David Andrawes, BSc

Design and Implementation of Controlling Bluetooth Low Energy Devices in Mobile Android Development

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme:
Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Institute of Software Technology

Graz, May 2023

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Acknowledgements

First and foremost, I would like to thank God for he has covered me, helped me, guarded me, accepted me to Himself, spared me, supported me and has brought me to this hour. This accomplishment would have been impossible without His blessings.

I would like to express my deepest gratitude to my supervisor Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany, who always offered me advice and active support and assisted me with his knowledge in this area.

Words cannot express my gratitude to my dear parents and my dear brother for their constant encouragement and unfailing support over all the years. None of this would have been possible without them, for they have made me the person I am today.

I would like to extend my sincere thanks to all my relatives and friends for their tremendous support.

Moreover, I am grateful to my close fellow students for the ups and downs we have shared during our studies, especially during group work, over the past five years.

Finally, I would like to thank the entire Catrobat team for their excellent cooperation and team spirit, especially the Catroid IDE team.

Graz, May 2023 David Andrawes

Abstract

Internet of Things (IoT) devices have increased dramatically in recent years. They can now be found in various applications in everyday life, and it is hard to imagine life without them. Bluetooth Low Energy (BLE) is increasingly being used for networking such devices for transmission due to its extremely low power consumption and compatibility with many smartphones, tablets and computers. However, there is a lack of applications for the quick and uncomplicated implementation of own ideas or projects without profound knowledge in these areas.

In the present work, the existing mobile Android application Pocket Code, with the visual programming language Catrobat, is extended to control external BLE devices wirelessly. The bidirectional communication between IoT devices and the smartphone or tablet is achieved by arranging graphical blocks in the visual programming environment. The practical implementation includes, as an example, the complete control of Arduino boards through the Firmata protocol.

The visual programming language makes it not only easier for non-experts to implement their own IoT projects directly via a smartphone or tablet but also for experts to quickly realise innovative ideas according to the principle of rapid prototyping. This includes creating applications to control external BLE devices with a user-defined graphical user interface (GUI) for interaction. In addition, the implementation design allows for easy expandability to other devices and communication protocols in the future.

Keywords: Pocket Code · Arduino · Bluetooth Low Energy · Firmata · Internet of Things · Rapid Prototyping · Visual Programming

Kurzfassung

Der Einsatz von Internet der Dinge (IoT) Geräten ist in den vergangenen Jahren drastisch gestiegen und ist heutzutage in verschiedenen Anwendungen des Alltags wiederzufinden und kaum wegzudenken. Zur Vernetzung solcher Geräte kommt für die Übertragung vermehrt Bluetooth Low Energy (BLE) zum Einsatz, aufgrund des äußerst geringen Stromverbrauchs und dessen Kompatibilität mit einer großen Anzahl von Smartphones, Tablets und Computern. Jedoch fehlt es an Applikationen für die schnelle und unkomplizierte Umsetzung von eigenen Ideen oder Projekten, ohne dabei über ein fundiertes Fachwissen in diesen Bereichen zu verfügen.

In der vorliegenden Arbeit wird die bestehende Mobile Android Applikation Pocket Code, mit der visuellen Programmiersprache Catrobat, um die drahtlose Steuerung von externen BLE-Geräten erweitert. Dabei wird durch die Anordnung von grafischen Blöcken in der visuellen Programmierumgebung die bidirektionale Kommunikation zwischen IoT-Geräten und dem Smartphone oder Tablet erzielt. Die praktische Implementierung beinhaltet im Zuge dessen exemplarisch die vollständige Steuerung von Arduino-Boards durch das Firmata Protokoll.

Die visuelle Programmiersprache erleichtert nicht nur Laien die Umsetzung ihrer eigenen IoT-Projekte direkt über ein Smartphone oder Tablet, sondern auch Experten bei der schnellen Realisierung innovativer Ideen nach dem Prinzip des Rapid Prototyping. Dies umfasst die Erstellung eigener Applikationen in Verbindung mit der Steuerung von externen BLE-Geräten mit einer benutzerdefinierten grafischen Benutzeroberfläche (GUI) für die Interaktion. Zudem erlaubt das Design der Implementierung künftig die leichte Erweiterbarkeit um weitere Geräte und Kommunikationsprotokolle.

Schlagwörter: Pocket Code · Arduino · Bluetooth Low Energy · Firmata · Internet der Dinge · Rapid Prototyping · Visuelle Programmierung

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADC	analog-to-digital converter
AI	Analog Input
AO	Analog Output
API	Application Programming Interface
ATT	Attribute Protocol
BLE	Bluetooth Low Energy
CC BY-SA	Creative Commons Attribution-ShareAlike
CCCD	Client Characteristic Configuration Descriptor
CI	Continuous Integration
DAC	digital-to-analog converter
DI	Digital Input
DIY	do-it-yourself
DO	Digital Output
FHSS	Frequency-Hopping Spread Spectrum
FIFO	First In - First Out
FOSS	Free and Open Source Software
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GFSK	Gaussian Frequency Shift Keying
GPIO	General Purpose Input/Output
GSM	Global System for Mobile communication
GSoC	Google Summer of Code
GUI	Graphical User Interface
HCI	Host Controller Interface
I^{2}C	Inter-Integrated Circuit
I/O	Input/Output
IC	integrated circuit

IDE	Integrated Development Environment
IDII	Interaction Design Institute Ivrea
IoT	Internet of Things
IPSP	Internet Protocol Support Profile
ISM	Industrial, Scientific, and Medical
ISP	in-system programming
L2CAP	Logical Link Control and Adaption Protocol
LED	Light-Emitting Diode
LoRa	Long Range
MAC	media access control
MIDI	Musical Instrument Digital Interface
MIT	Massachusetts Institute of Technology
MMA	MIDI Manufacturers Association
MSB	most significant bit
NB-IoT	Narrowband Internet of things
OTG	On-The-Go
PC	Personal Computer
PWM	Pulse Width Modulation
RX	Receiver
SIG	Special Interest Group
SRAM	Static Random-Access Memory
Sysex	System exclusive
TDD	Test-Driven Development
TX	Transmitter
UI	user interface
USB	Universal Serial Bus
USI	Universal Synthesizer Interface
UUID	Universally Unique Identifier
UX	User Experience
VPL	visual programming language
Wi-Fi	Wireless Fidelity

XML	Extensible Markup Language
XP	Extreme Programming

1 Introduction

In contemporary times, an increasing number of devices are interconnected and communicate with each other, thereby creating the Internet of Things (IoT). This exponential growth is evidently noticeable and profoundly impacts various aspects of human life, making ordinary activities more convenient. These devices are essentially small embedded systems equipped with sensors and actuators to gather data and execute specific actions based on the acquired information. Due to their reliance on batteries or even button cells for extended periods, the energy consumption of these devices is very constrained.

Since wired connections are unavailable in certain contexts, many devices resort to wireless connections for data exchange. While wireless communication is a simpler and more cost-effective alternative to wired connections, it has its drawbacks. These include heightened security concerns and diminished reliability due to the potential for radio interference. The required range and transmission speed determine the selection of the appropriate communication protocol. However, it is crucial to consider energy efficiency when selecting a protocol since a device's maximum power consumption occurs during transmissions.

In recent years, the wireless communication protocol Bluetooth Low Energy (BLE) has emerged as a clear leader in this field. This is due to its optimised low power consumption and extensive device support. Additionally, it is inherently integrated into all modern smartphones, tablets, and computers, allowing for direct interaction with other IoT devices through these common devices. The technology's advantages include its low cost and flexibility to be configured by certain parameters to support a wide range of applications. It is particularly suitable for applications with short distances and minimal data transfer requirements.

1.1 Motivation

The motivation for this thesis stems from the possibility of employing IoT devices in many ways to realise personal projects and ideas. Nevertheless, this endeavour requires fundamental knowledge of electronics and programming, which can present a formidable obstacle, particularly for non-experts. Even for hobbyists or specialists in this domain, the fast development of prototypes necessitates extensive labour.

Catrobat is a visual programming language that allows programming without prior knowledge by playfully arranging functional blocks. This takes place on a mobile device, such as a smartphone or tablet, using the mobile app Pocket Code, an Integrated Development Environment (IDE). In addition to accessing sensors and actuators of the mobile end device, the app also offers the possibility of connecting external hardware. However, to remain up to date, continuous development is necessary to support the latest technologies.

For this reason, this thesis expands the existing app Pocket Code to include wireless control of external BLE devices, with the first representative symbolically being an Arduino. Arduino, an open-source platform with a large community, is suitable for rapid prototyping, particularly in the hardware context. Its use ranges from small experimental projects for beginners to market-ready professional applications.

The combination of Arduino and Pocket Code unites simplicity in both hardware and software, thus promising immense potential for implementing IoT projects. The visual programming language not only provides non-experts with a simple way to execute their projects but also enables experts to realise innovative ideas quickly in terms of rapid prototyping. Moreover, unlike other solutions, everything runs on the mobile device, from the application's programming to its control via a self-designed Graphical User Interface (GUI). These aspects significantly lower the entry threshold into these domains and enable promising IoT applications that make daily life easier.

1.2 Contribution

This thesis extends the existing Android app Pocket Code to control BLE devices in order to comply with state-of-the-art and thus enable own IoT projects in this respect. Specifically, the thesis integrates the widely used Arduino as the first device. In summary, the thesis includes the following points:

- Implementation of a BLE framework to manage BLE connections;
- Implementation of the Firmata protocol with support for BLE;
- Integration of controlling the Arduino via BLE into the existing Android app Pocket Code.

The implementation of the BLE framework and Firmata protocol are designed in a general way, making their use also outside this context possible. This is particularly significant since there is no open-source implementation of the Firmata protocol with BLE support for Android applications. Moreover, this thesis places substantial emphasis on the design for easy extensibility, allowing the addition of more devices and communication protocols in the future.

Additionally, given the sparse description of the Android Application Programming Interface (API) regarding BLE, this thesis details the handling of BLE in Android development and highlights potential issues in this regard. Furthermore, the thesis provides a comprehensive evaluation and presents promising future ideas based on its findings.

1.3 Outline

The remaining part of this thesis is divided into several chapters as follows. Chapter 2 presents the relevant background for the thesis. This includes a brief introduction to Catrobat, the organisation whose Android app Pocket Code was extended during this thesis. Furthermore, the theory behind the BLE technology used is discussed. Subsequently, there is also an introduction to the employed Arduino hardware and the Firmata protocol used for communication. Following this, Chapter 3 goes into more detail about the design used in the implementation. For this purpose, the general structure is described, and the individual design decisions are explicitly clarified. Thereafter, a detailed explanation of the practical implementation follows in Chapter 4. This includes a presentation of the implementation of the BLE framework and the Firmata protocol in Android, as well as its integration into Pocket Code and its testing. The results and limitations are discussed and evaluated in detail in Chapter 5. Finally, Chapter 6 concludes the thesis with a final summary and an outlook on future work.

2 Background

This chapter provides some background information on aspects of this thesis and is divided into four parts. Section 2.1 starts by giving some information about the Catrobat organisation behind the Pocket Code app, which is extended in the context of this thesis. This is followed by a description of their mobile Android app Pocket Code and the visual programming language (VPL) used. Section 2.2 then introduces BLE, the wireless communication protocol used in this thesis. The differences and advantages compared to the previous version Bluetooth Classic are highlighted, and the individual components of the protocol are explained in detail. Section 2.3 gives an overview of the general functionality of the widely used Arduino hardware, which will be controlled by Pocket Code via BLE in the practical implementation of this thesis. Finally, Section 2.4 explains the theoretical background of the Firmata protocol used for communication.

2.1 Catrobat

The following sections describe Catrobat, the organisation for which the practical part of this thesis is carried out. First, in Section 2.1.1 the organisation itself is presented. Then the Pocket Code application and its features are presented in Section 2.1.2, and finally, the VPL used, also called Catrobat, is described with its components and functionality in Section 2.1.3.

2.1.1 Catrobat Project

The Catrobat[1] Organisation is a non-profit Free and Open Source Software (FOSS) project initiated by Slany [1] at Graz University of Technology in 2010. Its aim is to simplify the introduction to programming, especially for children from the age of eight, to create their own applications. The project was inspired by Scratch, a VPL for children which was developed by the Lifelong Kindergarten Group at the Massachusetts Institute of Technology (MIT) Media Lab [2].

[1] `https://catrobat.org`, accessed: Jan 2, 2023

In contrast to similar visual languages like Scratch[2] or MIT App Inventor[3], Catrobat requires only a smartphone for both development and execution and no additional Personal Computer (PC) is needed. The VPL is based on LEGO® blocks which are stacked together to form a program. This gives the impression of playing for children while coding and avoids text-based programming language problems like syntax errors.

The organisation is divided into different teams, each pursuing another project or task area. These teams mainly consist of students from the Graz University of Technology but also of external developers, especially in the context of programming campaigns for open-source projects such as Google Summer of Code (GSoC)[4]. For this reason, as usual in FOSS projects, the fluctuating number of members results in various challenges. These range from a poor flow of information between members to a lack of expertise due to a deficiency of prior knowledge and a short contribution period. To counteract this, various concepts are used, such as the promotion of pair programming units to pass on information or the need for code reviews before merging to avoid bugs and increase the code quality. Furthermore, agile development is pursued using Kanban with Extreme Programming (XP) and Test-Driven Development (TDD).

2.1.2 Pocket Code

The main project of Catrobat is the mobile app Pocket Code [3], initially Catroid [4], consisting of an IDE and interpreter for the VPL Catrobat. Combining visual blocks makes it possible to easily and playfully create individual apps, ranging from games to animations and much more, without any previous knowledge. Figure 2.1 shows the main screen of Pocket Code. What sets it apart from comparable applications is that it only requires a smartphone or tablet, which are widely available, making programming accessible anywhere and at any time. The application is freely available for Android[5] as well as for iOS[6] and Huawei[7], whereby reference will be made to the Android version in the following, as this will be expanded in the practical part of this thesis.

The aim is to provide an easy introduction to programming and teach programming concepts, especially to children. For this purpose, VPL is used instead of

[2]`https://scratch.mit.edu/`, accessed: Jan 2, 2023

[3]`https://appinventor.mit.edu/`, accessed: Jan 2, 2023

[4]`https://summerofcode.withgoogle.com/`, accessed: Jan 2, 2023

[5]`https://play.google.com/store/apps/details?id=org.catrobat.catroid`, accessed: Jan 2, 2023

[6]`https://apps.apple.com/at/app/pocket-code/id1117935892`, accessed: Jan 2, 2023

[7]`https://appgallery.huawei.com/#/app/C100085769`, accessed: Jan 2, 2023

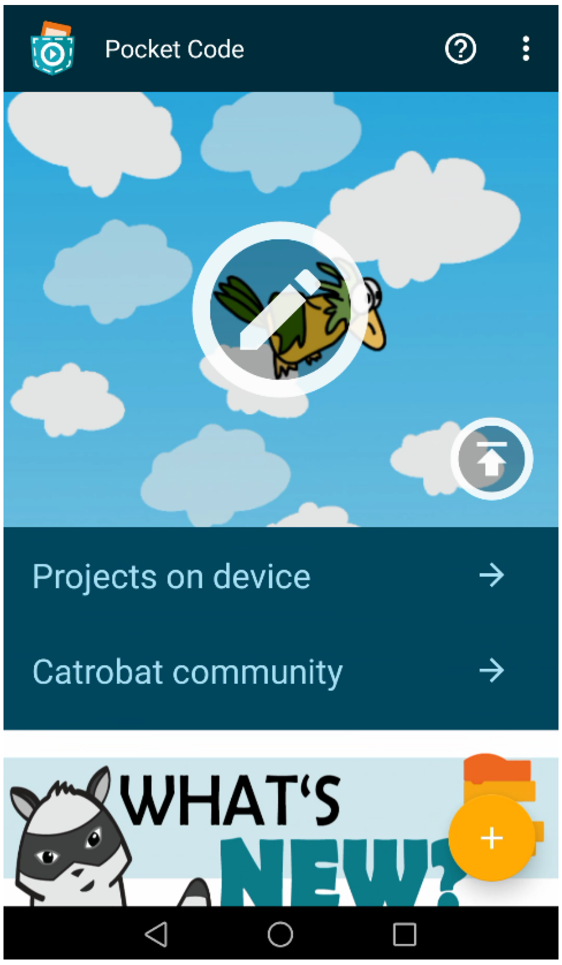
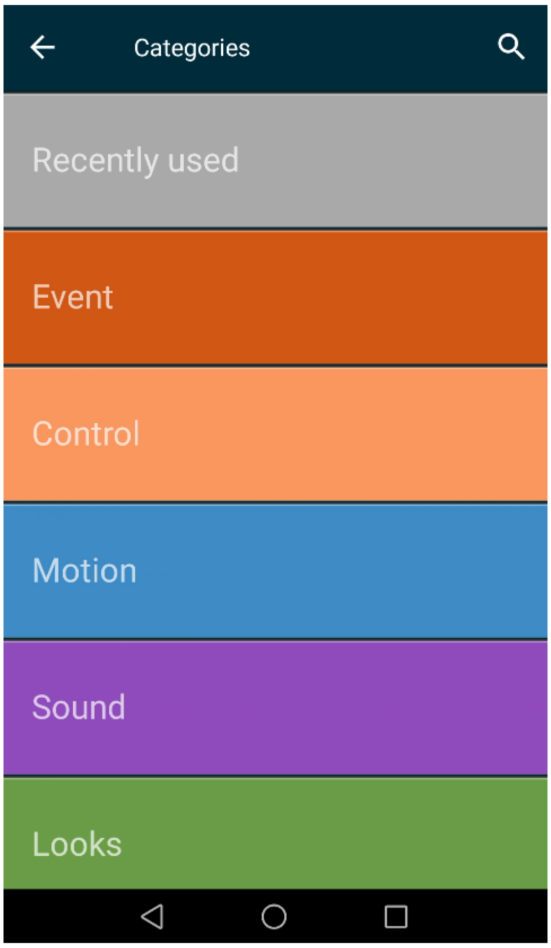


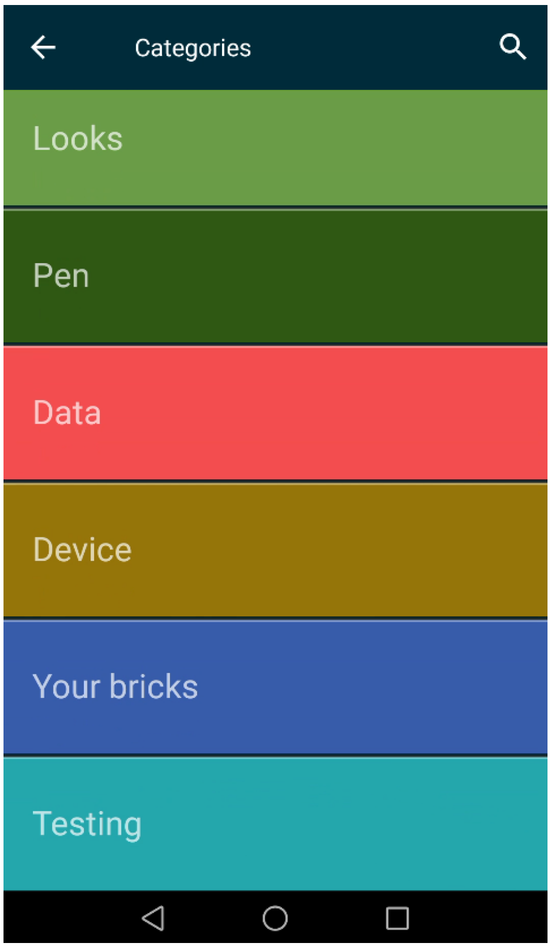
Figure 2.1: Pocket Code Main Screen

text-based programming language to avoid hurdles such as learning a programming language or following a strict syntax. Functional blocks, so-called bricks, are arranged into scripts executed parallel to form a program. Further facilitation, according to the principle of internationalisation (i18n) and localisation (L10n), is the possibility of using the app and the blocks for programming in the respective native language. This includes support for more than 60 languages, making it even easier for children to get started.

