Universität Bremen

Supporting Bluetooth Low Energy Keyboards in Environments without a Bluetooth Stack

Bachelors's Thesis at the Group for Computer Architecture (AGRA)
Prof. Dr. Rolf Drechsler
Department 3
University of Bremen

by

Wilhelm Jochim

Advisors:
Sallar Ahmadi-Pour
Sören Tempel

Reviewers:
Prof. Dr. Rolf Drechsler
Dr. Olaf Bergmann

Day of registration: 16th of April 2022
Day of submission: 7th of Febuary 2023

Declaration of Authorship

Hereby I declare that I have composed the presented thesis independently on my own and without other resources than the ones indicated. All thoughts taken directly or indirectly from other sources are properly denoted as such. This thesis has neither been previously submitted to another authority nor has it been published yet.

Bremen, 7th of Febuary 2023

Wilhelm Jochim

Zusammenfassung

Diese Arbeit beabsichtigt, einen Prototypen für einen Bluetooth Low Energy zu USB Tastatur-Adapter zu implementieren, der eingeschränkte Umgebungen ohne Bluetooth Low Energy Funktionalität unterstützt. Dieser Prototyp ist Basis für zukünftige Entwicklung und wird als Open-Source-Software veröffentlicht. Um die Nützlichkeit des Adapters zu evaluieren, wurden zwei Kriterien abgeleitet: Wie viel Kompatibilität kann der Adapter erzielen und führt die Nutzung des Adapters zu wahrnehmbar zusätzlicher Latenz? Der Adapter zeigte Kompatibilität mit einer breiten Anzahl an USB Umgebungen und ist voraussichtlich kompatibel mit vielen Bluetooth Low Energy Tastaturen und Microcontroller Boards, während es nur ungefähr 3,54 Millisekunden an Latenz hinzufügt im Vergleich zu einer direkten Bluetooth Low Energy Verbindung. Diese Latenz ist zwar wahrnehmbar, schränkt aber die Effektivität des Nutzers selbst in den anspruchsvollsten Aufgaben nicht ein.

Abstract

This work aims to implement a prototype for Bluetooth Low Energy keyboards to USB adapter, in constrained environments where the host system does not support Bluetooth Low Energy. This prototype is a foundation for further development and will be released as open-source software. To evaluate the usefulness, two criteria were derived: How much compatibility can the adapter reasonably achieve and does the usage of the adapter noticeably increase the input latency? The adapter was found to be broadly compatible with different host environments and is expected to be compatible with many Bluetooth Low Energy keyboards and Microcontroller boards, while only adding approximately 3.54 milliseconds of additional latency over a direct Bluetooth Low Energy connection. The latency is noticeable but does not impair the user's effectiveness in even the most demanding of tasks.

Contents

List of Figures

List of Tables

1. Introduction

The usage of a personal computing device requires the ability to accept input from the human user. This is achieved through input devices. Examples of such input devices are the classical desktop computer peripherals, namely the mouse and keyboard. Wireless peripherals are becoming increasingly popular. Whether it is about being able to increase the distance to the computer without needing to consider cable routing, increasing the freedom of movement in interactive entertainment peripherals, or just having a cleaner, cable-free work environment on the desk, there are many reasons to prefer wireless solutions. A widely adopted technology for wireless peripherals is Bluetooth. Bluetooth is supported by many personal computing devices, for example, smartphones, laptops, and many desktops. For desktop usage, the keyboard remains a popular input peripheral.

There are limitations to the usage of a Bluetooth keyboard as the main input device. Since Bluetooth input devices require a Bluetooth stack on the computer they are connected to, constrained environments without a Bluetooth stack can not be accommodated. Even on some devices which provide a Bluetooth Stack, some modes of operation do not. For example, a desktop computer might have an operating system supporting Bluetooth, but the BIOS on this device does not.

One method of overcoming these limitations is the usage of a USB device which adapts the Bluetooth communication from the keyboard to a wired USB connection for the computer. While some wireless keyboards do use such a USB adapter as the connection method, these tend to use proprietary wireless protocols. An example of such a proprietary protocol is Logitech Lightspeed used to connect Logitech peripherals to a USB adapter [8]. A lack of such adapters for generic Bluetooth keyboards was observed. To address this gap, this work aims to implement a limited-scope, open-source prototype for a Bluetooth to USB keyboard adapter. Publishing the prototype as open-source software allows more users to be able to use this project to satisfy such usage scenarios, but also broadens the scope of hardware a user might desire to run the software on. This broadened scope requires further design considerations in the implementation of the prototype.

Before further specification of the requirements and scope of the work can be established, background knowledge is required. After this background information is given in chapter 2, the detailed problem statement is specified. The problem statement chapter is followed by a description of the implementation including hardware and software solutions. The metrics of evaluation, established in the problem statement, are experimentally captured in chapter 5 and further discussed in chapter 6. The discussion includes possibilities where future work could continue. Lastly, the work is concluded.

2. Background

This chapter will examine the communication methods involved in this project, such as the wired USB protocol as well as the wireless Bluetooth and Bluetooth Low Energy protocols. Further, a way of describing human input events in the form of human interface devices (HID) is explored. Lastly, the necessary information for the implementation of embedded firmware on the USB adapter is given.

Readers familiar with the terminology of these technologies might choose to continue with the problem statement in chapter 3 but are nonetheless encouraged to refer back to this chapter as necessary.

2.1 USB

Universal Serial Bus (USB) is a wired personal computer extension specification that is hot-pluggable, meaning devices can be added and removed while the computer is operating. The connection which USB specifies is hierarchical with a host and device model. The host can be connected to multiple devices through the usage of hubs, making the connectivity graph a tree structure [2, Section Topology]. USB is designed to support a wide variety of devices, such as audio, mass storage, printers, or input devices [2, Chapter Introduction]. To support such a variety of devices, USB Descriptors are defined, which are data structures that can be read by the USB host to determine a USB device's function [2, Chapter 2, Descriptors]. Furthermore, USB devices can be powered over USB. This capability has evolved into the preferred method of charging modern mobile devices, such as smartphones.

Version 1.0 of USB was first introduced in 1996, with future iterations increasing the maximum transfer speed. Until USB 2.0, four pins were used in the USB connectors, while supporting USB 3.0 up to USB 3.2 Gen 2x2 necessitated an increase of the pin count to 9 [2, Chapter 1, Evolution of an interface].

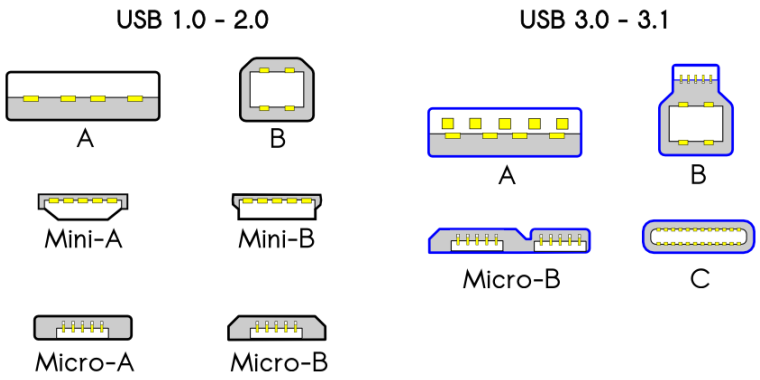


Figure 2.1: Comparison of USB connectors [12]

Every USB device has a buffer, which is referred to as an endpoint. These endpoints are connected to the host via pipes. A pipe can be one of four types:

1. **Control pipes**, used for short control messages and is mandatory to implement
2. **Bulk**, used for large transfers such as mass storage
3. **Isochronous**, used for predictable arrival times in real-time usages such as audio
4. **Interrupt**, for latency critical usages such as input devices

Previously, the USB connectors were different for the USB host, the Type A connector, and the USB device called Type B. Physical space constraints for mobile devices required smaller connectors, such as Mini-A and Mini-B, as well as Micro-A and Micro-B. B and Micro-B were changed in form to add the additional pins required for USB 3.0. The newest connector, called Type C, is used for both USB host and device roles and can be connected in two orientations. Adapters can be used to connect USB devices utilizing a different connector. The connectors are depicted in Figure 2.1.

2.2 HID

The concept of *Human Interface Devices* (HID) is integral to this work, so an explanation of the required parts of the HID specification is necessary. Originally introduced as a device class for USB [19, p. 1], it is now a Protocol that is also used over other transport methods, such as Bluetooth Classic [5] and Bluetooth Low Energy [4]. While intended for interfacing between a person and a computer, it may also be used for devices sending similarly structured data between each other [19, p. 2]. Examples of devices used for human input are mice, gamepads, rotary encoders, and keyboards.

