive if voltage potential increases in the selected direction (e.g. U_1) and
- voltage is negative if voltage potential decreases in this direction (e.g. U_2).

Practical Example in Robotics:

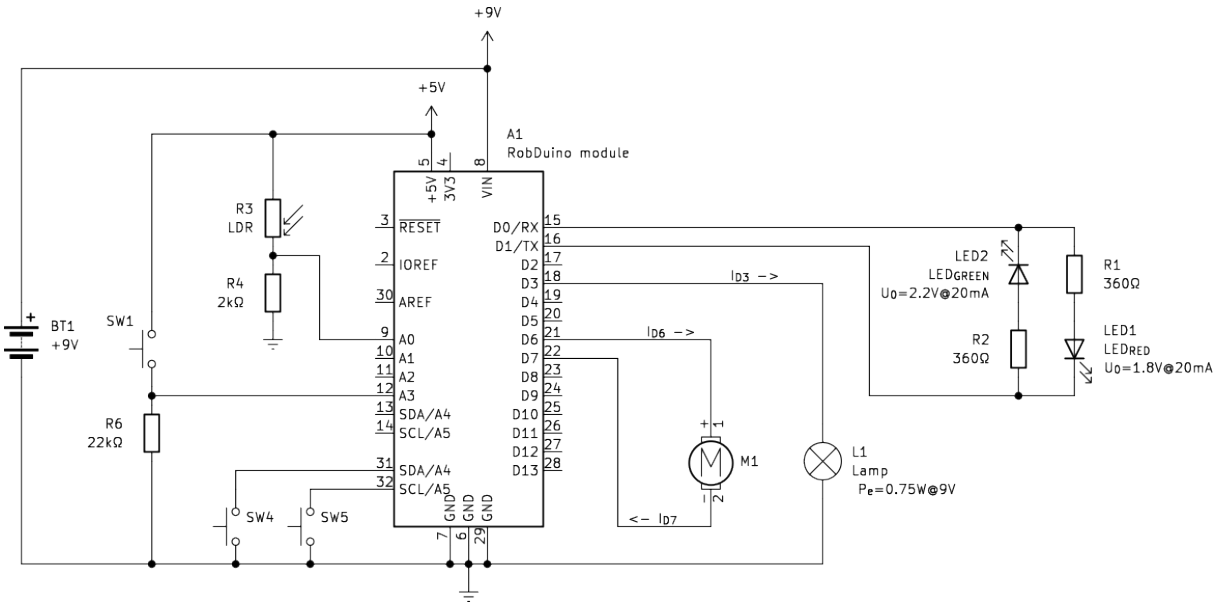

Figure 4.5: Scheme of robotic device.

Consider a circuit in a robotic device that includes an LED circuit as a signal light (e.g., $D_0 \rightarrow R_1 \rightarrow LED_1 \rightarrow D_1$. If we assume that across output pins $D1$ and D_0 is a positive voltage potential difference $U_{(D_1-D_0)} = 9V$, according to KVR, we can write an eq. 4.8 for this loop:

$$U_{(D_1-D_0)} - U_{R_1} - U_{LED_1} = 0V \tag{4.8}$$

Rearranging the eq. 4.8 we can calculate the voltage across resistor R_1:

$$U_{R_1} = U_{(D_1-D_0)} - U_{LED_1} = 9V - 1.8V = 7.2V \tag{4.9}$$

This ensures that the energy supplied by the controller is completely used by the resistor and led.

Kirchhoff's voltage rule is instrumental in analyzing and designing circuits for energy efficiency and proper component operation in robotics.

Understanding the basic principles of electricity and electronics, epitomized by Ohm's Law and Kirchhoff's rules, is crucial for anyone venturing into robotics. These principles not only guide the design and analysis of robotic systems but also ensure their safe and efficient operation. By applying these laws, we can predict how circuits will behave under various conditions, optimize energy consumption, and troubleshoot potential issues, laying the groundwork for more advanced explorations into the electrifying world of robotics.

4.4.1 Questions

1. Calculate the voltage across resistor R_2 when voltage potencial of $V_{D0} = 0V$ and voltage potencial of $V_{D1} = 9V$!
2. What is the voltage across resistor R_3 if we measured voltage potencial $V_{A_0} = 2V$ at the input pin A_0?

4.5 Digital output

On an Arduino Uno board, a digital output is a pin that can be used to output a digital signal, which can be either high (5 volts) or low (0 volts). Digital outputs are useful for controlling devices that are either on or off, such as LEDs, motors, and relays.

To use a digital output on an Arduino Uno board, you will need to specify which pin you want to use as an output in your code. You can do this using the pinMode function, which takes two arguments: the pin number and the mode (OUTPUT or INPUT). For example, the following code sets digital pin 13 as an output:

```
1    pinMode(13, OUTPUT);
```

Once you have set a pin as an output, you can use the digitalWrite function to set the pin to either a high or low state. For example, the following code sets digital pin 13 to a high state:

```
1    digitalWrite(13, HIGH);
```

4.6 Digital input

> to-do

1. Push Button: a push button can be used to trigger a digital input. By connecting a push button to an Arduino digital pin and writing a sketch to register when the button is pressed, digital input can be used to trigger an action.
2. Touch Sensor: a touch sensor can be used to detect contact with a particular surface and can act as a digital input. By connecting the sensor to an Arduino digital pin and writing a sketch to listen for contact, digital input can be used to trigger an action.
3. Light Sensor: a light sensor can be used to detect light levels and can act as a digital input. By connecting the sensor to an Arduino digital pin and writing a sketch to listen for changes in light levels, digital input can be used to trigger an action."

5 INTRODUCTION TO C++

C++ is a high-performance programming language that is widely used for building software applications. It was developed by Bjarne Stroustrup in 1979 as an extension of the C programming language. C++ is an object-oriented language, which means that it provides features for organizing and modularizing code in the form of "objects." C++ is also a compiled language, which means that the source code is converted into machine code by a compiler before it can be run on a computer.

Here are some basic concepts in C++:

Variables: A variable is a named location in memory that stores a value. In C++, you must specify the data type of a variable when you declare it. For example:

```
int x;      // declares a variable x of type int
float y;    // declares a variable y of type float
char c;     // declares a variable c of type char
```

Operators: Operators are special symbols that perform specific operations on one or more operands. C++ has a variety of operators, including arithmetic operators (e.g., +, -, *, /), comparison operators (e.g., ==, !=, >, <), and logical operators (e.g., &&, ||, !).

Control structures: Control structures are statements that control the flow of execution in a program. C++ has several types of control structures, including if statements, for loops, and while loops.

Functions: A function is a block of code that performs a specific task. C++ has a large standard library of functions, and you can also define your own functions. A function definition has the following syntax:

```
return_type function_name(parameter list) {
   // function body
}
```

Object-oriented programming: As I mentioned earlier, C++ is an object-oriented language, which means that it provides features for organizing and modularizing code in the form of "objects." An object is a self-contained unit of code that represents a real-world entity, such as a person, a car, or a bank account. Objects have attributes (data) and behaviors (functions). In C++, you can define classes to create objects.

5.1 Basic syntax and structure of a C++

A C++ program begins with preprocessor directives, an example of which is including header files. Preprocessor directives provide instructions to the compiler and tell it what additional files to include in the compilation process.

Following the preprocessor directives are declarations, which include variables, constants, and user-defined functions.

The main function is the entry point of any C++ program, and contains all of the program's executable code. Within the main function are more definitions, which are additional declarations of data types, variables, constants, and user-defined functions. Finally, the program is concluded with the return 0 statement, indicating success.

Developing a C++ program requires careful attention to the order in which the preprocessor directives, declarations, main function, and definitions are written. Only by understanding the basic structure of a C++ program can a programmer write effective, efficient, and bug-free code.

Here is an example of a basic C++ program that blinks LED on a 13-th pin of an Arduino Uno controller and can be written in Arguing IDE:

Listing 5.1: Native C++ program for ATmega328.

```
#include <avr/io.h>
#include <util/delay.h>
int time_ms = 1000;     // Variable declaration
void setup();           // Function declaration
void loop();

int main()
{
    setup();            // Function call
    while (true){       // Main LOOP
        loop();
    }
    return 0;
}
void setup() {          // Function definition
    PDDB |= (1<<PINB5);
}
void loop(){
    PORTB |= (1<<PINB5);
    _delay_ms(time_ms);
    PORTB &= !(1<<PINB5);
    _delay_ms(time_ms);
}
```

Programming an Arduino Uno board in native C++ is much more difficult than in Arduino IDE. Arduino IDE makes it easier for users to write and debug code without having to know the details of the underlying hardware. In addition, the IDE provides many additional functions which simplify the usage of additional peripherals and actuators such as serial communication, LCDs, servo motors, step motors… This is especially true and important for beginners.

5.1.1 Task 1: Writing a Basic C++ Program for Arduino (Without Arduino IDE Functions)

To better understand the fundamentals of C++ before using Arduino-specific functions, write a simple C++ program that runs on an Arduino board but does **not** rely on `setup()` and `loop()`. The program should:

- Blink an LED connected to **pin 13** of an Arduino Uno.
- Use **direct register manipulation** instead of `digitalWrite()`.
- Implement a **main function** as in standard C++ programs.

Modify and complete the following template:

```
#include <avr/io.h>
#include <util/delay.h>

int main() {
    DDRB |= (1 << PINB5); // Set pin 13 as output

    while (true) {
        PORTB |= (1 << PINB5);  // Turn LED on
        _delay_ms(500);
        PORTB &= ~(1 << PINB5); // Turn LED off
        _delay_ms(500);
    }

    return 0;
}
```

5.1.2 Questions to consider:

- Why does this program use `DDRB`, `PORTB`, and `PINB5` instead of `pinMode()` and `digitalWrite()`?
- How does the `while(true)` loop function compared to `loop()` in Arduino IDE?
- What happens if the `return 0;` line is removed?

5.1.3 Task 2: Analyzing the Structure of a Native C++ Arduino Program

Use the provided **native C++ code for Arduino** (from the learning material) and identify the following elements **by line number**:

- Preprocessor Directive (`#include` statement)
- Variable Declaration
- Function Declaration
- Function Definition
- Function Call
- Main Function
- Comment

By completing these tasks, you will gain a deeper understanding of **how C++ works independently of Arduino's simplified environment**, giving you a solid foundation for more advanced programming.

5.1.4 Summary:

5.1.4.1 <++>

5.1.5 Issues:

5.1.5.1 Not including a semicolon at the end of each statement:

Every statement in C++ must end with a semicolon. If a semicolon is omitted, the code will not compile correctly.

5.1.5.2 Not properly formatting the code:

Properly indenting and spacing code is important in C++ to make the code easier to read. Not formatting the code correctly can lead to syntactical errors.

5.1.5.3 Not using correct capitalization:

C++ is a case sensitive language and therefore proper capitalization is important. If the wrong capitalization is used, it can lead to syntax errors.

5.2 Writing clean code

The Arduino IDE (Integrated Development Environment) has contributed to clean and readable code by promoting a simple and structured coding approach. One of the ways it achieves this is by introducing the `setup` and `loop` functions.

The "setup" function is called only once when the Arduino board starts up. It is typically used for initializing variables, setting pin modes, and configuring any necessary settings. By separating this initialization code into a designated function, it becomes easier to identify and understand the setup process, making the code more organized and readable.

The "loop" function is called repeatedly after the setup function. This function contains the main logic of the program, where actions are performed continuously. By having a dedicated loop function, it enhances the readability of the code as it clarifies the flow of execution.

The Arduino IDE enforces the presence of these two functions, which serves as a guideline for developers to structure their code. This standardization promotes cleaner code because it encourages encapsulating specific functionalities in separate functions. This helps to compartmentalize different tasks and improves code modularity and maintainability.

Additionally, the Arduino IDE provides various built-in libraries and functions that simplify common tasks, such as reading sensor values or controlling actuators. These libraries follow consistent naming conventions and provide well-documented APIs, making it easier for developers to understand and use them. By leveraging these libraries, developers can write cleaner and more readable code, as they can focus on the high-level functionality and rely on the underlying library for the low-level details.

In order to make your code readable you have to clean your code regularly. This step is very important to not to slow down the programming process in the future programming. You will probably spent the same amount of time cleaning the code that you needed for writing a working version.