The blocks are divided into different categories according to their functionality and affiliation, as shown in Figure 2.2. The integration of actuators and sensors built into the mobile phone, such as the acceleration sensor, gyro sensor or the touchscreen itself, makes it possible to create even more complex programmes. Access to the data is provided by the specially developed formula editor, shown in Figure 2.3, which also includes more complex functions such as object recognition using machine vision. In addition, the app contains numerous built-in extensions for wireless control of external hardware, which can be activated via the app's settings. Furthermore, the app Pocket Paint[8], specially developed by Catrobat, is directly integrated into Pocket Code so that its functions are fully accessible within the app. This powerful graphics editor allows the graphic creation and editing of objects that can be used straight in Pocket Code.



a) Categories Part 1



b) Categories Part 2

Figure 2.2: Pocket Code Categories

Furthermore, the giant online community offers the possibility to exchange experiences and share projects via the community website[9]. It is possible to upload personal projects, browse projects or download others directly within the app. An important concept is to download other projects, modify or extend them yourself and upload them again, which is called "remixing". In addition, projects can be used to generate APK files that can be installed on any device and run as a standalone application.

[8]`https://play.google.com/store/apps/details?id=org.catrobat.paintroid`, accessed:Jan 2, 2023

[9]`https://share.catrob.at/`, accessed: Jan 2, 2023

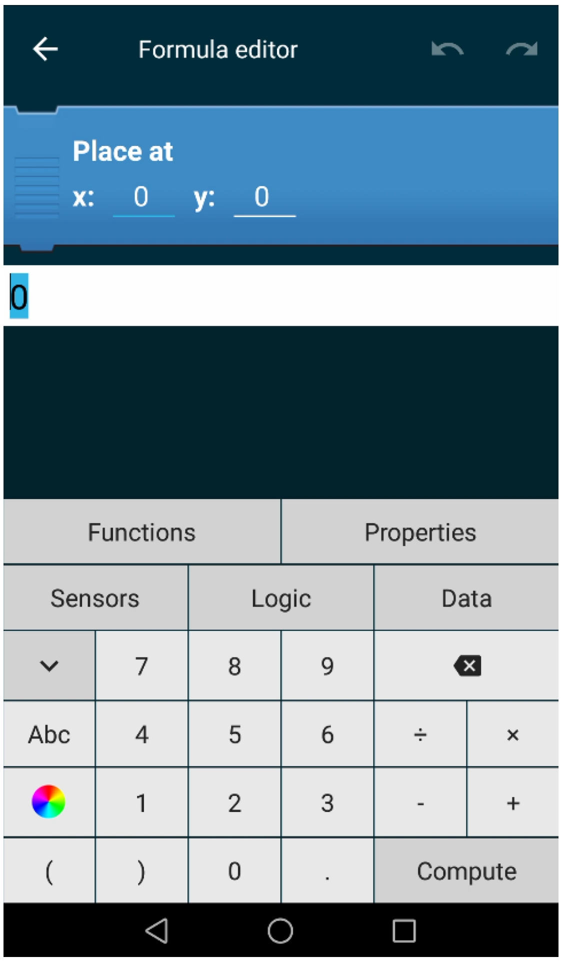


Figure 2.3: Pocket Code Formula Editor

2.1.3 Catrobat Programming Language

The programming language used in Pocket Code has the same name as the organisation, namely Catrobat. This is a pure VPL to facilitate the introduction to programming. For this purpose, individual blocks, so-called bricks, are stacked together and visually arranged to form more complex programmes. An essential difference to other block-based languages is that the blocks are arranged directly below each other, and there are no indentations when nesting. These bricks are divided into categories according to their functional affiliation and are therefore distinguishable by colour. Each of these bricks performs a specific predefined function, but it is also possible to create individual bricks.

A program consists of scenes that can combine several related objects (sprites). Each of these objects, in turn, contains scripts that are executed in parallel. A script consists of a series of bricks that define the logical sequence. Figure 2.4 shows an example of the script view of an object composed of different bricks.

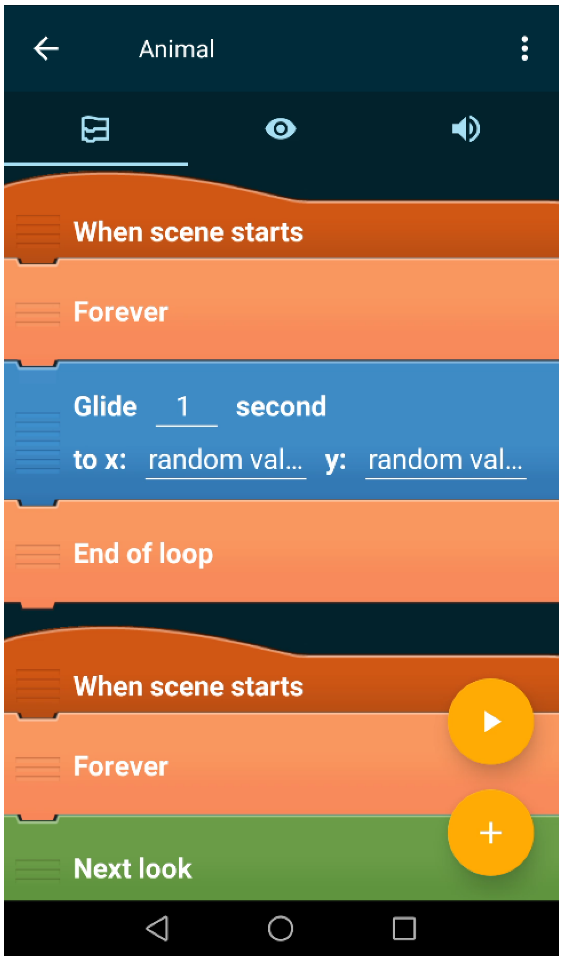


Figure 2.4: Pocket Code Example Script

In the background, the entire VPL Catrobat project is saved in Extensible Markup Language (XML). When the program is executed, the blocks are transformed into XML, which the built-in interpreter can execute in Pocket Code. Conversely, the IDE in Pocket Code displays the graphic blocks by converting them back into the VPL Catrobat from the XML.

The VPL has several advantages, starting with a more accessible introduction to programming without any previous knowledge and ending with a playful method of learning programming concepts. This allows total concentration on creativity during implementation instead of dealing with syntax errors in text-based programming languages.

2.2 Bluetooth Low Energy (BLE)

Intending to develop a communication protocol that can transmit data wirelessly without cables, the Bluetooth Special Interest Group (SIG) [5] created the Bluetooth communication protocol in 1999. This short-range wireless technology for data exchange has since become widespread and developed into a standard. Today, the Bluetooth SIG, responsible for this, comprises more than 30,000 companies in various fields. It is also responsible for continuously developing and releasing new specifications that integrate new features. The latest published specification is version 5.4.

BLE, formerly also called Bluetooth Smart, is a wireless communication protocol published in 2009 by the Bluetooth SIG as the successor to Bluetooth Classic in the Bluetooth 4.0 Core Specification. The main focus was on optimising low energy consumption and cost-effectiveness during its development [6]. It is particularly suitable for applications with short distances and small data volumes. Its widespread use today is mainly due to the fact that most modern user devices, i.e. smartphones, tablets and computers, support the protocol. Its broad application areas range from home automation to its use in wearables.

The following sections explain the properties and the BLE protocol's internal design in detail. In particular, the essential parts for understanding the use of the BLE API in Android and its background concerning the practical implementation in this thesis are discussed in more detail. Section 2.2.1 goes into the basics of BLE and describes the general operating mode as well as its limitations. Section 2.2.2 illustrates the BLE protocol stack and describes its individual components. Finally, Section 2.2.3 discusses the usage of BLE as a wireless communication protocol in IoT applications and compares it with other wireless communication protocols.

2.2.1 Fundamentals

BLE differs from its predecessor, Bluetooth Classic, in many aspects. The most significant difference is in power consumption and data throughput, as Bluetooth Classic is designed for high data throughput and BLE, in contrast, is optimised for low power consumption. On the one hand, Bluetooth Classic is suitable for large transmissions and is, therefore, more power-intensive than BLE, which is designed for small and short transmissions. This also makes both protocols incompatible. However, dual-mode devices are compatible with both Bluetooth Classic devices and BLE devices by combining both protocol stacks.

The protocol has been optimised for use in IoT devices with limited resources due to their size. Regarding communication protocols, the energy resource is particularly affected, as the transmission times are the main factor in terms of power consumption. In order to achieve low power consumption with BLE, two main measures are taken. On the one hand, the general power consumption must be reduced, and on the other hand, the power consumption peaks during transmission times must be flattened. The first is achieved by including many long so-called sleep phases in which the radio, mainly responsible for the power consumption, is switched off. For the second, small amounts of data are transmitted at a slow speed, which results in a flattening of the current peaks.

This section is based on [6]. The BLE radio uses the 2.4 GHz Industrial, Scientific, and Medical (ISM) band for communication and divides it into 40 channels. The three channels, 37, 38 and 39, are used as advertising channels, and the rest as data channels. Since other technologies, such as Wireless Fidelity (Wi-Fi) and Bluetooth, also use this frequency band, Frequency-Hopping Spread Spectrum (FHSS) is used to minimise interference. This means the channel is changed for each new connection according to a certain formula. Like Bluetooth Classic, BLE also uses Gaussian Frequency Shift Keying (GFSK) for modulation with a 1 Mbit/s modulation rate. The theoretical throughput is 1 Mbps, which in practice is significantly lower. A trade-off must be made for the maximum range, as the range increases with a higher transmission power but also increases power consumption. For this reason, BLE concentrates mainly on short ranges in the range of 2-5 m, although ranges of 30-100 m are also possible.

2.2.2 BLE Protocol Stack

The BLE stack is formed from several already existing protocol layers. It can basically be divided into two parts: the controller, the lower part, and the host, the upper part. On top of this is the application for the execution itself. These three together form the main components of a BLE device and can either be installed in a common integrated circuit (IC) or in separate ICs communicating via a serial interface. In this case, the controller and the host are connected to each other via the optional Host Controller Interface (HCI) interface to enable standardised communication between them. This is especially important to ensure interoperability between controllers and hosts from different manufacturers. This described BLE stack is shown in Figure 2.5. When sending, the layers are passed through from top to bottom, and the BLE packets are fragmented accordingly. When receiving, however, the sequence is reversed from bottom to top, and the raw data is recombined so that the original BLE packet is re-created.

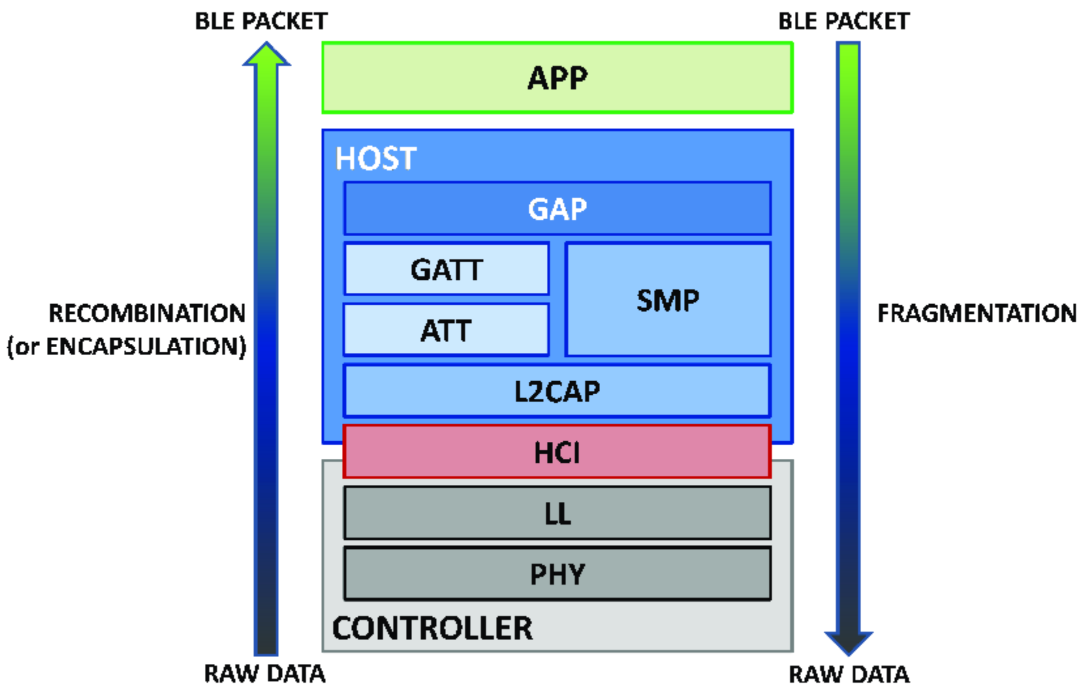


Figure 2.5: BLE Protocol Stack [7]

Application

The application is located at the top of the protocol stack. It serves to provide the user with an interface to the BLE protocol stack, consisting of the host and controller.

Host

The upper layers are referred to as the host. These build on the lower layers and thus allow more complex functions based on them. These are typically executed on a host such as a smartphone. It consists of the following layers:

- Generic Access Profile (GAP)
- Generic Attribute Profile (GATT)
- Logical Link Control and Adaption Protocol (L2CAP)
- Attribute Protocol (ATT)
- Security Manager Protocol (SMP)
- Host Controller Interface (HCI), Host side

Controller

The controller comprises the lower layers. It takes care of the low-level functions on which the upper layers are based. This is usually implemented in a Bluetooth chip. The controller consists of the following layers:

- Host Controller Interface (HCI), Controller side
- Link Layer (LL)
- Physical Layer (PHY)

The most important parts of the BLE protocol stack in terms of practical implementation in this thesis are the upper layers Generic Access Profile (GAP), Generic Attribute Profile (GATT) and Attribute Protocol (ATT), as their backgrounds need to be understood for using the BLE API in Android.

Generic Access Profile (GAP)

GAP is the highest layer in the BLE protocol stack and must be implemented by the devices. It enables BLE devices from different manufacturers to work together, as it defines a standard for this. This includes everything from searching for connections to establishing connections and exchanging data. Communication can take place either connectionless via broadcasting or connection-oriented via connections. In the former case, data is transmitted in one direction only, with the sender broadcasting data and several receivers listening to it being able to receive it in return. Whereas with the second, a direct connection is established with the partner after a defined connection setup, and the data can be exchanged bidirectionally with only this partner. For this purpose, the following four roles are assigned to BLE devices:

- Broadcaster:

  A device that sends out data and can be read by interested devices. For this purpose, advertising packets containing the data are broadcast periodically, which can be received by all interested devices listening to them.

- Observer:

  A device for receiving broadcasted data. For this purpose, advertising packets are listened to and read out if there is interest in the data.

- Central:

  A device that listens to advertising packets from possible connection participants and then actively initiates a connection setup through a request.

- Peripheral:

  A device that sends out advertising packets in order to be found by a central for a connection setup and to be able to exchange data after a successful connection setup.

A device can establish several connections and assume different roles in each connection. Specification version 4.0 did not allow this for the role peripheral because, after a successful connection to a central, the device no longer sends out advertising packets and can consequently no longer be found by another central. However, this has been extended since specification version 4.1, so multiple connections are also possible for peripherals.

Generic Attribute Profile (GATT)

GATT is located in the BLE protocol stack above ATT and thus builds on it. It regulates the exchange of data between two connected BLE participants [8]. For this purpose, a hierarchical system is defined that organises the data. This consists of profiles, services, characteristics and descriptors. Two roles are defined: GATT Server and GATT Client, which coincide with ATT. The GATT client can explore the services offered by the GATT server and subsequently read and write its characteristics. Figure 2.6 shows the structure of the GATT server consisting of several services, and these, in turn, of different characteristics and descriptors. SIG predefines profiles, but user-defined profiles with their services, characteristics and descriptors can also be created for their applications.

- Service:

  Represents a specific function and bundles associated characteristics.
- Characteristic:

  An attribute with all its properties. It is included in a service and described in more detail by descriptors.
- Descriptor:

  Describes the characteristics in more detail.

