

BACHELOR PAPER

Thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Electronics and Business Distance Study (BEW-DL)

Wireless Button for FABI

By: Friedrich König, MSc
Student Number: 2210255002

Supervisor 1: Alijia Sabic, MSc

Munich, 05.05.2025

Declaration of Authenticity

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz/ Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and according to the rules currently applicable at the UAS Technikum Wien and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.”

Munich, 05.05.2025

Place, Date

Digital Signature

Kurzfassung

Diese Arbeit untersucht die Verbesserung assistiver Technologien durch die Einführung drahtloser Konnektivität in das Flexible Assistive Button Interface (FABI). Traditionell mit kabelgebundenen Verbindungen implementiert, schränkt FABI die Flexibilität und Anpassungsfähigkeit der Benutzer ein. Um diese Einschränkungen zu überwinden, integriert das vorgestellte System Bluetooth-Konnektivität über einen Raspberry Pi Pico W, der als Bluetooth Low Energy (BLE) Central-Gerät fungiert, und einen Puck.js, der als BLE-Peripheriegerät dient. Die Lösung ist in zwei Betriebsmodi gegliedert: einen Wi-Fi-Konfigurationsmodus, in dem eine webbasierte Benutzeroberfläche den Nutzern die individuelle Anpassung der Tastenaktionen ermöglicht, und einen BLE-Betriebsmodus, der drahtlose Tastenanschläge in spezifische Befehle über die USB Human Interface Device (HID)-Emulation umwandelt. Benutzerbewertungen zeigten die erfolgreiche Umsetzung zentraler Aktionen wie die Simulation von Tastaturanschlägen und die Ausgabe von Text, was die Machbarkeit des Ansatzes bestätigt. Trotz einiger Einschränkungen - wie der Unterstützung lediglich einer einzelnen Tastenverbindung und der grundlegenden Eingabedetektion - bietet das drahtlose Tasteninterface vielversprechende Fortschritte für zugängliche und anpassbare assistive Technologien. Zukünftige Arbeiten sollen diese Herausforderungen adressieren und die Skalierbarkeit sowie Funktionalität des Systems weiter verbessern.

Schlagwörter: Puck.js, Raspberry Pi Pico W, FABI, Assistive Technologies

Abstract

This thesis investigates the enhancement of assistive technologies by introducing wireless connectivity into the Flexible Assistive Button Interface (FABI). Traditionally implemented with wired connections, FABI limits user flexibility and adaptability. To overcome these constraints, the proposed system integrates Bluetooth connectivity through a Raspberry Pi Pico W acting as a Bluetooth Low Energy (BLE) central device and a Puck.js serving as the BLE peripheral. The solution is structured into two operational phases: a Wi-Fi configuration mode, where a web - based interface enables users to customize button actions, and a BLE operation mode that translates wireless button presses into specific commands via USB Human Interface Device (HID) emulation. User evaluations demonstrated successful execution of key actions such as keystroke simulations and text outputs, confirming the feasibility of the approach. Despite limitations - such as support for only a single button connection and basic input recognition - the wireless button interface offers promising advancements for accessible, customizable assistive technology. Future work is suggested to address these challenges and further enhance system scalability and functionality.

Schlagwörter: Puck.js, Raspberry Pi Pico W, FABI, Assistive Technologies

Table of Contents

1	Introduction	6
2	State of the Art of Alternative Input Devices	7
2.1	Research	7
2.2	Commercially available Alternative Input Devices	7
2.3	Open-Source Alternative Input Devices	8
3	Requirements	9
3.1	Non-functional Requirements	9
3.1.1	Raspberry Pi Pico W	9
3.1.2	Puck.js	9
3.1.3	C++ and Arduino IDE	9
3.1.4	JavaScript and Espruino IDE	10
3.2	Functional Requirements	10
3.2.1	User Interface	10
3.2.2	Connection of Puck.js	10
3.2.3	Button Presses	11
4	Design	11
4.1	FABI	11
4.2	Microcontrollers	11
4.2.1	Raspberry Pi Pico W	11
4.2.2	Arduino Nano RP2040 Connect	12
4.2.3	Raspberry Pi Pico 2 W	12
4.2.4	Puck.js	12
4.3	Earle Philhower Core: Arduino-Pico (by Earle F. Philhower, III).	13
4.4	Hardware Design	13
4.4.1.1	Wi-Fi Mode	13
4.4.1.2	BLE Mode	14
4.5	Software Design	14
5	Implementation	17
5.1	Puck.js	17
5.2	Raspberry Pi Pico W	18

5.2.1	Setup	19
5.2.2	Wi-Fi User Interface.....	20
5.2.3	BLE	20
5.2.4	Button Mapping	21
5.2.5	Key Presses	22
5.2.6	Main Loop.....	22
6	Evaluation.....	23
6.1	User Test.....	23
6.1.1	Connecting to the UI.....	23
6.1.2	Button Action “KEY_ENTER”	24
6.1.3	Button Action “Hello World”	24
6.2	Comparing the Project to the Requirements	25
6.2.1	Nonfunctional Requirements	25
6.2.2	Functional Requirements.....	25
6.2.2.1	User Interface	25
6.2.2.2	Connection of Puck.js.....	25
6.2.2.3	Button Presses	26
6.3	Limitations	26
6.3.1	One Button vs Multiple Buttons	26
6.3.2	Wi-Fi vs BLE	26
6.3.3	BLE Central vs BLE Peripheral	26
7	Conclusion.....	27
	Bibliography.....	28
	List of Figures.....	32
	List of Listings.....	33
	List of Abbreviations	34
	Documentation table of AI-based tools	35
	Appendix A: Code for Puck.js.....	36
	Appendix B: Code for Raspberry Pico W	37

Appendix C: Code for Arduino Nano RP2040 Connect	46
Appendix D: Code for Puck.js (for the Connection with Arduino Nano RP2040 Connect).....	52
Appendix E: Mermaid Code for the Flowchart of Figure 3	53
Appendix F: Link to the Github Repository.....	54

1 Introduction

Assistive user interface technologies play a crucial role in empowering individuals with disabilities or special needs by providing them with alternative ways to interact with digital systems and environments. These technologies span a broad spectrum - from software solutions such as speech recognition and screen reading systems, which translate digital content into audible or tactile feedback, to innovative hardware devices that employ eye tracking, head movement detection, or switch-based interfaces. Both open source and commercial solutions contribute significantly to this field, offering varying advantages in terms of flexibility, cost, and technical support.

In recent years, there has been a growing recognition of the importance of modular, customizable input devices that can be tailored to the unique requirements of each user. Commercial offerings, like the Xbox Adaptive Controller [1] and Microsoft Adaptive Hub [2], have set new benchmarks by providing highly configurable platforms that allow users to connect a variety of external assistive peripherals. These systems exemplify how targeted design, and robust engineering can create interfaces that are both accessible and versatile. However, despite these advances in the commercial sector, open-source projects remain critical for fostering innovation and ensuring that assistive technologies are accessible to a wider community.

One such open-source project is Flexible Assistive Button Interface (FABI) [3], which has been developed with the goal of creating an accessible and customizable user interface. While FABI has proven to be a valuable tool in the assistive technology landscape, it currently lacks native Bluetooth connectivity - a feature that is increasingly essential for seamless integration with modern devices and wireless peripherals. FABI can connect via Bluetooth by installing an additional board, but the buttons itself still need to be wired then. Bluetooth connectivity offers the advantage of cable-free operation, greater mobility, and the potential to connect multiple devices simultaneously, all of which can significantly enhance the user experience for individuals with motor impairments.

This thesis addresses this gap by designing and prototyping a system that incorporates Bluetooth-connected buttons, that may later be implemented into the FABI device. The aim is to combine the open-source flexibility of FABI with the convenience and modern connectivity offered by Bluetooth technology. By adding this functionality, the project seeks to improve the adaptability of assistive interfaces, enabling users to interact with digital environments more intuitively and efficiently. In doing so, the work contributes to the advancement of open-source assistive technologies and enhancing inclusivity and independence for users with disabilities.

2 State of the Art of Alternative Input Devices

Alternative input devices, play a pivotal role in creating inclusive digital environments for individuals with limited motor abilities. These devices are engineered to overcome the challenges posed by traditional input methods - such as keyboards, mice, or standard game controllers - by offering customizable and modular interfaces that adapt to a wide range of user needs. This chapter examines both commercial and open-source solutions in this domain.

2.1 Research

Alternative input devices, when used by non-disabled users, can enhance the user's entertainment when playing video games [4]. However, for disabled users, it may become necessary to operate computers or consoles via alternative input devices, as their motor abilities may be impeded [5]. There are a wide range of possible alternative input devices. Garcia et al. [6] developed an alternative input device "that converts discrete breathing patterns into pre-defined words to help paralysis victims communicate their needs" [6, p. 1]. Similar to that, Fatnani et al. [7] developed a device, that registers the user's blows and executes predefined actions. Users with more motor abilities may use a system designed by Chen et al. [8] controlling a computer mouse with their head movements.

2.2 Commercially available Alternative Input Devices

Users, that are less motorically impeded may even use commercially available alternative user input devices such as the Microsoft Adaptive Hub [2]. This plug and play solution may be configured according to the user's demand [2]. Input devices such as joysticks or buttons are available and may be connected to a computer via Universal Serial Bus (USB) or Bluetooth and may execute user defined actions [2].

Similarly, the Xbox Adaptive Controller is commercially available to control the video game console Xbox for motorically impeded users [1]. This device also acts as a hub, that can be extended by connecting buttons, joysticks or other input devices, that the user can operate [1]. Connections can be established via USB, 3.5mm-ports or Bluetooth [1].

2.3 Open-Source Alternative Input Devices

Alongside commercial offerings, the open-source community has been actively developing alternative input devices. Open-source projects harness the flexibility of platforms like Arduino and Raspberry Pi to create Do-it-yourself (DIY) adaptive controllers that are both affordable and customizable, such as the FLipMouse [9, 10]. The FLipMouse is an open-source device, that lets users control a computer mouse via a very sensitive joystick, while the click action is performed via a sip-puff sensor [9, 10].