In general you can follow some rules:

1. Use FUNCTIONS for every single action,
2. COMMENT the code only where is necessary,
3. Use EXPLANATORY CONSTANTS and VARIABLES

to make your code clean.

Our aim is to write more readable code like in prog. example 5.2:

Listing 5.2: Writing Clean Code.

```
#include "RobotMoves.h"
void setup()
{
  setIOpins();
  moveForward();
  delay(3000);
  stopTheRobot();
}

void loop()
{

}
```

... we will do it in several steps.

5.2.1 Tasks:

1. Write programming functions for moving the robot in several dirrections:
    1. `moveForward()`,
    2. `moveLeft()`,
    3. `moveRight()`,
    4. `moveBackward()`,
    5. `stopTheRobot()`.
2. Save all the functions into header file: `RobotMoves.h`. An example of header file is shown in prog. example 5.3

Listing 5.3: Robot Moves.

```
/*****************************
 * IO pins of the Robot
 *****************************/
const int LEFT_MOTOR_PIN_1  = 7;
const int LEFT_MOTOR_PIN_2  = 6;
const int RIGHT_MOTOR_PIN_2 = 5;
const int RIGHT_MOTOR_PIN_1 = 4;
/*****************************
 * Function declarations
 *****************************/
void setIOpins();
void moveForward();
/*****************************
 * Function definitions
 *****************************/
void setIOpins(){
  pinMode( LEFT_MOTOR_PIN_1, OUTPUT);
  pinMode( LEFT_MOTOR_PIN_2, OUTPUT);
  pinMode(RIGHT_MOTOR_PIN_1, OUTPUT);
  pinMode(RIGHT_MOTOR_PIN_2, OUTPUT);
}
void moveForward(){
  digitalWrite( LEFT_MOTOR_PIN_1, LOW);
  digitalWrite( LEFT_MOTOR_PIN_2, HIGH);
  digitalWrite(RIGHT_MOTOR_PIN_1, LOW);
  digitalWrite(RIGHT_MOTOR_PIN_2, HIGH);
}
```

5.2.2 Questions:

1. Explain why functions contribute to more readable code.
2. Why is good to use explanatory variables?
3. <++>

5.2.3 CLEAN CODE EXPLAINED

5.2.3.1 Comments - YES and NO

Comments are very helpful and necessary. Keep them short and meaningful whenever is needed. May also help during thinking process while beginning designing the code.

```
   // robot will go forward
   digitalWrite(7,HIGH);
   digitalWrite(6,LOW);
   digitalWrite(5,HIGH);
   digitalWrite(4,LOW);
```

Don't use comments where the code is self-explanatory, for example:

```
     delay(3000); //wait for 3000ms
```

5.2.3.2 Functions

Concatenate programming code into meaningful functions is a must! Previous example of code for `driving a robot forward` is very difficult to understand at first sight. We can make cleaner code as is shown in nex example where is easier to understand what-is-what:

```
   void robotForward()
   {
     digitalWrite(LEFT_MOTOR_PIN_1,HIGH);
     digitalWrite(LEFT_MOTOR_PIN_2,LOW);
     digitalWrite(RIGHT_MOTOR_PIN_1,HIGH);
     digitalWrite(RIGHT_MOTOR_PIN_2,LOW);
   }
```

Compact code is more understandable than large one, see next example:

```
   void setup()
   {
     setIOpins();
     moveForward();
     delay(3000);
     robotStop();
   }
```

5.2.3.2.1 Function declaration Function declaration is highly advisable since allow you a quick overview of available functions in a current file. It is like a table of functions with it's return type and parameters. All declarations are tipically found at the beginig of the file.

```
   void moveForward();
   void moveLeft();
   void moveLeft_PWM(int pwm_value);
```

5.2.3.2.2 Function Definition A function definition provides the actual body of the function.

```
1   void robotForward()
2   {
3     digitalWrite(LEFT_MOTOR_PIN_1,HIGH);
4     digitalWrite(LEFT_MOTOR_PIN_2,LOW);
5     digitalWrite(RIGHT_MOTOR_PIN_1,HIGH);
6     digitalWrite(RIGHT_MOTOR_PIN_2,LOW);
7   }
```

5.2.3.3 Constants

Use explanatory constants to more clearly represent unintuitive numbers or other abstract values. Use these constants instead of comments since these numbers will appear several times during programming code.

```
1   const int LEFT_MOTOR_PIN_1 = 7;
2   const int LEFT_MOTOR_PIN_2 = 6;
```

Now you can easily see why the pins are set as OUTPUT. Because there is `Left motor` attached.

```
1   void setIOpins()
2   {
3     pinMode(LEFT_MOTOR_PIN_1, OUTPUT);
4     pinMode(LEFT_MOTOR_PIN_2, OUTPUT);
5   }
```

5.2.3.4 Variables

Use explanatory variables to make if-statements easily readable and thus understandable. Make `boolean` variables as short statements with no inverting logic.

For example we will use the case where the robot should stop as soon it hits the obstacle with front bumper. The worst case scenario of the program could look like this (we have all done it at some point):

```
1   void loop()
2   {
3     if (digitalRead(A0) == FALSE){
4       digitalWrite(7, HIGH);
5       digitalWrite(6, LOW);
6       digitalWrite(5, HIGH);
7       digitalWrite(4, LOW);
8     }else{
9       digitalWrite(7, LOW);
10      digitalWrite(6, LOW);
11      digitalWrite(5, LOW);
12      digitalWrite(4, LOW);
13    }
14  }
```

And more clean representation of same functionality is shown in next example of the code. Line 3 is easy readable, simple, clear and easy understandable.

```
1   void loop()
2   {
3       int front_bumper_is_pressed = digitalRead(BUMPER_INPUT);
4       if (front_bumper_is_pressed) robotStop(); else robotForward();
5   }
```

5.2.3.5 Header files

To keep our main program file short and transparent as possible we can put supporting code (e.g. functions, settings, ...) into separate file and include it at the beginning of the main program. These files are called header files. We can write a function and save it into header file called “calculate.h”

```
1   int sumTwoNumbers(int A, int B)
2   {
3     return A+B;
4   }
```

In our main program we can include the header file and use the function:

```
1   #include "calculate.h"
2
3   int main()
4   {
5     int a = 5, b = 3;
6     int sum = sumTwoNumbers(a, b);
7   }
```

There are several reasons to use header files in C++:

Code organization: Header files allow you to organize your code into logical units, which can make it easier to understand and maintain. For example, you can use a header file to group together related function declarations, constants, and data types.

Code reuse: Header files can be included in multiple source files, which allows you to reuse the same code in multiple places without having to copy and paste it. This can save time and reduce the risk of errors.

Compilation speed: When you include a header file in a source file, the compiler does not need to recompile the code in the header file every time it compiles the source file. This can significantly improve the compilation speed of your program, especially if the header file contains a large amount of code.

Separation of interface and implementation: Header files can be used to separate the interface (the function declarations and data types that are visible to the rest of the program) from the implementation (the actual code that performs the tasks). This can make it easier to change the implementation of a module without affecting the rest of the program.

5.2.3.6 Pre-process

The preprocessors are the directives, which give instructions to the compiler to pre-process the information before actual compilation starts (e.g. `#include` is one of them). You can easily use as such text substitutions for more clear code reading.

```
1    #define LEFT_MOTOR_PIN_1 7
2    #define LEFT_MOTOR_PIN_2 6
```

Remember! `#define` is really a simple text substitution and is not type-safe. Furthermore, we have to be certain that our definition will not interfere with other code used outside of our scope e.g. `libraries`. The last example is not the best representation of `#define` usage. In these case the `const int` is more proper way to go (allowed type checking, debugging). But `#define` has other benefits where `const` can not be used.

5.2.3.6.1 Translations The substitutions can be used as a translation and simplification of code. Such code can be introduced to very young children to get involved in programming.

```
#define vkljuci_led digitalWrite(13,HIGH)
#define izkljuci_led digitalWrite(13, LOW)
#define pocakaj(time) delay(time)
void loop(){
  vkljuci_led;
  pocakaj(1000);
  izkljuci_led;
  pocakaj(1000);
}
```

5.2.3.6.2 Debugging You can even substitute function names e.g. `debug(txt)` with `Serial.println(txt)` and easily separate debugging code lines from necessary serial print of data.

```
#define debug(txt) Serial.porintln(txt)
void setup()
{
  Serial.begin(9600);
  debug("Running...")
}
void loop()
{
  unsigned long myTime = millis();
  Serial.println(myTime);
  delay(1000);
}
```

When we are done with programming and debugging is not needed anymore we can simply change `#define` line to nothing:

```
#define debug(txt)
```

And these programming sentences will not be used. More sophisticated example is shown where programmer can switch between debugging mode (with `#define DBG 1`) and normal operation (with `#define DBG 0`) where code statement `debug("Running...")` will not even compile into program.

```
#define DBG 1
#if DBG == 1
#define debug(txt) Serial.porintln(txt)
#else
#define debug(txt)
#endif
void setup()
{
  Serial.begin(9600);
  debug("Running...")
}
```

5.2.4 Summary:

5.2.5 Issues:

5.2.5.1 What is the difference between `const int` and #define?

`#define` is textual replacement, so it is as fast as it can get. Also it can save some RAM. The downside is that it's not type-safe.

`const` variables may or may not be replaced inline in the code. It is guaranteed to be type-safe though since it carries its own type with it.

5.3 Testing programming code

Testing code in Arduino is important because it helps to ensure that the code is working correctly and producing the desired results. Testing can help to catch bugs and errors in the code, and can also help to verify that the code is performing the tasks that it is intended to perform. By thoroughly testing code, you can improve the reliability and functionality of your Arduino projects.

The `Serial.println()` function is a useful tool for debugging Arduino code because it allows you to print information to the serial monitor, which can help you understand what your code is doing and troubleshoot any problems.

To use `Serial.println()` for debugging, you will need to include the Serial library at the top of your sketch and initialize the serial monitor using the `Serial.begin()` function. Then, you can use Serial.println() to print strings or variables to the serial monitor.

Here is an example of how to use `Serial.println()` for debugging in an Arduino sketch:

Listing 5.4: Testing programming code.

```
#include <RobotMovingFunctions.h>
//include <RobDuinoSerialTesting.h>

void setup()
{
  Serial.begin(115200);

  Serial.print("Setting IO pins .."); // Reporting start of function.
  setIOpins();                       // Function execution.
  Serial.println("..DONE");          // Reporting end of function.

  moveForward();
  delay(3000);
  stopTheRobot();

}

void loop() { }
```

To view the output of the Serial.println() statements, you will need to open the serial monitor in the Arduino IDE. You can do this by clicking on the “magnifying glass” icon in the top right corner of the window.

5.3.1 RobDuino Testing Mode

Since testing programming code and hardware is one of the key features in designing a robot it is recommended that testing functions are a part of your main program.

In further lectures we will be using more advances `Testing mode` where single digital outputs can be controlled; and inputs will be measured in digital and analog manner. This testing process is available if you have installed `RobDuino Library` (see Program Installing chapter). The testing mode will be trigged by the command `testing`. The output will show every output state:

```
******* Testing mode ********************
Dig. Out    Dig. In.    An.In.
D0 = 1      A0 = 0      A0 = 293
D1 = 0      A1 = 0      A1 = 334
D2 = 0      A2 = 0      A2 = 353
D3 = 0      A3 = 0      A3 = 369
D4 = 0      A4 = 0      A4 = 339
D5 = 0      A5 = 0      A5 = 264
D6 = 0
D7 = 0
----------------------------------------
```

5.3.2 Task: RobDuino module testing

3. Uncomment line 2 in prog. example 5.4:
   '#include <RobDuinoSerialTesting.h>
4. Explore testing functions with command `testing` writing it into Serial Monitor and you will get this respond:

```
*** Testing mode - menu - ******************
*    help - prints this text menu
*    D5   - toggles output state of D5
*    Dx   - toggles output state of any Dx,
*           x is any num. from 0 .. 13.
*    run  - toggles monitoring od I/O pins
*    exit - exits the Testing mode.
*----------------------------------------
Type any command to continue ...
```

5.3.3 Questions:

1. Explain why testing is important.
2. Describe the techniques of testing.
3. What parts of the robot should be tested regularly.