Furthermore, HID introduces the concept of a Report. Reports are arbitrary data structures that enable the broad applicability of this protocol for different kinds of input devices. Reports contain the data exchanged between the HID devices. The structure of this data is encoded in a Report Descriptor, usually located in

read-only memory [19, p. 4]. At initiation, this Report Descriptor is read by a HID parser located on the HID host. The parser is intended to be implemented as a state machine, reading the contents of the Report Descriptor. This descriptor is structured in the form of an item list, changing the state of the parser. There are two categories of items: local and global. Local items will be dropped after defining a so-called Main item [19, p. 15]. An item can be one of:

- **Usage:** a 32-bit value that describes the information of what the specific report is measuring
- **Usage Page:** the higher order 16 bits of a Usage
- **Usage ID:** the lower order 16 bits of a Usage
- **Usage Minimum:** specifying the minimum usage for a range; consecutive usages for multiple reports at once
- **Usage Maximum:** specifying the maximum usage for a range; consecutive usages for multiple reports at once
- **Report Size:** the bit length of a report
- **Report Count:** the number of reports
- **Logical Minimum:** the minimal numerical value of report data
- **Logical Maximum:** The maximal numerical value of report data
- **Main Items:** special group of items responsible for allocating a new report structure. Main items are:
  - **Begin Collection:** starting a grouping of items
  - **End Collection:** ending a grouping of items
  - **Input:** data sent from the device to the host
  - **Output:** data sent from the host to the device
  - **Feature:** bidirectional data

However, requiring a HID parser is not always a preferable solution. For environments where the complexity of a HID parser, which would have to be able to handle arbitrary data and devices, is infeasible, the Boot Protocol is specified. One example of such an environment is the BIOS, which was the intended target of the Boot Protocol [19, p. 8]. The Boot Protocol is currently defined for two classes of devices, a computer mouse and a keyboard. This is specified with in the *bInterfaceProtocol* field in the USB Interface Descriptor. The possible values for this field are:

Protocol Code	Description
0	no Boot Protocol supported
1	keyboard Boot Protocol supported
2	mouse Boot Protocol supported
3-255	reserved values

Bitrange	Usage
0	Left Control
1	Left Shift
2	Left Alt
3	Left Super
4	Right Control
5	Right Shift
6	Right Alt
7	Right Super
8-15	Reserved field
16-23	1st keypress
24-31	2nd keypress
32-39	3rd keypress
40-47	4th keypress
48-55	5th keypress
56-63	6th keypress

Table 2.1: Boot Protocol bit interpretation table

A HID host using a HID device in Boot Protocol mode does not read the Report Descriptor, making it possible for HID devices to simultaneously support a device-specific Report format in addition to the Boot Protocol. If the HID device only sends the data in form of the Boot Protocol, an equivalent Report Descriptor can be defined, as the Boot Protocol is a valid subset of the Report Protocol. An implementation of such a Report Descriptor for Boot keyboard devices is specified in listing A.

The keyboard Boot Protocol consists of a 64-bit data structure. How each bit in this data structure is interpreted by the Host is depicted in Table 2.1. Each keypress is represented by a one-byte value stored in the higher 6 bytes. Each key has an associated keycode, which is mapped onto the character for the operating systems chosen keyboard language. On keypress, the next free byte will be set to the keycode. If additional keys are pressed, and the first key is not released, the byte continues to contain the keycode. At release, the byte is set to 0. The first byte of the datastructure is used for modifier keys, which are handled separately from the remaining keys. Modifier keys are: Left and Right Control, Left and Right Shift, Left and Right Alt, and Left and Right Super. Each modifier key has a dedicated bit in the byte, set to 1 while depressed and to 0 when released. The second byte is a reserved field that should not be interpreted by the host and may contain manufacturer-specific data. If not utilized, the byte should be set to 0.

While highly compatible, the Boot Protocol structure limits the number of keys that can be pressed simultaneously. At most six pressed keys, in addition to modifier keys, can be recognized at the same time. This limitation is one reason to define an alternative Report Protocol with more flexibility.

While HID is a very flexible protocol which allows for many device classes, the prototype will only consider the keyboard class of device.

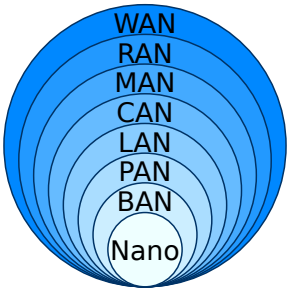


Figure 2.2:
Illustration of computer network types by spatial scope [3]

2.3 Bluetooth

Computer networks can be classified by their spatial scope. Figure 2.2 visualizes 8 distinct scopes for networks, including *Wide Area Networks* (WAN) for networks spanning wide geographical locations, *Local Area Networks* (LAN) for interconnection in a building and *Body Area Networks* (BAN) for wearable devices [7, p. 16].

Bluetooth is a technology for creating *Personal Area Networks* (PAN), located between LAN and BAN. A PAN connects personal devices in a workspace, such as input and output devices, smartphones, and printers. While a PAN can be wired (as, for example, USB is) Bluetooth creates *Wireless Personal Area Networks* (WPAN) [15]. It operates in the unlicensed 2.4 GHz ISM band [6, p. 190].

While both are referred to as Bluetooth, the specification consists of two distinct technologies: Bluetooth *Basic Rate* (BR) with an optional *Enhanced Data Rate* (EDR) extension, subsequently referred to as Bluetooth Classic, and *Bluetooth Low Energy* (BLE) [6, p. 187]. Bluetooth Classic and BLE are further described in subsection 2.3.1 and subsection 2.3.2 respectively.

2.3.1 Bluetooth Classic

Bluetooth was first introduced in 1998 [6, p. 103]. It is a connection-oriented technology, defining the roles of Bluetooth Central and Bluetooth Peripheral. These devices can fulfill different functionality and Bluetooth profiles describe the class of the device. The Bluetooth profile describes the interactions between Bluetooth devices, as well as between the Bluetooth subsystems [6, p. 277].

Input devices are a class of device present in the PAN of a workspace, which could be connected wirelessly by leveraging the Bluetooth Protocol. This was achieved in Bluetooth version v2.1 by adapting the HID Protocol for usage as a Bluetooth profile [5, p. 14]. The support for the limited-scope Boot Protocol was retained [5, p. 21].

2.3.2 BLE

BLE was introduced in 2010 with Bluetooth version 4.0 [6, p. 102]. While Classic Bluetooth and BLE are capable of coexisting on the same hardware, the following will focus on a BLE-only stack. An overview of the components composing the BLE stack is depicted in Figure 2.3. Bluetooth is split into two components: the Host (blue) and the Controller (green). These two are connected via the *Host Controller*

Interface (HCI). The protocol for this interface can be used over different transport methods, such as UART, USB, or through traces on the same chip package. Parts of the Host and Controller can either be implemented in hardware or software on a general-purpose CPU.

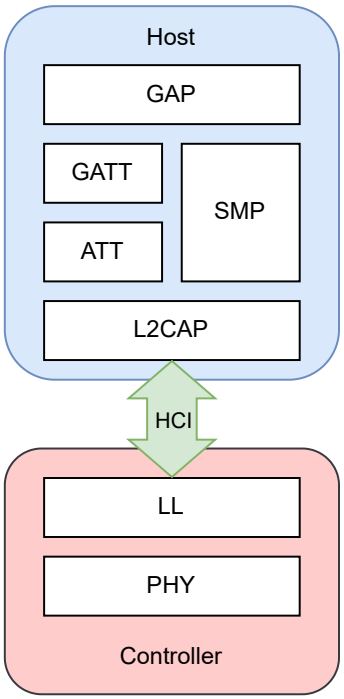


Figure 2.3: The BLE stack with the controller (red), the host (blue) and HCI (green) connecting them

The Controller is composed of two layers: the *Physical Layer* (PHY) and the *Link Layer* (LL). The PHY is nearest to the hardware and therefore the wireless antenna. It defines maximum transmission power, channels, transmission mechanisms, etc. [6, p. 2639]. The LL directly interfaces with the PHY, and is responsible for the different operational states of a BLE device. The states are defined [6, p. 2662] as:

- Standby State
- Advertising State
- Scanning State
- Initiating State
- Connection State
- Synchronization State
- Isochronous Broadcasting State

A BLE device will go through the States of Advertising, Initiating, and Scanning to establish a Connection state. In contrast, Isochronous Broadcasting and Synchronization state are used in non-connection-oriented operation. Furthermore, device addressing is handled on the LL, where Bluetooth addresses identify Bluetooth devices.

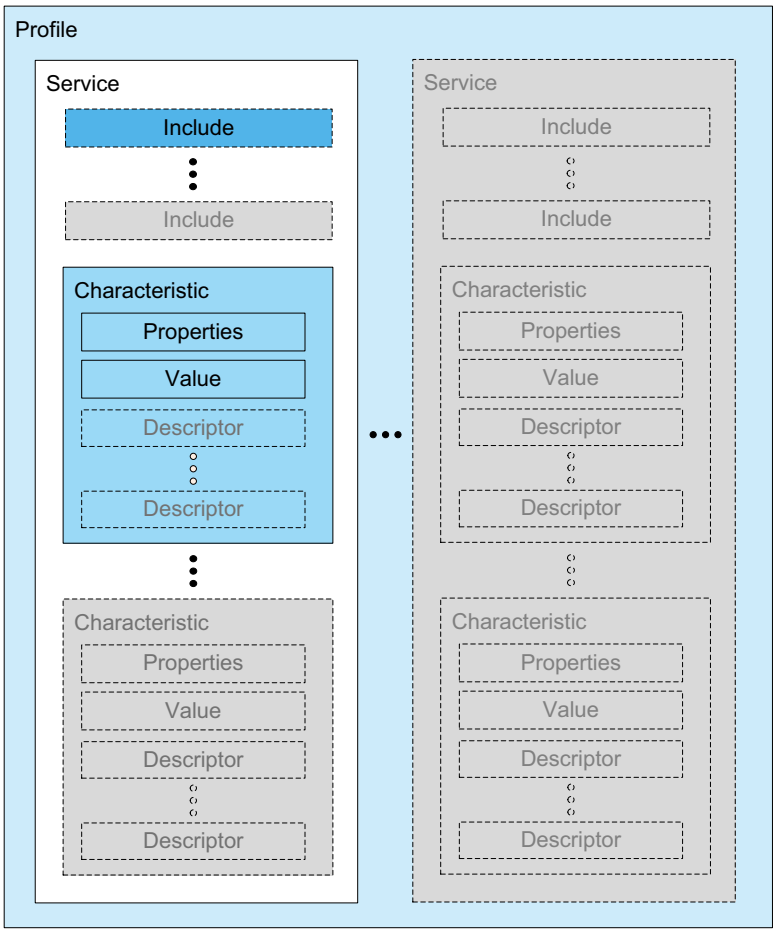


Figure 2.4: GATT-based profile hierarchy [6, p. 281]