Another open-source alternative input device by the the AsTeRICS Foundation is FABI [3]. According to the AsTeRICS Foundation, “[t]he FABI (Flexible Assistive Button Interface) is an interface box which allows the connection of up to 9 momentary switches (buttons) or self-made electric contacts to a computer, tablet or smartphone” [3]. The user may specify the actions of each button in a “web-based config manager” [11, 12]. Buttons are either “momentary switches or self-made electrical contacts” [11] or “sip-puff sensor[s]” [11], that have to be connected via wire to the FABI system. The FABI itself may either communicate via USB to the target computer, or utilize a Bluetooth connection, if the user attaches a Bluetooth module to the FABI. The FABI system is an open-source project, wherefore it is possible to build a device oneself, using the provided building guides and user manual [13 - 16]. The main processing unit of the FABI system is an Arduino Pro Micro microcontroller, that is based on the ATmega32U4 chip [17], according to SparkFun Electronics [18], a vendor of electronic components, whose Pro Micro board is recommended by the AsTeRICS Foundation in the construction of a FABI system. The Pro Micro board has been chosen, as it supports the USB Human Interface Design (HID) protocol, meaning the device can act as an interface device, such as a mouse or a keyboard, when connected via an USB port [14]. It features 12 digital I/O pins, and is “[s]upported unter Arduino IDE v1.0.1” [18]. However, the Pro Micro board does not feature an onboard Bluetooth or Wireless-Fidelity (Wi-Fi) chip, wherefore the FABI currently relies on an external Bluetooth module for wireless connections. To make the external Bluetooth module obsolete, a different board will be chosen for this project. The following chapters will explore the requirements, that the project will need to fulfill.

3 Requirements

This chapter will give an overview of the requirements of this project, i.e. which features need to be implemented. First, the non-functional requirements, such as the hardware on which the project must run will be stated. Afterwards, the functional requirements will be discussed.

3.1 Non-functional Requirements

As this project features BLE, the required hardware features both an BLE central and a BLE peripheral device. Both will be stated in the next parts.

3.1.1 Raspberry Pi Pico W

To provide an update to FABI, the main processing board and the BLE central device has to be the Raspberry Pi Pico W, which will be introduced in Chapter 4.2.1. While the Raspberry Pico 2 W is an updated version, it is rather more expensive and does not provide any additional features, that are required for this project. While not explicitly testing for compatibility, it is assumed, that the project will work on the Raspberry Pi Pico 2 W, as well as the Philhower Core [19 - 20] explicitly also covers the board, and the wireless communication chip has not changed from the predecessor.

3.1.2 Puck.js

The BLE peripheral device, that recognizes the user induced button presses has to be the Espruino Puck.js, which will be introduced in depth in Chapter 4.2.4. As stated before, that device fulfills all needs of this project, as it features an onboard push-button and is able to establish a BLE connection.

3.1.3 C++ and Arduino IDE

After having introduced the required hardware, next the required software will be introduced. This includes both the programming language, as well as the Integrated Development Environments (IDEs).

The software, that will run on the Raspberry Pi Pico W, has to be programmed in C++ using the Arduino IDE, analogous to the original FABI project. The required core is the Philhower Core, that will be introduced in Chapter 4.3. This implies, that the included libraries in the Philhower Core have to be used as well.

3.1.4 JavaScript and Espruino IDE

The Puck.js has to be programmed in JavaScript using the Espruino IDE. The Espruino IDE is a web-based IDE, that uses Web BLE to connect to the Puck.js [21 - 23]. It is therefore not needed to connect the Puck.js via cable to a computer nor the BLE Central device.

3.2 Functional Requirements

This section outlines the key functional requirements (FR) that underpin the interactive nature of the system. These features include the user interface, the connection mechanism for the Puck.js board, and the recognition of button presses and button actions. Each of these elements contributes to a seamless and responsive user experience.

3.2.1 User Interface

The user interface (UI) must be designed to be intuitive and accessible, ensuring that users can easily interact with the system. Its core elements include:

- **FR-1 Intuitive Layout:** The UI must be organized in a clear and structured manner, presenting essential information and controls prominently. This design approach minimizes user confusion and enhances overall usability.
- **FR-2 Immediate Feedback:** Dynamic visual indicators provide immediate feedback on settings such as the number of Puck.js devices to be connected and the button actions. This real-time interaction helps users understand the current operation and any changes that occur.
- **FR-3 Customizable Elements:** The interface can be adapted to suit individual user needs. Whether through configurable controls or adaptable display options, the UI offers flexibility in how information is presented and interacted with.

3.2.2 Connection of Puck.js

A critical functional aspect of the system is its ability to establish a reliable connection with at least one Puck.js board. The connection process involves:

- **FR-4 BLE:** The Puck.js is connected via BLE to the BLE Central device, i.e. a Raspberry Pi Pico W.
- **FR-5 Automated Discovery and Pairing:** The system scans for nearby Puck.js devices and initiates a secure pairing process automatically. This ensures a quick and efficient connection without requiring manual intervention.
- **FR-6 Connection Monitoring:** Once connected, the system continuously monitors the connection status. Any disconnects have to restart the discovery and pairing process.

3.2.3 Button Presses

User interaction through physical button inputs on the Puck.js are the core component of the system's functionality. The system handles these interactions by:

- **FR-7 Accurate Button Detection:** The software distinguishes between various types of button presses - such as single taps, double taps, and long presses - to ensure that each interaction is correctly interpreted.
- **FR-8 Action Mapping:** Each type of button press is linked to a specific function or command, that the user configured in the UI.
- **FR-9 Feedback Mechanisms:** Upon detecting a button action, the system provides immediate feedback (visual or auditory) to confirm that the input has been received and processed. This helps users feel confident in their interactions.
- **FR-10 Robust Debouncing:** To prevent unintended multiple triggers from a single press, a debouncing mechanism has to be implemented. This ensures that only deliberate and distinct button actions are registered, improving overall system reliability.

4 Design

Before implementing any ideas, it is important to gain an understanding of the used hardware and software. This chapter explains the design of the project. First, the hardware design and necessary connections will be explained. In a second step, the software design is shown.

4.1 FABI

As explained in chapter 2.3, the current version of FABI utilizes a Pro Micro board, which does not feature an onboard Bluetooth or Wi-Fi chip. Therefore, the FABI currently relies on an external Bluetooth module for wireless connections. To make the external Bluetooth module obsolete, a different board will be chosen for this project. Hence, the following chapter explores the available options for alternative boards.

4.2 Microcontrollers

An alternative to the Pro Micro board has to be compatible with Arduino IDE, support the USB HID protocol and needs to have digital I/O pins. One such candidate is the RP2040 chip developed and produced by Raspberry Pi Ltd [24], that is the basis of microcontrollers such as the Raspberry Pi Pico W or the Arduino Nano RP2040 Connect [25].

4.2.1 Raspberry Pi Pico W

The RP2040 present on the Raspberry Pi Pico W board features a dual-core ARM Cortex-M0+ processor running at up to 133 MHz [26], significantly outperforming the ATmega32U4, which

has a single-core 8-bit AVR processor operating at 16 MHz [17]. This increased clock speed and 32-bit architecture allow the RP2040 to handle more computationally intensive tasks efficiently. Both chips support native USB support without needing any additional hardware [8, 10]. While the Pro Micro costs \$12.50 from the vendor SparkFun [18] at the time of writing, the Raspberry Pi Pico W is significantly cheaper priced at 6.70€ (\$7.04) from the official vendor BerryBase GmbH [27]. The key advantage of the Raspberry Pi Pico W over the Arduino Pro Micro is the inclusion of the Infineon CYW43439 [28]. This chip supports “Wi-Fi 4 (802.11n), Single-band (2.4 GHz)” [28], as well as Bluetooth 5.4 [28], which both are necessary for the present project.

4.2.2 Arduino Nano RP2040 Connect

The Arduino Nano RP2040 Connect features the same RP2040 chip as the aforementioned Raspberry Pi Pico W. The main differences are the replacement of the CYW43439 chip by the u-blox Nina W102 module, which also provides single-band 2.5GHz Wi-Fi operation and supports Bluetooth 4.2 [25, 29]. It is currently priced at 30.74€ (\$31.35), meaning it is significantly more expensive than the Raspberry Pi Pico W. It is more advanced technically speaking, as it features various additional sensors, such as a gyroscope, microphone or a cryptographic co-processor, that are not relevant for this project [25].

4.2.3 Raspberry Pi Pico 2 W

In August of 2024 Raspberry Pi Ltd announced a successor to the Raspberry Pi Pico W, the Raspberry Pico 2 W, that is based on the enhanced RP2350 chip [30 - 32]. While the new board is more powerful than its predecessor, featuring a dual core ARM Cortex-M33 processor running at up to 150MHz [32], the same CYW43439 is present on the board, leading to no changes in wireless connectivity. The price of the Raspberry Pi Pico 2 W is given at 7.90€ (\$8.30) by the official vendor BerryBase GmbH [33].

4.2.4 Puck.js

The Puck.js is a compact, energy-efficient Bluetooth Low Energy (BLE) development board designed for wireless applications. It is based on the Nordic Semiconductor nRF52832 chipset [34], featuring an ARM Cortex-M4 processor and integrated radio communication capabilities [23]. The onboard nRF52832 chipset supports BLE 5.0, allowing for reliable data transmission and interaction with other BLE-enabled devices such as smartphones, computers, and embedded systems [23, 34, 35]. The Puck.js board is equipped with various built-in sensors and features, most importantly for this project a push-button for user interaction [35]. The Puck.js is designed for long battery life, operating on a standard coin-cell battery (CR2032) for extended periods [35]. Unlike many microcontroller boards that require compiled firmware development, Puck.js can be programmed using JavaScript through the Espruino interpreter, which

makes development highly accessible, as code can be written and executed directly from a web browser via Web Bluetooth [23]. It currently retails for £32.40 (\$40.88) in the official Espruino webshop [35].

4.3 Earle Philhower Core: Arduino-Pico (by Earle F. Philhower, III).

The Earle Philhower Core is an Arduino-compatible core designed specifically for the RP2040 and RP2350 families of microcontrollers [19]. By bridging the gap between the robust features of the RP2040 & RP2350 and the ease-of-use provided by the Arduino ecosystem, this core allows developers to write, compile, and deploy code using the Arduino IDE [19].

The core is designed to be compatible with a wide range of existing Arduino libraries [20]. This compatibility minimizes the need for extensive code rewrites and allows for the quick migration of projects from other Arduino boards, such as the Arduino Pro Micro, that is used in the FABI system. The Earle Philhower Core provides many Software Development Kits (SDKs), that are needed in this project, such as an SDK for BLE and Wi-Fi, that are used in this project [20]. Additionally, Philhower [19] provides many detailed examples on how to use the SDKs on his GitHub page.

4.4 Hardware Design

The CYW43439 chip is limited by providing “[s]imultaneous BT/WLAN reception with a single antenna” [36, p. 6]. While that does not mean it is impossible to host a web server while acting as a BLE Central device as the same, the author has not managed to get it to work. Therefore, the design was split into two phases. The Wi-Fi phase and the BLE phase.

4.4.1.1 Wi-Fi Mode

During the Wi-Fi mode, the Raspberry Pi Pico W needs to be connected via USB cable to user device, such as a PC or laptop. In this phase, the Raspberry Pi Pico W does not utilize the USB 1.1 protocol yet, as it only needs power from the user device. During the Wi-Fi mode, the Raspberry Pi Pico W hosts an Access Point (AP), to which the user device has to connect. A simplified schematic of the hardware connections is shown in figure 1.