5.3.4 Summary:

5.3.4.1 Testing mode

5.3.5 Issues:

5.3.5.1 How can I get RobDuinoSerialTesting working.

Basically you need to do these steps:

1. install RobDuino Library
2. put this code at the top of your porgram:
   `#include <RobDuinoSerialTesting.h>`
3. Compile and write the porgram to your Arduino UNO controller
4. Open `Serial Monitor` window

5. and write `testing` command into prompt.

5.4 Flow control

Flow control in C++ programming is the mechanism that allows the execution path of a program to change based on conditions, loops, or jumps. It is fundamental to creating dynamic and responsive programs. The primary ways to control flow in C++ include:

- **Jump Statements**: Facilitate the control flow by jumping to other parts of the program. The break, continue, and goto statements are examples of jump statements.
- **Loop Statements**: Enable executing a block of code repeatedly as long as a condition remains true. C++ offers for, while, and do-while loops for this purpose.
- **Conditional Statements**: Direct the program flow based on boolean conditions. Examples include if, if-else, and switch statements.

The **`goto`** statement in C++ provides a way to jump to another part of the program, altering the normal sequential flow of execution. It's generally recommended to use goto sparingly, as it can make code harder to read and maintain, but it can be useful in certain contexts, such as breaking out of deeply nested loops.

5.4.1 Tasks:

1. Mark the moving instructions with the label `repeating_moves:`.
2. At the end of the moves put the **`goto`** statement and jump to repeating_moves label.

Listing 5.5: Flow control with goto statement.

```
#include <RobotMovingFunctions.h>

void setup()
{
  setIOpins();

repeating_moves:
  moveForeward();
  delay(1000);
  moveLeft();
  delay(550);
  robotStop();
  delay(1000);
goto repeating_moves;

}

void loop()
{

}
```

5.4.2 Questions:

1. Why is using goto statement not the best programming practice.
2. Which form two is programming instruction: a) repeating_moves or b) goto repeating_moves, and ; is needed?

5.4.3 Summary:

The goto statement in C++ programming is a control flow instruction that allows the program to jump to another point in the code. It is used to transfer control to a labeled statement within the same function. Despite its capability to alter the execution flow in a very straightforward manner, goto is often discouraged in modern programming practices due to several reasons:

- **Readability**: Frequent use of goto can make code difficult to read and understand. It breaks the structured programming paradigm, making the flow of execution non-linear and less predictable.
- **Maintainability**: Programs that rely on goto statements can be harder to maintain and debug. The non-linear flow can introduce bugs that are difficult to trace and fix.
- **Alternative Constructs**: C++ provides structured control flow constructs such as loops (for, while, do-while) and conditionals (if, else if, else, switch) that can handle nearly all the use cases

for goto in a cleaner, more structured way.

However, there are specific situations where goto might be considered useful or necessary, such as:

- **Breaking out of nested loops**: When a break is needed from deeply nested loops, a goto statement can provide a straightforward solution without having to refactor large portions of code.
- **Error handling**: In some low-level programming scenarios, especially in system-level programming, goto can be used for cleanup tasks and to jump to error handling routines.

Despite these use cases, it's important to approach goto with caution. Its use should be limited to scenarios where the benefits outweigh the potential drawbacks in terms of code clarity and maintainability. Modern C++ programming encourages structured programming practices, with goto largely being considered a relic of earlier programming styles.

5.4.4 Issues:

5.4.4.1 <++>

<++>

5.5 Programming loops

It is very often needed, that we want to repeat some part of code several times. In that case we can use programming loops where we can specify which code should be repeated. In general there are two very often situation where we are using the programming loops:

1. We know `how many times` some code should repeat and
2. The code is `repeated` **`while`** `the condition` is met.

5.5.1 For-Next Loop

So called `For-Next` loop is used whenever the repetition of the code can be controlled by a `counter`. Counter is a number with some **starting value** and gets incremented by each repetition of the code. When `counter` reaches the given **ending value** repetition will stop. Typical examples where `For-Next` loop is used are:

- filling an array of data,
- summarising of all the costs in the bill
- robot should turn for **8 times** with 45 degree step to complete full rotation.

5.5.2 While Loop

`While` loop is used in situations where we can not predict the numbers of repetitions in advanced. In this case we must state the `condition` that must be met to repeat the code. The repetition of the code will be terminated when the `condition` will not hold anymore. Typical examples are:

- read the content to end of file,
- divide some number by 2 while we can,
- while no obstacle is in front of the robot it should drive forward

5.5.3 Do-While Loop

The Do-While loop in C++ programming is a control flow statement that executes a block of code at least once and then either repeatedly or until a particular condition is met. The condition is evaluated after the execution of the block of code. If the condition is true, the block of code is executed again. This repeats until the condition becomes false.

Here are three examples where a do-while loop can be suitable in programming a mobile robot:

- Navigating a Maze: A do-while loop can be used to control a robot to navigate through a maze by repeating movements (forward, turn left or right) until it finds the exit.
- Obstacle Avoidance: A do-while loop can be used to program a robot to continuously move in a particular direction until it detects an obstacle, then it changes direction.
- Searching for a Specific Object: A robot can be programmed using a do-while loop to keep searching in an environment until a particular object is found. This can be useful in search and rescue missions, or in a manufacturing setting where a robot is used to find and retrieve specific items.

5.5.4 Task: FOR-NEXT LOOP

1. For example the next prog. example 5.6 repeats the functions **robotLeft()** and **robotRight()** for **10 times** and robot will do a funny "dancing" move.

Listing 5.6: Programming Loops.

```
#include <RobotMovingFunctions.h>

void setup()
{
  setIOpins();
  // Repeating Left and Right movement
  // for 10 times to make a danging move
  for (int i = 0; i < 10; i++)
  {
    robotLeft();
    delay(100);
    robotRight();
    delay(100);
  }
  stopTheRobot();
}

void loop()
{

}
```

2. Experiment a bit more with such programming techniques and change some code:

   - value of `i`,
   - duration of `delay()` function,
   - add some other functions to the **`for`**`-next` loop…

5.5.5 Task: WHILE LOOP

3. Change the **`for`**`-next` loop with this **`while`** loop. Can you predict the result?

```
while ( 1 == 1 ){
    robotLeft();
    delay(100);
    robotRight();
    delay(100);
}
```

Presented **`while`** loop is not an useful example as the condition `( 1 == 1 )` will never change and will be always **`true`**. So, we created an infinite loop. `While` loop is far more usable if in the condition is some sensor's value, as we will see in next sections.

5.5.6 Questions:

1. Name the situation where `for-next` loop can be used.
2. What is the purpose of a `counter` in `for-next` loop?
3. What is the difference between `for-next` and `while` loops?

5.5.7 Summary:

Loops in C++ programming are used for flow control, allowing developers to execute a block of code repeatedly until a certain condition is met. There are three types of loops: - for, - while, and - do-while.

The `for` loop is typically used when the number of iterations is known. It contains an initializer, a condition, and an iterator. The `while` loop executes a block of code as long as the condition remains true. Unlike the `for` loop, the number of iterations in a `while` loop is indeterminate and depends on when the condition becomes false. The `do-while` loop is similar to the `while` loop but executes the block of code at least once before checking the condition. Loops are fundamental for flow control in C++, allowing for efficient and organized code execution.

5.5.7.1 For Loop:

Executes a block of code a specific number of times.

```
for (initialization; condition; increment) {
    // Code to execute
}
```

5.5.7.2 While Loop

Executes a block of code as long as a condition remains true.

```
while (condition) {
    // Code to execute
}
```

5.5.7.3 Do-While Loop

Similar to the while loop, but it executes the block of code at least once before checking the condition.

```
do {
    // Code to execute
} while (condition);
```

<++>

5.5.8 Issues:

5.5.8.1 *Can I measure the execution time of the loop?*

Yes, you can. You must save the time before the loop and save the time after the loop is executed. The difference in these two values is the spent in the execution of the loop. A minimal working example counld look like this:

```
unsigned long start_time = millis();
for (int i = 0; i<100; i++)
{
  //some code in this loop
}
unsigned long stop_time = millis();
unsigned long loop_duration = stop_time - start_time;
```

5.5.8.2 *Can I exit a while loop.*

Yes, you can use the “break” statement to exit a while loop in C++. However, this is not a common practice it is advised to set appropriate condition to exit a while loop. Here is an example of using “brake” statement:

```
int x = 0;
while (x < 10) {
  Serial.println(x);
  x++;
  if (x == 5) {
    break;
  }
}
```

This code will output the following to the serial port:

```
0
1
2
3
4
```

In this example, the “break” statement is used to exit the while loop when the value of “x” becomes 5. As a result, the loop only executes 5 times, rather than 10 times.

It is also possible to use the “continue” statement to skip the remainder of the current iteration of a loop, without exiting the loop entirely. For example:

```
int x = 0;
while (x < 10) {
  x++;
  if (x % 2 == 1) {
    continue;
  }
  Serial.println(x);
}
```

This code will output the following to the serial port:

```
2
4
6
8
10
```

In this example, the “continue” statement is used to skip the remainder of the current iteration of the loop if the value of “x” is odd. As a result, only the even values of “x” are printed.

5.6 Variables and data types

In earlier examples we have stored some values into `variables` (e.g counting **`for`** `loop` repetition). Variables are the containers for storing data values usually located in RAM (also in EPROM, FLASH ...). In order to store different data (e.g. numbers, words ...) we have to use different type of variables. The declaration of the variable (=creation) has next syntax:

```
type varialble_name = value;
```

With next example we will solve the problem how to make light blinking while the robot is driving in reverse.

5.6.1 Task: USING VARIABLES

1. Start with this example of driving the robot for 3s forward and then for 3s backward. Test program example in prog. example 5.7. Then try to add some code to blink the light while the robot is driving backward.

Listing 5.7: Variables and Data Types.

```
#include <RobotMovingFunctions.h>
void setup()
{
  setIOpins();

  moveForward();
  delay(3000);
  moveBack();
  deay(3000);
  stopTheRobot();
}
void loop()
{
}
```

2. As you probably find out you have to divide the duration of 3000 ms into smaller durations and meanwhile controlling the light output. This can be done with `for-next` loop which repeats 10 times.

   Change the 9^{th} line `delay(3000)` in previous example into `for-next` loop with 10 repetition, but with the same overall duration of 3000 ms.

```
...
moveBack();
for (int i = 0; i < 10; i++)
{
  delay(150);
  delay(150);
}
stopTheRobot();
...
```

3. Add some code for blinking the LED in the `for loop` during the robot is driving backward.

   > Don't forget to set the REVERSE_LIGHT_PIN value and its `pinMode(...)`.

```
...
moveBack();
for (int i = 0; i < 10; i++)
{
  digitalWrite(REVERSE_LIGHT_PIN, HIGH);
  delay(150);
  digitalWrite(REVERSE_LIGHT_PIN, LOW);
  delay(150);
}
stopTheRobot();
...
```

4. More advanced way to do a time conditioned loop is shown in next example:

```
...
robotBack();
unsigned long start_time = millis();
int time_diff = 0;
while (time_diff < 3000)
{
  digitalWrite(REVERSE_LIGHT_PIN,HIGH);
  delay(150);
  digitalWrite(REVERSE_LIGHT_PIN,LOW);
  delay(150);
  unsigned long now = millis();
  time_diff = now - start_time;
}
stopTheRobot();
```

5.6.2 Questions:

1. Show some examples of programming assignment statement!
2. What is the operator for assign the value to the variable?

5.6.3 Summary:

5.6.3.1 What is variable?

In computer programming, a variable is a storage location in memory that is used to hold a value. The value of a variable can be changed during the execution of a program.

Each variable has a name, which is used to refer to the variable in the code, and a data type, which determines the kind of value that the variable can hold.

There are several different data types in C++, including:

Integers: Integers are whole numbers that can be positive, negative, or zero. In C++, there are several different integer data types, including char, short, int, and long.

Floating-point numbers: Floating-point numbers are numbers with decimal points. In C++, the float and double data types are used to represent floating-point numbers.