Attribute Protocol (ATT)

ATT is located in the stack in the host part above Logical Link Control and Adaption Protocol (L2CAP) and uses this to transmit data. It also offers its services to the GATT layer above it. It regulates the data exchange, which includes finding out as well as writing and reading attributes. It operates according to a client-server model. The participant acting as the server offers

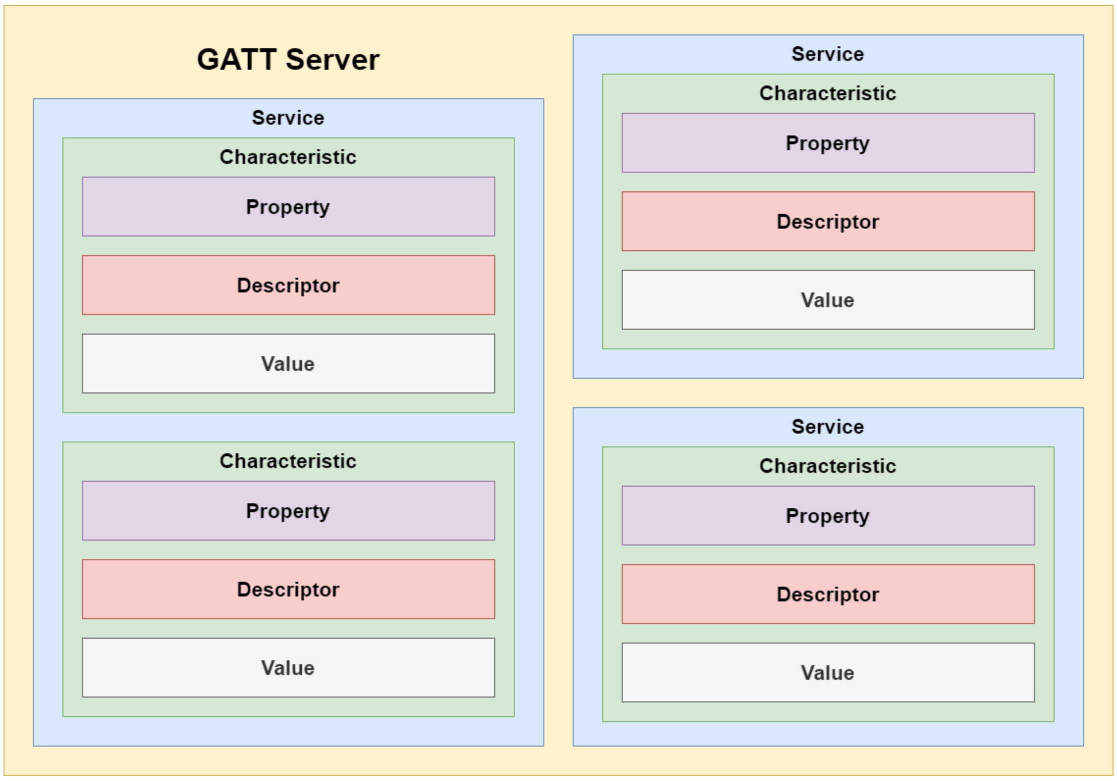


Figure 2.6: GATT Hierarchy [6]

various attributes to the client. On the one hand, the participant acting as a client can find out these attributes and, on the other hand, read or write to them. For this purpose, the device can lead one or the other role, but also both simultaneously. However, only one server may run per device at the same time.

So-called attributes are used to map the data. An attribute comprises the following components [8]:

- Attribute Handle:

  A unique indexing of the attribute by a number (0x0001-0xFFFF, as 0x0000 is reserved) on the server side so that the attributes are uniquely identifiable for the client. Thus, this unique handle for each attribute can be used for all operations between server and client.

- Attribute Type:

  Describes what this attribute is about using a Universally Unique Identifier (UUID).

- Attribute Value:

  The attribute's current value can be written or read depending on the permissions.

- Attribute Permissions:

  Determine the permissions for access to the respective attribute by the client. On the one hand, write and read access, but on the other hand also whether encryption, authentication or authorisation on the client's part are necessary for access.

In addition, the server can also send data to the client by itself. This happens when the client sends a command in which the attribute is subscribed. This has two significant advantages: on the one hand, the client does not have to constantly query the value, so-called polling, but is notified directly by the server when it changes. On the other hand, this method is much more energy-efficient, as unnecessary polling with a constant value is spared, and thus the sleep phases are not constantly interrupted. A distinction is made between two types of notification from the server, namely Notification and Indication. The essential difference is that with Indication, after the notification of the changes in the value from the server to the client, the client still has to send a confirmation message back to the server.

Universally Unique Identifier (UUID)

BLE uses so-called UUIDs for the identification of attributes. These consist of 128 bits (16 bytes). This length allows random UUIDs to be generated that are globally unique with an extremely high probability [6].

The BLE specification also defines two other shortened formats consisting of only 16 or 32 bits, as the total length (16 bytes) would take up a lot of space in the 27-byte data payload of the link layer. However, these shortened formats are only used for UUIDs standardised by the Bluetooth SIG for, e.g. services and characteristics. For non-standardised UUIDs or UUIDs not based on the Bluetooth base for own applications, however, the original format with 16 bytes must be used. Formula 2.1 can be used to calculate the 128-bit UUID from the shortened format, which is always used in the background.

$$\mathrm{UUID}_{\text{128-bit}} = \mathrm{UUID}_{\text{16/32-bit}} \cdot 2^{96} + \mathrm{UUID}_{\text{Bluetooth-Base}} \tag{2.1}$$

The Bluetooth base UUID required for this is the following:

$$00000000 - 0000 - 1000 - 8000 - 00805F9B34FB$$

The formula used for conversion shifts the shortened UUID by 96 bits and adds the Bluetooth base UUID to it. The shortened format is therefore added to the remaining 32 bits (front 4 bytes), consisting only of zeros. This shortened format can consequently be inserted directly. This means that the positions marked with x are replaced directly with the shortened UUID starting from the right and filled with leading zeros in the following UUID:

$$xxxxxxxx - 0000 - 1000 - 8000 - 00805F9B34FB$$

2.2.3 BLE and Internet of Things (IoT)

The number of IoT devices is exploding and can be found in various areas of everyday life. Connectivity is an essential component of networking in these IoT applications. Limited power consumption is an important factor to consider with these increasingly smaller embedded systems [9]. This is due to the fact that such devices have to operate on a battery or even a coin cell for a long period. Since the power consumption peaks occur during the transmissions, the selection of the respective communication protocol significantly influences the possible operating time. Therefore, the most suitable communication protocol is selected based on the application area and its requirements.

Figure 2.7 shows a diagram of various wireless communication protocols used in IoT applications. They are compared and classified according to the following characteristics: range, data rate & energy consumption and cost. It can be seen that, in contrast to other technologies, BLE is characterised by its low power consumption and low cost. It is suitable for applications with small distances, which are, however, sufficient in most IoT applications. Thus, BLE falls into the same category as the Zigbee and Z-Wave communication protocols. However, due to its broad support, especially in consumer devices such as smartphones, BLE is well-suited and preferred over the other two comparable protocols [10].

BLE is particularly suitable for applications with power consumption limitations where little data is to be exchanged over short distances, such as for so-called BLE beacons. It is increasingly used in IoT devices as a wireless communication protocol for such applications. On the one hand, it was developed exclusively for the IoT sector and on the other hand, it is also constantly being optimised in this direction to adapt to the latest state of the art. One such example is the introduction of Internet Protocol Support Profile (IPSP) in Specification 4.1, which allows BLE devices to exchange IPv6 packets directly over the internet via gateway devices such as a router or smartphone.

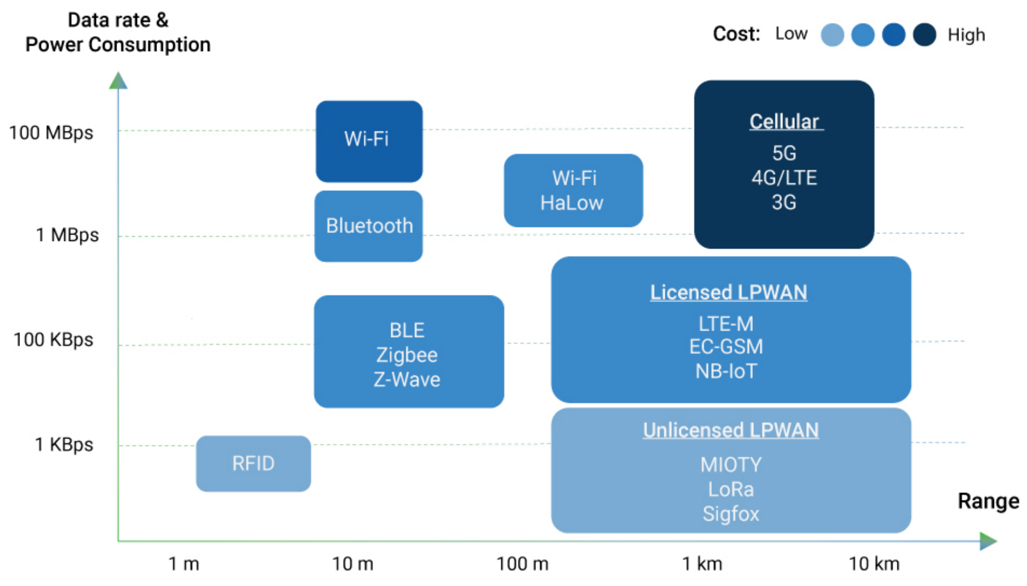


Figure 2.7: Comparison of IoT Wireless Communication Protocols [11]

2.3 Arduino

Arduino was launched in 2005 at the Interaction Design Institute Ivrea (IDII) and was initially intended for students with no prior knowledge of electronics and programming for rapid prototyping [12]. Through its widespread use, it has evolved into an open-source platform that significantly facilitates entry into hardware prototyping in the field of electronics. Its applications range from small do-it-yourself (DIY) projects to professional industrial use. Their vision is to break down barriers to entry and make Arduino accessible to everyone, making people's lives easier. Due to the open-source aspect, a gigantic worldwide community is also behind it, which shares its knowledge extensively. Consequently, many resources are available, from numerous detailed instructions on completed projects to exchanging experiences and help in forums. The founders of Arduino see the following advantages over comparable microcontrollers [13]:

- Inexpensive
- Cross-platform
- Simple, clear programming environment
- Open source and extensible software
- Open source and extensible hardware

The following sections describe the Arduino hardware (Section 2.3.1) and the software (Section 2.3.2). Finally, Section 2.3.3 describes its relation to IoT, as it is used as such in the practical part of this thesis.

2.3.1 Arduino Hardware

The hardware, the Arduino board, is a microcontroller board and can be extended with sensors and actuators depending on the application, thus forming a so-called physical computing system [14]. These boards are available in different versions, which differ in their equipment. Arduino owes its large community to the open-source concept it uses, both in hardware and software. The original files, from the circuit diagrams of the boards to the source code, are shared under the Creative Commons Attribution-ShareAlike (CC BY-SA) 4.0 license. This allows their use, extension at will and dissemination, which explains the reason for the many available imitations. The only condition is to reuse the same license, thus forming a never-ending cycle and leading to numerous contributions by the community. The Arduino hardware, by its design, is not only suitable for entry into the world of electronics but is also designed for fast hardware prototyping, which is a tremendous added value even for experienced users. This allows users to build different circuits easily and quickly with the help of a breadboard.

Over the years, many variations of the Arduino board have been developed, each representing an improvement over its predecessor or being adapted to a specific application area. However, their essential equipment is almost the same for all of them. Figure 2.8 shows an Arduino Uno and its main components, as it is one of the most common boards due to its beginner-friendliness.

Most boards consist of a Microchip (formerly Atmel) AVR microcontroller from the megaAVR series and are operated with a supply voltage of 5 V or 3.3 V. These differ among themselves in the size of the Flash memory, Static Random-Access Memory (SRAM), the clock frequency and the provision of other additional features. The power is supplied directly via Universal Serial Bus (USB) or an external power source. Additionally, the USB port is also used to program the board with a preloaded bootloader using a separate USB to Serial chip. They are also equipped with Input/Output (I/O) pins, both digital and analog, to connect external components. Some of these pins are also equipped with an alternative function, such as necessary connections for different communication protocols or provide an additional function like Pulse Width Modulation (PWM). Furthermore, it also incorporates other pins, like pins for powering external components or pins for an external reference voltage. Moreover, the hardware can be extended externally via modules or via so-called shields, which can be plugged on directly.

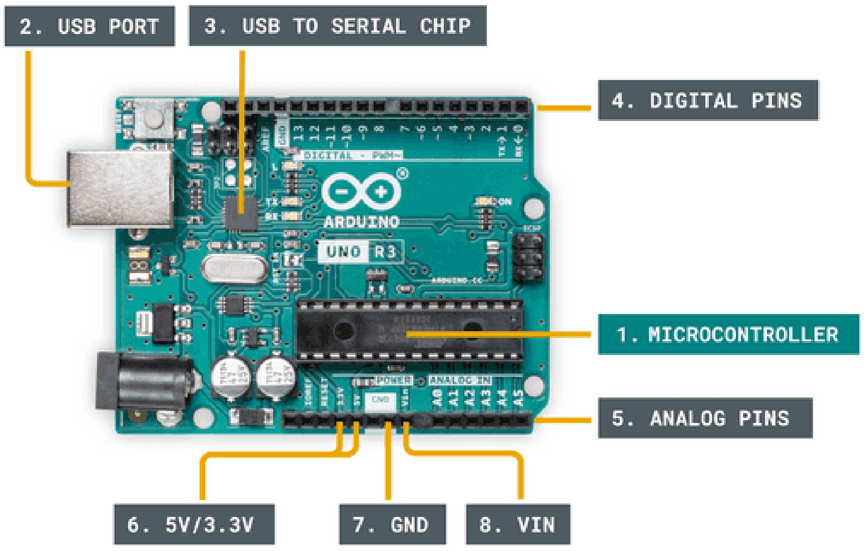


Figure 2.8: Main Components of an Arduino Board [15]

2.3.2 Arduino Software

After the electronic circuit has been built, the software behind it and thus its behaviour can be programmed. The great advantage of such programmable microcontroller boards, such as Arduino, is the possibility of achieving a different behaviour with an unchanged circuit solely through software. The Arduino is programmed in the Arduino IDE, which is based on Processing[10]. It uses its own Arduino programming language, which is based on Wiring[11]. This is a simplified language based on C/C++ to make programming easier. This results in the following procedure for each desired function change:

1. Connecting the board to the computer
2. Writing the sketch (program)
3. Uploading the sketch to the board
4. Executing the sketch in a continuous loop on the board

When uploading, the code is converted into the C language, the programming language usually used for system-level programming. This C source code is then translated by the avr-gcc compiler into machine code executable by the microcontroller. Unlike many other microcontrollers, the Arduino is programmed directly through the USB port rather than through an in-system programming (ISP). For this purpose, the board is reset during the upload, and the so-called

[10] `https://processing.org/`, accessed: Jan 10, 2023
[11] `http://wiring.org.co/`, accessed: Jan 10, 2023

bootloader (runs at each reset) checks whether a sketch is to be transferred. If this is the case, the translated machine code is written to the non-volatile flash memory, which is retained even after a restart, so the program can be executed again as soon as the board is re-powered. On the other hand, stored values are written to the volatile SRAM at runtime, which is erased when the board is powered off or restarted.

Listing 2.1 shows the basic structure of each Arduino program, called sketch. Each sketch consists of at least these two functions, *void setup()* and *void loop()*. The former is executed once the microcontroller is started or reset, mainly containing initialisations. The second is executed after *setup()* function in a continuous loop as long as the board is powered and contains all the logic related to I/Os.

Listing 2.1: Arduino Code Structure

```
void setup() {

}

void loop() {

}
```

The Arduino acts as an interactive device and follows a constantly repeating cycle consisting of input, process and output when carrying out certain tasks. This concept of a closed loop system or a so-called feedback control system can be seen in Figure 2.9. This involves the microcontroller reading in the sensors, evaluating them according to the logic predefined in the software, and ultimately setting the corresponding actuators to interact with the environment. For this, libraries can be used for programming, simplifying communication to these external devices.

2.3.3 Arduino and IoT

Especially the newer versions of the Arduino boards have integrated radio modules and can therefore communicate via wireless communication protocols. Wi-Fi, Bluetooth, Long Range (LoRa), Global System for Mobile communication (GSM), Narrowband Internet of things (NB-IoT) and many other protocols are supported [15]. Depending on the application area and its requirements, the most suitable protocol can be identified, and the appropriate compatible board selected. This makes Arduino boards also well-suited as IoT devices and can be used as such.

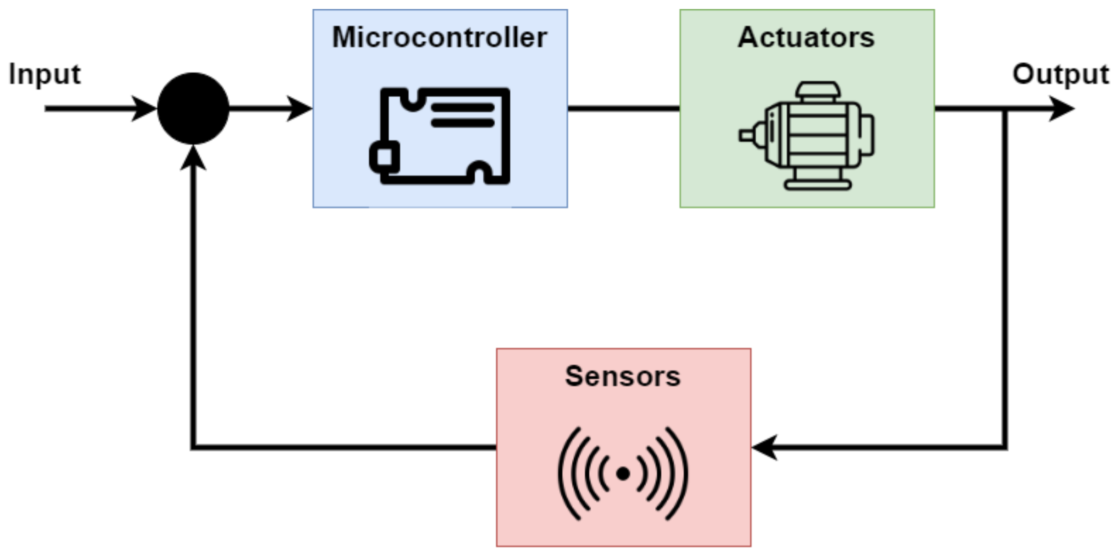


Figure 2.9: Microcontroller Operating Cycle

In order to cover the rapidly growing sector of IoT applications, specially designed boards have been developed. For regular IoT projects, the MKR family is suitable on the one hand, which is aimed at computationally intensive applications, and the Nano family, on the other, stands out due to its small size and directly integrated sensors. In contrast, the Portenta family addresses advanced and industrial IoT applications in combination with artificial intelligence applications. These boards offer high processing power and allow secure and reliable operation even in critical infrastructures through the integrated secure element. Furthermore, the functionality can be extended correspondingly with external modules or shields so that almost any board can be used for such purposes.