Meanwhile, the Host is composed of a multitude of Protocols operating on different abstraction levels: The *Logical Link Control and Adaptation Protocol* facilitates and multiplexes the higher order protocols and communicates with the LL [6, p. 1013]. *Security Manager Protocol* (SMP) is the BLE-specific protocol for encryption key exchange and management [6, p. 207]. The *Attribute Protocol* (ATT) defines how data in the form of attributes is delivered between the ATT server and ATT client [6, p.207]. ATT is encapsulated in the *Generic Attribute Profile* (GATT). GATT includes definitions for logically structuring data sent between the GATT Server and GATT client, visualized in Figure 2.4. The highest level definition is a profile, describing what kind of application the BLE device is implementing. For example, there are audio, heart rate monitor, and input device profiles. A profile contains one or more Services, having distinct functions. An example of a Service present in multiple profiles is the Battery Service, which allows access to the battery charge percentage of a BLE device. Furthermore, a Service can include other Services and is composed of Characteristics. Characteristics contain properties, values, and descriptors. To manipulate the data in a Characteristic, it can be read, written, notified, or subscribed to. Additionally, permissions can be set on Characteristics, such as limiting read access to clients with an encrypted connection.

The *Generic Access Profile* (GAP) is the highest abstraction layer and allows for connections and broadcasting, device discovery, and security by utilizing the underlying layers.

2.3.3 HOGP

To implement HID devices with BLE technology, the *HID-over-GATT Protocol* (HOGP) profile is defined. A BLE device can consist of one or more HID Services, with the HID device implementing the GATT server role and the HID host operating as the BLE GATT client. The Report Descriptor is given in the Report Map Characteristic, with Reports being implemented in further Characteristics [4].

Furthermore, Boot Protocol support is retained in HOGP. This is realized by introducing a Protocol Mode Characteristic switching the operation between Report Mode, Boot keyboard, or Boot mouse. Additionally, the Boot keyboard Input Report Characteristic, the Boot keyboard Output Report Characteristic, and the Boot Mouse Input Report Characteristic are present. The Boot keyboard Input Report Characteristic is used to receive the 8 Byte array seen in Table 2.1. Since this mode of operation is separate from the Report Protocol, these Characteristics are not specified in the Report Descriptor. Supporting the Boot Protocol is mandatory for all HOGP keyboard devices.

2.4 OS

An *Operating System* (OS) is an abstraction of the hardware for applications. Examples of this kind of abstraction are file systems, device drivers, and networking interfaces. The OS itself runs in a privileged mode of execution called kernel mode. In kernel mode, the software has full access to the underlying hardware. The applications run in user mode, where access to the hardware is achieved via system calls to the kernel. In addition, OSes allow for multiple applications to run concurrently. This is achieved with the concept of processes, a program in execution, and a scheduling algorithm deciding when switching between multiple processes occurs. While not technically part of the OS, the user interfaces on top of the OS are grouped with them. An example of the distinction is the X11 graphical user interface server, sometimes also referred to as an X-Server or similary, which is used in both Linux and BSD OSes.

2.5 RTOS

A *Real-Time Operating System* (RTOS) is a specialization of an OS with a dependency on real time. For example, an external device generating events which the OS has to respond to is a form of real time dependency. Therefore, an RTOS guarantees a maximum response time for such events. This is different from minimizing the latency of responses, which OSes also strive towards, since the relevant consideration in an RTOS is meeting the guaranteed deadline [16, p. 164]. Achieving this goal requires architectural considerations, such as the scheduling algorithm employed. When used in embedded systems, RTOS and application code can be compiled into one firmware, in contrast to the dynamically loaded binary of a typical application.

2.6 Zephyr

Zephyr is an open-source RTOS hosted by The Linux Foundation [18]. It supports a wide variety of target platforms with software libraries for many protocols, including Ethernet, WiFi, CoAP, Thread, USB, Bluetooth Classic, BLE, and HID [20]. Zephyr is implemented in the C programming language.

2.6.1 ZMK

ZMK is an open-source firmware for keyboard devices. Being an open-source project with many contributors and users, it allows for a high degree of customization. While ZMK has support for USB, it focuses on wireless connectivity over BLE [21], which allows for long battery life in ZMK devices.

2.7 Constrained Environments

During the usage of a Bluetooth keyboard in a desktop scenario, there are situations where such a keyboard is not supported. This occurs in constrained environments, where no Bluetooth stack exists. In these constrained environments the usage of a wired alternative is required. Some situations where this applies in a desktop setting are:

- BIOS/UEFI
  The BIOS, and its replacement UEFI, is software embedded on the motherboard of a desktop computer. It provides a configuration menu for the system's hardware. While using the BIOS/UEFI menu is an infrequent occurrence for most users, when needed, Bluetooth support is absent.
- Bootloader
  Actions such as booting a different OS in multiboot environments or other tasks performed by interaction with the bootloader require wired keyboards.
- initramfs (with FDE)
  The *initial RAM filesystem* (initramfs), is the first stage of execution in Linux. Most initramfs environments do not require user input, but systems using *Full Disk Encryption* (FDE) display password prompts in this stage.
- Operating Systems
  The main OS of the system might also not contain Bluetooth support, either through the lack of Bluetooth hardware or the absence of a Bluetooth software stack for various reasons.

2.8 USB Adapters

An approach for using wireless keyboards in constrained environments is the use of an adapter which handles the wireless communication to the peripheral, and presents itself like a regular USB device to the host. This solution is used by some keyboard manufacturers. The manufacturers bundle the keyboard with a proprietary adapter, which might use Bluetooth or a different proprietary communication protocol. Some only work with this specific keyboard model, others are more broadly compatible with the manufacturer's lineup. An example of the latter is Logitech Unifying Receiver [9].

3. Problem Statement

As mentioned in the introduction, Bluetooth keyboards are not usable in constrained environments. One possible solution to close the gap is implementing a Bluetooth to USB adapter for generic Bluetooth keyboards. It should translate wireless Bluetooth HID device data to a wired USB device. The prototype for such an adapter is limited to BLE HID keyboards. While many keyboards use the Bluetooth Classic protocol, supporting both BLE and Bluetooth Classic would require separate implementations for each. This would not be feasible in the required timeframe. BLE was chosen to be implemented first, since BLE HID has lower input latency, longer battery life, and a versatile open-source firmware implementation in the form of ZMK.

To measure the success of the prototype, some metrics are required. Compatibility and latency were chosen as metrics. This results in three research questions:

1. To what extent is the *implementation* of a BLE to USB HID keyboard adapter feasible?
2. How much *compatibility* can the adapter reasonably achieve?
3. Does the usage of this adapter noticeably increase the *input latency*?

These questions are further discussed in section 3.1, section 3.2, and section 3.3, respectively. Finally, in chapter 7, they are answered.

3.1 Implementation

For the success of the prototype, the scope had to be clearly defined and limited. Bluetooth Classic is not included in the scope. Furthermore, the first prototype only implements the Boot Protocol. Since every BLE keyboard and every USB host environment is required to support the Boot Protocol, no device compatibility should be lost due to this constraint. Further work should not be restricted to the Boot Protocol, however, since it involves other compromises, such as no more than 6 simultaneous keypresses.

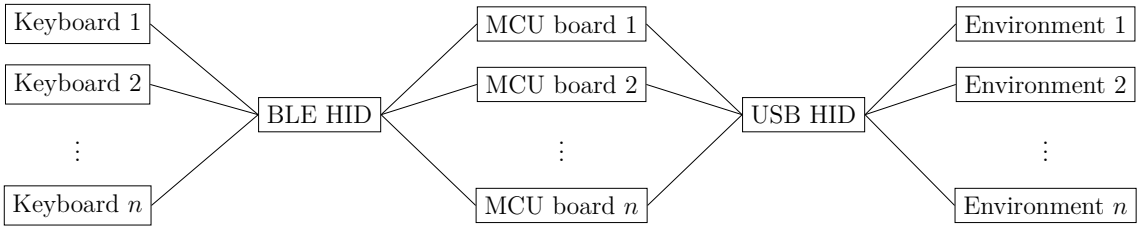


Figure 3.1: Three aspects of compatibility

3.2 Compatibility

Since the adapter is a generic communication bridge between two physical devices, maximizing the compatibility for each of these is a desirable goal. The importance of compatibility becomes even more apparent when considering the publication of the implementation as open-source software. The, possibly broad, audience of open-source software increases the range of hardware the user might possess and which would ideally be supported by the prototype. As seen in Figure 3.1, there are three aspects to this compatibility: the number of supported keyboards, the number of supported constrained environments, and the number of *microcontroller* (MCU) boards the software can run on.

The left part of the figure depicts the variety of keyboards. There are many keyboards from different manufacturers, each having the possibility of device-specific edge cases. Unfortunately, keyboard compatibility is hard to evaluate without having access to a wide variety of devices, but useful evaluation is still possible by selecting representative devices. The keyboards connect to the MCU boards via BLE HID, depicted in the middle. The MCU boards are a further compatibility factor, as the prototype should, ideally, support multiple such boards. This compatibility can also be referred to as the portability of the prototype. The presence of hardware support for both BLE and USB is obligatory, but if these constraints are satisfied, the MCU Board should be a viable target platform for the software. The MCU boards connect to the host environments over USB HID, depicted on the right side of the graphic. *USB host environments* have different HID parser implementations. Evaluating the USB adapter with multiple environments will be necessary to ensure broad compatibility. As an example, some USB host environments do not support the Report Protocol mode, and require the keyboard to implement the Boot Protocol.