Characters: Characters are single letters, digits, or symbols. In C++, the char data type is used to represent characters.

Booleans: Booleans are values that can either be true or false. In C++, the bool data type is used to represent booleans.

To use variables in C++, you will need to declare them and assign them values. Here is an example:

```
int x;        // Declare an integer variable called x
x = 10;       // Assign the value 10 to x

char c;       // Declare a character variable called c
c = 'A';      // Assign the value 'A' to c

double d;     // Declare a double variable called d
d = 3.14;     // Assign the value 3.14 to d
```

5.6.3.2 Variable definition and initialization in C++

A variable definition means that the programmer writes some instructions to tell the compiler to create the storage in a memory location. The syntax for defining variables is:

```
data_type variable_name;
```

Here `data_type` means the valid C++ data type which includes int, float, double, char, wchar_t, bool and variable list is the lists of variable names to be declared which is separated by commas. Variables are declared in the above example, but none of them has been assigned any value. Variables can be initialized, and the initial value can be assigned along with their declaration.

```
data_type variable_name = value;
```

Examples:

```
int value = 1234;        // whole numbers from -32768 .. 32767
char smalVal = 123;      // whole numbers from 0 .. 255
char letterA = 'A';      // character value like !"#0123..ABC..xyz
bool logicVal = true;    // 0 and 1 or false and true
float pi_value = 3.14;   // from -3.4E+38 .. +3.4E+38
char text[32] = "Some text.";
```

In next fig. 5.1 we can find previous variables stored in controllers' RAM memory (upper window of fig. 5.1). In the lover left corner of the fig. 5.1 we can find printed memory addresses of these variables. In the memory table we can first notice `text` variable from the address `0x0100` within next 32 bytes (2 rows of the memory table). Next 4 bytes are occupied by `pi_value` variable, at the memory address `0x0124` `logicVal` is stored (1 byte), following with character letter A stored in variable named `letterA` at the address `0x0125` with the HEX value of `0x41`. At the memory address `0x0126` we can find `smalVal` variable which storing the value 123 (DEC) or `0x7B` in HEX. The last 2 bytes are occupied by the integer variable named `value` where the nuber 1234 is stored or in HEX 0x04 0xD2.

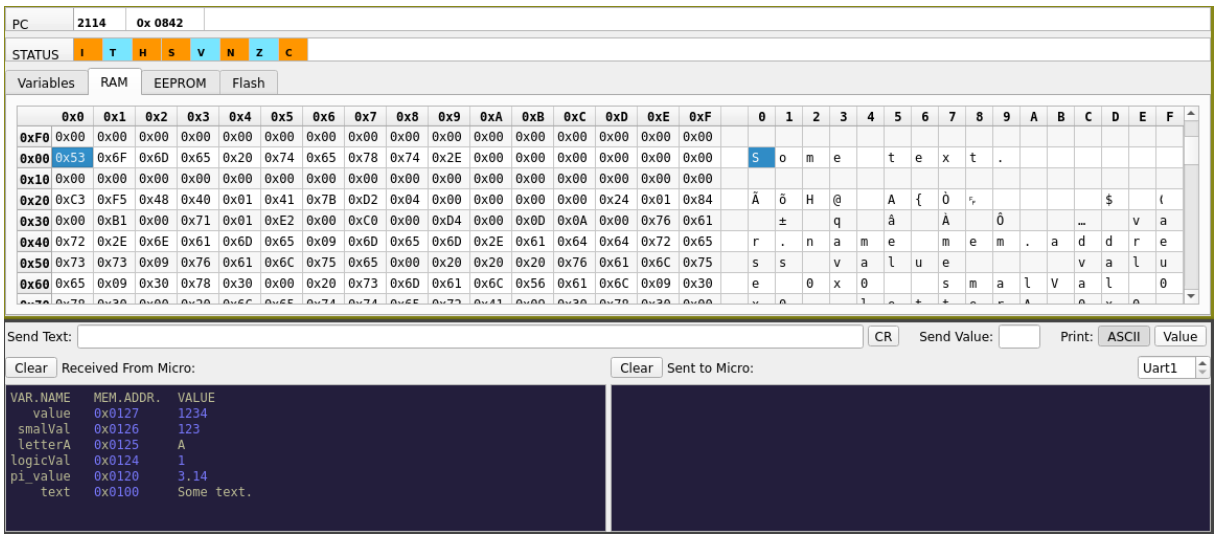

Figure 5.1: Table of values stroed in RAM memory of Arduino UNO controller.

5.6.3.3 Measuring Time with programming loops

The easiest way to measure time is to simply count the number of loop's executions. And if we know how long is one execution of the loop - we can easily determine the time lapsed for the whole process.

Example:

```
1    int t = 0;
2    while (t<10){
3    t++;
4    delay(100);
5    }
```

In the previous example the **`while`** loop is executed 10 times (t = [0 .. 9]), since each execution of the loop last 100 ms (determined by `delay(100);`) the whole **`while`** loop last 1 s.

5.6.3.4 Time measuring with Timers

More proper way of measuring the time is by using the timer's values. More on that can be read here.

Example:

```
    unsigned long start_time;
    unsigned long stop_time;
    start_time = millis();
    // time measured process goes here
    // ...
    stop_time = millis();
    unsigned long duration = stop_time - start_time;
```

Where the `duration` is time measured in milliseconds.

5.6.3.5 Structures

n C++, a struct is a user-defined data type that groups together a collection of variables. It is similar to a class in that it can contain variables and functions, but there are a few key differences between the two.

One of the main differences between a struct and a class in C++ is that structs have public members by default, while classes have private members by default. This means that, by default, all members of a struct can be accessed directly from outside the struct, while members of a class can only be accessed through its member functions.

Another difference is that structs are often used for small, simple data structures that do not require the encapsulation and data hiding features provided by classes. Structs are commonly used for situations where you simply want to group together related data, such as representing a point in two-dimensional space, a date, or a color.

Here is an example of a simple struct in C++:

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

This struct defines a new type called Point, which contains two variables of type int, x and y, representing the coordinates of a point in a two-dimensional space.

```
    Point p1;
    p1.x = 3;
    p1.y = 4;
```

In this example, we create a variable p1 of type Point and assign values to its members x and y.

It's also worth noting that C++ has also a keyword class which is semantically equivalent to struct except for the default access level of its members.

5.6.3.6 Enumeration

In C++, an **enum** (short for "enumeration") is a user-defined data type that consists of a set of named values. It is used to create a new type with a fixed set of possible values, which can make your code more readable and maintainable.

Here's an example of an enumeration that could be used in a mobile robot program to represent the different states of the robot:

```
enum class RobotMoves{
    FORWARD,
    BACKWARD,
    MOVE_LEFT,
    MOVE_RIGHT,
    STOP
};
```

You can use this enumeration in the robot's control loop to check and update the current state of the robot:

```
RobotMoves currentRobotState = RobotMoves::STOP;

while (true) {
    // Some other logic here
    // ...

    // Sampling the sensors based on the state of the robot
    switch (currentRobotState){
      case RobotMoves::FORWARD    : checkFrontSensors(); break;
      case RobotMoves::BACKWARD   : checkBackSensors(); break;
      case RobotMoves::MOVE_LEFT  : checkLeftSensors(); break;
      case RobotMoves::MOVE_RIGHT : checkRightSensors(); break;
      default: //nothing to do...
    }
}
```

This way, it's clear and easy to understand the current state of the robot, and it can also help to implement logic and different behaviors for each state. It's also easy to add or remove states in the future if needed, without having to modify the code in many different places.

5.6.4 Issues:

5.6.4.1 <++>

<++>

5.7 Conditional Statements

Conditional statements in C++ are foundational constructs that allow programmers to execute specific sections of code based on certain conditions. These statements enable decision-making within a program, allowing it to respond differently to various inputs or situations. The most commonly used conditional statements in C++ are if, else if, and else.

Imagine you are programming a mobile robot that uses a bumper sensor to detect obstacles. The bumper sensor can return two states: 0 (no contact), 1 (contact with the obstacle). Based on the sensor's input, the robot should make decisions: stop moving forward when an obstacle is detected and adjust its path accordingly. This will be our future task in next chapter. First we have to construct robot's bumper with push button key and test it.

5.7.1 Tasks:

1. Construct the bumper of the robot with push-button-switch as is shown in this video instructions.

2. And connect the push-button-switch (PBSW) terminals with module RobDuino according to tbl. 5.1:

Table 5.1: Connection of push-button-switch to the Robduino module.

PBSW con.	RobDuino connectors
No. 1	A0
No. 2	GND
No. 3	+5V

3. Test the push-button-switch in the bumper with next prog. example 5.8:

Listing 5.8: Conditional Statements.

```
const int BUMPER_PIN          = A0;
const int TEST_BUMPER_LED_PIN = 3;
void setup()
{
  pinMode(BUMPER_PIN, INPUT);
  pinMode(TEST_BUMPER_LED_PIN, OUTPUT);
}

void loop()
{
  bool bumperIsPressed = digitalRead(BUMPER_PIN);
  if ( bumperIsPressed ) digitalWrite(TEST_BUMPER_LED_PIN, HIGH);
}
```

2. Then... complete the program to turn OFF the LED when the bumper is not touching anything.
3. Next... Change IF statements into single one IF-THEN-ELSE statement.
4. Complete the IF-statement with a block of code so that the LED will blink when the bumper is pressed.

5.7.2 Questions:

1. Check if the LED on the output terminal D3 is ON when the bumper is pressed.
2. Measure the voltage potencial at the terminal A0 when the bumper is pressed.
3. Explain when the curly braces `{}` are necessary in the if-statement.

5.7.3 Summary:

Conditional statements in C++ programming are utilized for flow control within a program. These statements allow the program to make decisions and execute certain blocks of code based on specified conditions. The primary conditional statements in C++ include **`if`**, **`if-else`**, nested **`if-else-if`**, and **`switch-case`**.

`If` executes a block of code if a specified condition is true.

`If-`**`else`** provides an alternate block of code if the initial condition is false.

Nested **`if-else-if`** involves multiple layers of if-else conditions within one another for complex decision making.

`Switch-`**`case`** allows a variable to be tested for equality against a list of values and executes the first match.

Thus, conditional statements provide essential control flow mechanisms in C++ programming.

5.7.3.1 IF, IF-ELSE, IF-ELSE-IF

can be written in several forms. The easiest one is:

```
   if (value_one) statement1;
```

In this case the variable named `value_one` can hold some numerical number. If `value_one` is **`true`** or greater than `0` the program will execute `statement1`. But this simple example is not used so often due its simplicity. We rather use it in this form:

```
   if ( value_one == value_two ){
    statement1;
    statement2;
   }
```

In this case `value_one` can be any number and the `statement1` and `statement2` will be executed if the `value_one` will be equal to `value_two`. These command can be expanded into IF-ELSE form:

```
   if ( value_one == value_two ){
    statement1;
    statement2;
   }else{
    statement3;
   }
```

An else if ladder can be used to decide among multiple conditions.

```
   if (condition1) {
       // Code to execute if condition1 is true
   } else if (condition2) {
       // Code to execute if condition2 is true
   } else {
       // Code to execute if none of the above conditions is true
   }
```

5.7.3.2 SWITCH statement

The switch statement allows you to execute one block of code out of many, based on the value of a variable. It's often more convenient than multiple if-else statements when dealing with variable values.

```
int x = 2;

switch (x) {
  case 1: printf("x is 1"); break;
  case 2: printf("x is 2"); break;
  case 3: printf("x is 3"); break;
  default: printf("x is something else"); break;
}
```

In this example, the switch statement checks the value of x and executes the code block corresponding to the first case label that matches the value. The break statements are used to exit the switch statement once a match is found. If no match is found, the code block for the default label is executed.