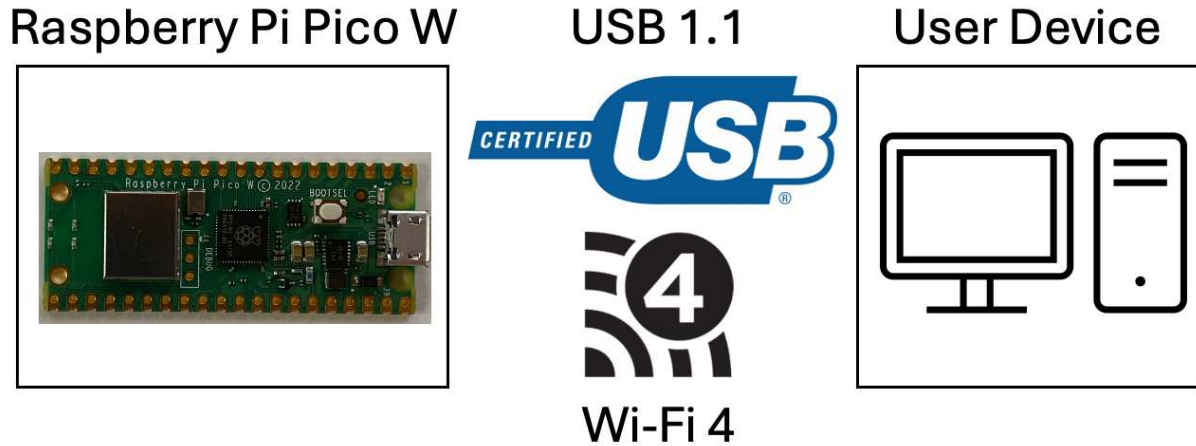


Figure 1: Hardware connections at the Wi-Fi phase (Source: Own depiction. Raspberry Pi Pico W: Own photograph. USB 1.1 logo: [37]. Wi-Fi 4 logo: [38])

4.4.1.2 BLE Mode

During the BLE mode, the Raspberry Pi Pico W has to stay connected to the user device via USB cable. It connects to at least one Puck.js via BLE. In the BLE mode the Raspberry Pi Pico W utilizes the USB 1.1 protocol, specifically the HID protocol [39], to send the respective button actions to the user device. A simplified schematic of the hardware connections is shown in figure 2.

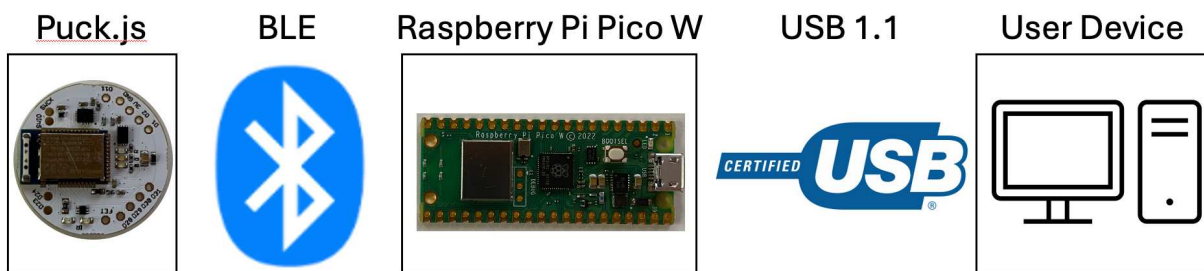


Figure 2: Hardware connections at the BLE phase (Source: Own depiction. Puck.js: Own photograph. BLE logo: [40]. Raspberry Pi Pico W: Own photograph. USB 1.1 logo: [37])

4.5 Software Design

The developed software runs on a Raspberry Pi Pico W and is designed to operate in two distinct modes: a Wi-Fi configuration mode and a Bluetooth Low Energy (BLE) operation mode. Its overall structure follows a state-based architecture where the system's behavior is determined by its operating current mode, ensuring a clear separation of concerns between user configuration and real-time operation. A simplified flowchart illustrating the software design is shown in figure 3.

At startup, the Raspberry Pi Pico W initializes its hardware interfaces - including Serial communication, USB Keyboard, and the built-in LED - and enters the Wi-Fi configuration mode. In this initial state, the device creates a Wi-Fi Access Point (AP) and hosts a web server that provides an HTML-based configuration interface. Through this interface, users can define parameters such as the maximum number of Bluetooth buttons (Puck.js devices) to be managed and assign a specific action to each button. When a configuration request is received via an Hypertext Transfer Protocol (HTTP) GET request, the software parses the query parameters, updates internal variables, and acknowledges the update. This design enables non-technical users to modify the device's behavior using a standard web browser. Additionally, a separate command on the web page allows the user to trigger the switch from Wi-Fi mode to BLE mode, effectively stopping the configuration mode and starting the operation mode.

Once the user saves the configuration and issues the command to start BLE operation, the software disables the Wi-Fi AP and transitions into BLE mode. In this mode, the software utilizes an event-driven architecture powered by the BTstack library (as integrated in the Philhower core [19, 41]) to scan for, connect to, and communicate with Puck.js devices that advertise a predefined BLE service. After the configuration is complete and the Wi-Fi connection is terminated to free system resources, the BTstack library is initialized with several callback functions that manage the entire BLE lifecycle. These include continuously scanning for advertisements from nearby Puck.js devices that broadcast a specific service Universally Unique Identifier (UUID), temporarily halting the scan and initiating a connection upon detecting a matching advertisement, discovering available services and identifying the service and characteristic associated with button notifications, and finally subscribing to notifications to receive events such as button presses. This modular approach ensures that BLE-related operations are managed asynchronously, with callbacks responding to events like connection successes, service discoveries, and incoming notifications.

The core functionality in BLE mode is driven by notifications received from connected Puck.js devices. When a button press is detected via a BLE notification, the software first determines which connected device generated the notification by referencing an internal array indexed by the connection order. It then retrieves the corresponding action that was configured during the Wi-Fi mode. If the action string begins with a special prefix (e.g., "KEY_"), the software interprets it as a keystroke command and simulates a key press through the USB HID interface; otherwise, it outputs the configured text using keyboard emulation. A brief Light Emitting Diode (LED) flash provides visual feedback that the action has been executed. This design allows for a flexible and dynamic mapping of button inputs to user-defined actions, supporting both standard text output and simulated keystrokes.

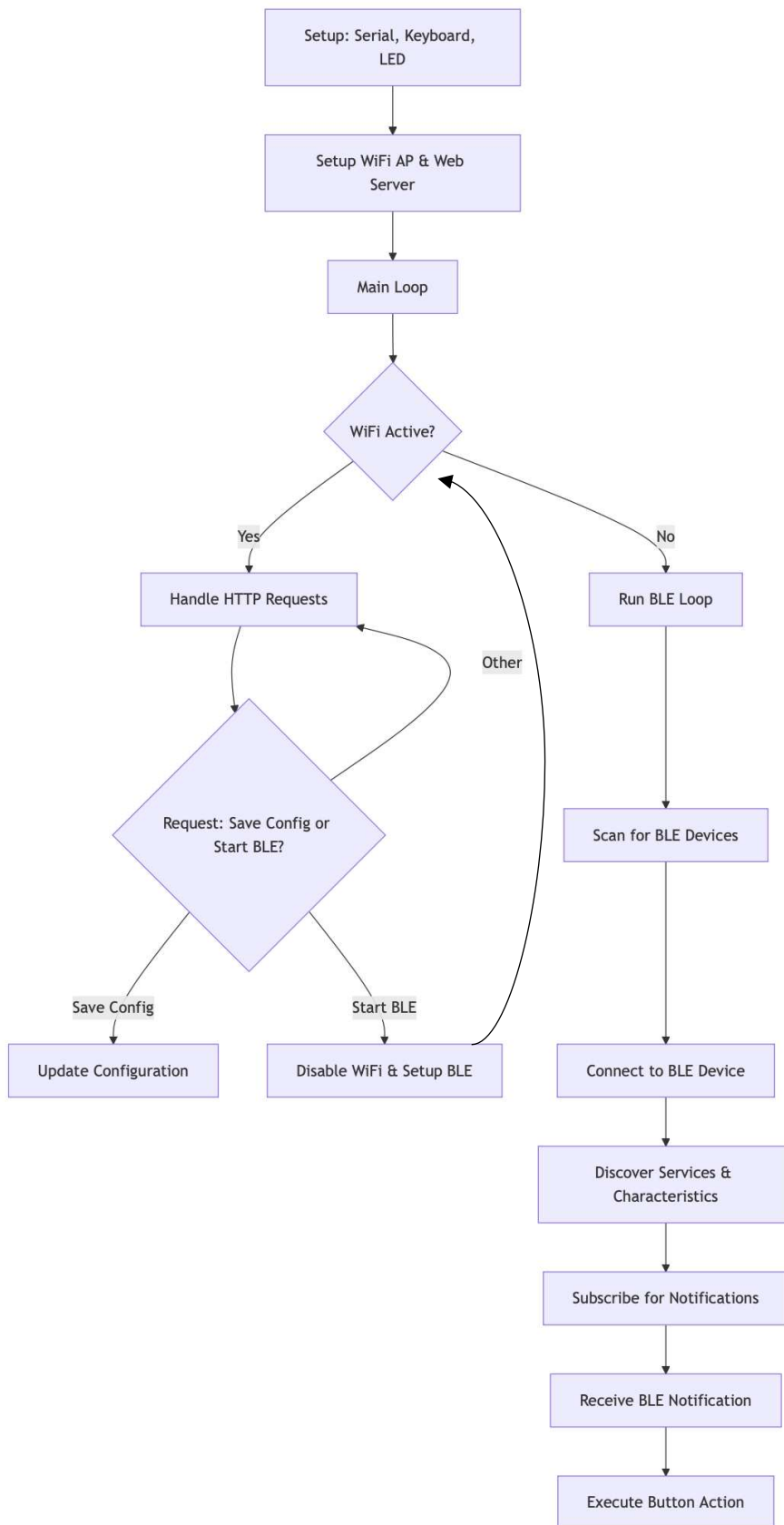


Figure 3: Simplified Flowchart (Source: Own depiction using mermaid.live. Code in Appendix E)

5 Implementation

After having introduced the hardware and software design of the project, this chapter delves into detail on how the design has been implemented in code. The code itself can be seen in appendices A and B, as well as in [42].

5.1 Puck.js

The Puck.js firmware implements a custom Bluetooth Low Energy (BLE) service to broadcast the state of a physical button. This implementation leverages the Espruino environment's built-in APIs to configure the BLE service, manage the BLE stack, and monitor hardware events in an event-driven manner.