3.3 Latency

Adding a middleman in the communication between the BLE keyboard and Host may introduce additional latency. In addition to the latency of the BLE communication, the microcontroller will have to process the data, and add the latency of the USB communication. To answer whether this addition is perceptible by humans, an evaluation of the added latency is required. Implementation details need to be aware of possible latency implications. Having too much latency impacts the user's ability to efficiently complete the desktop computing tasks, and in a highly interactive task, such as interactive entertainment, even small increases in latency are perceptible [1].

3.4 Summary

To summarize, from these research questions a set of requirements can be derived:

1. Fundamentally, the project must offer a method to support the use of BLE keyboards in constrained environments.
2. A reasonable level of compatibility should be achieved by the project.
3. While this work investigates the noticeability of the added latency, the actual requirement derived here is that the offered method's added latency must not impair the user in completing desktop computing tasks.

4. Implementation

This chapter describes the approach used in the prototype. This includes the hardware used in the development process and why it was chosen, as well as the operational details of the software and the architectural decisions leading to them.

Figure 4.1 describes a general overview of an adapter between a BLE HID device and a USB HID host Environment. The BLE HID device (pictured in blue) is a keyboard supporting the HOGP for communication. For the applicability of this adapter, the additional native support for USB of the keyboard is not a factor, although the keyboard used in chapter 5 will be required to support USB. The USB HID host (pictured in yellow) is a constrained environment, which does support the USB HID protocol as a HID host, but does not have support for being a BLE HID host.

The implementation scope encompasses the adapter (pictured in green). It has three primary functions: implementing a BLE HID host for communicating with the BLE keyboard, implementing a USB HID device for communicating with the USB HID host, and relay the HID messages between those two internal components. While two different transports are used, the common HID Protocol is present, simplifying the internal mapping between the two components.

As a software basis, the Zephyr RTOS was used. Using Zephyr broadens the portability of the code to those of Zephyr's target platforms which have BLE and USB device support. Using Zephyr's USB, BLE, and HID software implementations gives the additional benefit of being able to leverage the capabilities of this large and continually improving project, on which other third parties, such as the ZMK firmware project, also rely; the adapter can benefit from improvements and fixes for edge cases contributed to Zephyr by other projects and the Zephyr team.

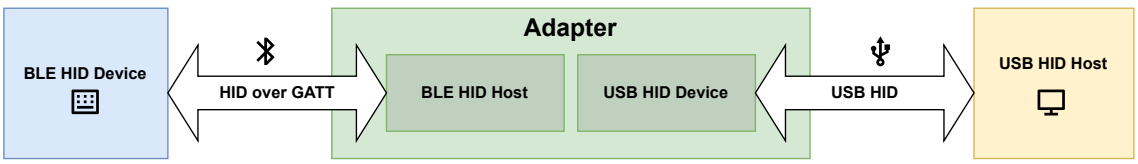


Figure 4.1: General overview

4.1 Hardware

While the prototype is intended to support a broad range of hardware, some hardware needed to be chosen for developmental purposes. Searching for an MCU board containing BLE and USB device support, boards based on the Nordic Semiconductor NRF52840 *System on a Chip* (SOC) are a good fit, as it already contains a BLE and USB 2.0 controller on package [13]. In essence, no additional components would be necessary to support this project. The code is running on an ARM Cortex M4 clocked at 64 MHz.

Choosing a board with the NRF52840 SOC was limited by the official Zephyr support and availability. The main board used in development is the NRF52840-MDK by Makerdiary [10], seen in Figure 4.2. It contains additional hardware with a DAPLink debugger, making development simpler. It also has the ability to access the serial output via USB. A disadvantage is the form factor. The board contains a Type C USB port, requiring a cable to connect to a USB host. For daily usage, a dongle form factor would be preferred. Figure 4.3 shows this in the form of the NRF52840 dongle from Nordic Semiconductor [14]. It contains the same SOC, but lacks the additional benefits of the NRF52840-MDK. Both devices were used in the development process and are verified compilation targets.

Figure 4.2: Makerdiary NRF52840 MDK

Figure 4.3: NRF52840 Dongle

Two BLE keyboards were used for development purposes. The Microsoft Bluetooth Keyboard (Figure 4.4) is a wireless keyboard offered by Microsoft [11]. It supports the BLE Boot Protocol mode. The Nice!60 (Figure 4.5) is a BLE keyboard board with an NRF52840 SOC and hot-swappable Cherry style switches. It supports ZMK firmware. The Nice!60 has the ability to use a wired USB connection, in contrast to the Microsoft Bluetooth Keyboard, which does not feature a USB port. Additionally, using an open-source keyboard allows access to the firmware logs of the keyboard. The logs aid in the debugging process during development. ZMK did not support Boot Protocol mode, but had the compilation option of a Boot Protocol-like Report. The firmware was patched to add the necessary GATT Characteristics required for Boot Protocol mode operation. This addition is a possible contribution to the ZMK firmware, but is out of scope for this work. Both keyboards were used in development and thereby tested for compatibility with the project.

The development was performed on multiple Linux desktop devices, meaning the first USB host environment compatibility reference was a Linux system. In later stages of development, other operating systems were tested, such as Windows 10, and compatibility problems with those were resolved. In section 5.1 the final USB host compatibility is examined extensively.

Figure 4.4: Microsoft Bluetooth Keyboard

Figure 4.5: Nice!60

The primary combination of development hardware was the NRF52840MDK with the Nice!60 keyboard and a Linux desktop.

4.2 Software

The following section will examine the architecture of the implementation and follows with an explanation of the program flow. The program flow includes multiple threads. Firstly, the initialization procedure will be examined followed by the repeated handling of keypress events.

The code is structured into four units:

- `main.c` contains the `main` function and is responsible for initializing the program.
- `usb.c` implements the USB HID device role.
- `ble.c` implements the BLE HID host role.
- `hid.c` is an abstraction bridge between BLE and USB.

The initialization call order is visualized in Figure 4.6. The entry point of the program is the `main` function. It starts by calling `init_usb` function exported by `usb.h`. This synchronous call in turn calls the corresponding Zephyr functions for USB initialization, returning to the main function if successful. This is repeated for BLE by calling `ble_init` exported by `ble.h`. Next, `handle_ble` is called, which registers Zephyr callbacks and starts to asynchronously scan for BLE HID keyboards and connect, pair, and subscribe when possible. `handle_ble`, meanwhile, returns, and the `main` function calls `handle_usb`. In contrast to BLE, USB uses the main thread of the program. With this, the initialization is complete.

The BLE compontent starts a discovery and bonding process. After bonding, it continues by discovering the HID Service, setting it to Boot Protocol mode and subscribing to key events. This is described in detail in subsection 4.2.1. BLE receives a callback when key events are available in the form of the `read_keys` function. `ble.c` imports the `hid_write` function from `hid.c`, which subsequently calls `usb_hid_write` from `usb.c`. This sequence is performed in the callback thread from BLE. `usb_hid_write` copies the HID data into a threaded message queue to be executed by the main `handle_usb` thread.