5.7.3.3 Condition operators

Also other logical condition operators can be used:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- Equal to `a == b`
- Not Equal to: `a != b`

5.7.4 Issues:

5.7.4.1 <++>

<++>

5.8 Pointers and references

5.9 Classes and objects

5.10 Exception handling

5.11 Input and output

5.12 Debugging and testing

5.13 Advanced topics threading memory management templates

5.13.1 Bit-field variable type

Variable bit fields are a specific type of data structure in C++ that allows a user to store multiple bit-sized values within a single variable. This can be useful for storing several different values in the same memory space or for compressing data. An example of a variable bit field in C++ is shown below:

```
1   struct example {
2    unsigned int value1 : 4; // Use 4 bits
3    unsigned int value2 : 8; // Use 8 bits
4    unsigned int value3 : 12; // Use 12 bits
5   } myStruct;
```

In this example, we have defined a structure called ‘example’ which contains three members - ‘value1’, ‘value2’, and ‘value3’. Each of these members has been defined as a variable bit field using the ‘unsigned int’ data type and the ‘:’ syntax, which allows us to specify the number of bits that each member should use. In this case, ‘value1’ will use 4 bits, ‘value2’ will use 8 bits, and ‘value3’ will use 12 bits. To access these values, we can use the members of the structure, for example, ‘myStruct.value2’.

```
struct adc4 {
  unsigned int value1 : 10;
  unsigned int value2 : 10;
  unsigned int value3 : 10;
  unsigned int value4 : 10;
};

unsigned int adc_val[40];           //40 values
adc4 myAdc[10];                     //40 values

void setup() {
  Serial.begin(9600);
  Serial.println(sizeof(adc_val));  //print 80
  Serial.println(sizeof(myAdc));    //print 50
}

void loop() {

}
```

6 SENSING REASONING ACTING LOOP

Robotics is a field of engineering that involves the design and operation of robotic systems. One of the most fundamental principles underlying robotic systems is the S-R-A (sensor-response-actuation) loop. This concept is at the heart of all robotic systems and is essential for understanding the behavior of robots.

The S-R-A loop involves a robot continually sensing its environment, interpreting the data, and then taking some action in response. In other words, the robot is constantly interpreting sensory input and responding with a motor action. It is a continuous cycle of sensing, reasoning, and acting.

The sensing component of the S-R-A loop generally involves the use of sensors such as cameras, ultrasound, or infrared sensors. These sensors detect the robot's surroundings and provide the robot with the data necessary to make decisions. The response component of the loop involves the robot using its artificial intelligence to interpret the data and make decisions. This decision-making process is what gives robots the ability to respond to their environment.

The actuation component of the S-R-A loop is where the robot takes action. This action may involve a physical movement, such as walking, or it may involve activating a motor to perform a task, such as picking up an object.

The S-R-A loop is the basic building block of any robotic system. All robots use this concept as it is essential for a robot to be able to interact with its environment. Without it, robots would not be able to make decisions or take action. This concept is also important for enabling robots to learn, as it allows them to continually increase their knowledge and abilities.

Overall, the S-R-A loop is the cornerstone of robotics. It is essential for robots to be able to interact with their environment and learn from it. Without the S-R-A loop, robots would be unable to take any action or make decisions. It is an integral part of any robotic system.

From the S-R-A loop, lets start at the very beggining of the loop - at reading input signals by emphasizing the importance of received input signal. In other words, it is critical that the system be able to detect and interpret input signals in order to produce the appropriate responses. Once these input signals are received, they must be accurately processed and acted upon. This is the primary task of the S-R-A loop, and is the basis for any successful input processing system.

To read an input signal on an Arduino, you can use one of the digital input pins or one of the analog

input pins. Digital input pins can only read two states: high (5 volts) or low (0 volts). They are often used to read switches or buttons, or to detect the presence or absence of a signal.

To read a digital input signal on an Arduino, you can use the digitalRead function, which takes a pin number as an argument and returns either HIGH or LOW. For example, to read the state of digital pin 2, you could use the following code:

```
int pin = 2;
int state = digitalRead(pin);
```

Analog input pins, on the other hand, can read a range of voltage levels, from 0 to 5 volts. They are often used to read sensors that output an analog signal, such as a temperature sensor or a potentiometer.

To read an analog input signal on an Arduino, you can use the analogRead function, which takes a pin number as an argument and returns a value between 0 and 1023, corresponding to the voltage level on the pin. For example, to read the voltage on analog pin 0, you could use the following code:

```
int pin = 0;
int value = analogRead(pin);
```

6.1 S-R-A loop

S-R-A loop is repeating process where:

1. Sensing,
2. Reasoning and
3. Acting

is involved during the procedure of controlling the robot. This is the most important part of software in robotics. Remember the `autonomous` control is `ability to perform intended tasks based on current state and sensing, without human intervention`.

The S-R-A loop is a common design pattern in robotics. It refers to the process of using sensors to gather information about the environment, processing the information to determine an appropriate response, and then executing the response using actuators.

Here is an pseudo example of how the S-R-A loop could be implemented in C++:

```
  while (true) {
    // 1. Sense the environment using sensors
    sensor_data = gatherSensorData();

    // 2. Process the sensor data to determine an appropriate response
    response = processSensorData(sensor_data);

    // 3. Execute the response using actuators
    executeResponse(response);
  }
```

In this example, the `gatherSensorData` function is used to gather data from the robot's sensors, the `processSensorData` function is used to determine an appropriate response based on the sensor data, and the `executeResponse` function is used to execute the response using the robot's actuators. The loop is executed continuously, allowing the robot to constantly sense and respond to its environment.

6.1.1 Tasks:

1. Using the S-R-A loop technique you should write the program in particular order:

    1. Check the sensor. IF the bumper ...
    2. ... Is pressed the robot has to stop/go back/turn.
    3. ... Is not pressed the robot can drive forward.

    Test the prog. example 6.1 and **find out why the robot does not stop.** (Such mistake is quite often - can you fix it):

Listing 6.1: SRA Loop.

```
#include <RobotMovingFunctions.h>
const int BUMPER_PIN = A0;
void setup()
{
  setIOpins();
  pinMode(BUMPER_PIN, INPUT);

  bool bumperIsPressed = digitalRead(BUMPER_PIN);
  if ( bumperIsPressed )
  {
    stopTheRobot();
  }
  else
  {
    moveForward();
  }
}
void loop()
{
}
```

2. Hint for fixing the prog. example 6.1: *S-R-A must be a loop function!*
3. Write a program to drive the robot around the class and avoid the obstacles.
4. Make split bumper design (left and right) as presented in fig. 6.1 and write a program so that the robot avoids the obstacle depending on which side the obstacle is present.

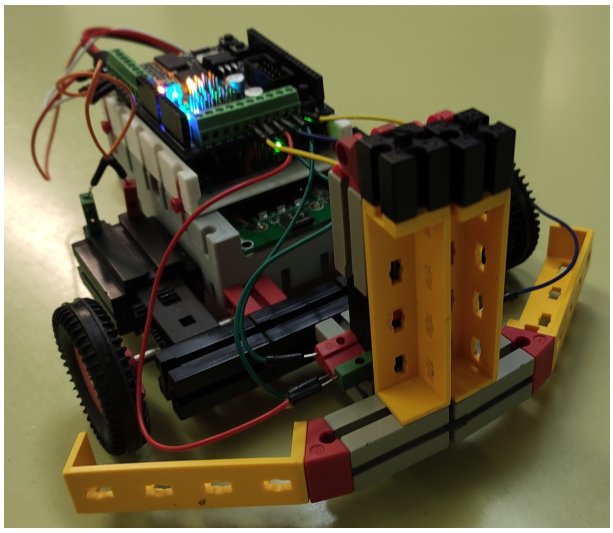

Figure 6.1: Split bumper design allowing detecting obstacles from left and right.

6.1.2 Questions:

1. What for `S-A-R loop` stands for?
2. Mark all three basic S-A-R processes in previous code example.
3. Can the `line 8` of the prog. example 6.1 be written outside of `loop()` function? What would happened if so?

6.1.3 Summary:

6.1.3.1 <++>

<++>

6.1.4 Issues:

6.1.4.1 <++>

<++>

6.2 Digital input

Digital inputs can only measure 2 different values. As such they are binary inputs and it's values are represented as logical `0` and `1` or in other words **`false`** and **`true`** or `LOW` and `HIGH`. However from electrical point of view those values are basically different voltage potencials. Usually potencial `0 V` is presented as logical `0` and potencial `+5 V` is indicated as logical `1`. Digital inputs are often used for detecting state of switches, board keys and push buttons…

Lets go back to fundamentals of digital inputs and explore some options we have to connect a push-button-switch.

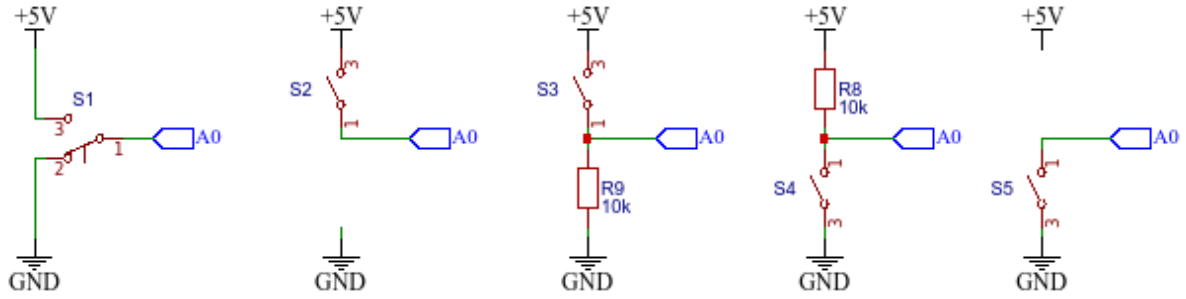

Figure 6.2: Different options of wireing the bush-button-switch.

6.2.1 Tasks:

1. Connect the push-button-switch according to first diagram on fig. 6.2 and test the program prog. example 6.2

Listing 6.2: Digital Input.

```
const int BUMPER_PIN = A0;
void setup()
{
 pinMode(BUMPER_PIN, INPUT);
}

void loop()
{
  bool bumperIsPressed = digitalRead(BUMPER_PIN);
 if ( bumperIsPressed ) digitalWrite(3, HIGH);else digitalWrite(3, LOW);
}
```

2. Try to connect the bush-button-switch according to second diagran on fig. 6.2

Table 6.1: Connection of push-button-switch with only 2 terminals.

PBSW con.	RobDuino connectors
No. 1	A0
No. 2	not connected
No. 3	+5V

Try to understand why this setup is not working. And test all other options in fig. 6.2

3. Solve the problem by constructing a `voltage divider` with `pull-down` resistor (third diagran on fig. 6.2).
4. Try to understand how the voltage potencial is spread among the components in electrical loop and how we can calculate this by using 2nd Kirchhoff's Rule.
5. Change the setup of PBSW and resistor to a `pull-up` setup (fourth diagran on fig. 6.2). What is changed?
6. Enable internal `pull-up` resistor (and remove external one - fifth diagran on fig. 6.2).

6.2.2 Questions:

1. Measure the voltage potencial on pin A0 where the bumper is in ether position.
2. Why the setup is not working properly if we connect the PBSW only to +5V voltage potencial?
3. Draw a schematic circuit of the bush-button-switch connected to controller.
4. What is determined by 2nd Kirshhoff's Rule?
5. How can we enable `pull-up` resistor?

6.2.3 Summary:

6.2.3.1 2nd Kirshhoff's Rule

Kirchhoffs Voltage Rule states that `in any closed loop network, the total voltage around the loop is equal to the sum of all the voltage drops within the same loop` which is also equal to zero. In other words the algebraic sum of all voltages within the loop must be equal to zero. This idea by Kirchhoff is known as the Conservation of Energy.

6.2.4 Issues:

6.2.4.1 <++>

<++>

6.3 Pull-up resistors on digital input

On the module RobDuino we can find two "on-board push button switches". Wiring of this switches is presented in fig. 6.3, where can we noticed that both switches are connected to ground voltage potential.

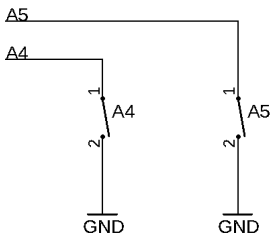

Figure 6.3: Wiring of on-board switches.

To properly use this on-board push-button switches we must enable the `pull-up` resistors of A4 and A5 input of microcontroller.

6.3.1 Tasks:

1. Configure pins `A4` and `A5` as inputs with `pull-up` resistor.
2. At the end of the `setup()` function add the **`while`**`-loop` which will delay the execution of the program until we press the `A4` key - acting as a “START BUTTON”.
3. Use the `A5` key to stop the robot and terminate the execution of the program.