Another helpful feature is the Arduino IoT Cloud, which can be used to control boards and monitor their data via a user-friendly interface. However, this option is limited to 2 devices in the free version and has further limitations compared to the paid subscription. In summary, the Arduino fulfils the requirements of an IoT device and, thanks to its numerous simplifications, makes it easy to realise IoT projects.

2.4 Firmata

In order to be able to communicate with other participants, an orderly and clearly defined process is required, which is achieved by means of a standardised protocol. Therefore, this chapter deals with the so-called Firmata protocol. For this purpose, the protocol is first described, and the used Musical Instrument Digital Interface (MIDI) message format is introduced. Afterwards, the different message types of the protocol are described.

2.4.1 Protocol

The Firmata protocol was introduced by Steiner [16] with the goal to communicate with a microcontroller, such as Arduino, through software on a host computer. This makes it possible to read and control I/O pins without having to reprogram the microcontroller itself for each modification. It is universally applicable because it is neither bound to specific hardware nor requires a specific communication protocol. Only the messages to be exchanged are clearly defined, and thus every microcontroller can implement the protocol in its firmware and every host computer in its software.

2.4.2 Musical Instrument Digital Interface (MIDI) Message Format

The MIDI protocol is a communication protocol that allows the communication and synchronisation of electronic musical instruments, computers and other devices [17]. The first prototype was initially introduced by Smith and Wood [18] in 1981 under the name Universal Synthesizer Interface (USI), which subsequently got renamed to MIDI after several changes and improvements in 1982. It was standardised by MIDI Manufacturers Association (MMA)[12] the year after [19].

Message Data Format

Each message in MIDI consists of several consecutive bytes. It generally includes a status byte characterised with the most significant bit (MSB) set to one and one or two data bytes with the MSB set to zero. The exception to this are the Real-Time messages which consist only of a status byte and the Exclusive messages which can consist of any number of data bytes.

The Firmata protocol does not use the MIDI protocol as a whole but reuses the MIDI message format because of its efficiency and the easiness in implementation [16].

[12] `https://www.midi.org/`, accessed: Jan 17, 2023

2.4.3 Message Types

The Firmata protocol [20] defines different message types, which are listed in Table 2.1. As mentioned before, the protocol uses the MIDI message format. All messages consist of 1–3 bytes depending on the message type, except System exclusive (Sysex) messages, which can consist of any number of bytes. Since the MSB distinguishes between a command message and a data message, only 7 bits remain for the command area and the data area respectively, whereby the channel information is still included in the command area. Consequently, the protocol supports 16 analog pins with a 14-bit resolution and 128 digital pins (16 * 8-bit ports). Due to its modularity, the protocol can be extended very easily, as only a new command identifier and its parameters need to be defined.

type	command	MIDI channel	first byte	second byte
analog I/O message	0xE0	pin #	LSB (bits 0-6)	MSB (bits 7-13)
digital I/O message	0x90	port	LSB (bits 0-6)	MSB (bits 7-13)
report analog pin	0xC0	pin #	disable/enable (0/1)	- n/a -
report digital port	0xD0	port	disable/enable (0/1)	- n/a -
sysex start	0xF0			
set pin mode (I/O)	0xF4		pin # (0-127)	pin mode (0=in)
set digital pin value	0xF5		pin # (0-127)	pin value (0/1)
sysex end	0xF7			
protocol version	0xF9		major version	minor version
system reset	0xFF			

Table 2.1: Overview of different Firmata Message Types [21]

System exclusive (Sysex) Messages

Sysex messages are a special message type in the Firmata protocol to extend the command set [21]. The main difference is that unlike standard MIDI messages, which are limited to a maximum of 2 data bytes, an unlimited number of data bytes can be exchanged. They are mainly used for configuration messages and allow the easy extension of the protocol with new features.

The structure of such Sysex messages comprises a Start_Sysex message at the beginning, followed by the data bytes and ending with an End_Sysex message. These intermediate data bytes must not have a one as MSB. Otherwise, they will be interpreted as a command. As shown in the structure of a Sysex message in Table 2.2, the Feature ID following the Start_Sysex command can be used to distinguish between the different Sysex message types. This ID is 1 byte in size and differs from the extended ID, which consists of 3 bytes, whereas the first is always zero.

byte 0	byte 1	byte 2 to (N-1)	byte N
Start Sysex (0xF0)	ID (0x01 - 0x7D)	payload	End Sysex (0xF7)
Start Sysex (0xF0)	ID (0x00)	Extended ID + payload (0x00 0x00 - 0x7F 0x7F)	End Sysex (0xF7)

Table 2.2: Sysex Message Structure [21]

These types of messages allow functionalities beyond the standard possibility. These include extended digital and analog I/O possibilities and information queries about the status and capabilities of the microcontroller and its firmware. They also allow the extension to hardware support, such as a servo or stepper motor, or protocols, such as Inter-Integrated Circuit (I^2C).

2.4.4 Usage

For communication using the Firmata protocol, both partners must support the protocol [22]. The Firmata firmware must be uploaded once to the microcontroller to be controlled. This Firmata firmware can be found in the sample programs of the Arduino IDE in the Firmata category. The sketch *StandardFirmata* can be used for serial communication or *StandardFirmataBLE* in the case of communication via BLE. In the case of the latter, the broadcast name to be displayed can be changed in the configuration file *bleConfig.h* beforehand. For some microcontrollers, the program must be adapted due to the use of a different BLE chip or lack of flash memory. For this purpose, the respective manufacturer either provides a suitable sketch or describes the necessary changes. The other participant, the client, must also implement the Firmata protocol on its side for communication with the Firmata firmware on the microcontroller. To implement this, there are numerous client libraries for the most diverse languages, which, however, do not all correspond to the latest state of the Firmata protocol. There is also a Firmata library[13] for the Arduino IDE that can be used to implement custom firmware.

As for implementations of the Firmata protocol for Android applications, there are a few Java implementations and an outdated Kotlin version that can be used as libraries. However, these do not support transmission via BLE and thus cannot be used in this form for the practical part of this thesis. Therefore, this thesis builds on this and extends the implementation to support BLE transmission. This is important because there is no open-source implementation for Android applications.

[13] `https://github.com/firmata/protocol#firmata-client-libraries`, accessed: Jan 17, 2023

3 Design

In this chapter, the design of the practical part of this thesis is explained in more detail. The idea is to be able to control an Arduino via BLE using the existing Pocket Code app. The planning regarding the integration of this new feature into Pocket Code is essential because the app is already very extensive. In addition, this should also ensure easy future expansion to include further communication protocols and devices.

Section 3.1 illustrates the structure of the system and describes its intended functionality. Section 3.2 lists the necessary requirements for integrating these new possibilities into the Pocket Code app. Section 3.3 then describes the hardware used in this thesis, including the Arduino board and exemplary circuits built to verify the functionality. Finally, in Section 3.4, the design for the concrete implementation is discussed, and its structure is shown.

3.1 System Architecture

The system setup for the practical part of this thesis is shown in Figure3.1. This basically consists of two components, an Android smartphone or tablet and an Arduino. The Pocket Code app runs on a smartphone or tablet. The Arduino, on the other hand, must be equipped with the Firmata firmware, as the Firmata protocol is used for message exchange. They communicate via the BLE wireless communication protocol, which requires both to be equipped with BLE capability. Due to this, the maximum BLE range also limits the maximum distance between the two devices.

The goal is to be able to control an Arduino wirelessly via the Pocket Code App. This results in the following simplified procedure:

1. Programming in Pocket Code
2. Connecting to the Arduino via BLE
3. Running the project

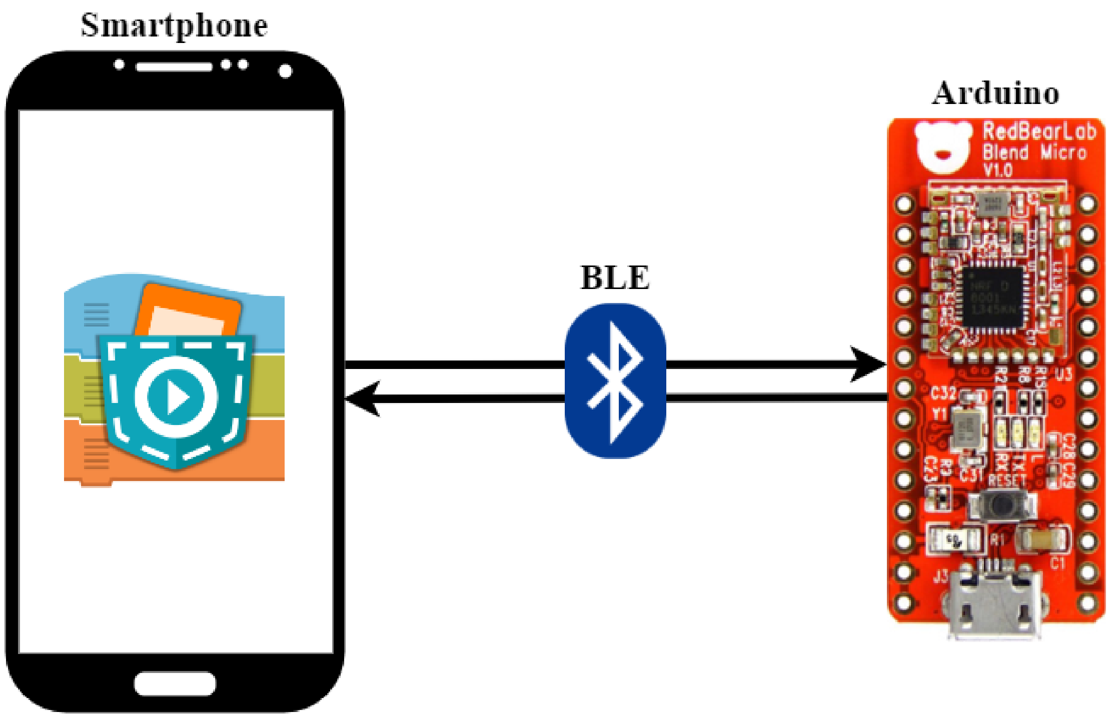


Figure 3.1: Schematic Structure

3.2 Requirements

The central requirement for the implementation is to control an Arduino via BLE from the existing app Pocket Code. Primarily, controlling the digital and analog I/Os should be possible. These actions should be carried out without a significant noticeable time delay. The handling for establishing the connection with the Arduino should be as simple as possible. Furthermore, the newly integrated functions should be bundled in a separate Arduino category in order to remain consistent.

Since the app is also intended for beginners in both software and hardware, this should be considered in the design process. This requires the integration of certain error detection mechanisms to avoid, at best, a crash of the app during the execution of the program on the software side and to protect the used hardware against possible irreversible damage to the components on the hardware side. In addition, its use should be user-friendly.

As the app is constantly being extended, the design should focus primarily on easy extensibility. This should make it possible to add further devices or communication protocols in a simple way in the future. Since the app already has numerous features, the newly added functions must be well covered with automatic tests to detect changes that lead to a malfunction quickly.

3.3 Hardware

This section describes the hardware components used for the implementation in this thesis. Section 3.3.1 describes the Arduino-based Blend Micro Board to be controlled. The circuits built with it to test the new functions integrated into the Pocket Code app are illustrated in Section 3.3.2.

3.3.1 Blend Micro

In this thesis, BLE devices are to be controlled via the app Pocket Code. For this purpose, support for the widely used Arduino was chosen as the first device. However, the Arduino family includes various boards, each with properties and features adapted to its application area.

The only requirement for the Arduino board in this thesis is BLE support. There are boards with BLE already integrated, as well as boards that become BLE compatible by connecting an external module, such as the HM10 module[1]. The two options are basically the same in terms of communication itself. The only difference is the additional configuration of the module in the program on the Arduino side.

Thus, it does not matter which board is used for this thesis, as they only differ in the number of I/O pins and the pin functions. Other boards can easily be added by integrating their configuration and subsequently selecting them in the app. This information is used for error detection to intercept actions on non-existent pins and eventually avoid malfunctions.

In the context of this thesis, the Arduino-based Blend Micro Board, shown in Figure 3.2, is used. This board was developed by the company RedBearLab and is part of the Arduino@Heart program. The aim was to make IoT projects more feasible for Makers by combining the Arduino with BLE on a single board [23], which coincides with the intention of this thesis.

[1]`http://www.jnhuamao.cn/bluetooth.asp`, accessed: Jan 25, 2023

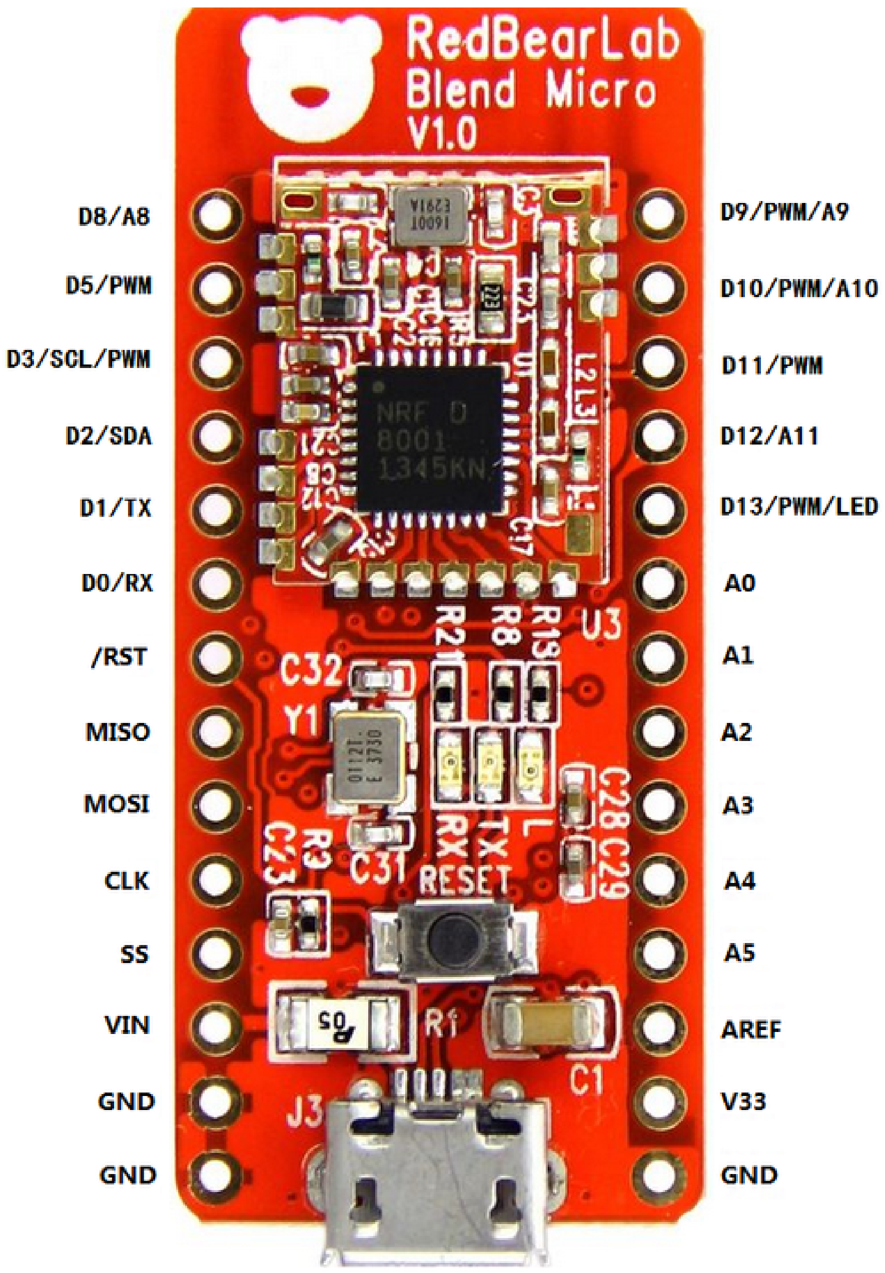


Figure 3.2: Blend Micro Pinout [24]

As stated in the specification [23] the Blend Micro Board runs on an Atmel ATmega32U4 with 8 MHz and supports BLE 4.0 through the Nordic nRF8001 chip. It can be powered by its Micro USB connector or an external input voltage between 3.3 V-12 V. However, the operating voltage is limited to 3.3 V as the nRF8001 chip only supports this. Table 3.1 lists all the specifications of the Blend Micro Board. Its programming is done as usual via the Arduino IDE by adding the board through the Board Manager[2]. In order to use its BLE functions, the relevant BLE libraries must be installed via the Library Manager[3].

Microcontroller	Atmel ATmega32U4
Wireless Chip	Nordic nRF8001
Operating Voltage	3.3 V
Input Voltage	5 V (USB) 3.3 V-12 V (VIN)
Clock Speed	8 MHz
Connectivity	Bluetooth 4.0 Low Energy micro-USB Serial (TX/RX) I2C SPI
Flash Memory	32 KB
SRAM	2.5 KB
EEPROM	1 KB
Dimensions	43.6 x 18.4 x 4.3 mm
Weight	4 g
Power Consumption	2 mA
I/O Pins	17

Table 3.1: Blend Micro Specification [23]

3.3.2 Experimental Setup

This section describes the circuits built to test the various functions integrated into this thesis in practice. These are only basic circuits to illustrate the functionality. However, these can be changed or extended as desired, as there are no restrictions regarding the circuit. The Firmata firmware must first be loaded onto the Arduino in order to be able to communicate with the Pocket Code

[2]`https://github.com/RedBearLab/Blend/blob/master/Docs/BoardsManager.pdf`, accessed: Jan 25, 2023

[3]`https://github.com/RedBearLab/Blend/blob/master/Docs/LibraryManager.pdf`, accessed: Jan 25, 2023

app. The following circuits were created with the software Fritzing[4], which is a software to design electronic circuits on the computer.

The I/O pins can be configured as digital or analog input or output. The difference between analog and digital is that analog pins can have many different values, depending on the resolution, between the low and the high level. Some pins are also equipped with additional functions and can be used as alternative. To use a pin, its mode must be defined beforehand. The pins are set as inputs by default but should still be designated as such. In the following, these different modes are described in more detail and illustrated by an exemplary circuit.