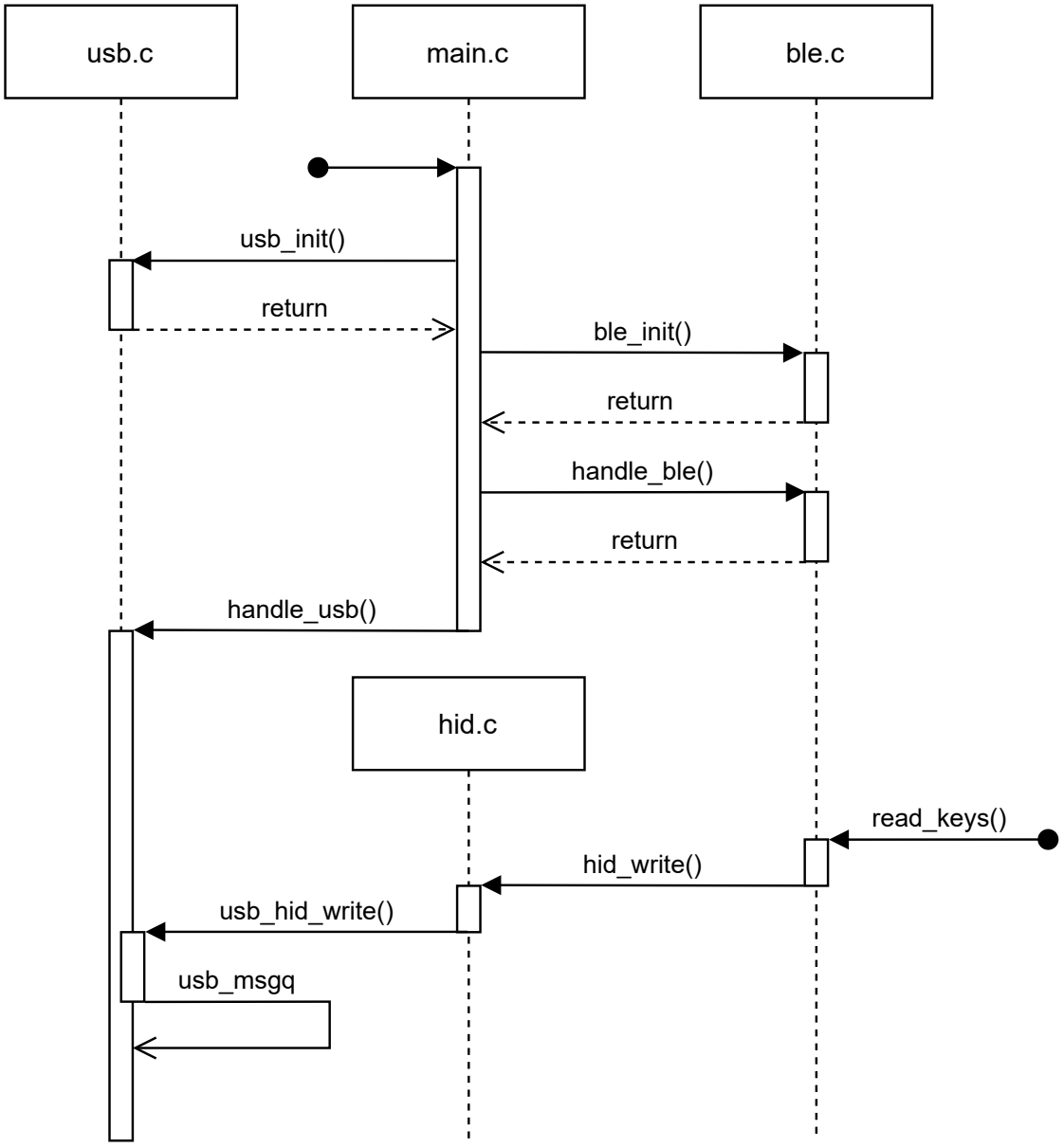


Figure 4.6: Sequence diagram for the program flow of the adapter

4.2.1 BLE

Zephyr's Bluetooth library is structured in an asynchronous manner. The state management is handled in the library and the application code is notified of changes via callbacks. This means the adapter's BLE component, implemented in `ble.c`, contains a collection of callbacks, which are registered at initialization time. After the main component calls `handle_ble`, a device scanning procedure is initiated. This corresponds to the Scanning State mentioned in chapter 2 and continuously discovers devices.

For every BLE device found, the *extended inquiry response* (EIR) is checked to detect if the device is a HID device. EIR is a data structure containing additional information about the device made available to the scanning device [6, p. 1304–1305]. If the device found does contain a HID device Service in the EIR, the creation of a connection to the device is attempted.

If the device connects, it starts out in security mode 1, the lowest possible of the four security modes defined by Bluetooth [6, p. 1267]. The security aspects of the modes as described by Zephyr are:

- Level 1: neither encryption nor authentication
- Level 2: encryption without authentication
- Level 3: encryption and authentication
- Level 4: encryption with a 128-bit key and authentication

The adapter requests security mode 2. Zephyr considers any mode higher or equal to it as fulfilling this request. This is relevant as the ZMK firmware does not support pin entry fully, a requirement for authentication, and the Microsoft Bluetooth Keyboard requires a minimum of Level 4. If pin entry is used, Zephyr generates a random 6 digit number, which is printed out via serial. The pin should be entered into the BLE keyboard via its own mechanism.

Zephyr defines a callback for pairing confirmation. In this step, user input is requested to confirm the pairing for this particular device found in scanning. For prototyping purposes, all connections are accepted. This should be changed to involve the user, possibly over the user buttons present on the boards.

A successful pairing results in the starting of the Service discovery procedure. If a HID Service is found, the HID protocol mode Characteristic in this Service is searched for. When found, the current value is read, a 1 value is written and, finally, the value is read again to ensure the mode switch has been applied. 0 is the initial value and corresponds to the Report Protocol, while our desired state of 1 is the Boot Protocol mode.

After switching the keyboard into the Boot Protocol mode, the HID boot input Characteristic is found and subsequently subscribed to. Being subscribed entails the keyboard sending the adapter the data of any key changes. The code receives this in the form of the `read_keys` callback.

4.2.2 HID

`hid.c` is a thin abstraction of `usb.h` with removing the import of `usb.h` in `ble.c` being the main objective. Future work would be able to utilize this abstraction for new features, for example, converting a non-Boot-Protocol-compatible HID Report to the Boot Protocol. Currently, HID contains a single function `usb_hid_write`, which calls an equivalent function with the same parameters in the USB component.

4.2.3 USB

The USB component calls multiple Zephyr functions required to initialize a new USB HID device and enable USB. One of these functions is `usb_hid_set_proto_code`, which registers the USB HID device to operate in the Boot Protocol mode. Zephyr specifies that this should be done before `usb_hid_init` is performed. During `usb_hid_register_device` a Report Descriptor is defined for the USB HID device. In this implementation, the Report Descriptor is a Report structured equally to the Boot Protocol.

The thread synchronization necessary for the USB device is entirely contained in `usb.c` without the caller having to be aware of it. The synchronization is implemented in the form of a threaded message queue provided by Zephyr. The HID data received is copied into the ring buffer of the message queue, while `handle_usb` is waiting for new elements to arrive in the queue, which it then passes to the Zephyr function `hid_int_ep_write` without any modification to the data.

With this implementation, the project reaches the prototype stage. Evaluation of the prototype's success is examined in the following chapter. The prototype will be compared against the requirements.

5. Evaluation

This chapter consists of two parts: the compatibility evaluation and the latency experiments. While three aspects of compatibility are defined in this work, and all three were considered in the implementation process, only the USB environment compatibility could be tested extensively.

5.1 Compatibility

The compatibility of the adapter with as many environments as possible is of special interest, since it is intended to be applicable in environments already constrained due to the lack of a Bluetooth stack. Evaluating compatibility requires testing with a broad variety of devices and software. Each environment implementing a USB stack could handle USB HID devices differently. Major operating systems should be evaluated, as well as device-specific BIOS and UEFI environments. Since testing many physical devices is not feasible, one BIOS and two UEFI devices were tested. Since the implementation operates in either Boot Protocol mode, or the equivalently structured Report Protocol mode, a high compatibility is expected. Any correct implementation of the USB HID host should be able to understand the adapter sending Boot Protocol data, if the adapter was implemented correctly.

All tests were performed with the NRF52840-MDK board, with logging disabled at compilation time. The keyboard used was the Nice!60, since, without being able to see the serial output of the adapter, pairing the Microsoft Bluetooth Keyboard, which requires pin entry, is not possible. During the test, the Nice!60 was powered and discoverable, while the adapter was plugged into the USB HID host under examination. After a short delay, waiting for the completion of the bonding process, the interactions with the environment were performed. In most cases, an exemplary standard graphical application was used. If the test involved this, the application used to receive the key input events was a web browser, since this is an application present on most machines. If a website with a text field was necessary, the google.com search field was used.

5.1.1 BIOS

The ThinkPad R60 laptop still uses a BIOS, making it a viable test candidate for BIOS support. It is unclear whether this BIOS does contain a HID parser or requires Boot Protocol operation, since the Boot Protocol was specifically designed for this usecase [19, p. 8]. The adapter was connected and the F12 key was used to access the Boot selection menu. The menu point <enter setup> was selected, and the keyboard was used to navigate the BIOS. The BIOS would not have supported BLE keyboards without the usage of the adapter.

5.1.2 UEFI

Two different UEFI devices were examined: A Lenovo ThinkPad E14 and an ASUS ROG STRIX x470-I GAMING based desktop. The adapter was connected to the ThinkPad E14. During the boot sequence while the Lenovo logo is visible, the Enter key was pressed. Afterward, the F1 key was pressed to select entering the UEFI. The arrow keys were used to navigate the UEFI. On the desktop, the F2 key was used to enter the UEFI. Afterwards the arrow keys were used for navigation. These UEFI devices would not have supported connecting a BLE keyboard but were usable with the adapter.

5.1.3 Windows

The adapter was tested against a machine running the Build 19044 of Windows 10. The motherboard was an ASUS M5A99DX PRO R.20 with an AMD FX-6300 CPU. Since this was a shared, public machine, testing compatibility with the BIOS/UEFI of this computer was not possible. The adapter was plugged into the frontal USB port. The interaction with the Windows system started by typing in the username and password into the Windows login screen. After successful login, the Windows desktop environment was used, specifically the search bar, to open the Microsoft Edge web browser. Lastly, inputting text into the address bar and text input fields on websites was tested. As everything worked as expected, the test was successful, and the adapter is compatible with Windows 10.

5.1.4 MacOS

MacOS was tested on a public 5K iMac 27-inch model, year 2020, on which the console mode had been made inaccessible. Specifically, the machine was running MacOS Catalina 10.15.7 with an Intel i9 10910 and an AMD Radeon Pro 5700 XT graphics card. The adapter was connected to one of the USB Type A ports at the back. The interaction consisted of inputting the credentials into MacOS' login screen, interacting with the desktop environment, specifically the launchpad search, to find safari, and interact with the address bar and website text input fields. There were no compatibility problems observed with MacOS; accordingly, the adapter passed this test.

5.1.5 Android

For Android compatibility evaluation, a Pixel 6 running version 12 (Build SQ3A.220605.009.B1) was used. It contains a USB Type C host port, meaning the adapter could be connected by using a Type C to Type C cable. The Pixel Launcher search was used to locate the Chrome browser and enter text into the address bar and website text field. The result was successful keyboard operation and compatibility with the Android OS.

5.1.6 iOS/iPadOS

Some models of Apple iPads use the USB Type C connector, making connecting a wired USB keyboard easier. The device chosen was an iPad Air 4th Gen running iPadOS 15.5. iPadOS and iOS share internal implementations, with iPadOS having a user interface better optimized for the bigger screen sizes of iPads. Meaning, compatibility with iPadOS gives a strong indication for probable compatibility with iOS as well. The test involved searching for Safari in the App Drawer, opening the web browser, inputting into the address bar and typing into the search field of google.com. No compatibility problems were observed.