Listing 6.3: Pull Up Resistors on Digital Input.

```
#include <RobotMovingFunctions.h>
const int KEY_A4 = A4;
const int KEY_A5 = A5;

void setup()
{
  setIOpins();
  pinMode(KEY_A4, INPUT_PULLUP);
  // KEY_A5 setup here...
}

void loop()
{
  moveForward();
  //to-do: the key reading
  bool stopTheRobotKey = 0;
  if (stopTheRobotKey == 1)
  {
    stopTheRobot();
    exit(0);        //terminate the program
  }
}
```

6.3.2 Questions:

1. What is the programming instruction of reading the value form digital input?
2. Which values can be assigned to `bool` type variable?
3. Explain the programming instruction `exit(0)`.

6.3.3 Summary:

6.3.3.1 <++>

<++>

6.3.4 Issues:

6.3.4.1 <++>

<++>

6.4 Pulse width as digital input

Digital input can also be used to transferee other data. One way is to modulate the data into pulse duration e.g. longer the duration of the pulse, bigger the value. This modulation of data is called **Pulse-width modulatio** or **PWM**. Such an example is ultrasonic distance sensor. Where the distance is hidden in the time duration that sound needed of travel the distance from source to object and back as presented in fig. 6.4.

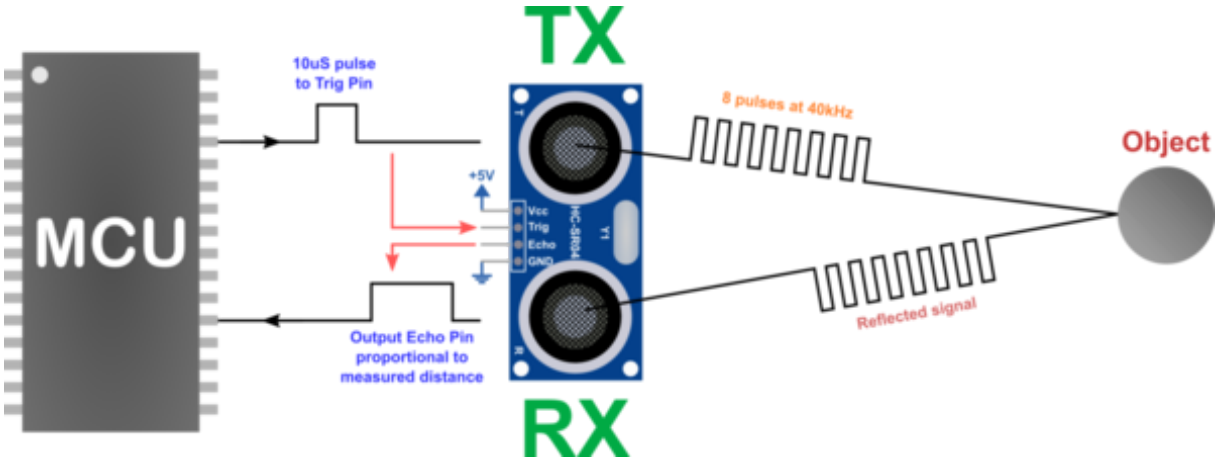

Figure 6.4: How Ultrasonic sensor works.

Since the speed of sound in air is constant ($v_s = 340m/s$) we can easily calculate the distance according to eq. 6.1.

$$distance = \frac{1}{2} v_s t_{duration} \tag{6.1}$$

6.4.1 Tasks:

1. Connect the ultrasonic distance sensor to module Robduino according to tbl. 6.2

Table 6.2: Connestion of ultrasonic distance sensor.

HC-SR04 pins	RobDuino pins
+5V	+5V
Trigg.	A0
Echo	A1
GND	GND

2. Test next program if you get reasonable data of time duration in `Serial` window.

Listing 6.4: PWM as Digital Input.

```
const char TRIGGER_PIN  = A0;
const char ECHO_PIN     = A1;

void setup()
{
  pinMode(TRIGGER_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);
  Serial.begin(9600);
}

int getPulseWidth_us()
{
  digitalWrite(TRIGGER_PIN,HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIGGER_PIN,LOW);
  return pulseIn(ECHO_PIN, HIGH);
}

float getDistance_cm()
{
  // do distance calculation here...
  return 0
}
void loop()
{
  float distance_cm = getDistance_cm();
  int duration_us = getPulseWidth_us();
  Serial.println(duration_us);
  delay(2000);
}
```

3. Add needed code in function `getDistance_cm()` to calculate the distance in cm. Also change the `Serial.println(duration_us)` program line to output `distance_cm` value.

6.4.2 Questions:

1. What is PWM?
2. How are PWM data presented in digital signal?
3. What voltage is used to transmit PWM values?

6.4.3 Summary:

6.4.3.1 <++>

6.4.4 Issues:

6.4.4.1 <++>

6.5 Analog input

In general, controllers are equipped with `Analog to Digital Converters` or short `ADC`. This internal devices converts voltage potencial into numeric value which can be further used by written program. This is also the case in Arduino UNO converter by the function `analogRead(pin_number)`. In this case the voltage range $[0.0\,V.. + 5.0\,V]$ is converted in to range of numbers $[0..1024]$.

6.5.1 Tasks:

1. Unmount robot's bumper and all connections to the switch.
2. Equip the robot with distance sensor according to video and scheme (see fig. 6.5).

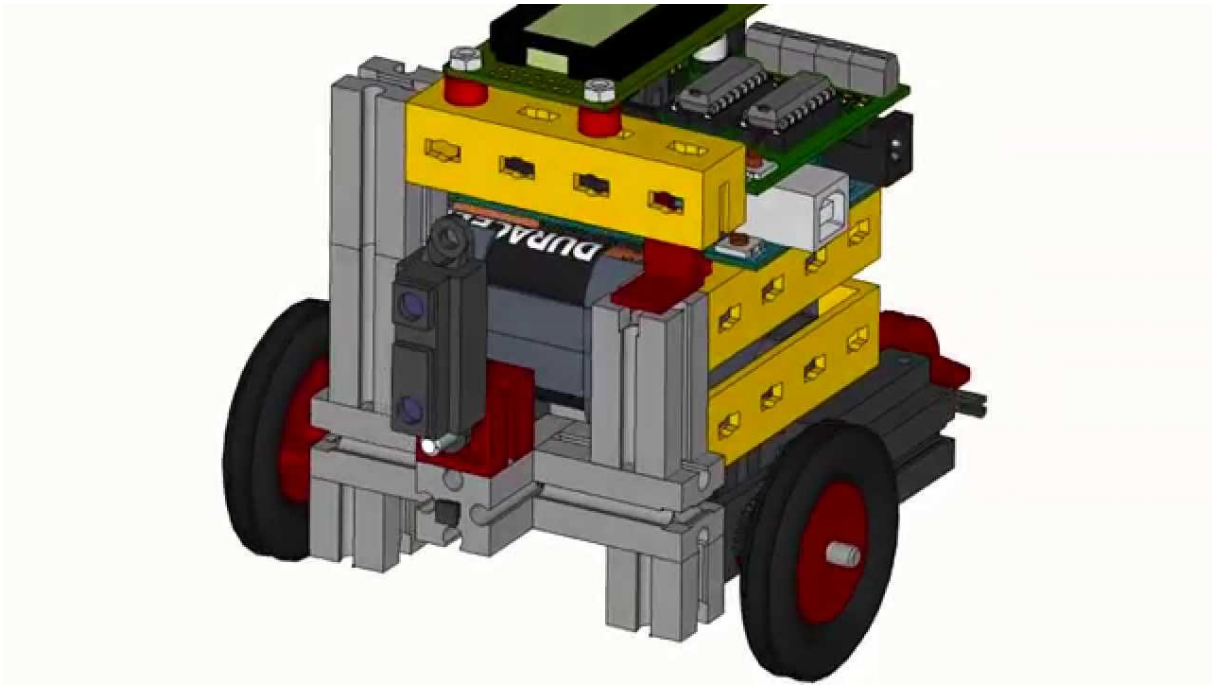

Figure 6.5: Mounting possition of analog distance sensor.

3. Try next prog. example 6.5 and check the output of distance sensor in Serial monitor.

Listing 6.5: Analog Input.

```
const int DIST_SEN_PIN = A0;
void setup()
{
  pinMode(DIST_SEN_PIN, INPUT);
  Serial.begin(9600);
}

void loop()
{
  int adc_value = analogRead(DIST_SEN_PIN);
  Serial.println(adc_value);
  delay(1000);
}
```

4. Convert the `analog_sensor_value` into `input_voltage` and measure the input voltage potencial with volt-meter. The formula for conversion can be programmed as:

```
float input_voltage = 5.0/1024 * adc_value;
```

5. From the datasheet for the distance sensor try to code the function for measuring the distance in cm. According to documentation there is almost linear trend between output voltage and $distance^{-1}$. Thus we can get good result with eq. 6.2.

$$distance^{-1}[cm] = 0.045 V_{out} \tag{6.2}$$

Next example can be your guide to code the function.

```
float getDistance_cm()
{
  int adc_value = analogRead(DIST_SEN_PIN);
  float input_voltage = 5.0/1024 * adc_value;
  float distance = 1/(0.045 * input_voltage);
  return distance;
}
```

6.5.2 Questions:

1. What kind of values do you getting from the reading of the distance sensor with the function `analogRead(A0)`?
2. Find the reasonable value where you should stop the robot.
3. Measure the voltage potencial of the sensor's output.

6.5.3 Summary:

6.5.3.1 Analog to digital converter - ADC

ADC is an electronic sistem that converts analog signal (voltage) to a digitalized values. In our particular case the range of an analog voltage from 0V to 5V is converted to range of numbers from 0 to 1024.

6.5.4 Issues:

6.5.4.1 <++>

<++>

6.6 Avoiding obstacles

6.6.1 Tasks:

Write the program to drive the robot around the class and avoid the obstacles.

1. Check the value of distance sensor. If the distance is greater than …
2. ... the robot can drive forward.
3. ...else ... the robot must to stop/go back/turn.

Listing 6.6: Avoiding Obstacles.

```
#include <RobotMovingFunctions.h>
const int DIST_SEN_PIN   = A0;
const int DISTANCE_LIMIT = 20;
void setup()
{
  setIOpins();
  pinMode(DIST_SEN_PIN,INPUT);
}
float getDistance_cm()
{
  int adc_value  = analogRead(DIST_SEN_PIN);
  float distance = 1/(0.045 * 5.0/1024 * adc_value);
  return distance;
}
void loop()
{
  if ( getDistance_cm() > DISTANCE_LIMIT )
  {
    moveForward();
  }
  else
  {
    stopTheRobot();
  }
}
```

6.6.2 Questions:

1. What are the values of the distance sensor (use `Serial.println(distance)` to verify)?
2. Robot stil hits the obstacles that are not in view angle of the distance sensor. Write and use new function for moving the robot forward more carefully.

6.6.3 Summary:

6.6.3.1 Moving the robot and checking the sensor simultaneously

The main important proces in robotics is S-R-A loop. This process is used in different situations and many times. One can be where we are moving the robot forward and at the same time observing the sensors value with the intention to stop it when the specific condition is met.

```
   void goForwardCarefully()
   {
      for (int i = 0; i < 10; i++)
      {
         robotLeft();delay(50);
         if (getDistance_cm() < DISTANCE_LIMIT) brake;
      }

      for (int i = 0; i < 10; i++)
      {
         robotRight();delay(50);
         if (getDistance_cm() < DISTANCE_LIMIT) brake;
      }
   }
```

<++>

6.6.4 Issues:

6.6.4.1 <++>

<++>

6.7 Light sensor

6.7.1 Tasks:

1. To construct a light sensor using a voltage divider configuration with a phototransistor and a resistor (with a value in the hundreds of kilohms range), you would set it up as follows:
    1. Components:
        - Phototransistor: This acts as a variable resistor whose resistance changes based on the amount of light it receives. The more light that hits the phototransistor, the lower its resistance.
        - Fixed Resistor: This is a resistor with a high resistance value, typically in the range of several hundred kilohms, to form the other half of the voltage divider.
    2. Configuration:
        - Connect one end of the phototransistor to the positive voltage supply (Vcc).
        - Connect the other end of the phototransistor to one end of the fixed resistor.
        - Connect the other end of the fixed resistor to the ground. More detailed construction of light sensor is show on video and scheme.