At the beginning of the code, the UUIDs are defined as constants. The default Nordic UART UUIDs have been chosen [43], shown in listing 1.

```
1. const SERVICE_UUID = '6e400001-b5a3-f393-e0a9-e50e24dcca9e'; // Service UUID
2. const CHARACTERISTIC_UUID = '6e400003-b5a3-f393-e0a9-e50e24dcca9e'; // Characteristic UUID
```

Listing 1: Defining the UUIDs

The global variable, `buttonState`, is initialized to 0 to indicate that the button is released; it will later change to 1 when the button is pressed.

```
4. let buttonState = 0; // 0 = Released, 1 = Pressed
```

Listing 2: The variable `buttonState`

The core BLE functionality is established in the `setupBLE()` function shown in listing 3. This function uses `NRF.setServices()` to define a custom GATT service. Within the service definition, a characteristic is configured with the following properties:

- **Value:** Initialized with the current button state (`[buttonState]`).
- **Readable:** Enabled, so that BLE clients can read the characteristic.
- **Writable:** Disabled, preventing external devices from modifying the value.
- **Notify:** Enabled, which means that if notifications are sent, connected devices could receive updates when the characteristic value changes.

The service is advertised using the provided `SERVICE_UUID`, and the UART functionality is enabled (`uart: true`). A console message confirms that the BLE services have been set up and that advertising has commenced. To ensure robustness, a `restartBLE()` function is provided. This function logs a restart message and calls `NRF.restart()`. Upon restarting, the BLE services are reinitialized by calling `setupBLE()` again. This mechanism ensures that if any issues arise with the BLE stack, the device can recover without needing a complete power cycle.

```

7. function setupBLE() {
8.   // Set up GATT services and characteristics
9.   NRF.setServices({
10.    '6e400001-b5a3-f393-e0a9-e50e24dcca9e': {
11.      '6e400003-b5a3-f393-e0a9-e50e24dcca9e': {
12.        value: [buttonState], // Initial value
13.        readable: true, // Allow reading the value
14.        writable: false, // Disable writing to the characteristic
15.        notify: true // Enable notifications
16.      }
17.    }
18.  }, { advertise: [SERVICE_UUID], uart: false });
19.
20.  console.log("BLE services set up and advertising started.");
21. }

```

Listing 3: The setupBLE function

Button interactions are captured using two separate `setWatch()` calls shown in listing 4. One watch monitors for the rising edge of the button signal (indicating a press), and the other monitors for the falling edge (indicating a release). When the button is pressed, the corresponding callback sets `buttonState` to 1, logs the state to the console, and updates the onboard LED (LED1) to reflect the change. Conversely, when the button is released, the callback sets `buttonState` back to 0, logs the change, and updates the LED accordingly.

```

48. // Watch for button press
49. setWatch(() => {
50.   buttonState = 1;
51.   console.log(buttonState); //Raspi recognizes only this
52.   notifyButtonState();
53.   LED1.write(buttonState);
54. }, BTN, { edge: "rising", debounce: 50, repeat: true });
55.
56. // Watch for button release
57. setWatch(() => {
58.   buttonState = 0;
59.   console.log(buttonState); //Raspi recognizes only this
60.   notifyButtonState();
61.   LED1.write(buttonState);
62. }, BTN, { edge: "falling", debounce: 50, repeat: true });

```

Listing 4: The setWatch functions

It is important to note that while the BLE characteristic is configured with notifications enabled, the code that would actively send notifications upon a change in the button state (via a function such as `notifyButtonState()`) has been commented out. This is due to the fact, that testing has shown, that the Raspberry Pi Pico W will receive the values written to the console, but not any updated values from `notifyButtonState()`. This behavior is explained in detail in chapter 6.2.4.

5.2 Raspberry Pi Pico W

Next the implementation of the code on the Raspberry Pi Pico W is explained in detail.

5.2.1 Setup

Firstly, all used libraries need to be included as shown in listing 5.

```
13. #include <BTstackLib.h>
14. #include <SPI.h>
15. #include <Keyboard.h>
16. #include <WiFi.h>
```

Listing 5: Including the libraries

Afterwards all global variables and constants are initialized, such as the Service Set Identifier (SSID) and Password of the Wi-Fi AP, as well as a variable to save the maximal number of Puck.js devices to connect to, the UUIDs that are used for BLE connections, as shown in listing 6. Notable is the constant `*button_pressed`, which is set to "[J1 \n". Experimenting has shown, that button presses send the characters "[J1 \n" to the Raspberry Pi Pico W. Button releases send "[J0 \n".

```
17. // ----- USER-CONFIGURABLE WIFI CREDENTIALS -----
18. char ssid[] = "FABI";           // Access Point SSID
19. char pass[] = "asterics";       // Access Point password
20.
21. // ----- GLOBALS -----
22.
23. // Wi-Fi
24. bool wifiActive = true;         // Tracks if WiFi is active (AP mode)
25. int status; // = WL_IDLE_STATUS;
26. WiFiServer server(80);
27.
28. // Number of buttons to connect to
29. int maxButtons = 3;
30. String buttonActions[100]; // Action strings for each "button"
31.
32. // BLE (BTstack) Definitions
33. // The Puck.js UUIDs as 128-bit. BTstack uses a UUID class:
34. UUID puckServiceUUID("6e400001-b5a3-f393-e0a9-e50e24dcca9e");
35. UUID puckCharacteristicUUID("6e400003-b5a3-f393-e0a9-e50e24dcca9e");
36.
37. // We'll allow up to 100 discovered Puck.js devices
38. BLEDevice puckDevices[100];
39. bool puckConnected[100];        // Keep track of whether device is connected
40. BLEService puckServices[100];   // The service object for each device
41. BLECharacteristic puckChars[100]; // The characteristic object for each device
42. bool puckServiceFound[100];
43. bool puckCharFound[100];
44. String puckNames[100];          // Puck.js device name from advertisement
45.
46. int puckCount = 0;              // How many Puck.js have been connected so far
47. int currentConnectIndex = -1;   // Which array index is currently connecting
48.
49. // Forward declarations of our BLE callbacks:
50. void advertisementCallback(BLEAdvertisement *advertisement);
51. void deviceConnectedCallback(BLEStatus status, BLEDevice *device);
52. void deviceDisconnectedCallback(BLEDevice *device);
53. void gattServiceDiscovered(BLEStatus status, BLEDevice *device, BLEService *service);
54. void gattCharacteristicDiscovered(BLEStatus status, BLEDevice *device, BLECharacteristic
    *characteristic);
55. void gattSubscribedCallback(BLEStatus status, BLEDevice *device);
56. void gattCharacteristicNotification(BLEDevice *device, uint16_t value_handle, uint8_t *value,
    uint16_t length);
57. void gattReadCallback(BLEStatus status, BLEDevice *device, uint8_t *value, uint16_t length);
58. void gattWrittenCallback(BLEStatus status, BLEDevice *device);
```

```
60. const char *button_pressed = "[J1 \n"; //Used to determine a button press
61. bool currently_pressed = false;
```

Listing 6: Initializing the global variables

At startup, the software initializes the core hardware components on the Raspberry Pi Pico W. The `setup()` function is the entry point and is responsible for initializing Serial communication (set to a baud rate of 9600), the USB Keyboard interface, and the built-in LED (configured as an output). This initialization is critical to ensure that all subsequent operations - from debugging output over Serial to keyboard emulation and visual feedback via the LED - function correctly. After these initializations, the software immediately calls the Wi-Fi configuration routine. This early establishment of the Wi-Fi mode guarantees that the user is presented with a configuration interface as soon as the device powers up.

5.2.2 Wi-Fi User Interface

The Wi-Fi user interface is implemented by setting up the Pico W as a Wi-Fi Access Point (AP), utilizing the WiFi library implemented in the Philower Core [44]. The `setup_Wifi()` function configures the device with predefined credentials (SSID: "FABI" and password: "asterics") and starts a web server on port 80. This server hosts an HTML-based configuration page on the default IP address 192.168.42.1 that allows the user to set parameters such as the maximum number of Bluetooth buttons (Puck.js devices) and the action to be assigned to each button. When a client (typically a browser) connects to the AP, the `loop_Wifi()` function handles the HTTP request. The function reads the incoming request line by line, detecting the type of request sent by the user. Two primary request types are supported: one for saving configuration parameters (triggered by a "GET /save?" request) and another for transitioning the device from Wi-Fi mode to BLE mode (triggered by a "GET /startBLE" request).

Upon receiving a configuration request, the software calls `parseConfig()`, which extracts query parameters from the URL. It updates the internal state variables - for example, setting the maximum number of buttons and their associated actions - thereby allowing non-technical users to control the device's behavior without needing to modify the code directly. The design of the HTML form and the parsing logic are deliberately kept simple to streamline the configuration process and reduce the potential for user error.

5.2.3 BLE

The transition from Wi-Fi mode to BLE mode marks the shift from configuration to operation. Once the user triggers the BLE mode via the web interface, the `startFABI()` function is invoked. This function first disables the Wi-Fi Access and then calls `setup_BLE()` to initialize the BLE subsystem using the BTstack library (integrated in the Philower core) [41].

The BLE functionality is highly event-driven. The software registers several callback functions that manage the BLE lifecycle:

- **BLE Scanning:** The device continuously scans for BLE advertisements from nearby Puck.js devices. The `advertisementCallback()` function inspects each advertisement to determine if it contains the specific Puck.js service UUID 6e400001-b5a3-f393-e0a9-e50e24dcca9e.
- **Connection Establishment:** Upon detecting a matching advertisement, scanning is halted temporarily, and a connection is attempted using a timeout mechanism. The `deviceConnectedCallback()` confirms if the connection was successful and initiates GATT service discovery.
- **Service and Characteristic Discovery:** Once connected, the `gattServiceDiscovered()` function is responsible for identifying the correct service (based on its UUID 6e400003-b5a3-f393-e0a9-e50e24dcca9e), and then the `gattCharacteristicDiscovered()` function searches for the corresponding characteristic associated with button notifications.
- **Notification Subscription:** When the correct characteristic is identified, the software subscribes to it so that any notifications (e.g., button presses) sent from the Puck.js devices are received. The `gattSubscribedCallback()` verifies that the subscription was successful.

This modular design, relying on asynchronous callbacks, allows the system to respond quickly to BLE events, such as new device advertisements or incoming notifications, and ensures a robust and responsive BLE operation.

5.2.4 Button Mapping

Button mapping is a crucial component of the system that links user-defined actions to the physical button events received via BLE. The software stores the configuration for each button in an array (`buttonActions`), where each entry represents the action associated with a specific Puck.js device (i.e. button).

The mapping logic is encapsulated in the function `buttonMapping(String txt)`. This function checks whether the received action string starts with the prefix "KEY_", indicating that it should be interpreted as a special keystroke command. The function then compares the action string to several predefined constants (e.g., "KEY_ENTER", "KEY_SPACE", "KEY_RIGHT", etc.) and calls the corresponding function from the USB Keyboard library [45] (such as `Keyboard.consumerPress()`) to simulate a key press. This mechanism allows for a flexible configuration where a button press can either trigger a standard text output or a specific keystroke command based on user preferences defined in the Wi-Fi configuration interface. The mapping of function keys has been done exemplary and may be adapted for specific use cases. As Philhower [45] uses the Arduino Keyboard library [46] as a base for his own library, a list of supported keys may be found in [47].