5.1.7 OpenBSD

The OpenBSD OS is a constrained environment since it does not currently contain any Bluetooth support[17]. So a keyboard only supporting BLE, such as the Microsoft Bluetooth Keyboard, would not be usable in OpenBSD without the usage of the adapter. Version 7.1 of OpenBSD was installed on a ThinkPad R60. The keyboard input was tested on the OpenBSD console. Logging into the console with the credentials and typing in the console was successful, demonstrating OpenBSD support.

5.1.8 Linux

Linux support was tested on three different platforms: the Raspberry Pi 3B+, a Lenovo ThinkPad E14 and an ASUS ROG STRIX x470-I Gaming based desktop. Using the Raspberry Pi 3B+ platform was necessary for the experiment setup in section 5.2. Linux's options for customization also bring the ability to have incompatibilities with specific setups. The laptop and the desktop where equipped with vastly different software.

The Raspberry Pi 3B+ was running Raspberry Pi OS version 11. This Linux distribution uses Version 5.10 of the Linux kernel. The test consisted of entering the user credentials in the Linux console and inputting text into the console. This was successful.

Manjaro Linux was used on the ThinkPad E14. Manjaro is a rolling release distribution without version numbering. During installation, the Manjaro installers full disk encryption was used. In this setup, the GRUB bootloader is used, and itself resides on an encrypted partition. Entering the decryption password is thereby handled by GRUB, and while other full disk encryption setups might have the initramfs responsible for unlocking the Linux root partition, neither would support the use of a Bluetooth keyboard without an adapter such as the one used here. Linux Kernel 5.18 was used, in combination with GRUB 2.06.

The adapter was connected to a USB port. During startup, the decryption password was entered in the GRUB password prompt. Afterward, the escape key was pressed to bring up the GRUB boot selection menu. Navigation in the menu was tested, and the Manjaro installation was selected for boot. The next step was using a graphical login screen to authenticate the Linux user. This was accomplished using version 0.19.0-8 of SDDM. The desktop environment used was Gnome 42.3.1-1 using the Wayland display protocol. In Gnome, the application search was used to locate the Firefox web browser, enter text in the address bar and enter text in a website text field. All steps were performed with a BLE keyboard connected via the adapter and no problems were encountered during the test.

The desktop uses the same Manjaro installation, but instead after SDDM the desktop environment used was based on an X-Server in the form of i3-gaps version 4.20.1-2 with the picom 9.1-3 compositor. The adapter was connected, the desktop booted, the GRUB password entry dialog was used, and the escape key was pressed to enter the boot selection menu. The Manjaro installation was selected and booted. The user was logged in using SDDM. In the i3-gaps desktop environment, Firefox was started and text entry in the address bar and on websites was tested.

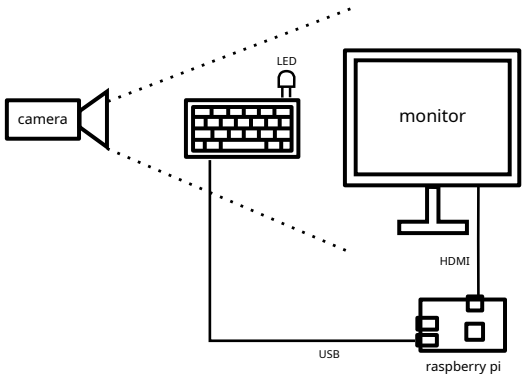
The adapter was compatible with Linux in the console, in Wayland, and in X-Server environments.

5.2 Latency

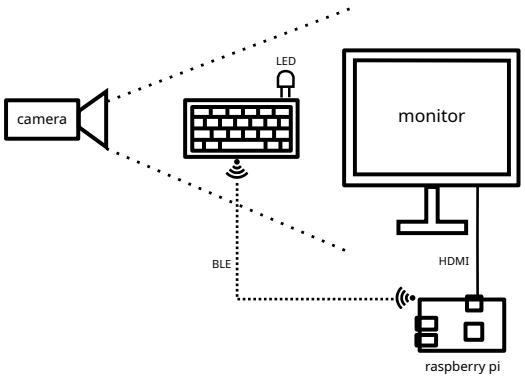
Latency in interactive computing environments stems from multiple components and accumulates through the input/output chain. Latency is added by:

- The latency inherent to the keyboard. The hardware and software of the keyboard require time to register a key press and process it.
- The keyboard communicating the key press to the computing device. This includes transports like USB, Bluetooth Classic, BLE or proprietary wireless protocols.
- The software on the computing device. This includes the operating system's processing time as well as the application responding to the input and generating the graphics necessary to display the change.
- The time necessary to transmit the data from the computing device to the monitor.
- The firmware running on the monitor and possibly processing the signal before displaying it.
- The reaction time of the display technology used by the monitor.

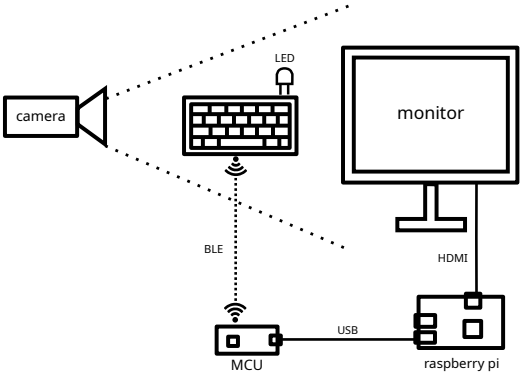
The latency relevant to the project is the latency of transmitting the data between the keyboard and the computing device. This latency is hard to measure on its own. As the relevant information is the noticeability of the latency added by the introduction of the adapter into the latency chain, the measurement of the total time



(a) USB test setup



(b) BLE test setup



(c) adapter test setup

Figure 5.1: latency test setup overview

elapsed from key press to light emitted by the display (the end-to-end latency) is a sufficient metric. This approximates the latency the user observes while interacting with the computing device.

An experimental setup, optimized for minimizing required testing equipment, is seen in Figure 5.1. The key press event is indicated by an LED connected to a singular key on the keyboard. In this case, the "equal sign" key was used, as it is near a corner of the keyboard and produces a printable character. The LED does have an unknown amount of latency before turning on, but this latency will be the same between all experiments and will not hinder the comparison between transportation methods. The LED is placed next to the monitor in the frame of a high-speed camera. The latency is measured by counting the video frames from the camera between the LED turning on and the monitor displaying the character.

To further eliminate the variable latency from each specific keyboard, the same keyboard is used in BLE and USB tests. The Nice!60 (Figure 4.5) supports both USB and BLE natively. The host device must support BLE HID host, USB HID host and be able to display to a high refresh rate monitor. These requirements are easiest to fulfill with an environment running a conventional operating system, for example, Linux. An affordable and widely available hardware platform for this use case is the Raspberry Pi 3B+. Running on version 11 of the Linux distribution Raspberry Pi OS, and configured to run in video mode 48. This Raspberry Pi video mode has an output resolution of 720 x 480 pixels and 120 Hz refresh rate. The Raspberry Pi is connected to a Samsung LX49HG90DMUXEN monitor via HDMI.

This monitor supports up to 144 Hz refresh rate but is limited to 120 Hz using HDMI. The camera used is a Pixel 6 smartphone recording at 240 fps. For tests involving the adapter, the NRF52840-MDK was used, with serial logging disabled at compile time, as logging might add additional latency.