Digital Output (DO)

For a pin to be used as a Digital Output (DO), it must first be configured as an output. In addition to the digital pins, all other pins can be used, which can also function as such. A DO can be set to LOW or HIGH, and depending on this, 0V or the operating voltage (in this case 3.3 V) is applied to the pin. The current that flows is sufficient for an Light-Emitting Diode (LED) or most sensors but not for more current-intensive components such as a motor, which would have to be supplied externally.

The simplest circuit to test the function of the DO is to drive an LED and switch it on and off. Figure 3.3 shows the experimental setup consisting of the Blend Micro Board, an LED and a resistor. In this case, pin D9 is used, but any other digital pin can be selected instead. The series resistor has a value of 220 Ω and is needed to limit the current through the LED. This value depends on the operating voltage and the used LED's forward voltage. In addition, the polarity must be observed when connecting the LED. If the output is switched to HIGH, the current flows, and the LED lights up. This can be switched off again by writing LOW to the output.

Digital Input (DI)

The pins are defined as input by default, but it is better to define them explicitly for Digital Input (DI) as well. As with the DOs, all pins can be used. The DI also distinguishes between the two states, LOW and HIGH, depending on whether the voltage at the input is below or above the threshold voltage (in this case 1.5 V). If there is no voltage at the input, a floating state is created at the pin, and the input reads random states mistakenly. Because of this, either a pull-down or pull-up resistor is needed to have a clearly defined voltage level at the input in

[4] `https://fritzing.org/`, accessed: Jan 25, 2023

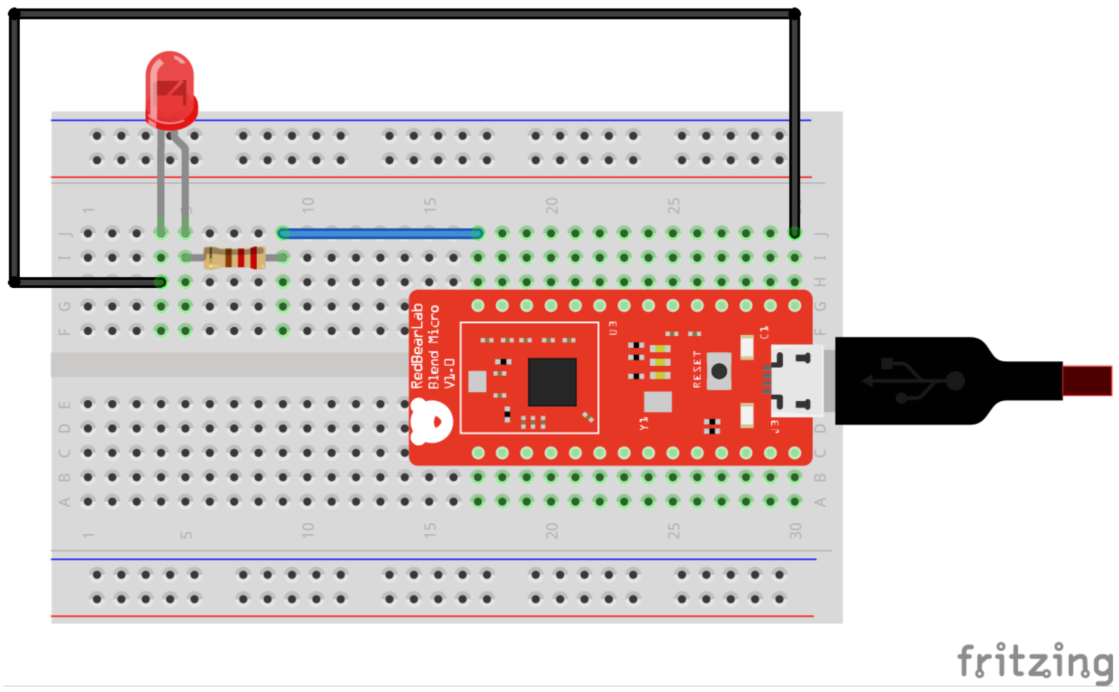


Figure 3.3: DO Circuit

such cases. These are connected externally, or in some boards already built in internally, so their use only has to be set when defining the pin as an input.

The corresponding circuit can be seen in Figure 3.4, which shows an exemplary use of a DI. Here, the state of a push button is read, which supplies LOW at the input in the unpressed case and HIGH in the pressed case. A pull-down resistor with 10 kΩ ensures that the LOW state is definitely present at the input to avoid a floating state in the unpressed case.

Analog Output (AO)

To output an analog signal, there are basically two different possibilities, on the one hand, via an Analog Output (AO) and, on the other hand, via a PWM signal. The difference is that an integrated digital-to-analog converter (DAC) is used with AOs, whereas with PWM, an analog signal is obtained through a digital pin. A square wave signal with a specific frequency is generated by quickly switching between the two states, LOW and HIGH. The duty cycle determines the ratio between the on and off time and thus simulates the AO signal, which is a continuous voltage between 0 V and the supply voltage (in this case 3.3 V). The duty cycle can be any value between 0-255 with 8-bit resolution, where 255 corresponds to 100 % duty cycle, which means that the output is continuously in the HIGH state. Furthermore, not any digital pin can be used for PWM, but

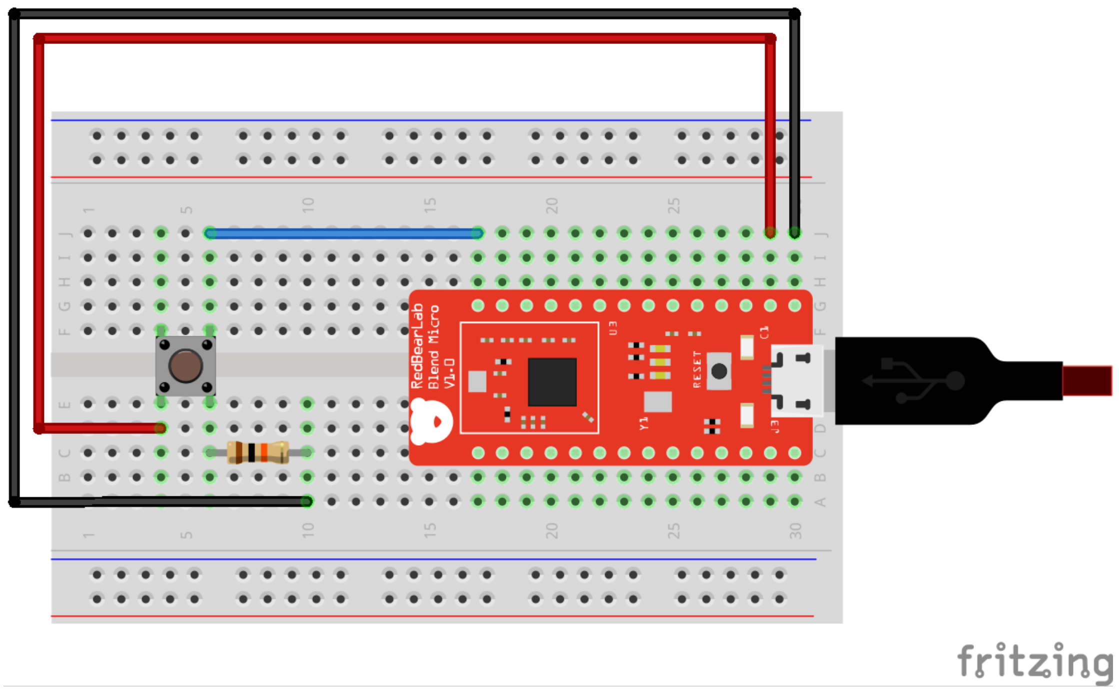


Figure 3.4: DI Circuit

only those specifically designated for it in the pinout. The AO can be used, for example, to adjust an LED's brightness or a motor's speed.

Since the Blend Micro Board does not have an integrated DAC, the PWM method must be used. For this purpose, the circuit in Figure 3.3 can be reused, as the digital pin D9 used is also a PWM pin. In this case, the LED can not only be switched on and off, but the brightness can be adjusted by the duty cycle, and thus the LED can be dimmed as desired. Any PWM value between 0–255 can be set, whereby the value 0 corresponds to a switched-off LED and the value 255 to the maximum brightness.

Analog Input (AI)

An Analog Input (AI) reads and converts analog signals into a proportional value using an integrated analog-to-digital converter (ADC). This value depends on the resolution of the ADC and is between 0-1023 for a 10-bit resolution, which corresponds to 3.22 mV per unit at a supply voltage of 3.3 V. The AIs are specially marked in the pinout but can also be used as General Purpose Input/Output (GPIO) pins. In this case, too, reading an input that is not connected will lead to incorrect random results. The AI can be used for analog sensors, such as a temperature sensor, to read the measured temperature constantly.

The circuit in Figure 3.5, consisting of the Blend Micro Board and a 10 kΩ potentiometer, shows a symbolic use of an AI. The two outer pins of the potentiometer are connected to the supply voltage and ground, respectively. The middle pin, the so-called wiper, is connected to an AI, in this case A0. Depending on the wiper's position, the resistance changes and, subsequently, the voltage applied to the input. The integrated ADC converts this voltage into a proportional value between 0-1023. In this case, 0 V at the input corresponds to the value 0, and the supply voltage at the input, in this case 3.3 V, corresponds to the value 1023. With this resolution, the smallest detectable voltage at the input corresponds to 3.22 mV.

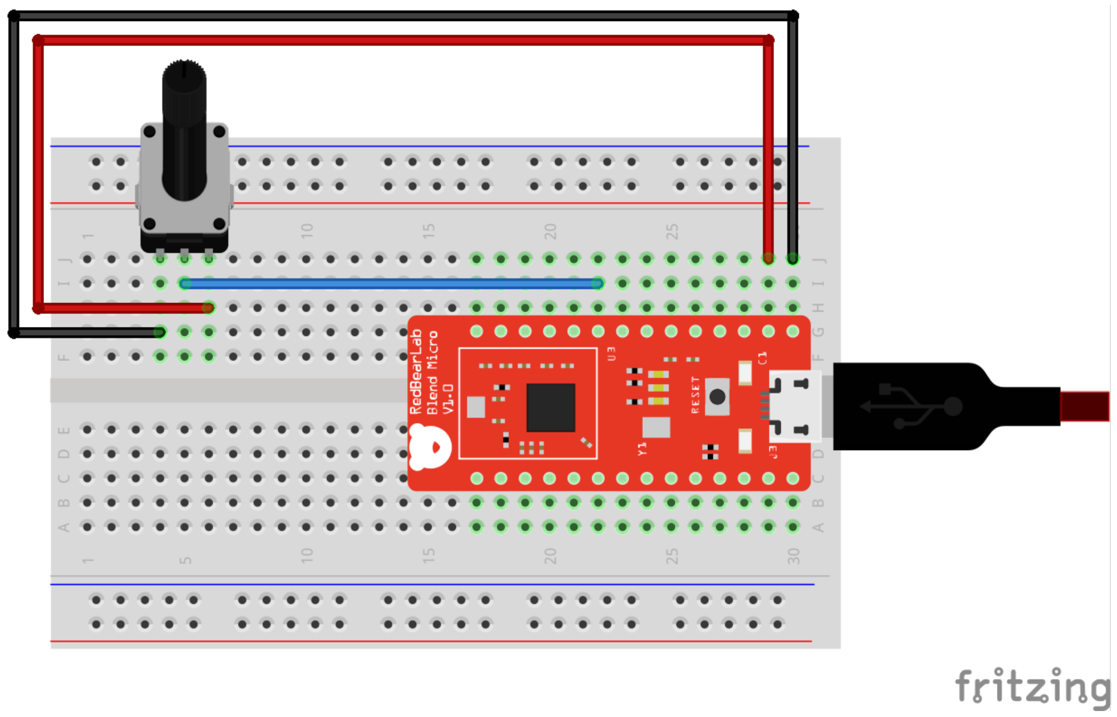


Figure 3.5: AI Circuit

3.4 Software

This section describes the software design to fulfil the desired function of this thesis. Section 3.4.1 explains the software architecture in more detail and describes the individual components. It also describes the design decisions that have been made. Section 3.4.2 describes the software flow of the whole system by combining the individual components.

3.4.1 Software Architecture

The components to be designed on the software side are, on the one hand, the modelling of the BLE communication and, on the other hand, the Firmata protocol. Since these should run independently of each other to ensure their reusability, clearly defined interfaces are required. For this purpose, their design is explained in detail in this section.

BLE Communication

For communication via BLE, the roles of the participants must be clearly defined. Figure 3.6 shows the devices used in the practical part of this thesis and defines their roles concerning the BLE connection. The Android smartphone or tablet with the Pocket Code app acts as the central device and the Arduino as the peripheral device. They communicate via a connection-oriented point-to-point BLE connection. To do this, the peripheral advertises and waits for a connection request from a central, which searches for surrounding devices. In principle, the central can connect to several peripherals, but a peripheral usually cannot because after a connection, its advertising is deactivated and thus can no longer be found by other centrals. In Android, the maximum number of simultaneous connections lies at seven devices.

With regard to the GATT in BLE, the two roles of the GATT server and GATT client can also be used and assigned concerning the data flow. The peripheral typically acts as the GATT server and the central as the GATT client and is also used as such in this thesis. The GATT server manages data in the form of GATT databases and makes them available with certain authorisations. These attributes can be read and written by the GATT client. In addition, the GATT client can subscribe to attributes in order to be automatically notified of their changes.

The GATT server, in this case, the Arduino, defines a GATT that regulates the data exchange via defined attributes. Standardised profiles can be used, or a new profile adapted to the application can be defined. In this thesis, the

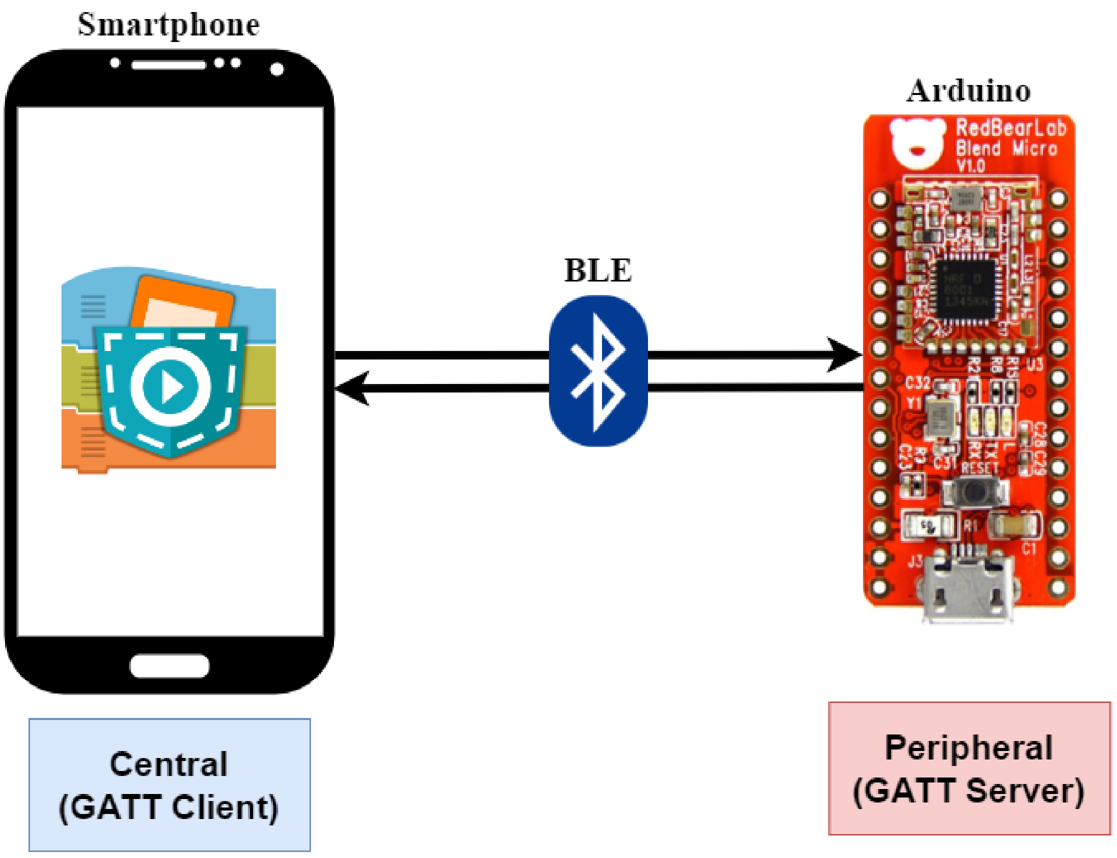


Figure 3.6: BLE Connection Device Roles

latter was used, and the profile was set up in the firmware of the Arduino board. This consists of several services, characteristics and descriptors. The two most important characteristics for the functionality in this application are the Transmitter (TX) and Receiver (RX) characteristics in order to be able to send commands to the Arduino for control and also to read out data.

To check the GATT, the app nRF Connect for Mobile[5], which is available for Android as well as iOS, can be used. This allows to connect to a BLE device, communicate and explore its features. Figure 3.7 shows the GATT of the Arduino used in this thesis, acting as a GATT server.