5.2.5 Key Presses

The implementation of key presses leverages the USB Keyboard library [45] to emulate key-strokes over the USB HID interface. Several key code definitions are established as macros (for example, `#define KEY_ENTER 0x28`, `#define KEY_SPACE 0x2C`, etc.) to correspond with standard keyboard keys. When a button press event is detected (via BLE notification) and the associated action string starts with "KEY_", the `buttonMapping()` function interprets this as a command to simulate a key press.

If the action string does not start with "KEY_", the system defaults to treating the action as text, and the `Keyboard.print()` function is used to output the text over the USB interface. Additionally, after executing a key press or text output, the software provides immediate feedback by flashing the built-in LED briefly. This visual cue confirms that the button press has been processed, enhancing the user experience, especially in scenarios where multiple actions may be triggered in quick succession. Debouncing is handled by defining the global variable `currently_pressed`, which is set to true once the Raspberry Pi Pico receives a notification, that a Puck.js button is pressed. As long as that value remains true, no other received button presses can trigger a button action. Only when the button is released, the variable `currently_pressed` is set to false and new button presses trigger button actions.

5.2.6 Main Loop

The main loop of the software is designed to handle both the configuration and operational phases efficiently. The loop is divided into two segments based on the current state of the device (Wi-Fi configuration or BLE operation):

Wi-Fi Mode: While the variable `wifiActive` is set to true, the main loop repeatedly calls the `loop_Wifi()` function. This function monitors for incoming HTTP connections, processes configuration requests, and serves the HTML interface to users. It ensures that the device remains accessible for configuration until the user explicitly switches to BLE mode.

BLE Mode: Once the user has finished configuring the device and has initiated BLE mode (which sets `wifiActive` to false), the main loop transitions to continuously calling `BTstack.loop()`. This call is essential for processing all BLE-related events, such as scanning for advertisements, handling connection events, managing GATT service and characteristic discoveries, and processing notifications. By continuously invoking `BTstack.loop()`, the software maintains responsiveness to BLE events, ensuring that all button presses and other BLE events are handled in real time.

This dual-phase main loop design ensures that the device operates in a resource-efficient manner, dedicating system resources to either configuration or BLE operation as needed. It also simplifies the overall control flow by clearly segregating the two functional modes, which in turn improves the maintainability and scalability of the code.

6 Evaluation

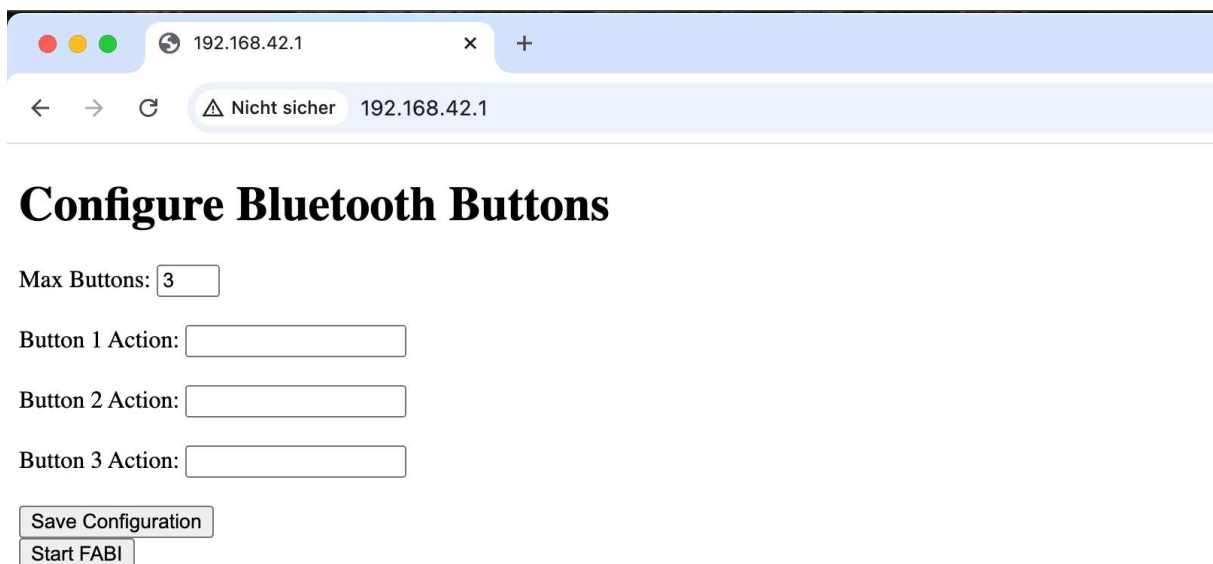
After having explained how the project is implemented in the previous chapter, this chapter will focus on the evaluation of the results. First, a short user test will be performed, showing how an average user might use the device and if it works as intended. Afterwards, the project will be compared to its requirements, while finally the limitations of the project are discussed.

6.1 User Test

The user test will focus on the main tasks a user is expected to perform using the project. Extreme cases or penetration testing will not be performed.

6.1.1 Connecting to the UI

The first test is whether users can open the web-based UI by connecting to the Wi-Fi AP FABI and open the IP address 192.168.42.1 in a browser. As shown in figure 4, opening the web UI works using the browser Google Chrome. As default values, three buttons are preset, so users may enter up to three button actions. Of course, it is possible to change the number of buttons as well.



The screenshot shows a web browser window with the address bar displaying '192.168.42.1'. Below the address bar, the page title 'Configure Bluetooth Buttons' is visible. The main content area contains a form with the following elements:

- 'Max Buttons:' followed by a text input field containing the value '3'.
- 'Button 1 Action:' followed by an empty text input field.
- 'Button 2 Action:' followed by an empty text input field.
- 'Button 3 Action:' followed by an empty text input field.
- Two buttons at the bottom: 'Save Configuration' and 'Start FABI'.

Figure 4: The Web UI with preset values (Source: Own depiction)

6.1.2 Button Action “KEY_ENTER”

The second test to be performed is by changing the preset values and letting one button perform the button action “KEY_ENTER”, as shown in figure 5. Once saving the configuration and entering the BLE operating mode by clicking “Start FABI”, the Puck.js is recognized by the Raspberry Pi Pico W, connected, and upon clicking the push-button, the enter keypress is recognized.



Figure 5: Setting the Button Action “KEY_ENTER” (Source: Own depiction)

6.1.3 Button Action “Hello World”

The third test to be performed is by changing the preset values and letting one button perform the button action write “Hello World”, as shown in figure 6. Once saving the configuration and entering the BLE operating mode by clicking “Start FABI”, the Puck.js is recognized by the Raspberry Pi Pico W, connected, and upon clicking the push-button, “Hello World” is written at the mouse cursor position. All three user tests are therefore successful for the most basic use cases.

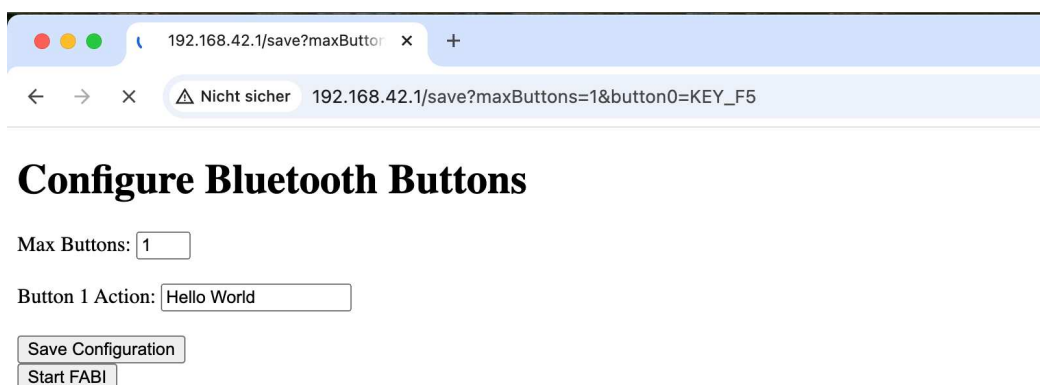


Figure 6: Setting the Button Action to “Hello World” (Source: Own depiction)

6.2 Comparing the Project to the Requirements

As shown, the user tests for the most basic use cases are successful. The next step is to compare the project to the requirements, that have been stated in chapter 3.

6.2.1 Nonfunctional Requirements

The nonfunctional requirements are all met in the project. To reiterate, the hardware Raspberry Pi Pico W and Puck.js are used and programmed using their respective IDEs in C++ and JavaScript.

6.2.2 Functional Requirements

The functional requirements have not all been met in the project as shown in the following.

6.2.2.1 User Interface

- **FR-1 Intuitive Layout:** While subject to personal preference, the UI is structured in a simple and intuitive manner. Distractions or unnecessary ornaments are avoided.
- **FR-2 Immediate Feedback:** Once the values are entered in the UI, it is immediately visible what was entered. The feedback is therefore given in real time. Also, by looking at the onboard LED of the Raspberry Pi Pico W, it is easily visible in which mode the user is currently in. If the LED is on, the user is in the Wi-Fi mode. If it is off, the user is in the BLE mode.
- **FR-3 Customizable Elements:** The UI is customizable by entering the number of Puck.js the user wants to connect. Depending on this value, the UI shows the same amount of user input boxes for the button actions.

6.2.2.2 Connection of Puck.js

- **FR-4 BLE:** The Puck.js connects to the Raspberry Pi Pico W via Bluetooth.
- **FR-5 Automated Discovery and Pairing:** This functional requirement is also met, as the Puck.js are discovered and paired automatically with the Raspberry Pi Pico W. However, it is only possible to connect one Puck.js and not multiple. This is a limitation that may be improved upon in later generations of this prototype.
- **FR-6 Connection Monitoring:** This functional requirement is also implemented. As soon as a Puck.js loses its connection, the Raspberry Pi Pico W starts to scan for devices again.

6.2.2.3 Button Presses

- **FR-7 Accurate Button Detection:** This function is implemented only for normal button presses. Long presses or double presses are not registered but could be improved in further generations of this prototype.
- **FR-8: Action Mapping:** Each type of button press is linked to a specific command. This is implemented and works as designed.
- **FR-9: Feedback Mechanisms:** The system provides immediate feedback to show the user the button press has been registered. The Puck.js uses its onboard LED to indicate button presses.
- **FR-10: Robust Debouncing:** To prevent unintended multiple button presses from being registered, a debugging mechanism is implemented both on the Puck.js, but also in the Raspberry Pi Pico W.

6.3 Limitations

During this project, several limitations emerged that impacted the functionality and overall design. The following sections detail each of these limitations.