2. Add also the light bulb which will help to lightning the area beneath the robot.

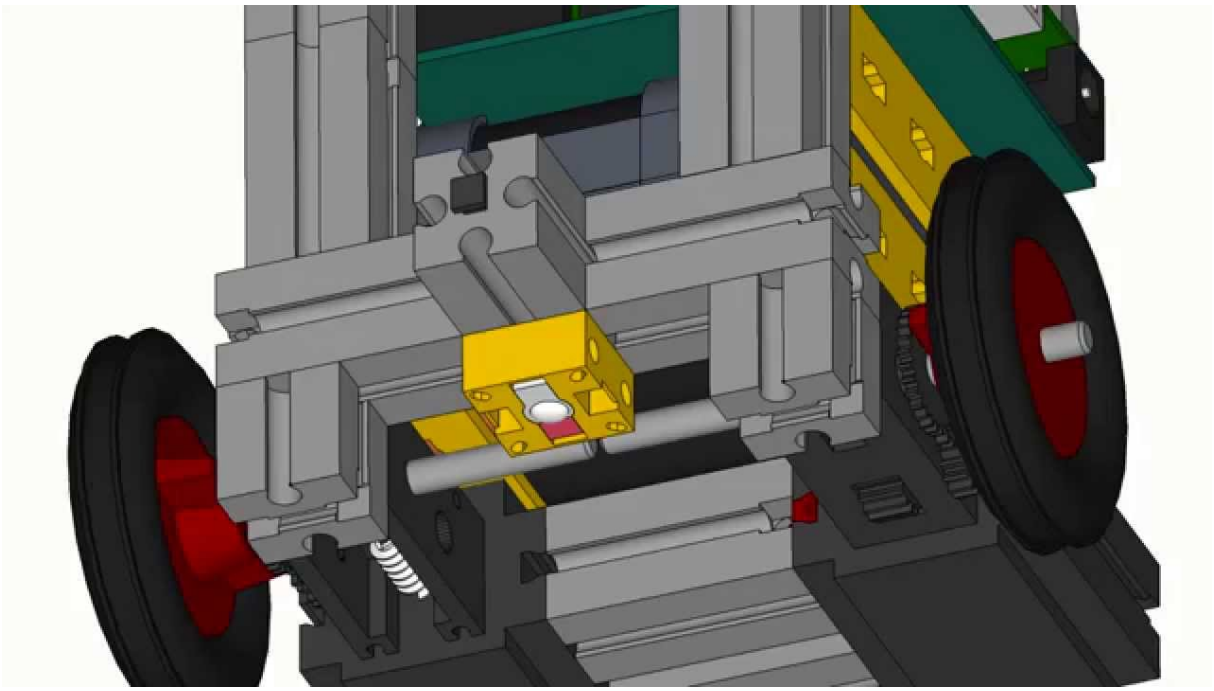

Figure 6.6: Mounting a light sensor.

1. To test the light sensor and light bulb test this example code and check the reported serial data.

Listing 6.7: Ligth Sensor.

```
const int LIGHT_SENSOR_PIN = A0;

void setup()
{
  pinMode(LIGHT_SENSOR_PIN , INPUT);
  Serial.begin(9600);
}

void loop()
{
  int light_sensor_value = analogRead(LIGHT_SENSOR_PIN );
  Serial.println(light_sensor_value );
  delay(200);
}
```

2. Try different resistors (1k, 10k, 100k, 1M) and find out at which the sensitivity of the sensor is greatest.

Table 6.3: Testing the sensitivity of the light sensor.

Resistance	(black) Sensor value	(whithe) Sensor value	Sensor difference
1 kOhm			
10 kOhm			
100 kOhm			
1 MOhm			

6.7.2 Questions:

1. What is the value of the sensor when the robot is over white/black area?
    - ADC value on white:
    - ADC value on black:
2. Calculate the average between those two values.
    - Average is:

6.7.3 Summary:

6.7.3.1 Sensors

Sensors are electronic devices which convert physical quantity into electrical quantity (usually voltage). In simplest setup, sensor can be constructed as voltage divider with two resistors - R_1 and R_2. One of the resistors is resistor with fixed resistance value (eg. $R_1 = 10k\Omega$). The second one is a bit special and it's resistance depends on some physical quantity (e.g. light, temperature, humidity...). When combining those two resistors into such voltage divider the output of the voltage divider can be calculated as:

$$U_{Out} = \frac{R_1}{R_1 + R_2} U_0$$

6.7.3.2 Voltage Divider Principle

The junction between the phototransistor and the fixed resistor is where the output voltage (Vout) is measured. As the light intensity increases, the resistance of the phototransistor decreases. This

causes the voltage across the fixed resistor to increase (because a smaller portion of the total voltage is dropped across the phototransistor). Conversely, when the light intensity decreases, the resistance of the phototransistor increases, causing more voltage to drop across it, and less across the fixed resistor, thus lowering Vout.

6.7.4 Issues:

6.7.4.1 *Value of the sensor is very small*

If the value of the sensor is less than 100 the resistance of R_2 (connected to GND) is to low in comparismant to the resistance of R_1 (connected to +5V).

6.7.4.2 *Value of the sensor is large*

If the value of the sensor is grater than 900 the resistance of R_2 (connected to GND) is to hi in comparismant to the resistance of R_1 (connected to +5V).

6.7.4.3 How to increase the sensors' response?

To increase the response of a light sensor built using a phototransistor and a resistor in a voltage divider configuration, you can try the following strategies:

- Fine-Tuning: Experiment with different resistor values to find the optimal balance between sensitivity and the operating range that suits your application. In general, sensors' response will be the biggest when the output voltage will change across $V_c/2$.
- Optimize Phototransistor Orientation and Placement: Ensure the phototransistor is optimally placed to receive the maximum amount of light from the light source. Avoid orientations where the phototransistor could be shaded or receive reflected light, which might decrease accuracy.
- Filter and Shield from Interference:
  - Use optical filters to limit the light spectrum that reaches the phototransistor, focusing on the wavelengths of interest. This can help in applications where specific types of light need to be detected.
  - Shield the sensor from ambient light or other light sources that might cause interference, ensuring that only light from the target source impacts the sensor's readings.

6.8 Line follower

6.8.1 Tasks:

1. Write the program to control the robot follow the line (actually above the edge between black and white area). Some programming hints you can find in prog. example 6.8 :

Listing 6.8: Line Follower.

```
#include <RobotMovingFunctions.h>
const int LIGHT_SENSOR_PIN = A0;
const int SURFACE_BRIGHTNESS_REFERENCE = 400;

void setup()
{
  setIOpins();
  pinMode(LIGHT_SENSOR_PIN , INPUT);
}

void loop()
{
  int light_sensor_value = analogRead(LIGHT_SENSOR_PIN );
  if ( light_sensor_value < treshold_value )
  {
    // do this if robot is over the black line
  }
  else
  {
    // do this if robot is over white area
  }
}
```

6.8.2 Questions:

1. What is the program function to get the `light_sensor_value`?
2. Determine the movements of the robot if the robot is over the black area and if the robot is over the white area.

6.8.3 Summary:

6.8.3.1 <++>

<++>

6.8.4 Issues:

6.8.4.1 <++>

<++>

7 CONTROLLING ACTUATORS

Motors and actuators are essential components of many robotic systems, as they allow robots to move and manipulate their environment. In Arduino robotics, there are several types of motors and actuators that you can use, depending on the specific needs of your application.

Some common types of motors and actuators that you can use with Arduino include:

DC motors: These are simple motors that rotate at a constant speed when a DC voltage is applied. They are commonly used to drive wheels or other mechanisms. To control a DC motor with an Arduino, you will need a motor driver, such as an H-bridge, which allows you to control the direction and speed of the motor.

Stepper motors: These motors have multiple coils that can be energized in a specific sequence, allowing them to rotate in precise increments. Stepper motors are commonly used in applications that require precise positioning, such as 3D printers or CNC machines. To control a stepper motor with an Arduino, you will need a stepper motor driver, such as a ULN2003 or L298N.

Servo motors: These motors have built-in feedback control and can rotate to a specific angle. They are commonly used to control the position of a mechanism, such as a robotic arm or a camera. To control a servo motor with an Arduino, you can use the Servo library and the write function, which takes an angle as an argument.

Linear actuators: These are motors that produce linear motion, rather than rotary motion. They are commonly used to move mechanisms or lift loads. To control a linear actuator with an Arduino, you will need a motor driver, such as an H-bridge, and you can use the analogWrite function to control the speed and direction of the actuator.

7.1 DC motor

7.2 PWM motor control

There is often the situation where the power of the motors must be controlled. One convenient way to do this is that we don't power the motor full time, but we can turn off the motor for short period of

time. For an example we can turn the motor on for 1 ms and turn it off for 1 ms. In this case the motor will not get 100% of power, but the motor's average power will be 50%.

Since we are changing the pulse width of logical 1 with the respect to width of logical 0, this technique is called `pulse width modulation` or shorter `PWM`.

This modulated output is controlled by the `analogWrite(pin, pwm)` function. Modulatio can be performed on pins: 3, 5 and 6 of the RobDuino modul. The value of `pwm` parameter can be on a scale of 0 - 255., where 0 is 0% and 255 is 100% of electrical power served.

7.2.1 Tasks:

1. Write new functions for driving the robot left and right with reduced power of the motors:

   - `moveLeftPWM();`
   - `moveRightPWM();`

   In one case you will might find yourself in trouble of controlling the power of the motor since both pins are not able to perform `PWM` output. In this case you can remember that the motor's power is 0 W also if both pins are in state of logical 1.

   An example of reducing power of both motors in function `moveForwardPWM()` is here:

```
1  void robotForwardPWM()
2  {
3    digitalWrite( LEFT_MOTOR_PIN_1, LOW);
4    analogWrite(  LEFT_MOTOR_PIN_2, 150);
5    digitalWrite( RIGHT_MOTOR_PIN_1, LOW);
6    analogWrite(  RIGHT_MOTOR_PIN_2, 150);
7  }
```

Similar to this function you can write other functions to.

2. Change the functions `moveLeft()` and `moveRight()` in S-R-A loop with new ones with less power on motors.

Listing 7.1: PWM motor control.

```
#include <RobotMovingFunctions.h>
const int LIGHT_SENSOR_PIN = A0;
const int SURFACE_BRIGHTNESS_REFERENCE = 400;

void setup()
{
  setIOpins();
  pinMode(LIGHT_SENSOR_PIN , INPUT);
}

void loop()
{
  int light_sensor_value = analogRead(LIGHT_SENSOR_PIN );
  if ( light_sensor_value < SURFACE_BRIGHTNESS_REFERENCE ){
      moveLeft();
  } else {
      moveRight();
  }
  delay(10);
}
```

3. Also add `analogWrite(LEFT_MOTOR_PIN_A, 0);` to function `stopTheRobot()` to stop the `PWM` control of the motor. And do similar code for the `right motor`.

4. Add a parameter `PWM_value` to each function to set the `duty cicle` of the controlled output.

   - `moveLeftPWM(int PWM_value)`
   - `moveRightPWM(int PWM_value)`

5. Save `moveRightPWM(int PWM_value)` and `moveLeftPWM(int PWM_value)` functions into header file `RobotMovingFunctions.h`

7.2.2 Questions:

1. How can we control the average power of the motor?
2. How can we control the average power of the motor in both directions if we are not able to control `PWM` both output pins of the motor?
3. Explain the purpose of programming function `analogWrite(pin, pwm)`.
4. Explain the meaning of the `pin` and `pwm` parameters in function `analogWrite`.

7.2.3 Summary:

7.2.3.1 <++>

<++>

7.2.4 Issues:

7.2.4.1 <++>

<++>

7.3 Servo motor

7.4 Stepper motor

Stepper motors are a type of electric motor that can precisely control a rotating shaft's angular position. They are the most commonly used type of motor in motion control applications. A stepper motor works by converting electrical pulses into mechanical shaft rotations, which can be used to move a device or position an object. Stepper motors produce precise, smooth, and repeatable motion and can be used in a variety of robotic applications. They are commonly used for positioning CNC machines, 3D printers, pick-and-place systems, and other robotic applications. Stepper motors are available in a variety of sizes and configurations, and can be used with a variety of drive systems and controllers.