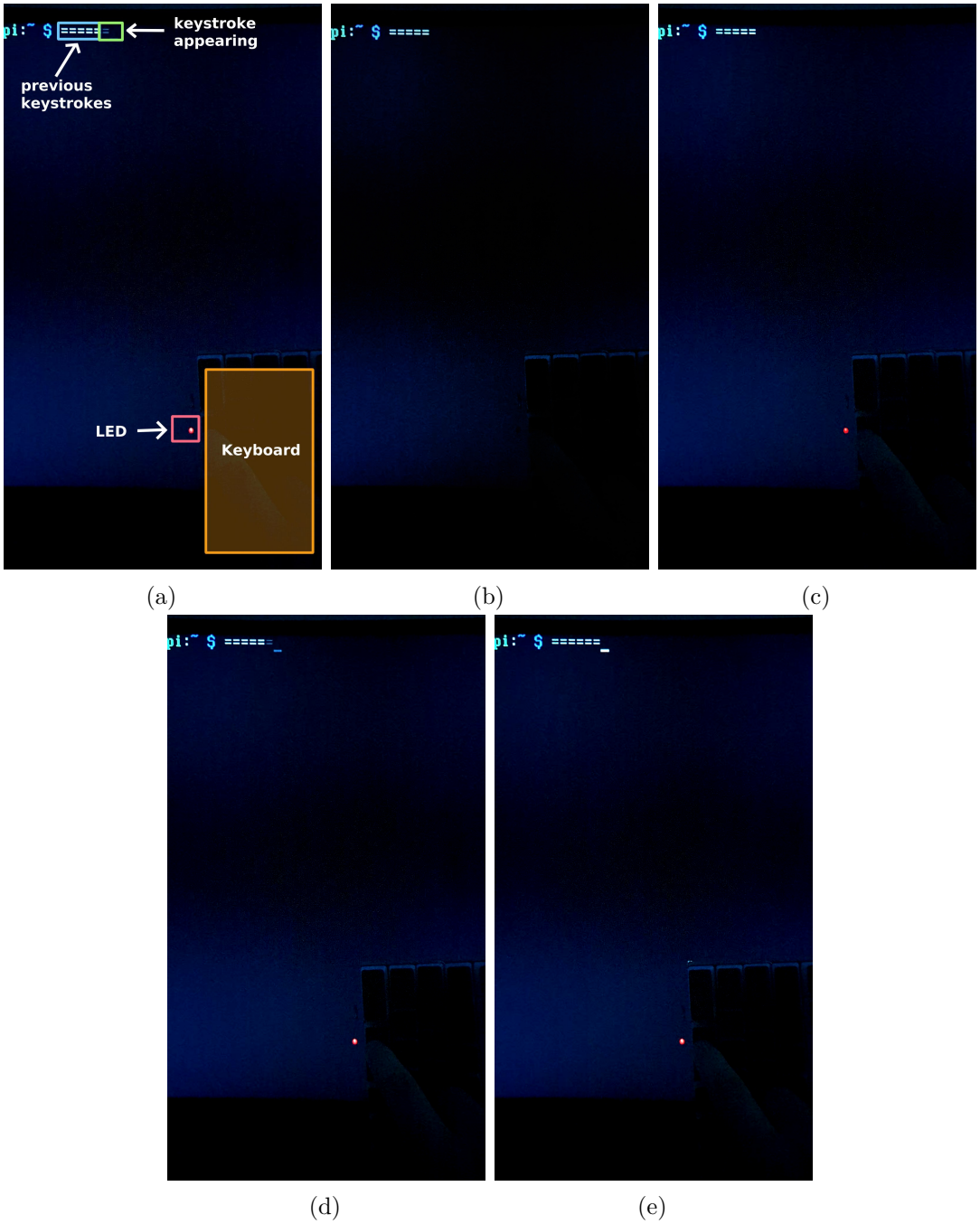


Figure 5.2: Four exemplary frames from the latency measurement

To minimize the latency added through host software, the key presses are displayed by the Linux console as seen in Figure 5.2. Figure 5.2a provides labels for the components visible in the camera frame. In 5.2b the initial state is visible. A single video contains 20 key presses of which five were already executed in this frame. 5.2c is the frame where the counting of the latency frames begins, with the first sign of the LED turning on. As observable between 5.2c and 5.2d, the LED does not achieve

full brightness in the first frame. Counting stops after the character starts appearing in 5.2d. 5.2e shows a fully present character on screen. After the key press event, the key is released, and another test is initiated.

The data captured in these measurements is represented in the histogram in Figure 5.3. Three Videos were recorded, for USB (yellow), BLE (blue), and the adapter (green), respectively. Each Video contained 20 key presses. On the x-axis, the latency is plotted in number of frames from key press to character on screen and on the y-axis the occurrences of this frame count. The USB results tended to be the lowest with 3 to 5 frames of latency counted, followed by BLE between 4 and 8 and lastly the adapter with 5 to 9.

The mean latencies are presented in the bar graph in Figure 5.4. The three connectivity methods on the y-axis are plotted against the milliseconds of latency on the x-axis. The number of camera frames can be converted into milliseconds by multiplying the values with the duration of a singular frame, which is $\frac{1}{240}$ of a second or approximately 4.2 ms. A USB connection offers the lowest mean latency, with the adapter being the slowest option and native BLE in between. The numerical means are 17.50 ms for USB, 24.79 ms for BLE and 28.33 ms for the adapter. Therefore, the mean latency added by switching from USB to BLE is higher at 7.29 ms, while introducing the adapter only adds another 3.54 ms to the total latency.

Figure 5.3 also describes a tendency for connections with wireless components such as BLE and the adapter to have a higher spread, which is particularly evident from the boxplot given in Figure 5.5. The y-axis plots the latency in milliseconds and the x-axis displays the three different methods of connectivity. The quarterlies and minima and maxima show the difference in spread. USB has a standard deviation of 2.8 ms, BLE 5.01 ms and the adapter 5.03 ms.

The latency figures show a favorable picture. While a camera frame has a duration of $\frac{1}{240}$s $\approx$ 4.2ms the monitor refreshes with $\frac{1}{120}$s $\approx$ 8.3ms. Due to this, the mean additional latency of 3.54 ms is less time delay than a monitor frame. Considering a monitor, which has a lower refresh rate of $\frac{1}{60}$ ms $\approx$ 16.6 ms, the differing latency figures between the connectivity methods are even less of a concern.

Attig et al. (2017) compiled a meta-review analyzing the previous total system latency guidelines in the field of human-computer interaction. It reevaluates the need for zero latency systems since, historically, latency guidelines did not consider latencies below 100 ms. The work distinguishes between zero-, first-, and second-order tasks. In zero-order tasks, a position change of the input device directly correlates with a position change on the output, while first-order tasks change the velocity on the output device and second-order tasks correlate input with acceleration at the screen. With the increase of the order of the task, the difficulty and sensitivity to latency increases. The HID protocol supports arbitrary data, and can thereby accommodate different classes of input devices with focus on different order tasks. Therefore, a generic BLE HID adapter has to consider latency requirements for the strictest second-order task. Attig et al. conclude:

> Users are indeed able to perceive latencies down to single milliseconds in specific tasks. Moreover, performance in zero-order and more demanding second-order tasks already gets impaired by latencies between 16-60 ms. ([1, p. 11])

Thus, the adapter adding 3.54 ms might be perceptible to humans in demanding situations, but will not noticeably impair performance. Considering the adapter enables the use of a whole class of keyboards for the constrained environments discussed previously, this downside can be considered negligible.

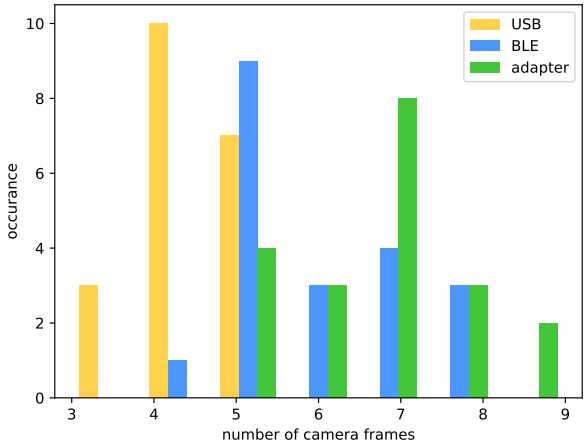


Figure 5.3: Histogram of the latency measurement results

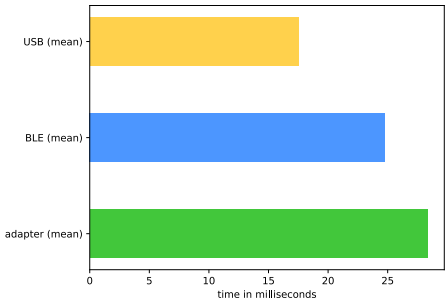


Figure 5.4: Mean latency of each connectivity method

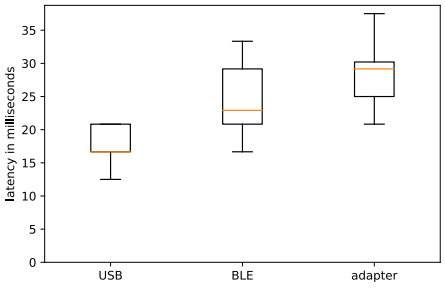


Figure 5.5: Boxplot of the latency measurement results

6. Discussion

The prototype succeeds in demonstrating the practicability of a generic BLE HID adapter. Three aspects of compatibility are considered: How many keyboards are supported, how many MCU boards can be targeted, and how many constrained environments can be accommodated. The adpater is shown to work with two BLE keyboards, the Nice!60 with a modified version of the ZMK firmware, and the Microsoft Bluetooth Keyboard. Only keyboards implementing the BLE Boot Protocol are currently supported. In theory, this should not be a big limitation, since the BLE HID specification declares Boot Protocol support mandatory for keyboard-type devices, but the example of the stock ZMK firmware shows the existence of keyboards that do not conform to this requirement. These keyboards are currently not supported by the prototype.

The source code was compiled and tested on two separate target MCU boards, the NRF52840MDK from Makerdiary and the NRF52840-dongle from Nordic Semiconductors. The architectural decision to use Zephyr RTOS as the basis for the implementation should allow for broad compatibility with other MCUs, but this was not further investigated.

Lastly, the compatibility with host environments was tested extensively. Every constrained environment tested here was demonstrated to be compatible with the adapter. This was to be expected since the adapter implements a USB Boot Protocol keyboard, a common use case for USB HID host implementations. Only implementing the Boot Protocol does impose the limitation of only being able to press 6 separate keys concurrently. This limitation might take effect in usage scenarios such as interactive entertainment, especially if the keyboard is used by two players simultaneously.

Meanwhile, the latency of the adapter is noticeably higher than a wired USB connection, which is to be expected, since the latency increase from USB to native BLE is also perceivable. Using two methods of connectivity combined, as is the case with the adapter, does add additional latency. In highly interactive tasks, this latency might be noticeable but does not impair the user's ability to complete the task. The downside of noticeable latency is offset by the additional usefulness of wireless keyboards in environments previously unable to connect with these.

6.1 Future Work

The implementation could be improved in smaller but significant ways with additional development time. Currently, bonding and reconnecting after the MCU board was powered down is not implemented. This means the keyboard tries to reestablish the previous connection, but the MCU board does not have the necessary data. The keyboard needs to be reset to forget the connection, and begin the pairing process anew. This can be circumvented by not powering down the MCU board, since desktop computers can power USB slots while powered down themselves. Accepting the connection should also be done by the user but the prototype does not support this (as noted already in subsection 4.2.1). The MCU boards typically allow for this by including physical buttons on them, meaning the implementation of this feature should be simple; it was only omitted from the prototype due to its lower priority (stemming from the research questions).