6.3.1 One Button vs Multiple Buttons

While the Arduino platform readily supports multiple BLE connections, as implemented in Appendix C, the implementation on the Raspberry Pi Pico W has proven challenging. In this project, only a single button was successfully integrated. Despite multiple attempts, it was not possible to connect multiple buttons with the Raspberry Pi Pico W. This limitation restricts the system's ability to handle more complex user interactions that multiple buttons could facilitate.

6.3.2 Wi-Fi vs BLE

As explained in chapter 4.4, the CYW43439 chip is limited by providing “[s]imultaneous BT/WLAN reception with a single antenna” [36, p. 6]. The author was not able to implement hosting a web server while acting as a BLE Central device at the same time, though it is theoretically possible. It is therefore not possible to configure the button actions while being in operation mode. More work is required, if it is necessary to operate Wi-Fi and BLE simultaneously.

6.3.3 BLE Central vs BLE Peripheral

Finally, the Raspberry Pi Pico W could not be configured to function simultaneously as a BLE central and a BLE peripheral at the same time, wherefore a USB connection is needed. That

may not be a problem for most PCs, but a connection to devices without USB port, such as tablets or smartphones is not possible, limiting the scope of usage to only computers.

7 Conclusion

This thesis set out to enhance the Flexible Assistive Button Interface (FABI) by incorporating wireless functionality via Bluetooth, thereby improving the usability and flexibility of assistive technologies. The project successfully integrated a Raspberry Pi Pico W as the BLE central device and employed a Puck.js for wireless button detection. Key functionalities - including a web-based configuration interface and BLE-driven button mapping - were implemented and validated through user tests.

The developed system effectively divides its operation into two distinct modes: a Wi-Fi configuration phase for user settings and a BLE operation phase for real-time button interactions. This dual-mode approach ensures that the user can easily customize button actions and then transition to wireless operation. The prototype reliably interprets button presses from the Puck.js and executes predefined actions (both keystrokes and text outputs), thereby confirming the viability of using open-source components in an assistive technology context.

Initial user tests demonstrated that the basic functionalities - connecting to the configuration UI, executing keystroke commands, and outputting text - are working as intended. The immediate visual feedback provided by both the interface and the onboard LED contributes to a positive user experience.

While the project has achieved its core objectives, several limitations remain:

- The current hardware and software setup restricts the prototype to a single Puck.js connection, limiting scalability.
- The inability to operate Wi-Fi and BLE concurrently points to a need for further hardware exploration.
- Future enhancements should include support for more complex button interactions (e.g., long presses, double taps) and additional testing with a broader user base.

Future work may involve investigating alternative microcontrollers or wireless modules that allow simultaneous dual-mode operation, refining the software architecture to manage multiple devices, and expanding the scope of user evaluations to better meet the diverse needs of the users. Also, as this was only a prototype, the next steps need to be implementing these functionalities into the FABI project.

In conclusion, the wireless button interface for FABI represents a promising step toward more adaptable and accessible assistive technologies. The integration of BLE connectivity with a flexible, user-configurable system lays the groundwork for future developments that can overcome current limitations and offer a more robust solution for users with disabilities. By building on the open-source principles inherent in the FABI project, this work contributes to the ongoing effort to empower users with greater independence and customization in their digital interactions.

Bibliography

- [1] Microsoft Corporation, "Xbox Adaptive Controller | Xbox." Accessed: Mar. 02, 2025. [Online]. Available: <https://www.xbox.com/de-DE/accessories/controllers/xbox-adaptive-controller>
- [2] Microsoft Corporation, "Kaufen Microsoft Adaptive Hub - Microsoft Store." Accessed: Mar. 02, 2025. [Online]. Available: <https://www.microsoft.com/de-de/d/microsoft-adaptive-hub/8pbjx6zn089b?activetab=pivot:%C3%BCbersichttab>
- [3] AsTeRICS Foundation, "FABI - Flexible Assistive Button Interface." Accessed: Feb. 14, 2025. [Online]. Available: <https://www.asterics-foundation.org/projects/fabi/>
- [4] A. Thorpe, M. Ma, and A. Oikonomou, "History and alternative game input methods," in *2011 16th International Conference on Computer Games (CGAMES)*, IEEE, Jul. 2011, pp. 76–93. doi: 10.1109/CGAMES.2011.6000321.
- [5] G. Turpin, J. Armstrong, P. Frost, B. Fine, C. Ward, and L. Pinnington, "Evaluation of alternative computer input devices used by people with disabilities," *J Med Eng Technol*, vol. 29, no. 3, pp. 119–129, Jan. 2005, doi: 10.1080/03091900500075317.
- [6] R. G. Garcia *et al.*, "Wearable augmentative and alternative communication device for paralysis victims using Brute Force Algorithm for pattern recognition," in *2017 IEEE 9th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, IEEE, Dec. 2017, pp. 1–6. doi: 10.1109/HNICEM.2017.8269554.
- [7] R. Fatnani, J. Fernandes, N. Shah, and A. Pabarekar, "Alternative and Augmentative Communication Device for Paralytic Patients," in *2024 3rd International Conference on Sentiment Analysis and Deep Learning (ICSADL)*, IEEE, Mar. 2024, pp. 697–701. doi: 10.1109/ICSADL61749.2024.00121.
- [8] Yu-Luen Chen, Te-Son Kuo, W. H. Chang, and Jin-Shin Lai, "A novel position sensors-controlled computer mouse for the disabled," in *Proceedings of the 22nd Annual International Conference of the IEEE Engineering in Medicine and Biology Society (Cat. No.00CH37143)*, IEEE, pp. 2263–2266. doi: 10.1109/IEMBS.2000.900591.
- [9] B. Aigner, V. David, M. Deinhofer, and C. Veigl, "FLipMouse," in *Proceedings of the 7th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*, New York, NY, USA: ACM, Dec. 2016, pp. 25–32. doi: 10.1145/3019943.3019948.
- [10] AsTeRICS Foundation, "Die FLipMouse - AsTeRICS Foundation." Accessed: Mar. 02, 2025. [Online]. Available: <https://www.asterics-foundation.org/projekte-2/flipmouse/>
- [11] AsTeRICS Foundation, "asterics/FABI: The repository for FABI (Flexible Assistive Button Interface)." Accessed: Feb. 14, 2025. [Online]. Available: <https://github.com/asterics/FABI>
- [12] AsTeRICS Foundation, "FABI Configuration." Accessed: Feb. 14, 2025. [Online]. Available: <https://fabi.asterics.eu/>

- [13] AsTeRICS Foundation, "FABI - User Manual." Accessed: Feb. 14, 2025. [Online]. Available: <https://github.com/asterics/FABI/blob/master/Documentation/UserManual/Markdown/Fabi%20User%20Manual.md>
- [14] AsTeRICS Foundation, "FABI the Flexible Assistive Button Interface 'Do-It-Yourself' - Building Guide." Accessed: Feb. 14, 2025. [Online]. Available: https://github.com/asterics/FABI/blob/master/Documentation/ConstructionManual/SelfmadeBox/FABI_Construction_SelfmadeBox.pdf
- [15] AsTeRICS Foundation, "FABI - Flexible Assistive Button Interface Construction Manual." Accessed: Feb. 14, 2025. [Online]. Available: https://github.com/asterics/FABI/blob/master/Documentation/ConstructionManual/3D-printedBox/en/FABI_ConstructionManual.pdf
- [16] AsTeRICS Foundation, "FABI Construction Manual." Accessed: Feb. 14, 2025. [Online]. Available: <https://github.com/asterics/FABI/blob/master/Documentation/ConstructionManual/PCB-Version/FABIManual.pdf>
- [17] Atmel Corporation, "ATmega32U4 Datasheet," 2010. Accessed: Feb. 14, 2025. [Online]. Available: <https://cdn.sparkfun.com/datasheets/Dev/Arduino/Boards/AT-Mega32U4.pdf>
- [18] SparkFun Electronics, "Pro Micro - 5V/16MHz." Accessed: Feb. 14, 2025. [Online]. Available: <https://www.sparkfun.com/pro-micro-5v-16mhz.html>
- [19] E. F. Philhower, "earlephilhower/arduino-pico: Raspberry Pi Pico Arduino core, for all RP2040 and RP2350 boards." Accessed: Feb. 14, 2025. [Online]. Available: <https://github.com/earlephilhower/arduino-pico>
- [20] E. F. Philhower, "Arduino-Pico — Arduino-Pico 4.4.3 documentation." Accessed: Feb. 14, 2025. [Online]. Available: <https://arduino-pico.readthedocs.io/en/latest/>
- [21] Espruino, "Espruino Web IDE." Accessed: Feb. 15, 2025. [Online]. Available: <https://www.espruino.com/ide/>
- [22] J. Dungen, "Puck.js Development Guide." Accessed: Feb. 15, 2025. [Online]. Available: <https://reelyactive.github.io/diy/puckjs-dev/>
- [23] Espruino, "Puck.js." Accessed: Feb. 14, 2025. [Online]. Available: <https://www.espruino.com/Puck.js>
- [24] Raspberry Pi Ltd, "RP2040 Datasheet," Oct. 14, 2024. Accessed: Feb. 14, 2025. [Online]. Available: <https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf>
- [25] Arduino, "Arduino Nano RP2040 Connect Datasheet."
- [26] Raspberry Pi Ltd, "Raspberry Pi Pico W Datasheet," Oct. 15, 2024. Accessed: Feb. 14, 2025. [Online]. Available: <https://datasheets.raspberrypi.com/picow/pico-w-datasheet.pdf>
- [27] BerryBase GmbH, "Raspberry Pi Pico W, RP2040 + WLAN Mikrocontroller-Board." Accessed: Feb. 14, 2025. [Online]. Available: <https://www.berrybase.de/raspberry-pi-pico-w-rp2040-wlan-mikrocontroller-board>