In general we differ two types of Stepper motors (regarding the coil wireing):

1. Bipolar Stepper Motor - This type of stepper motor has two sets of coils, each with a single winding per phase. The coils are wired in series or in parallel depending on the application. Each winding in the motor is energized, then de-energized in order to make the motor rotate.
2. Unipolar Stepper Motor - This type of stepper motor has two sets of coils, each with multiple windings per phase. The coils are wired in series or in parallel depending on the application. Only one winding in the motor is energized at a time to make the motor rotate."

7.4.1 Task

Stepper motors are used in many Arduino projects to control motion, such as turning a wheel or a motor shaft. By applying pulse-width modulation (PWM) signals, the Arduino can control the speed

and direction of the motor. Below is an example of Arduino code that can be used to control a stepper motor:

```
    //Define the pins to be used for the stepper motor
    #define STEPPER_PIN_1 8
    #define STEPPER_PIN_2 9
    #define STEPPER_PIN_3 10
    #define STEPPER_PIN_4 11

    //Define the delay between steps in milliseconds
    #define STEP_DELAY 10

    //Create an array of the pins to be used
    int pins[] = {STEPPER_PIN_1,STEPPER_PIN_2,STEPPER_PIN_3,STEPPER_PIN_4};

    //Initialize the stepper motor
    void setup()
    {
     //Set each pin as an output
     for(int i=0;i<4;i++)
     {
     pinMode(pins[i], OUTPUT);
     }
    }

    //Control the stepper motor
    void loop()
    {
     //Rotate clockwise
     for(int i=0;i<4;i++)
     {
     digitalWrite(pins[i],HIGH);
     delay(STEP_DELAY);
     }
     //Rotate counter-clockwise
     for(int i=3;i>=0;i--)
     {
     digitalWrite(pins[i],HIGH);
     delay(STEP_DELAY);
     }
    }"
    ---
    grand_parent: Basic Robotics
    parent: ACTUATORS
    title: I2C LCD
    nav_order: 4
    ---

    ## LCD(I2C)

    ### Tasks:

    1. Priključite LCD na I2C vodilo kot prikazuje

    ![Povezava LCD na I2C vodilo krmilnika.](./slike/I2C_LCD.png){#fig:
       test_I2C_LCD}

    2. Priskrbite si knjižnico `LiquidCristal-I2C` iz naslova:
    https://www.arduino.cc/reference/en/libraries/liquidcrystal-i2c/
    3. Knjižnico dodajte v Arduino IDE okolje tako, da dodate `ZIP` datoteko
        v :
    `Sketch >> Include Library >> Add .ZIP Library`
    3. V VSC in PlatformIO vtičniku si lahko knjižnico naložite tako, da v
       terminalno okno vpišete ukaz
```

Če niste prepričani kateri i2c naslov uporablja naprava na LCD-ju le tega lahko preverite s programom `I2C scanner` (https://playground.arduino.cc/Main/I2cScanner/). Običajno I2C LCD-ji, ki jih naredijo kitajski proizvajalci uporabljajo I2C naslov `0x27` , `0x3F` ali manj pogosto `0x38`.

7.4.2 Questions:

1. <++>
2. <++>

[Visual instructions.]

7.4.3 Summary:

7.4.3.1 <++>

<++>

7.4.4 Issues:

7.4.4.1 <++>

<++>

8 INTERMEDIATE C++

Welcome to Intermediate C++ programming! This course will dive deeper into the core aspects of C++ programming and provide you with a solid foundation for further development. We're going to cover some of the building blocks of C++, including arrays, strings, pointers, classes and objects, exception handlers, and much more.

Firstly, we'll explore arrays, which allow you to store multiple values of the same type in a single block of memory. This can be particularly useful when programming a mobile robot to follow a specified path, for example:

```
int path[5] = {1, 2, 3, 4, 5};
```

Next, we'll dissect strings – sequences of characters used to store and manipulate text. For instance, we may use a string to denote the robot's status:

```
std::string status = "Moving Forward";
```

Pointers are on our list as well. They are essential and powerful features in C++, storing memory addresses of other variables, which can be useful for dynamic memory allocation in robot's tasks:

```
int batteryLevel = 100;
int* p = &batteryLevel;
```

We will also delve into classes and objects – the backbone of Object-Oriented Programming (OOP). Classes act as blueprints for objects, while objects represent instances of a class. For mobile robot programming, we could have a class "Robot" and create objects representing specific robots:

```
class Robot {
 std::string name;
 int speed;
 // Other attributes and methods...
};

Robot MobileRobot;
MobileRobot.speed = 255; //full speed
```

Lastly, we'll look into exception handlers, they are mechanisms that handle runtime errors, ensuring our robot doesn't crash when it encounters an issue:

```
try {
 // Code that could throw an exception
} catch (const std::exception& e) {
 // Handle exception
}
```

By the end of this course, you'll have a solid understanding of these key C++ programming concepts and be able to apply them to real-world mobile robot programming scenarios. So, let's get started!

8.1 Arrays and strings

8.2 Pointers and references

8.3 Classes and objects

8.4 Exception handling

8.5 Input and output

8.6 Debugging and testing

8.7 Advanced topics threading memory management templates

8.7.1 Bit-field variable type

Variable bit fields are a specific type of data structure in C++ that allows a user to store multiple bit-sized values within a single variable. This can be useful for storing several different values in the same memory space or for compressing data. An example of a variable bit field in C++ is shown below:

```
struct example {
 unsigned int value1 : 4; // Use 4 bits
 unsigned int value2 : 8; // Use 8 bits
 unsigned int value3 : 12; // Use 12 bits
} myStruct;
```

In this example, we have defined a structure called ‘example’ which contains three members - ‘value1’, ‘value2’, and ‘value3’. Each of these members has been defined as a variable bit field using the ‘unsigned int’ data type and the ‘:’ syntax, which allows us to specify the number of bits that each member should use. In this case, ‘value1’ will use 4 bits, ‘value2’ will use 8 bits, and ‘value3’ will use 12 bits. To access these values, we can use the members of the structure, for example, ‘myStruct.value2’.

```
struct adc4 {
  unsigned int value1 : 10;
  unsigned int value2 : 10;
  unsigned int value3 : 10;
  unsigned int value4 : 10;
};

unsigned int adc_val[40];           //40 values
adc4 myAdc[10];                     //40 values

void setup() {
  Serial.begin(9600);
  Serial.println(sizeof(adc_val));  //print 80
  Serial.println(sizeof(myAdc));    //print 50
}

void loop() {

}
```

9 FUNDAMENTAL TASKS IN ROBOTICS

9.1 Move to reference position

9.2 Pick and place operations

9.3 PID Control

9.4 Navigation and mapping

9.4.1 Tasks:

1. Stop the robot when it reaches the end of line.
2. Detecting the end of line can be done by measuring the time that robot spend over the black and white area. E.g. if the robot is driving along the line - the time spent over black and time spent over white area will be quite the same. When line ends the robot will not detect the black area soon and the time spent over white area will increase significantly - and that is the trigger for detecting the end of line.
3. Advanced: Make a function to align (move) the robot back to the line.

9.4.2 Questions:

1. How can we store a data to the controller's memory?
2. How can we measure time in programming loops?
3. What is the purpose of the prog. instr. exit(0); ?

Listing 9.1: Edn of Line Detection.

```
#include <RobotMovingFunctions.h>
const int LIGHT_SENSOR_PIN = A0;
const int SURFACE_BRIGHTNESS_REFERENCE = 400;
int time_on_black = 0;
int time_on_white = 0;

void setup()
{
  setIOpins();
  pinMode(LIGHT_SENSOR_PIN , INPUT);
}
void loop()
{
  int light_sensor_value = analogRead(LIGHT_SENSOR_PIN );
  if ( light_sensor_value < SURFACE_BRIGHTNESS_REFERENCE )
  {
    // BLACK area
    moveLeft();
    time_on_white = 0;  // reset time on white
    time_on_black++;    // meas. time on black
    delay(100);
  }
  else
  {
   // WHITE area
    moveRight();
    // Do similar meas.
    // of time on white
    delay(100) ;
    // If time is signif. longer:
    //        robotStop();exit(0);
  }
}
```

9.4.3 Summary:

9.4.3.1 <++>

9.4.4 Issues:

9.4.4.1 <++>

9.5 Timers and time measurement

Timers and time measurement are important concepts in Arduino programming, as they allow you to perform tasks at specific intervals, measure elapsed time, or synchronize events. The Arduino has several built-in timer modules that you can use in your programs.

Here are some common ways to use timers and measure time in Arduino:

delay() function: This function causes the program to pause for a specific number of milliseconds. For example, the following code will cause the LED on digital pin 13 to blink every second:

```
void loop() {
  digitalWrite(13, HIGH);
  delay(1000);  // wait for 1 second
  digitalWrite(13, LOW);
  delay(1000);  // wait for 1 second
}
```

millis() function: This function returns the number of milliseconds that have elapsed since the Arduino was powered on or reset. You can use this function to measure elapsed time or to trigger events at specific intervals. For example, the following code will turn the LED on and off every 5 seconds:

```
unsigned long previous_time = 0;  // store the previous time

void loop() {
  unsigned long current_time = millis();  // get the current time
  if (current_time - previous_time >= 5000) {  // check if 5 seconds
     have passed
    digitalWrite(13, !digitalRead(13));  // toggle the LED
    previous_time = current_time;  // update the previous time
  }
}
```

Hardware timers: The Arduino has several hardware timers that can be used to generate periodic interrupts. You can use these timers to trigger events at specific intervals without using the delay() function. For example, the following code uses Timer 1 to toggle the LED on and off every second:

```
  void setup() {
    // set up Timer 1 to generate an interrupt every 1 second
    cli();  // disable global interrupts
    TCCR1A = 0;  // set Timer 1 to normal mode
```

9.6 Perception and recognition

10 ROBOTICS APPLICATIONS

10.1 Robotics projects for educational and research applications

10.2 Robotics in industry and everyday life

10.3 Robotics competitions and challenges

10.4 Robotics careers and future opportunities

11 ADVANCED ROBOTICS

11.1 Robotics in artificial intelligence and machine learning

This exercise introduces students to the concept of a simple neuron using a line follower robot. We'll start by creating a basic neuron model with input from a light sensor and output to a PWM-controlled LED.

To understand the basic function of a neuron and how it can be used to control an output device, like an LED, in response to sensor input. This lays the groundwork for understanding how a group of neurons can control a robot.

11.1.1 Steps

1. **Setup the Circuit**
    - Connect the light sensor to analog pin `A0`.
    - Connect the LED to digital pin `D6` (PWM capable), with a 220Ω resistor in series.
    - Make sure to common the ground and supply rail of the breadboard with the Arduino.
2. **Write the Arduino Code**

```
#include "neural_network.h"
const int LIGHT_INPUT_PIN = A0;
const int OUTPUT_PIN = 6;

Neuron my_first_neuron(LIGHT_INPUT_PIN, -0.5, OUTPUT_PIN);

void setup() {
}

void loop() {
  Left_neuron.update();
  delay(100);
}
```

3. **Experiment and Observe**

- Place different light sources or cover the light sensor to observe how the LED's brightness changes in response.
- Discuss how this mirrors the function of a simple neuron, where the input (light sensor) influences the output (LED brightness).

4. **Extensions**

- Replace the LED with a motor to simulate the neuron influencing a motor's speed for robot movement.
- Use two neurons: one for each motor, and discuss how this setup can control a line-following robot.

11.1.2 Conclusion

Through this exercise, students gain a foundational understanding of how simple neurons work by transforming input data into an actionable output. By relating this to a line follower robot, they can see a practical application of AI concepts in robotics. This hands-on approach enhances grasping the dynamic nature of artificial intelligence and its real-world implementations.

11.2 Robotics in computer vision and image processing

11.3 Robotics in natural language processing

11.4 Robotics in swarm intelligence and multi-agent systems