Firmata

The messages exchanged through BLE must be able to be interpreted by both parties to carry out the respective actions. For this purpose, either a new

[5]`https://www.nordicsemi.com/Products/Development-tools/nrf-connect-for-mobile`, accessed: Jan 25, 2023

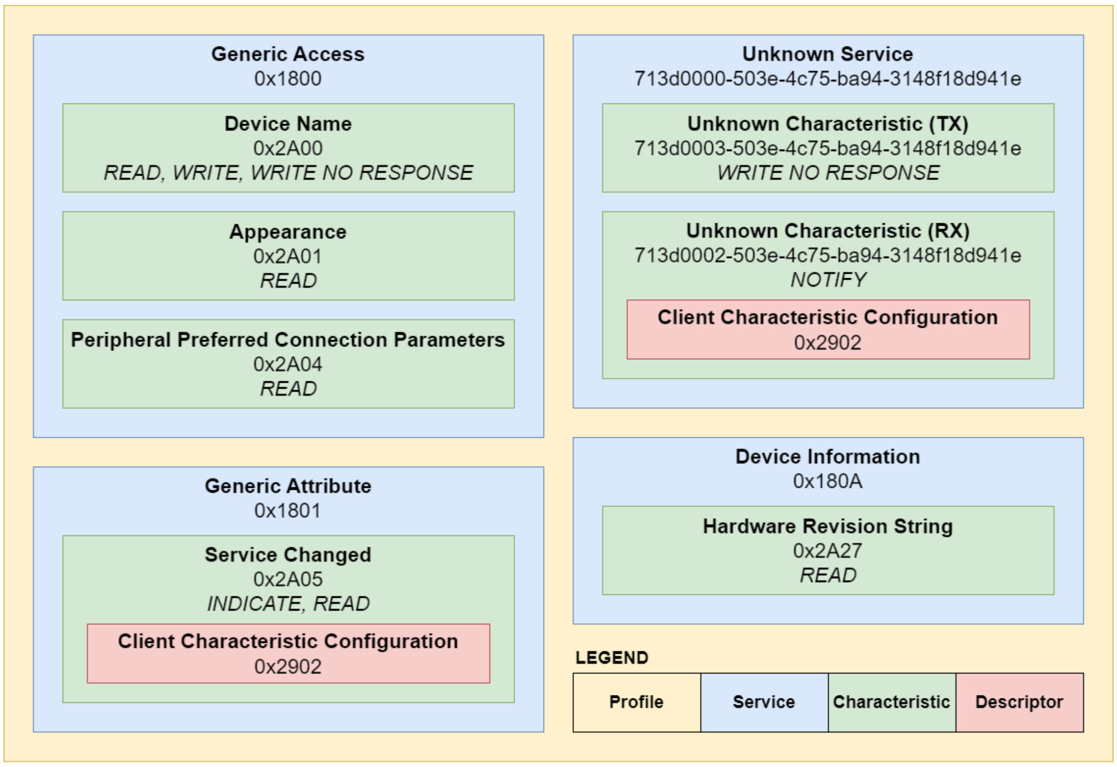

Generic Access
0x1800
Device Name
0x2A00
READ, WRITE, WRITE NO RESPONSE
Appearance
0x2A01
READ
Peripheral Preferred Connection Parameters
0x2A04
READ
Generic Attribute
0x1801
Service Changed
0x2A05
INDICATE, READ
Client Characteristic Configuration
0x2902
Unknown Service
713d0000-503e-4c75-ba94-3148f18d941e
Unknown Characteristic (TX)
713d0003-503e-4c75-ba94-3148f18d941e
WRITE NO RESPONSE
Unknown Characteristic (RX)
713d0002-503e-4c75-ba94-3148f18d941e
NOTIFY
Client Characteristic Configuration
0x2902
Device Information
0x180A
Hardware Revision String
0x2A27
READ
LEGEND
Profile
Service
Characteristic
Descriptor

Figure 3.7: GATT Profile

protocol can be designed, or an existing one can be used. The latter possibility was chosen in this case, and the widely used Firmata protocol was used. This is often used in combination with the Arduino.

In order to change the behaviour of the I/Os of the Arduino, it is necessary to change the program and re-upload it to the Arduino via the PC. Firmata, however, allows the behaviour of the Arduino to be changed dynamically without the need for a re-upload. To do this, the Firmata firmware is uploaded once to the Arduino via the PC. This step can also be done directly from the smartphone via various apps using an USB On-The-Go (OTG) cable. This would allow the user to do without a PC at all. This firmware can be found in the sample programs under the Firmata category of the Arduino IDE. Since this implementation is too large for the flash memory of the Blend Micro Board, the BLEFirmataSketch of the RBL_nRF8001 library[6], which is compatible with the Blend Micro Board, was used. This firmware implements the Firmata protocol and provides a service with RX and TX characteristics for communication via BLE with a BLE Central.

From the software's point of view, the process initially starts with creating

[6]`https://reference.arduino.cc/reference/en/libraries/rbl_nrf8001/`, accessed: Jan 25, 2023

the Firmata message according to the desired function. These messages are converted into the correct format, i.e. encoded, before they are sent. The other participant can then decode these messages and read out the data, and initiate further steps based on this.

3.4.2 Workflow

To achieve the goal of this thesis, the individual components described above together result in the following software workflow. The involved components are Pocket Code, Firmata, BLE and Arduino, as shown in Figure 3.8. Pocket Code runs on the smartphone and communicates with the Arduino via BLE. The Firmata protocol is directly integrated into both participants, i.e. implemented in Pocket Code as well as in the firmware running on the Arduino, and does not represent any hardware components of its own. These components are passed through from left to right when Pocket Code issues a command. In the case of a necessary response from the Arduino or the automatic reporting of value changes, on the other hand, the sequence is passed through from right to left.

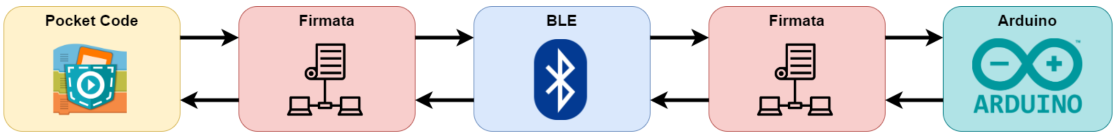


Figure 3.8: Workflow

The workflow is such that the Pocket Code project programmed by the user is executed on the smartphone after the successful connection with the Arduino via BLE. If an Arduino block or function is carried out during the execution, then an interaction with the Arduino is necessary. After checking the parameters of the command to be sent, the necessary Firmata message is created. This is transmitted to the Arduino via BLE. The Firmata firmware running on the Arduino finally decodes the command and performs the requested action.

If a command is sent out that requires a response from the Arduino, such as a request for an input pin, then this process is repeated in reverse order, meaning from right to left. This also happens when automatic notification is set for changes in the data on the Arduino side. The actual Firmata message is created on the Arduino side and sent via BLE to Pocket Code. It is decoded again through the Firmata implementation to process the transmitted data further.

4 Implementation

This chapter provides insight into the practical implementation of this thesis. The aim is to integrate the control of an Arduino via BLE into the existing Android app Pocket Code. Accordingly, the well-known Firmata protocol was used for the message exchange.

The implementation mainly consists of two large, independent parts, which were subsequently integrated into Pocket Code. On the one hand, there is the entire management of a BLE communication (Section 4.1) and, on the other hand, the mobile side support of the Firmata protocol (Section 4.2). These were designed to communicate through defined interfaces to ensure their universal applicability and can therefore be used outside of Pocket Code. Furthermore, Section 4.3 demonstrates their integration into the Pocket Code app and the changes made in this regard. In particular, a lot of focus was put on usability, so the new feature is easy to use and offers a good User Experience (UX). Finally, Section 4.4 deals with testing in more detail, as is usual in XP and other agile methods in the sense of TDD.

4.1 BLE Framework

This section deals with the individual components of the implementation for communication with external devices via BLE. This requires a well-thought-out and structured implementation to ensure its modularity so that further devices and other communication protocols can be easily integrated later.

Section 4.1.1 describes the BLE Manager, which is responsible for the basic functionalities of such a connection. Then, in Section 4.1.2, the various BLE operations are described that are necessary for bidirectional data exchange, i.e. for reading and writing data. Finally, Section 4.1.3 describes the BLE Queue Manager, which takes care of the synchronisation of successive BLE operations within the same connection.

4.1.1 BLE Manager

The first step in implementing the BLE framework was to create a central unit that takes care of all the management of the BLE connection. After searching for surrounding BLE devices, this BLE manager primarily takes care of establishing the connection and subsequently managing it.

For this purpose, after the request to establish a connection to a device, the corresponding callback is waited for, as this is an asynchronous function, like all other BLE operations, which are described in Section 4.1.2. For this reason, a BLE Queue Manager, described in Section 4.1.3, is required to synchronise these BLE operations. After a successful connection, the services are discovered first. This is necessary even if the GATT is already known and must be done once at the beginning before any other BLE operation is performed.

Afterwards, it is determined whether the device to be connected has the two characteristics, RX and TX, required in this case for data exchange. If this is not the case, the BLE connection is terminated. The next step is to enable notification for the RX characteristic in order to receive data changes when subscribing to this characteristic automatically. In Android, enabling notifications for a characteristic must also be signalled to the Android Bluetooth stack via *setCharacteristicNotification()*. Each characteristic supporting notification or indication also has a Client Characteristic Configuration Descriptor (CCCD) that must be set to the desired behaviour. In this case, notification was selected, which, unlike indication, does not confirm receipt of a message to the sender. Also, the Arduino's GATT only supports notification in this particular setting.

The BLE Manager is the central point that sends requests regarding BLE operations to the BLE Queue Manager for enqueueing. It also manages all callbacks of BLE operations and reports the completion of a BLE operation to the BLE Queue Manager for dequeuing.

4.1.2 BLE Operations

Since the BLE operations are asynchronous functions, i.e. a callback signals their completion, they must be synchronised. All possible BLE operations were implemented in their own classes based on an abstract operation class to achieve this. Each BLE operation also receives the necessary information as a parameter. The following BLE operations were implemented in this regard:

- Characteristic Read Operation
- Characteristic Set Operation
- Characteristic Write Operation
- Descriptor Read Operation

- Descriptor Write Operation
- Disconnect Operation
- Discover Services Operation

When a specific BLE operation needs to be performed, such as reading or writing a characteristic, the BLE Manager requests the BLE operation to be executed. The BLE Manager adds the operation to the queue of the BLE Queue Manager to be scheduled. When it is the turn of the respective BLE operation, its function gets executed. If it is an asynchronous function, its callback is waited for before performing the following operation in the queue.

4.1.3 BLE Queue Manager

The BLE operations listed in Section 4.1.2 are asynchronous functions. Unlike synchronous functions, the result is not returned immediately after the call but is provided afterwards by a callback function, since the Bluetooth stack needs some time for execution. Additionally, their execution must not overlap, as parallel execution leads to loss of operations and undefined behaviour. Contrary to expectations, the Android API does not handle synchronisation internally.

Consequently, a central unit is needed to coordinate the flow of operations. For this purpose, a BLE Queue Manager was implemented, which is responsible for executing the operations sequentially. It is not enough to wait for a certain amount of time after the execution, but it is necessary to wait for the corresponding callback. As Android internally ensures the synchronisation of different connections, it is sufficient to have one queue per BLE connection to the peripherals.

Each new operation is first queued and processed according to the First In - First Out (FIFO) principle. Once an operation is completed, the corresponding callback is called and the queue is notified of its completion. Following this, the next operation in the queue will be fetched and executed. This procedure is executed as long as the queue is not empty.

Figure 4.1 shows an example BLE queue, where the Discover Services Operation would be completed by triggering the corresponding callback and thus being dequeued. The next operation to be executed would be the operation at the beginning of the queue, which is the Characteristic Set Operation. In addition, a Disconnect Operation has been enqueued, but it is not executed immediately but queued at the back of the queue.

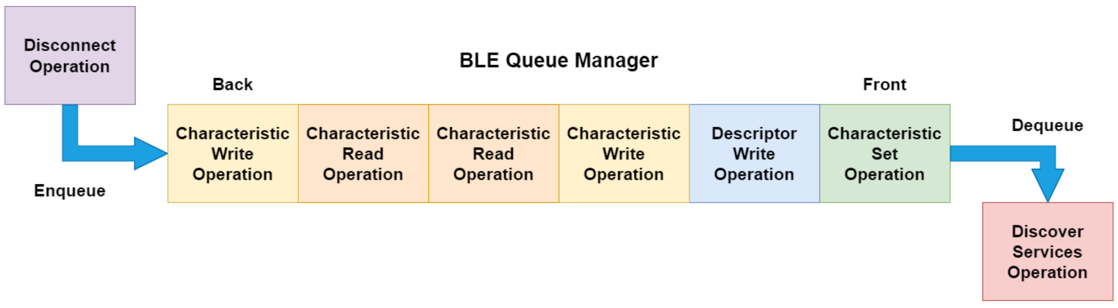

Disconnect Operation
BLE Queue Manager
Back
Front
Enqueue
Characteristic Write Operation
Characteristic Read Operation
Characteristic Read Operation
Characteristic Write Operation
Descriptor Write Operation
Characteristic Set Operation
Dequeue
Discover Services Operation

Figure 4.1: FIFO BLE Queue

4.2 Firmata

For dynamic control of the Arduino via an Android application, the Firmata protocol was used in this case. To exchange messages via Firmata, both participants in the conversation must implement this protocol. The Firmata firmware has to be loaded once on the Arduino, and the protocol has to be implemented on the client side. In this case, the client is the Android application.

As already mentioned in the Section 2.4.4, there is no existing library for the Firmata protocol for Android applications that supports communication via BLE. Therefore, in this thesis, the existing open-source Java implementation of Smirnov [25] was used and extended to this end. This extension is an enrichment because it is the only open-source implementation of the Firmata protocol that supports the wireless communication standard BLE. It can be used as a library not only in Android applications but also in standard Java or JavaFX applications. This is possible through its encapsulation as a stand-alone implementation for reuse.

The concrete implementation is divided into three parts: Implementing the different message types and transmitting and receiving messages. The messages are encoded before they are sent in order to comply with the defined format of the Firmata protocol. When a message is received, however, the message is decoded. The first byte, the command byte, determines the message type. Depending on this, the respective parameters are read out. These are passed on to registered listeners for further use.

This encapsulated implementation can easily be used in individual applications to use the Firmata protocol for communication. The various message types can be created, sent and received via defined interfaces. In order to receive messages, so-called listeners have to be registered, which are triggered when a message is received to use the read data further.

Section 4.2.1 describes the implementation of the message types of the Firmata protocol. Subsequently, Section 4.2.2 describes the sending of messages and

Section 4.2.3 the receiving of messages and the necessary steps.

4.2.1 Messages

To model the different message types in Firmata, all message types were implemented as separate classes based on an abstract base class. Depending on the message type, these classes contain the necessary parameters.

In addition, the profiles of the different Arduino boards were each modelled in their own class based on a base class. These are used to create the appropriate Firmata messages depending on the desired control behaviour. In addition, various error checks are performed during their creation, such as validation of pins and modes based on the hardware characteristics of the Arduino used. The generated Firmata messages can be sent directly via the Firmata instance. An exception is thrown if a command is requested with an incorrect configuration. The profiles contain the number of digital I/O and analog input pins. They also define the digital I/O pins that can also be used as PWM outputs.

This implementation allows easy future extension with further Firmata message types or Arduino boards. All that is required is to implement the appropriate Firmata message and its parameters or to implement the profile of the Arduino board.

4.2.2 Writer

The Writer part takes care of encoding the different message types to comply with the Firmata protocol specification and subsequently sends them. For this purpose, each message type implemented a common interface. These writer classes send the respective message consisting of a command byte and one or two data bytes, as specified in the Firmata protocol. Exceptions to this are the Sysex messages where there can be any number of data bytes between the two command bytes START_SYSEX and END_SYSEX.

The interface and its implementing classes have been extended to enable transmission via BLE in addition to serial transmission. For this purpose, the bytes to be sent are created, which can be sent via a TX characteristic.

4.2.3 Reader

The reader part takes care of decoding the incoming Firmata messages on the receiver side. For this purpose, the incoming message is read byte by byte, and the command byte at the beginning of the message is used to decide what type of message it is. Then the following data bytes are read to create an object of the respective message class and to determine its parameters. Finally, the respective listeners are informed of the incoming message. Again a uniform interface was defined for its implementation, which was implemented by the different message types.

4.3 Integration into Pocket Code

The most important part of the implementation was to integrate the BLE framework (Section 4.1) and the Firmata implementation (Section 4.2) into the existing Android app Pocket Code and thus enable the control of the Arduino via BLE. During the integration, the main focus was on usability to make it as easy as possible for the user to use the new functions. Apart from that, attention was paid to the clean implementation of the already very extensive Pocket Code app to ensure easy future expandability.

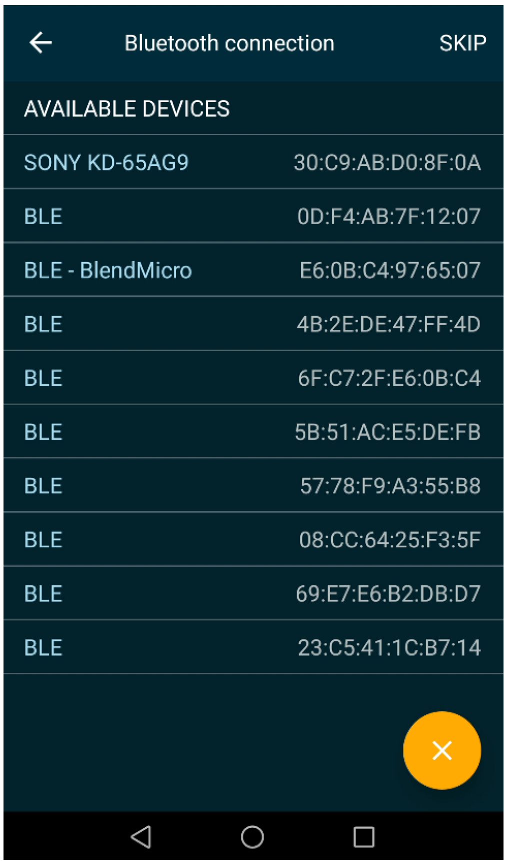
The integration can be divided into two major parts. Section 4.3.1 describes the integration of BLE as a communication protocol. This includes everything from device search and connection establishment to message exchange and disconnection within the app. Section 4.3.2 explains the changes that were necessary concerning controlling an Arduino.

4.3.1 BLE Connection

The first part of the integration was to extend the app to connect to BLE devices and exchange messages. For this purpose, the self-implemented BLE framework was integrated into Pocket Code. The framework takes care of the entire administration of the BLE connection within the app to achieve encapsulation.

Basically, as soon as a project is started in Pocket Code, the necessary resources are checked. This includes checking whether specially used blocks or functions are contained in the program that requires additional actions beforehand, such as initialising the hardware used, before the stage can be displayed and the program can be executed. The blocks and functions for controlling the Arduino also contain their own resource. This signals a necessary connection with the Arduino in advance. For this reason, if this resource is present, an activity starts to be able to connect to such a device.

The connection screen that takes care of the device search in order to subsequently establish a connection is shown in Figure 4.2. In this connection screen, it is first checked whether Bluetooth is supported and activated on the device. In the deactivated case, the user is prompted to activate Bluetooth. It is also possible to skip a connection using the button in the upper right corner. This is useful for inspecting the GUI, but without an upright connection, the functionalities related to Arduino will not lead to any actions. In the lower right corner, it is possible to start and stop a search for Bluetooth devices in the area. The screen is divided into two parts: the upper section shows the bounded (paired in Bluetooth Classic) devices, and the lower section shows the new devices found during this scan. The devices are listed with their name and media access control (MAC) address, whereby BLE devices are additionally marked with the prefix "BLE – ". A connection is established by clicking on the respective device in the list.