- [28] Infineon Technologies AG, "CYW43439." Accessed: Feb. 14, 2025. [Online]. Available: <https://www.infineon.com/cms/en/product/wireless-connectivity/airoc-wi-fi-plus-blue-tooth-combos/wi-fi-4-802.11n/cyw43439/>
- [29] u-blox, "NINA-W10 series Datasheet," Dec. 19, 2022. Accessed: Feb. 14, 2025. [Online]. Available: https://content.u-blox.com/sites/default/files/NINA-W10_DataSheet_UBX-17065507.pdf
- [30] C. Windeck, "Raspberry Pi Pico 2 W: Mikrocontroller-Board mit WLAN vorgestellt | heise online." Accessed: Feb. 14, 2025. [Online]. Available: <https://www.heise.de/news/Raspberry-Pi-Pico-2-W-Mikrocontroller-Board-mit-WLAN-vorgestellt-10082909.html>
- [31] Raspberry Pi Ltd, "Raspberry Pi Pico 2 W Datasheet," Nov. 26, 2024. Accessed: Feb. 14, 2025. [Online]. Available: <https://datasheets.raspberrypi.com/picow/pico-2-w-datasheet.pdf>
- [32] Raspberry Pi Ltd, "RP2350 Datasheet," Oct. 16, 2024. Accessed: Feb. 14, 2025. [Online]. Available: <https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf>
- [33] BerryBase GmbH, "Raspberry Pi Pico 2W, RP2350 + WLAN + Bluetooth Mikrocontroller-Board." Accessed: Feb. 14, 2025. [Online]. Available: https://www.berry-base.de/raspberry-pi-pico-2w-rp2350-wlan-bluetooth-mikrocontroller-board?utm_source=google&utm_medium=cpc&gad_source=1&gbraid=0AAAAADSQJK5VOj_pMDe3XJJt-wudJV3C0&gclid=CjwKCAiA8Lu9BhA8EiwAag16b6RfywPgD-TiNy7ovvn2tIR_ZSmJGJOElz2sBuxc0K8q9uFuRaz1SjxoCGOkQAvD_BwE
- [34] Nordic Semiconductor, "nRF52832 Datasheet".
- [35] Espruino, "Espruino Puck.js v2.1." Accessed: Feb. 14, 2025. [Online]. Available: <https://shop.espruino.com/puckjs>
- [36] Infineon Technologies AG, "CYW43439 Datasheet," Mar. 2021, Accessed: Feb. 15, 2025. [Online]. Available: https://www.mouser.com/datasheet/2/196/Infineon_CYW43439_DataSheet_v03_00_EN-3074791.pdf?srsltid=Afm-BOoqh1Fsa7ADgs_QJUSfQZn4mVc2rR7JT3VoX70H_gDAn5DZDs7Fx
- [37] USB Implementers Forum, "USB Logo Usage Guidelines," 2024, Accessed: Feb. 15, 2025. [Online]. Available: https://www.usb.org/sites/default/files/usb-if_original_logo_usage_guidelines_final_2024.02.8.pdf
- [38] Wi-Fi Alliance, "Generational Wi-Fi User Guide," Apr. 2023, Accessed: Feb. 15, 2025. [Online]. Available: https://www.wi-fi.org/system/files/Generational_Wi-Fi_User_Guide_202304.pdf
- [39] USB Implementers Forum, "Human Interface Devices (HID) Specifications and Tools." Accessed: Feb. 15, 2025. [Online]. Available: <https://www.usb.org/hid>
- [40] I. Bluetooth SIG, "Bluetooth Technology Website." Accessed: Feb. 15, 2025. [Online]. Available: <https://www.bluetooth.com/>
- [41] E. F. Philhower, "arduino-pico/libraries/BTstackLib at master · earlephilhower/arduino-pico." Accessed: Feb. 16, 2025. [Online]. Available: <https://github.com/earlephilhower/arduino-pico/tree/master/libraries/BTstackLib>

- [42] F. König, "FriedrichKoenig/WirelessButtonforFABI: Bachelor Thesis Friedrich Koenig." Accessed: Mar. 02, 2025. [Online]. Available: <https://github.com/FriedrichKoenig/WirelessButtonforFABI>
- [43] Espruino, "About Bluetooth LE (BLE)." Accessed: Feb. 16, 2025. [Online]. Available: <https://www.espruino.com/About+Bluetooth+LE>
- [44] E. F. Philhower, "arduino-pico/libraries/WiFi at master · earlephilhower/arduino-pico." Accessed: Feb. 16, 2025. [Online]. Available: <https://github.com/earlephilhower/arduino-pico/tree/master/libraries/WiFi>
- [45] E. F. Philhower, "arduino-pico/libraries/Keyboard at master · earlephilhower/arduino-pico." Accessed: Feb. 16, 2025. [Online]. Available: <https://github.com/earlephilhower/arduino-pico/tree/master/libraries/Keyboard>
- [46] Arduino, "Keyboard | Arduino Documentation." Accessed: Feb. 16, 2025. [Online]. Available: <https://docs.arduino.cc/language-reference/en/functions/usb/Keyboard/>
- [47] Arduino, "Keyboard Modifiers and Special Keys | Arduino Documentation." Accessed: Feb. 16, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/usb/Keyboard/keyboardModifiers/?_gl=1*162ul68*_up*MQ..*_ga*MTM4MTUzODYzMj4xNzM5NzA5NTk1*_ga_NEXN8H46L5*MTczOTcwOTU5My4xLjAuMTczOTcwOTU5My4wLjAuMTc0Nzk3OTU3OQ..

List of Figures

Figure 1: Hardware connections at the Wi-Fi phase	14
Figure 2: Hardware connections at the BLE phase.....	14
Figure 3: Simplified Flowchart.....	16
Figure 4: The Web UI with preset values	23
Figure 5: Setting the Button Action “KEY_ENTER”	24
Figure 6: Setting the Button Action to “Hello World”	24

List of Listings

Listing 1: Defining the UUIDs	17
Listing 2: The variable buttonState	17
Listing 3: The setupBLE function.....	18
Listing 4: The setWatch functions	18
Listing 5: Including the libraries	19
Listing 6: Initializing the global variables	20

List of Abbreviations

AP	Access Point
BLE	Bluetooth Low Energy
DIY	Do-it-yourself
FABI	Flexible Assistive Button Interface
GPIO	General Purpose Input/Output
HID	Human Interface Device
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
I/Os	Input/Outputs
LED	Light Emitting Diode
SDK	Software Development Kit
UI	User Interface
USB	Universal Serial Bus
UUID	Universally Unique Identifier
Wi-Fi	Wireless Fidelity
SSID	Service Set Identifier

Documentation table of AI-based tools

AI-based tools	Intended use	Prompt, source, page, paragraph...
ChatGPT (4o-mini)	Grammar and spelling	"Please list issues with spelling and grammar in the following text: " Upload of PDF document
DeepL	Translating of technical terms	"Bitte übersetze Benutzeranforderung auf Englisch"
ChatGPT (o3-mini)	Commenting the code	„Please comment my following code for better readability:" Upload of my code

Appendix A: Code for Puck.js

```
1. const SERVICE_UUID = '6e400001-b5a3-f393-e0a9-e50e24dcca9e'; // Service UUID
2. const CHARACTERISTIC_UUID = '6e400003-b5a3-f393-e0a9-e50e24dcca9e'; // Characteristic UUID
3.
4. let buttonState = 0; // 0 = Released, 1 = Pressed
5.
6. // Function to set up services and start advertising
7. function setupBLE() {
8.   // Set up GATT services and characteristics
9.   NRF.setServices({
10.     '6e400001-b5a3-f393-e0a9-e50e24dcca9e': {
11.       '6e400003-b5a3-f393-e0a9-e50e24dcca9e': {
12.         value: [buttonState], // Initial value
13.         readable: true, // Allow reading the value
14.         writable: false, // Disable writing to the characteristic
15.         notify: true // Enable notifications
16.       }
17.     }
18.   }, { advertise: [SERVICE_UUID], uart: false });
19.
20.   console.log("BLE services set up and advertising started.");
21. }
22.
23. // Function to notify the connected device of button state changes
24. function notifyButtonState() {
25.   NRF.updateServices({
26.     '6e400001-b5a3-f393-e0a9-e50e24dcca9e': {
27.       '6e400003-b5a3-f393-e0a9-e50e24dcca9e': {
28.         value: [buttonState], // Initial value
29.         readable: true, // Allow reading the value
30.         writable: false, // Disable writing to the characteristic
31.         notify: true // Enable notifications
32.       }
33.     }
34.   });
35.   //console.log("Notification sent: " + buttonState);
36. }
37.
38. // Restart BLE and set up services
39. function restartBLE() {
40.   console.log("Restarting BLE...");
41.   //NRF.disconnect(); // Disconnect any connected devices
42.   NRF.restart(() => {
43.     setupBLE(); // Set up BLE after restart
44.     console.log("BLE restarted.");
45.   });
46. }
47.
48. // Watch for button press
49. setWatch(() => {
50.   buttonState = 1;
51.   console.log(buttonState); //Raspi recognizes only this
52.   notifyButtonState();
53.   LED1.write(buttonState);
54. }, BTN, { edge: "rising", debounce: 50, repeat: true });
55.
56. // Watch for button release
57. setWatch(() => {
58.   buttonState = 0;
59.   console.log(buttonState); //Raspi recognizes only this
60.   notifyButtonState();
61.   LED1.write(buttonState);
62. }, BTN, { edge: "falling", debounce: 50, repeat: true });
```

Source: [42]