As was pointed out, for both communication partners, only the Boot Protocol is supported. The more capable Report Protocol is currently not supported and supporting it would necessitate bigger changes. The simplest addition would be that of adding Report Protocol keyboard support for those constrained environments that already support the USB Report Protocol. The adapter would only need to copy the keyboard's Report Descriptor and pass through the HID data unchanged. Meanwhile, supporting Report Protocol keyboards for constrained environments that only support the Boot Protocol would necessitate mapping the Report Protocol to the Boot Protocol. This would be an inherently lossy conversion, since the Boot Protocol imposes strict limitations, and would require the adapter to implement a HID Parser. In addition to the work needed to implement such a Parser, this step might introduce further latency.

The first step for improving the evaluation of latency is gathering more data. This might be improved by automating the manual step of detecting the video frames that contain the latency information. Furthermore, the current setup has limited timing precision through the capabilities of the camera and monitor hardware. This might be improved through the usage of more capable hardware, or changing to another method of measuring latency. Comparing the BLE keyboard observation to Bluetooth Classic keyboards would allow further insight into the user's acceptance of the adapter's latency since the Bluetooth Classic technology has greater latency. Adding Bluetooth Classic keyboards into the evaluation would introduce another variable, since this would require another keyboard with its inherent latency. This difference would be hard to account for in the current setup.

Despite the advantages of BLE for input devices, there are Bluetooth Classic keyboards in use that would benefit from a similar adapter for Bluetooth Classic. Broadening the compatibility to include these would require further development. Improving the evaluation of the current level of BLE keyboard compatibility could be easily achieved by acquiring more BLE keyboard hardware. Similarly, the portability of the project could be evaluated by acquiring further MCU boards to test. Many of the boards fulfilling the necessary hardware requirements of USB and BLE should already be supported, as the decision to use an RTOS was made with exactly this portability goal in mind. Were large amounts of hardware to be tested, automated and repeatable test environments would need to exist.

7. Conclusion

This thesis discussed a project which implements a generic BLE to USB HID adapter for keyboard devices with Boot Protocol mode support. This final chapter will summarize the answers found to the research questions posed in chapter 3.

To what extent is the *implementation* of a BLE to USB HID keyboard adapter feasible?

Despite the limitations discussed, the prototype has proven reliable and it successfully demonstrates the feasibility of such an adapter. Further, the limitations are undoubtedly solvable with additional development work.

How much *compatibility* can the adapter reasonably achieve?

The aspect of BLE keyboard compatibility was tested with two keyboards: the Microsoft Bluetooth Keyboard and the ZMK firmware in the form of the Nice!60. Since the keyboard compatibility relies on the promise of the BLE HID specification that every BLE keyboard device needs to support Boot Protocol operation, in theory, every BLE keyboard is supported by the adapter. In practice, ZMK is a counter-example, showing not every keyboard does abide by the specification. To increase keyboard compatibility further, a Report Protocol mode implementation would be necessary.

While portability was tested on two MCU boards, the NRF52840MDK and NRF52840-dongle, the project relies on the board support of Zephyr. This means that every Board with BLE hardware, USB host capability and Zephyr support is usable in theory. This can only be tested by acquiring more hardware, which is out of the scope of this work.

Constrained environment compatibility was more extensively evaluated. It was tested on the major desktop operating systems, Windows, MacOS, Linux, and OpenBSD. Mobile devices with USB support, such as iOS and Android, were also tested and compatible. The main target of desktop computers, namely BIOS/UEFI and bootloader, were also tested and found compatible. However, the BIOS/UEFI support is not easily generalizable for the different BIOS/UEFI software of other devices.

Does the usage of this adapter noticeably increase the *input latency*?

The latency between USB, a native BLE connection, and the adapter were compared. Changing from a wired connection to a wireless one adds a mean latency of 7.29 ms, while introducing the adapter adds an additional 3.54 ms. While this is, albeit barely, a humanly noticeable duration, it does not impair function, even in the most interactive higher-order tasks.

In conclusion, the prototype fulfills all requirements and is a foundation for further development. It has already proven useful in practice, for example, to enter the decryption password in encrypted Linux machines or to interface with the BIOS/UEFI. To allow others to benefit from its utility, and to accept public contributions, the project will be shared as an open-source project.

A. Appendix

Listing A.1: The Report Descriptor for a Report equal to the Boot Protocol

```
1  /**
2   * @brief BOOT HID keyboard report descriptor.
3   */
4  #define HID_BOOT_KEYBOARD_REPORT_DESC() {                    \
5      HID_USAGE_PAGE(HID_USAGE_GEN_DESKTOP),                   \
6      HID_USAGE(HID_USAGE_GEN_DESKTOP_KEYBOARD),               \
7      HID_COLLECTION(HID_COLLECTION_APPLICATION),              \
8          HID_REPORT_SIZE(1),                                  \
9          HID_REPORT_COUNT(8),                                 \
10         HID_USAGE_PAGE(HID_USAGE_GEN_DESKTOP_KEYPAD),        \
11         HID_USAGE_MIN8(224),                                 \
12         HID_USAGE_MAX8(231),                                 \
13         HID_LOGICAL_MIN8(0),                                 \
14         HID_LOGICAL_MAX8(1),                                 \
15         HID_INPUT(0x02),                                     \
16         HID_REPORT_COUNT(1),                                 \
17         HID_REPORT_SIZE(8),                                  \
18         HID_INPUT(0x03),                                     \
19         HID_REPORT_COUNT(6),                                 \
20         HID_REPORT_SIZE(8),                                  \
21         HID_LOGICAL_MIN8(0),                                 \
22         HID_LOGICAL_MAX8(255),                               \
23         HID_USAGE_MIN8(0),                                   \
24         HID_USAGE_MAX8(255),                                 \
25         HID_INPUT(0x00),                                     \
26     HID_END_COLLECTION,                                      \
27 }
```

USB	BLE	adapter
5	6	8
5	7	8
4	8	6
6	8	8
5	7	6
6	8	9
5	6	7
5	9	8
6	5	9
5	6	9
4	6	10
5	7	8
6	9	7
5	6	8
6	6	6
6	6	8
5	9	7
5	8	8
6	6	10
4	6	6

Table A.1: Raw latency measurement data. Each cell contains the total frame count, subtraction of 1 is necessary to obtain the latency figure.

Bibliography

[1] Attig, C., Rauh, N., Franke, T., and Krems, J. F. System latency guidelines then and now – is zero latency really considered necessary? In *Engineering Psychology and Cognitive Ergonomics: Cognition and Design* (Cham, 2017), D. Harris, Ed., Springer International Publishing, pp. 3–14.

[2] Axelson, J. *USB Complete: The Developer's Guide*, 5 ed. Complete Guides series. Lakeview Research, 2015.

[3] Bakni, M. Illustration of computer network types by spatial scope. https://en.wikipedia.org/wiki/File:Data_Networks_classification_by_spatial_scope.svg, 2018. accessed on 16.08.2022.

[4] Bluetooth SIG. HID over GATT Profile Specification. https://www.bluetooth.com/specifications/specs/hid-over-gatt-profile-1-0, 2011. Revision V10r00.

[5] Bluetooth SIG. Human Interface Device Profile 1.1. https://www.bluetooth.com/specifications/specs/human-interface-device-profile-1-1-1, 2015. Revision v1.1.1.

[6] Bluetooth SIG. Bluetooth Core Specification. https://www.bluetooth.com/specifications/specs/core-specification-5-3/, 2021. Revision v5.3.

[7] Gratton, D. A. *The Handbook of Personal Area Networking Technologies and Protocols*. Cambridge University Press, 2013.

[8] Logitech. High-performance wireless technology lightspeed. https://www.logitechg.com/en-us/innovation/lightspeed.html, 2002. accessed on 04.02.2022.

[9] Logitech. What is Unifiying? https://www.logitech.com/en-us/resource-center/what-is-unifying.html, 2022. accessed on 24.03.2022.

[10] makerdiary. nrf52840-MDK. https://wiki.makerdiary.com/nrf52840-mdk/, 2019. accessed on 24.03.2022.

[11] Microsoft. Microsoft Bluetooth Keyboard. https://www.microsoft.com/de-de/d/microsoft-bluetooth-keyboard, 2022. accessed on 12.09.2022.

[12] Milos634. Comparison of usb connectors. https://commons.wikimedia.org/wiki/File:USB_2.0_and_3.0_connectors.png, 2015. accessed on 24.08.2022.

[13] Nordic Semiconductor. nRF52840. ttps://www.nordicsemi.com/products/nrf52840, 2022. accessed on 12.09.2022.

[14] Nordic Semiconductor. nRF52840 Dongle. https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dongle, 2022. accessed on 12.09.2022.

[15] Siep, T., Gifford, I., Braley, R., and Heile, R. Paving the way for personal area network standards: an overview of the ieee p802.15 working group for wireless personal area networks. *IEEE Personal Communications 7*, 1 (2000), 37–43.

[16] Tanenbaum, A. S., and Bos, H. *Modern Operating Systems*, 4 ed. Pearson, Boston, MA, 2014.

[17] Ted Unangst. File removed. http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/sys/netbt/Attic/bluetooth.h, 2014. accssed on 25.01.2023.

[18] The Linux Foundation. https://linuxfoundation.org/, 2022. accessed on 12.09.2022.

[19] USB-Implementers' Forum. Device Class Definition for Human Interface Devices (HID). https://www.usb.org/document-library/device-class-definition-hid-111, 1996-2001.

[20] Zephyr Project. https://zephyrproject.org/, 2022. accessed on 24.03.2022.

[21] ZMK Firmware. https://zmk.dev/, 2022. accessed on 24.03.2022.