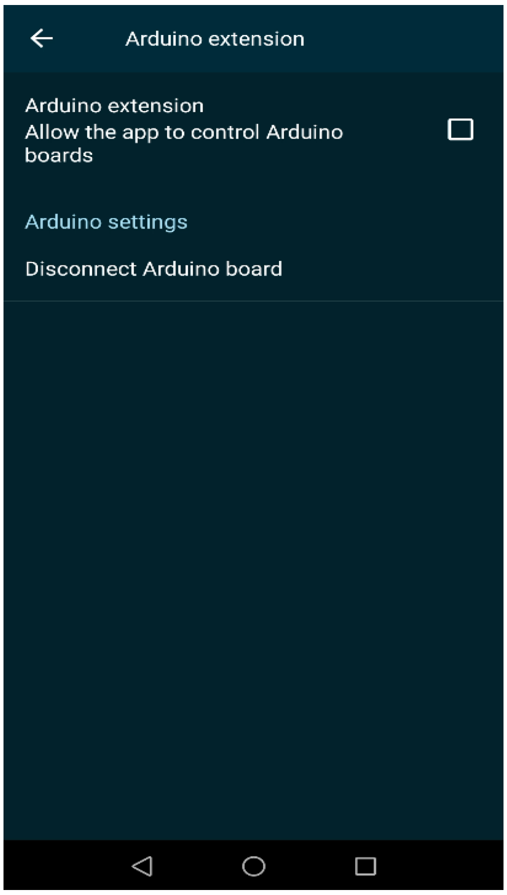


AVAILABLE DEVICES	
SONY KD-65AG9	30:C9:AB:D0:8F:0A
BLE	0D:F4:AB:7F:12:07
BLE - BlendMicro	E6:0B:C4:97:65:07
BLE	4B:2E:DE:47:FF:4D
BLE	6F:C7:2F:E6:0B:C4
BLE	5B:51:AC:E5:DE:FB
BLE	57:78:F9:A3:55:B8
BLE	08:CC:64:25:F3:5F
BLE	69:E7:E6:B2:DB:D7
BLE	23:C5:41:1C:B7:14

Figure 4.2: Bluetooth Device Search Screen

The connection is maintained for as long as possible to avoid a constant repetitive reconnection when starting the program. If it is determined that such a device is already connected when starting the program, the connection step is skipped, and the program is executed directly. The connection is terminated when the device goes outside the range or this is explicitly requested in the app settings. Figure 4.3 shows the settings screen for explicitly disconnecting.

In both cases, the user is notified with a message on the screen. In addition, the user is also notified if the BLE connection is lost for any reason during the execution of the project.



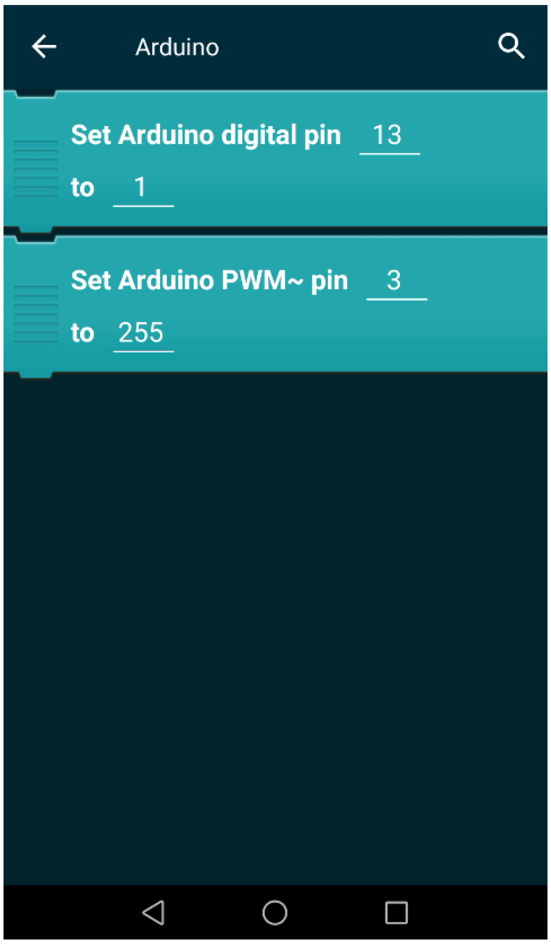
Arduino extension
Allow the app to control Arduino boards ☐

Figure 4.3: Settings remove BLE Device

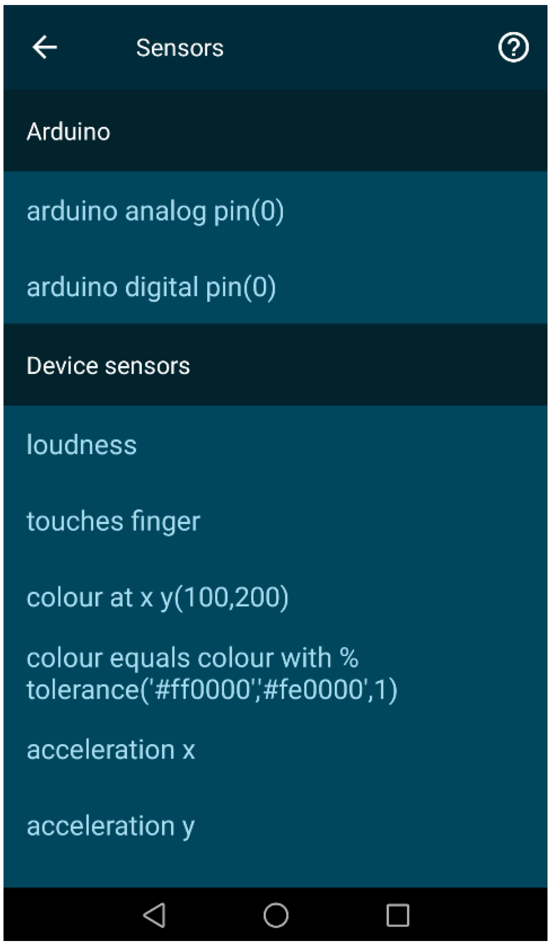
4.3.2 Arduino Control

The second part of the integration included the control of the Arduino itself, using the Firmata protocol in this case for message exchange. For this purpose, the implementation of the Firmata protocol from Section 4.2 was integrated, and some modifications were made to the app in this regard.

In order to be able to control an Arduino in Pocket Code, this extension must first be enabled in the settings, as it is disabled by default. This will show an additional Arduino section in the list of brick categories containing Arduino bricks as well as in the sensors category of the formula editor. These two sections are shown in Figure 4.4. The Arduino brick category contains the functions for setting an output as bricks, and the Arduino section in the formula editor includes the functions for reading an input. The reason for this separation lies in the app's concept since the setting is an action to be executed and is therefore represented as a brick, but the reading is used as input in other bricks. These bricks and functions contain an Arduino-specific resource. These resources are checked when a project is started, and in this case, signal the necessity of establishing a connection to an Arduino in advance when they are used.



a) Arduino Bricks



b) Arduino Functions

Figure 4.4: Arduino in Pocket Code

The internal procedure for using the Arduino functions is as follows: Checking the entered parameters, creating the necessary Firmata messages and sending them via BLE to the connected Arduino. The check of the parameters includes, on the one hand, the check if it is a valid pin and, on the other hand, if its value is in the valid range. Additionally, for analog I/Os, it is checked whether the respective pin supports this function. Then, the necessary Firmata messages are generated depending on the selected function. For digital I/Os and PWM, this includes the setting of the pin mode. Finally, these messages are sent to the

Arduino via BLE. This is not done directly, but the messages are inserted into the BLE Queue, as described in Section 4.1.2. All these described actions are again encapsulated in the Firmata implementation.

In addition, a copy of the current state of the I/Os is stored locally. This is especially necessary for the automatic notification feature of BLE, where value changes are automatically reported. This information can also be used to have a view in the app where all states of the I/Os can be viewed and changed. Apart from that, the chosen design allows the addition of further control functions for the Arduino quite easily by creating the respective brick or function with the linkage of sending the required Firmata messages in the background.

4.4 Testing

In order to guarantee the functionality of the implementation in the future and to avoid unintentional errors, the new features integrated within this thesis must be tested extensively. This is especially essential for a project as large as Pocket Code, as otherwise functionalities can be broken unnoticed. Therefore, the tests must have extensive code coverage and be meaningful to locate a malfunction immediately.

Basically, there are two ways to test the functionality, on the one hand, via real hardware and, on the other hand, simulated via an emulator. Since the tests are to run in the automated Continuous Integration (CI) pipeline, the first would require a real test setup to be constructed that interacts with the test server via a defined interface. This test setup would have to consist of an Android smartphone and an Arduino connected to each other via BLE. The great advantage of this approach is that it represents reality through the real test environment with all its characteristics, which reflect the real conditions of the end user. However, due to the usual instability of wireless connections, there might be random failed test runs, which do not necessarily represent a failure in the implementation. As such, they are not ideal for automated testing but could be done manually from time to time or before application releases.

The second method is also a bit trickier in this case, as the Android emulator, which is also used for the automated CI pipeline, does not support Bluetooth at all. Consequently, the BLE connection has to be simulated, as a real BLE connection to the Arduino is not feasible. This process of simulating objects is called mocking. It involves creating an imitation that mimics the behaviour of the real object. This proves to be a difficult matter, especially with complex objects, because their real conditions must be reflected very well for a promising result. On top of that, this has to be taken into account already in the implementation design to ensure its compatibility. The great advantage of this method is its reliability due to the reproducibility of the results. However, the main weakness lies in the ideal and perfect imitation of the real object, which does not occur in reality.

Since the app follows the TDD concept and thus requires reliable test results, the second method from above was used in this thesis. All parts described so far (Section 4.1, 4.2, 4.3) are tested independently to reuse the tests when the BLE framework or the Firmata protocol is used individually outside the app. They are mainly functional tests to ensure their correct functionality.

The tests can basically be divided into two types, which are discussed in the following sections. An overview of the setup of both methods is shown in Figure 4.5. Section 4.4.1 describes the concept of unit testing, which deals with verifying individual functionalities. This type of testing was applied to the BLE framework and Firmata protocol implementation. Section 4.4.2 deals with instrumentation testing, which mainly checks the user interface (UI) of the app via the emulator and was used for the tests of the third part concerning the integration into the Pocket Code app.

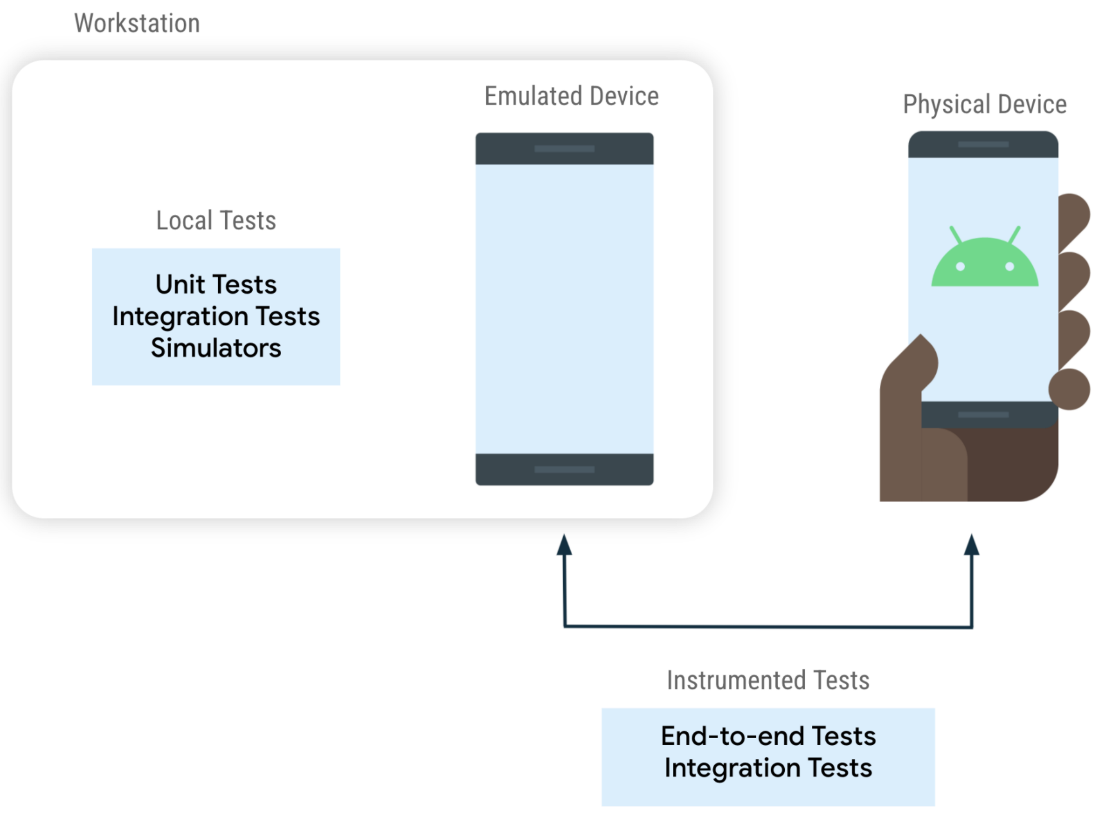


Figure 4.5: Android Testing Methods Overview [26]

4.4.1 Unit Testing

Unit tests are tests used to test small functionalities where it is not determined what this small functionality actually is. Mostly, however, it is a matter of testing individual functions. They are also called local tests because, unlike instrumentation tests, they do not require a device or emulator to run the tests and therefore run much faster. The only sticking point with these tests is the dependencies on other objects or on the Android API itself, which occur as parameters or directly as objects in the section to be tested. In order to be able to test the respective section individually, it must be isolated. The dependencies must be mocked to do this, thereby simulating their real behaviour.

Unit tests were written for the implemented BLE framework as well as for the implementation of the Firmata protocol. The testing of the objects was already considered during the design process of the implementation, which is especially important concerning mocking. For this purpose, the Mockito[1] framework was used in this thesis, which allows the imitation of an object and the determination of its behaviour.

The different aspects of a BLE connection were tested for the BLE framework. This includes searching for devices, establishing connections, exchanging messages and terminating connections. For this purpose, the external device, in the case of this thesis, an Arduino, was mocked. In this case, the framework interface communicates with a simulated external device and checks whether the communication is working correctly. The only drawback in this respect is that a trouble-free BLE connection is assumed, but in reality, interference and certain unpredictable errors occur.

The extended Firmata protocol implementation already included unit tests for serial transmission. These have been extended to test transmission over BLE as well. These tests involve testing the exchange of the different Firmata message types as well as their required utils classes. Instead of mocking, separate test object implementations were created for testing. Furthermore, the implementation includes real hardware tests that can be executed with a connected external device, in this case, an Arduino with Firmata firmware. These tests communicate directly with the board via Firmata and therefore do not require an Android smartphone or emulator to be executed.

4.4.2 Instrumentation Testing

Instrumentation tests are tests that run on real hardware or an emulator as opposed to unit tests. They, therefore, run longer but reflect the real behaviour

[1]`https://site.mockito.org/`, accessed: Jan 29, 2023

of the end user. The name comes from the fact that access is granted to the Instrumentation API, which allows the app to be controlled from within the code. This is mainly used for UI tests, where the GUI of the app is automatically checked by simulating user interactions. This includes checking the layout itself, the user flow and aspects of functionality. However, they can also test other logic, not just the UI. They are also suitable for testing objects with complex dependencies, as no mocking is necessary in this case. Since these tests run on the emulator, they are also well suited for automated test suites and lead to an increased UX.

This type of testing was mainly used to test the integration of the newly implemented features in Pocket Code. For this purpose, primarily, UI tests were written to check the correct display of the layout and the user flow. This was done using the frameworks Espresso[2] and JUnit[3], which make it possible to simulate user interaction within the app.

The tests include various checks in the settings screen, brick categories and formula editor regarding the Arduino extension. Furthermore, it is checked whether the correct event is triggered in the background when the extension is used. The only shortcoming of these UI tests is their flakiness, which can occur with the relatively slow emulator. Several measures were taken to reduce this to a minimum. On the one hand, the delays in displaying the UI elements by the emulator were solved by explicitly waiting for these elements at sensitive points. On the other hand, such annoying flaky tests can be annotated so that if they fail, they are executed again until a self-defined maximum number of retries is reached.

[2]`https://developer.android.com/training/testing/espresso`, accessed: Jan 29, 2023

[3]`https://junit.org/junit4/`, accessed: Jan 29, 2023

5 Evaluation

This chapter evaluates the developed system in this thesis in order to highlight its added value. Section 5.1 compares programming in Pocket Code with direct programming in the Arduino IDE concerning controlling an Arduino and derives results based on this. Section 5.2 discusses these findings in detail on the one hand and summarises the advantages of the developed system on the other hand.

5.1 Programming

In this section, the benefits of the practical part of this thesis, the integration of the control of an Arduino via BLE in Pocket Code, are evaluated with regard to the facilitation of programming. To this end, Section 5.1.1 describes the experimental test setup used for evaluation. This is followed in Section 5.1.2 by the implementation of the desired function both in the Arduino IDE and in Pocket Code in order to compare them and highlight the differences. Finally, in Section 5.1.3, the differences are highlighted, and the main results obtained from using Pocket Code to realise individual IoT projects are discussed.

5.1.1 Experimental Setup

A minimalistic test setup for evaluating the newly introduced feature is shown in Figure 5.1. The circuit built on the breadboard consists of a Blend Micro Board, an LED, a push button and two resistors. One resistor is 220 Ω and serves as a series resistor for the LED, and the other has 10 kΩ and serves as a pull-down resistor for the push button. The aim is to control the LED using the push button. The LED should light up when the push button is pressed and switch off again when it is released. The push button is not directly connected to the LED but to the pins of the Blend Board, which should read the status of the push button in order to control the LED. Two digital pins of the Blend Micro Board are used for this, pin 9 as DO for the LED and pin 10 as DI for the push button. The Firmata firmware must be uploaded once to the Blend Micro Board before it can be programmed using Pocket Code.

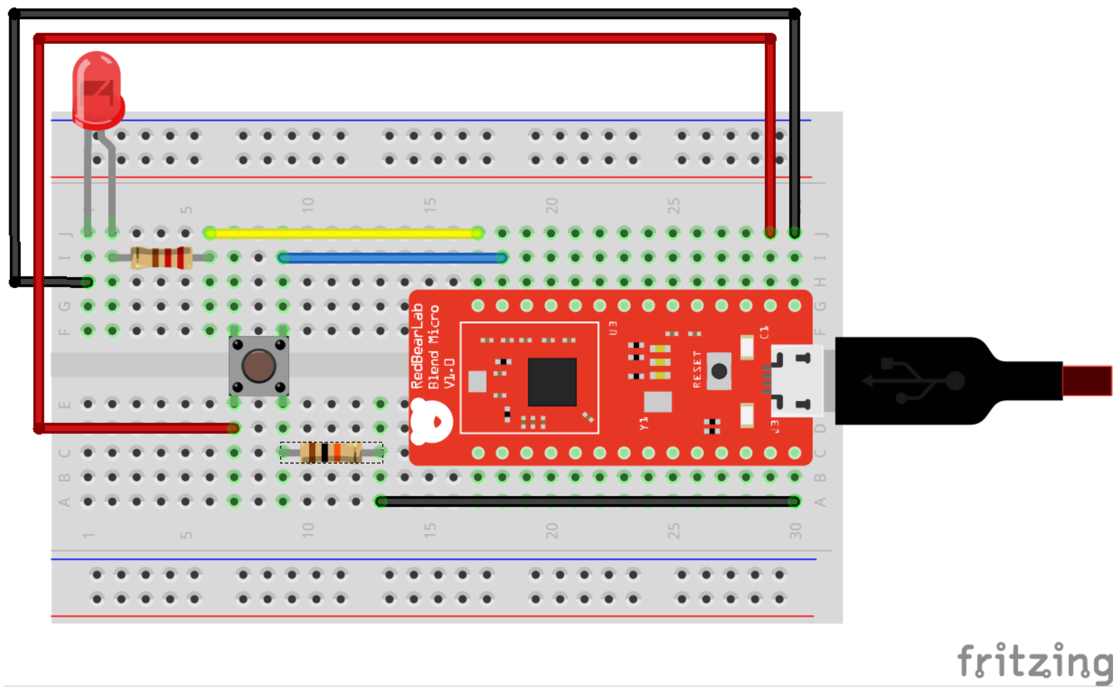


Figure 5.1: Evaluation Test Setup

5.1.2 Code

The desired function was programmed in the Arduino IDE on the one hand and in Pocket Code on the other for comparison. The code from the Arduino IDE can be found in Listing 5.1, and Figure 5.2 shows the project in Pocket Code.

The code in the Arduino IDE first defines two constants for the respective pins of the LED and the push button for reuse. Their mode is first configured in the *setup()* function, with the LED pin as DO and the push button pin as DI. Then the main program follows in the *loop()* function, in which the state of the push button is read. Depending on the state, the LED is switched on (set to HIGH) in the pressed state, meaning HIGH is applied to the input or otherwise switched off (set to LOW).

Listing 5.1: Arduino Evaluation Sketch

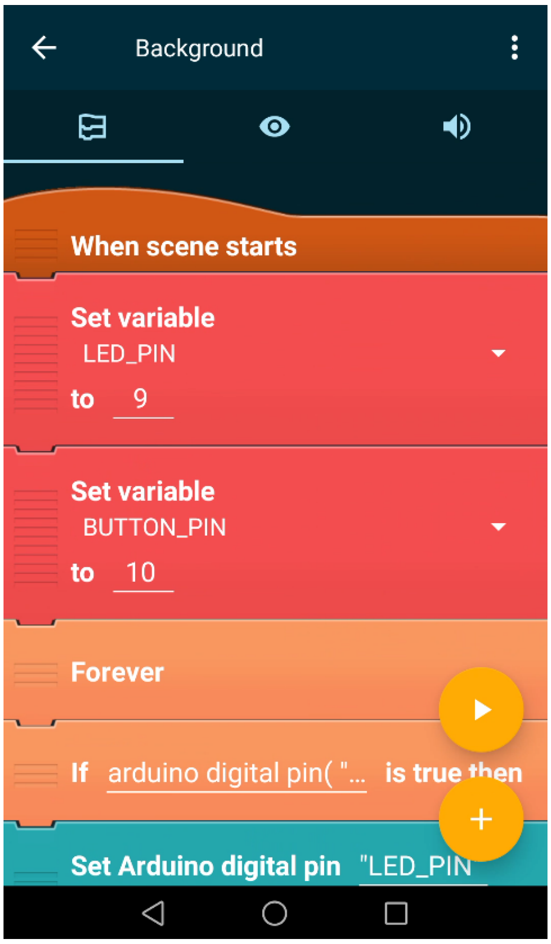
```
const int LED_PIN = 9;
const int BUTTON_PIN = 10;

void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(BUTTON_PIN, INPUT);
}
```

```

void loop() {
  if (digitalRead(BUTTON_PIN) == HIGH) {
    digitalWrite(LED_PIN, HIGH);
  } else {
    digitalWrite(LED_PIN, LOW);
  }
}
```

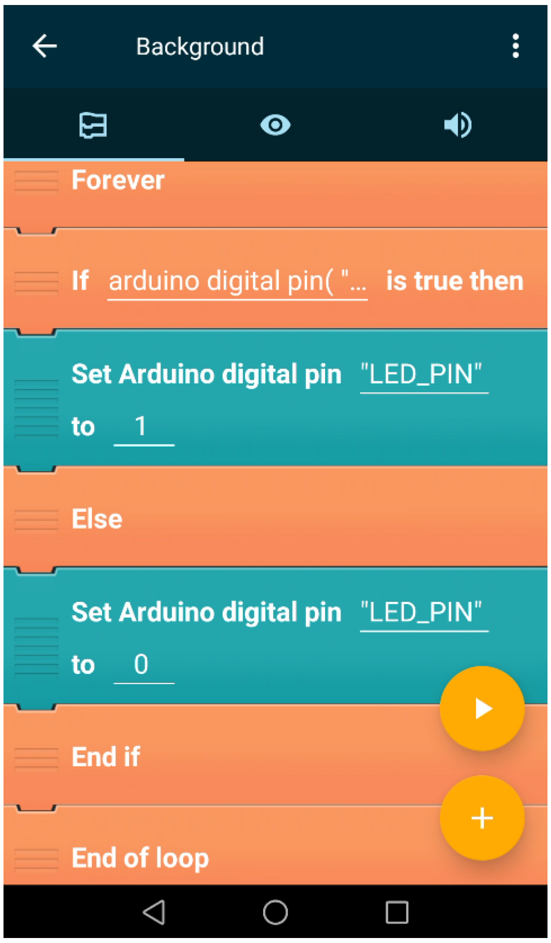
In the Pocket Code project, two variables are also defined at the beginning, one each for the LED pin and the push button pin. Then the state of the DI of the push button is constantly queried in a loop, and the DO of the LED is set based on this.



Background

When scene starts
Set variable LED_PIN to 9
Set variable BUTTON_PIN to 10
Forever
If arduino digital pin("... is true then
Set Arduino digital pin "LED_PIN

a) Evaluation Project Part 1



Background

Forever
If arduino digital pin("... is true then
Set Arduino digital pin "LED_PIN" to 1
Else
Set Arduino digital pin "LED_PIN" to 0
End if
End of loop

b) Evaluation Project Part 2

Figure 5.2: Pocket Code Evaluation Project

5.1.3 Result

Both the programming in the Arduino IDE as well as in Pocket Code leads to the desired functionality. However, they differ in some respects, both in the programming environment and especially in the programming language itself.

Programming in the Arduino IDE requires a PC, whereas Pocket Code runs entirely on a smartphone or tablet, which is much more common. In addition, uploading a program in the former requires a direct connection via a USB cable to the Arduino, which is done wirelessly via BLE in the latter.

However, there are some distinctions in the programming language itself. In Pocket Code, programming is done visually by dragging and dropping blocks. In the Arduino IDE, by contrast, programming is text-based and thus requires explicit code writing. This requires learning the programming language's syntax and thus introduces the possibility of syntax errors, which cannot occur in visual programming languages. In addition, in Pocket Code, it is possible to program in one's native language, which is only possible in English in the Arduino IDE. This feature makes programming much more accessible, especially for children, who usually only know their mother tongue.

In terms of code size, both types are pretty comparable and differ in the program's structure only in two essential points. The first difference is that the pin modes are not explicitly set in Pocket Code as in the *setup()* function of the Arduino IDE. This is because they are set implicitly in the background of the respective Arduino blocks. This makes it easier for beginners in software and hardware prototyping to get started, as no explicit prior knowledge is required. The second difference is that the continuous loop, through the *loop()* function in the Arduino IDE, must be explicitly added to the program in Pocket Code. Otherwise, the program would only be executed once. Apart from these two distinctions, both types can easily be converted into each other. In principle, programming in Pocket Code is a visualisation of the code in the Arduino IDE and is, therefore, easier to get started with.

5.2 Discussion

The new feature integrated in this thesis, the control of the Arduino board via Pocket Code, opens up many new possibilities with regard to the implementation of IoT projects. Typical hurdles for such projects are reduced to a minimum, and no profound knowledge in the area of software, as well as hardware, is required.

The built-in feature basically consists of two components, Pocket Code and Arduino, which communicate with each other via BLE. On the one hand, Pocket Code itself already offers a more accessible introduction to programming and on the other hand, Arduino provides a simplified introduction to hardware prototyping. Thus, combining both unites the respective advantages and enables promising projects in this respect, especially for beginners in these fields.

However, it also has an enormous added value for experienced users in implementing ideas according to the principle of rapid prototyping. Through this new feature, ideas can be realised quickly directly via the smartphone or tablet. The typically necessary constant changes in the prototyping process can easily be carried out wirelessly via BLE without constantly reconnecting the Arduino to the PC, which can be annoying when installed. In addition, the ability to easily and quickly create a custom GUI for user interaction via Pocket Code is especially advantageous. This is also very important for the Maker community when creating their individual projects.

The only drawback in this respect is that the control options are limited to the functions represented in the app by blocks, which somewhat restricts flexibility. Experienced users might also criticise the lack of clarity in large, complex programmes due to the blocks used in Pocket Code compared to the text-based code in the Arduino IDE. However, this can be remedied by dividing it into possible scenes and sprites and creating custom blocks in Pocket Code.

6 Conclusion

In conclusion, this chapter provides a summary of the present thesis in Section 6.1 and then presents future work based on it in Section 6.2.

6.1 Summary

The massive upswing of IoT devices in everyday life can no longer be overlooked. More and more devices support BLE as a wireless communication protocol due to its numerous advantages over other protocols, especially its predecessor Bluetooth Classic. However, there is a lack of applications to implement personal ideas and projects in this area, especially without sound prior knowledge in this field. For this reason, this thesis offers a simple way to realise such projects by extending the existing Android app Pocket Code.

The thesis focuses on designing and implementing the control of BLE devices in mobile Android development. For this purpose, Pocket Code is extended by the control via BLE of the widely used Arduino as an exemplary first device to be up-to-date with the latest technology. The implementation can be divided into three main parts, the implementation of a framework to manage the BLE connection, the implementation of the Firmata protocol with support for BLE for communication and finally, the integration of both in Pocket Code to control BLE devices.

This newly integrated feature in Pocket Code makes it much easier for non-professionals to develop their projects. Moreover, no prior knowledge is needed, both in terms of software through the visual language Catrobat and hardware through the Arduino board's flexibility. This combination allows the promising realisation of projects in a simple and straightforward way. But it is also suitable for experts and hobbyists, especially in the Maker movement, to quickly implement prototypes according to the principle of rapid prototyping. A significant advantage over other applications in this field is that everything is done entirely on the smartphone or tablet, from the programming to the execution.

Finally, the chosen design also allows the app to be easily expanded in the future to include other devices or communication protocols so that constant adaptation to the latest technologies is easily possible.

6.2 Future Work

In the design of the implementation in this thesis, much emphasis has been placed on easy extensibility in the future. Based on this, several future works are presented in this section.

6.2.1 Arduino Flavor

The first option for future work is to extend the functionalities in connection with the Arduino board to create a separate Arduino Flavor. This has the advantage that, as with the other flavors already existing on Pocket Code, the app can focus specifically on this target group and be adapted to their interests. This also allows the creation of its own community to stimulate user exchange. At the same time, users can share their projects to make them accessible to others and pass on their experiences.

The Firmata protocol allows many other message types to control the Arduino, such as controlling a servo motor or controlling via another interface, such as I^2C. For this purpose, the Arduino category can be extended with further functions and blocks. Furthermore, the support of other Arduino boards can be added, which is easily possible by incorporating the Firmata feature of automatically reading the pin assignment at the beginning. This even allows the control of any board equipped with the Firmata firmware.

It would also be a good idea to create a remote control, so to speak, for the connected board. For this purpose, a screen is created that displays all I/Os of the respective board. The function of each individual pin can be set using simple control elements such as buttons, switches and sliders, and depending on this, its value can be read or written.

6.2.2 Support BLE devices

Another possible idea would be to control BLE devices via Pocket Code in general. Especially concerning IoT devices that communicate via BLE, this would allow promising projects. The range here extends from small everyday devices such as smartwatches to distributed sensor nodes such as temperature sensors. This involves communicating directly via the GATT in BLE to read and write data. If the respective device uses a standardised profile for communication, whose services and characteristics can be stored in the app in advance, it allows even easier usage for the user.

This requires creating new blocks and functions in Pocket Code that enable communication to BLE devices in general. These range from service discovery, reading and writing characteristics to registering for automatic notifications of changes. In this way, applications can again be realised quickly and easily according to the principle of rapid prototyping.

Bibliography

[1] W. Slany, "A mobile visual programming system for android smartphones and tablets", in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2012, pp. 265–266 (cit. on p. 5).

[2] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment", *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, pp. 1–15, 2010 (cit. on p. 5).

[3] W. Slany, "Pocket code: A scratch-like integrated development environment for your phone", in *Proceedings of the companion publication of the 2014 ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity*, 2014, pp. 35–36 (cit. on p. 6).

[4] W. Slany, "Catroid: A mobile visual programming system for children", in *Proceedings of the 11th International Conference on Interaction Design and Children*, 2012, pp. 300–303. DOI: `10.1145/2307096.2307151` (cit. on p. 6).

[5] I. Bluetooth SIG. "Bluetooth homepage", [Online]. Available: `https://www.bluetooth.com/` (accessed: Feb. 2, 2023) (cit. on p. 11).

[6] K. Townsend, C. Cufı, R. Davidson, *et al.*, *Getting started with Bluetooth low energy: tools and techniques for low-power networking.* " O'Reilly Media, Inc.", 2014 (cit. on pp. 11, 12, 16, 17).

[7] J. Tosi, F. Taffoni, M. Santacatterina, R. Sannino, and D. Formica, "Performance evaluation of bluetooth low energy: A systematic review", *Sensors*, vol. 17, no. 12, p. 2898, 2017 (cit. on p. 13).

[8] N. K. Gupta, *Inside Bluetooth low energy*. Artech House, 2016 (cit. on pp. 15, 16).

[9] W. Kassab and K. A. Darabkh, "A–z survey of internet of things: Architectures, protocols, applications, recent advances, future directions and recommendations", *Journal of Network and Computer Applications*, vol. 163, p. 102 663, 2020 (cit. on p. 18).

[10] A. Dementyev, S. Hodges, S. Taylor, and J. Smith, "Power consumption analysis of bluetooth low energy, zigbee and ant sensor nodes in a cyclic sleep scenario", in *2013 IEEE International Wireless Symposium (IWS)*, IEEE, 2013, pp. 1–4 (cit. on p. 18).

[11] B. T. Inc. "6 leading types of iot wireless tech and their best use cases", [Online]. Available: https://behrtech.com/blog/6-leading-types-of-iot-wireless-tech-and-their-best-use-cases/ (accessed: Feb. 2, 2023) (cit. on p. 19).

[12] Arduino. "About arduino", [Online]. Available: https://www.arduino.cc/en/about (accessed: Jan. 10, 2023) (cit. on p. 19).

[13] Arduino. "What is arduino?", [Online]. Available: https://www.arduino.cc/en/Guide/Introduction (accessed: Jan. 10, 2023) (cit. on p. 19).

[14] M. Banzi and M. Shiloh, *Getting started with Arduino*. Maker Media, Inc., 2022 (cit. on p. 20).

[15] Arduino. "Getting started with arduino", [Online]. Available: https://docs.arduino.cc/learn/starting-guide/getting-started-arduino (accessed: Jan. 10, 2023) (cit. on pp. 21, 22).

[16] H.-C. Steiner, "Firmata: Towards making microcontrollers act like extensions of the computer.", in *NIME*, 2009, pp. 125–130 (cit. on p. 24).

[17] B. Moog, "Midi", *J. Audio Eng. Soc*, vol. 34, no. 5, 1986 (cit. on p. 24).

[18] D. Smith and C. Wood, "The 'usi', or universal synthesizer interface", in *Audio Engineering Society Convention 70*, Audio Engineering Society, 1981 (cit. on p. 24).

[19] M. M. Association. "Midi 1.0 detailed specification", [Online]. Available: https://www.midi.org/ (accessed: Jan. 17, 2023) (cit. on p. 24).

[20] "Firmata protocol", [Online]. Available: http://firmata.org/wiki/Main_Page (accessed: Jan. 17, 2023) (cit. on p. 25).

[21] "Firmata protocol documentation", [Online]. Available: https://github.com/firmata/protocol (accessed: Jan. 17, 2023) (cit. on pp. 25, 26).

[22] P. Desai, *Python programming for Arduino*. Packt Publishing Ltd, 2015, ISBN: 978-1783285938 (cit. on p. 26).

[23] "Redbearlab", [Online]. Available: https://web.archive.org/web/20160314200149/http://redbearlab.com/blendmicro/ (accessed: Jan. 25, 2023) (cit. on pp. 29, 31).

[24] "Blend micro", [Online]. Available: https://github.com/RedBearLab/Blend (accessed: Jan. 25, 2023) (cit. on p. 30).

[25] A. Smirnov. "Firmata pure java implementation", [Online]. Available: https://github.com/4ntoine/Firmata (accessed: Jan. 29, 2023) (cit. on p. 44).

[26] Android. "Fundamentals of testing android apps", [Online]. Available: https://developer.android.com/training/testing/fundamentals (accessed: Jan. 29, 2023) (cit. on p. 51).