Appendix B: Code for Raspberry Pico W

```
1.  /*
2.  * Raspberry Pi Pico W code for Wi-Fi AP + BLE (BTstack).
3.  *
4.  * Functionality:
5.  * - Runs a Wi-Fi AP on startup.
6.  * - Hosts a web page to set "maxButtons" and each button's key action.
7.  * - After clicking "Start FBI", disables Wi-Fi and starts BLE scanning.
8.  * - Connects to Puck.js devices advertising the given service/characteristic.
9.  * - On notifications (button press), sends the configured text or keystroke over USB.
10. */
11.
12. // ----- INCLUDES -----
13. #include <BTstackLib.h>
14. #include <SPI.h>
15. #include <Keyboard.h>
16. #include <WiFi.h>
17. // ----- USER-CONFIGURABLE WIFI CREDENTIALS -----
18. char ssid[] = "FABI"; // Access Point SSID
19. char pass[] = "asterics"; // Access Point password
20.
21. // ----- GLOBALS -----
22.
23. // Wi-Fi
24. bool wifiActive = true; // Tracks if WiFi is active (AP mode)
25. int status; // = WL_IDLE_STATUS;
26. WiFiServer server(80);
27.
28. // Number of buttons to connect to
29. int maxButtons = 3;
30. String buttonActions[100]; // Action strings for each "button"
31.
32. // BLE (BTstack) Definitions
33. // The Puck.js UUIDs as 128-bit. BTstack uses a UUID class:
34. UUID puckServiceUUID("6e400001-b5a3-f393-e0a9-e50e24dcca9e");
35. UUID puckCharacteristicUUID("6e400003-b5a3-f393-e0a9-e50e24dcca9e");
36.
37. // We'll allow up to 100 discovered Puck.js devices
38. BLEDevice puckDevices[100];
39. bool puckConnected[100]; // Keep track of whether device is connected
40. BLEService puckServices[100]; // The service object for each device
41. BLECharacteristic puckChars[100]; // The characteristic object for each device
42. bool puckServiceFound[100];
43. bool puckCharFound[100];
44. String puckNames[100]; // Puck.js device name from advertisement
45.
46. int puckCount = 0; // How many Puck.js have been connected so far
47. int currentConnectIndex = -1; // Which array index is currently connecting
48.
49. // Forward declarations of our BLE callbacks:
50. void advertisementCallback(BLEAdvertisement *advertisement);
51. void deviceConnectedCallback(BLEStatus status, BLEDevice *device);
52. void deviceDisconnectedCallback(BLEDevice *device);
53. void gattServiceDiscovered(BLEStatus status, BLEDevice *device, BLEService *service);
54. void gattCharacteristicDiscovered(BLEStatus status, BLEDevice *device, BLECharacteristic
*characteristic);
55. void gattSubscribedCallback(BLEStatus status, BLEDevice *device);
56. void gattCharacteristicNotification(BLEDevice *device, uint16_t value_handle, uint8_t
*value, uint16_t length);
57. void gattReadCallback(BLEStatus status, BLEDevice *device, uint8_t *value, uint16_t length);
58. void gattWrittenCallback(BLEStatus status, BLEDevice *device);
59.
60. const char *button_pressed = "[J1 \n"; //Used to determine a button press
61. bool currently_pressed = false;
```

```

62.
63. // ----- SETUP -----
64. void setup() {
65.   Serial.begin(9600);
66.   Keyboard.begin();
67.   while (!Serial);
68.
69.   // Initialize built-in LED
70.   pinMode(LED_BUILTIN, OUTPUT);
71.
72.
73.   // Start Wi-Fi as Access Point
74.   setup_Wifi();
75.   //setup_BLE();
76. }
77.
78. // ----- MAIN LOOP -----
79. void loop() {
80.   // As long as Wi-Fi (AP mode) is active, serve the config page
81.   while (wifiActive) {
82.     loop_Wifi();
83.   }
84.
85.   // Once Wi-Fi is disabled and BLE is started, let BTstack handle BLE
86.   while (!wifiActive) {
87.     // The BTstack library is event/callback-driven, but you
88.     // must regularly call BTstack.loop() in your main loop.
89.     BTstack.loop();
90.   }
91. }
92.
93. // ----- BLE SETUP/START -----
94. void setup_BLE() {
95.   // Initialize all BTstack callbacks we need
96.   BTstack.setBLEAdvertisementCallback(advertisementCallback);
97.   BTstack.setBLEDeviceConnectedCallback(deviceConnectedCallback);
98.   BTstack.setBLEDeviceDisconnectedCallback(deviceDisconnectedCallback);
99.   BTstack.setGATTServiceDiscoveredCallback(gattServiceDiscovered);
100.  BTstack.setGATTCharacteristicDiscoveredCallback(gattCharacteristicDiscovered);
101.  BTstack.setGATTCharacteristicNotificationCallback(gattCharacteristicNotification);
102.  //BTstack.setGATTCharacteristicReadCallback(gattReadCallback);
103.  //BTstack.setGATTCharacteristicWrittenCallback(gattWrittenCallback);
104.  BTstack.setGATTCharacteristicSubscribedCallback(gattSubscribedCallback);
105.  //BTstack.setGATTCharacteristicIndicationCallback(gattCharacteristicIndication);
106.
107.  // Start BTstack
108.  BTstack.setup();
109.
110.  // Begin scanning for BLE advertisements
111.  puckCount = 0;
112.  BTstack.bleStartScanning();
113.
114.  Serial.println("BTstack BLE scanning started...");
115. }
116.
117. // ----- BLE CALLBACKS -----
118.
119. // Called whenever an advertisement is received
120. void advertisementCallback(BLEAdvertisement *advertisement) {
121.   // If we already have enough connected Puck.js, ignore
122.   if (puckCount >= maxButtons) return;
123.
124.   // Check if this advertisement has our Puck.js service
125.   if (advertisement->containsService(&puckServiceUUID)) {
126.     // Attempt to connect
127.     Serial.println("Found a Puck.js device advertising our service!");
128.

```



```

129. // Store the device's name
130. currentConnectIndex = puckCount;
131. puckNames[currentConnectIndex] = advertisement->getBdAddr()->getAddressString();
132. Serial.print("Assigning device index ");
133. Serial.print(currentConnectIndex);
134. Serial.print(" name: ");
135. Serial.println(puckNames[currentConnectIndex]);
136. // Stop scanning so we can connect
137. BTstack.bleStopScanning();
138. // Connect with a 10s timeout
139. BTstack.bleConnect(advertisement, 1000);
140. }
141. }
142.
143. // Called when the connection attempt completes
144. void deviceConnectedCallback(BLEStatus status, BLEDevice *device) {
145.     if (status == BLE_STATUS_OK) {
146.         // Connection success
147.         Serial.println("Device connected!");
148.         puckDevices[currentConnectIndex] = *device;
149.         puckConnected[currentConnectIndex] = true;
150.
151.         // Start service discovery
152.         puckServiceFound[currentConnectIndex] = false;
153.         puckCharFound[currentConnectIndex] = false;
154.         //BuhBuhdevice->discoverGATTServices();
155.         //Serial.println(currentConnectIndex);
156.         Serial.print("Services: ");
157.         Serial.println(puckDevices[currentConnectIndex].discoverGATTServices());
158.         //Serial.println(device->discoverGATTServices());
159.         puckDevices[currentConnectIndex].discoverGATTServices();
160.
161.     } else if (status == BLE_STATUS_CONNECTION_TIMEOUT) {
162.         Serial.println("Connection timeout or error while connecting!");
163.         // Resume scanning for more Puck.js
164.         BTstack.bleStartScanning();
165.     }
166.     else {
167.         Serial.print("Unknown connect error: ");
168.         Serial.println(status);
169.         // Resume scanning
170.         BTstack.bleStartScanning();
171.     }
172. }
173.
174. // Called if device gets disconnected
175. void deviceDisconnectedCallback(BLEDevice *device) {
176.     // Find which device index this is
177.     int disconnectedIndex = -1;
178.     for (int i = 0; i <= puckCount; i++) {
179.         if (puckDevices[i].getHandle() == device->getHandle()){
180.             disconnectedIndex = i;
181.             break;
182.         }
183.     }
184.
185.     Serial.println("Device disconnected, scanning again...");
186.     if (disconnectedIndex != -1) {
187.         puckConnected[disconnectedIndex] = false;
188.     }
189.     // Keep scanning for more devices
190.     BTstack.bleStartScanning();
191. }
192.
193. // Service discovery results
194. void gattServiceDiscovered(BLEStatus status, BLEDevice *device, BLEService *service) {
195.     Serial.print("Status: ");

```

```

196. Serial.println(status);
197. if (status == BLE_STATUS_OK) {
198.     Serial.print("Service discovered: ");
199.     Serial.println(service->getUUID()->getUuidString());
200.
201.     // Check if it's the Puck.js service
202.     if (service->matches(&puckServiceUUID)) {
203.         Serial.println("Puck.js service found!");
204.         // Store service
205.         for (int i = 0; i < puckCount + 1; i++) {
206.             if (puckDevices[i].getHandle() == device->getHandle()){
207.                 puckServices[i] = *service;
208.                 puckServiceFound[i] = true;
209.                 break;
210.             }
211.         }
212.     }
213. }
214. else if (status == BLE_STATUS_DONE) {
215.     Serial.println("Service discovery finished.");
216.
217.     // If we found the service, discover its characteristic(s).
218.     for (int i = 0; i < puckCount + 1; i++) {
219.         //if (puckDevices[i].isSameDevice(device) && puckServiceFound[i]) {
220.             if (puckDevices[i].getHandle() == device->getHandle()){
221.                 device->discoverCharacteristicsForService(&puckServices[i]);
222.             }
223.         }
224.     }
225.     else {
226.         Serial.println("Service discovery error.");
227.     }
228. }
229.
230. // Characteristic discovery results
231. void gattCharacteristicDiscovered(BLEStatus status, BLEDevice *device, BLECharacteristic
*characteristic) {
232.     if (status == BLE_STATUS_OK) {
233.         Serial.print("Characteristic discovered: ");
234.         Serial.print(characteristic->getUUID()->getUuidString());
235.         Serial.print(", handle 0x");
236.         Serial.println(characteristic->getCharacteristic()->value_handle, HEX);
237.
238.         // Check if it's the Puck.js button characteristic
239.         if (characteristic->matches(&puckCharacteristicUUID)) {
240.             Serial.println("Puck.js characteristic found!");
241.             for (int i = 0; i < puckCount + 1; i++) {
242.                 if (puckDevices[i].getHandle() == device->getHandle()){
243.                     puckChars[i] = *characteristic;
244.                     puckCharFound[i] = true;
245.                     Serial.println(puckDevices[i].getHandle());
246.                 }
247.             }
248.         }
249.     }
250.     else if (status == BLE_STATUS_DONE) {
251.         Serial.println("Characteristic discovery finished.");
252.
253.         // Subscribe for notifications if we found the characteristic
254.         for (int i = 0; i < puckCount + 1; i++) {
255.             //if (puckDevices[i].isSameDevice(device) && puckCharFound[i]) {
256.             if (puckDevices[i].getHandle() == device->getHandle()) {
257.                 // Subscribe
258.                 //BLEStatus subscribeStatus =
259.                 device->subscribeForNotifications(&puckChars[i]);
260.                 if (status == BLE_STATUS_OK) {
261.                     Serial.println("Subscription to notifications requested.");

```

```

262.     }
263.   }
264. }
265.
266. // Increase the puckCount since we have successfully connected & discovered
267. // for the current device index
268. puckCount++;
269. Serial.print("Total devices connected: ");
270. Serial.println(puckCount);
271. // If we can still connect more devices, resume scanning
272. if (puckCount < maxButtons) {
273.   BTstack.bleStartScanning();
274. }
275. }
276. else {
277.   Serial.println("Characteristic discovery error.");
278. }
279. }
280.
281. // Called after we attempt to subscribe
282. void gattSubscribedCallback(BLEStatus status, BLEDevice *device) {
283.   if (status == BLE_STATUS_OK) {
284.     Serial.println("Successfully subscribed to Puck.js notifications!");
285.   } else {
286.     Serial.print("Subscription failed, status = ");
287.     Serial.println(status);
288.   }
289. }
290.
291. // Called when we receive a notification from a characteristic
292. void gattCharacteristicNotification(BLEDevice *device, uint16_t value_handle, uint8_t
*value, uint16_t length) {
293.   //'value' is the data from the remote device
294.   // We want the second bit of the first byte (0x02):
295.   (void) device;
296.   (void) value_handle;
297.   (void) length;
298.   Serial.print("Notification: ");
299.   Serial.println((const char *)value);
300.   if (length > 0) {
301.     char* v = (char *)value;
302.     Serial.print("Value: ");
303.     Serial.println(v[4]);
304.     if (v[4] == '1' and currently_pressed == false) {
305.       currently_pressed = true;
306.       // Find which device index this is
307.       int devIndex = -1;
308.       for (int i = 0; i < puckCount; i++) {
309.         if (puckDevices[i].getHandle() == device->getHandle()) {
310.           devIndex = i;
311.           break;
312.         }
313.       }
314.       if (devIndex != -1) {
315.         Serial.print(puckNames[devIndex]);
316.         Serial.println(" button pressed");
317.
318.         // Retrieve the action string
319.         String buttonText = buttonActions[devIndex];
320.         // If it starts with KEY_, treat as special key
321.         if (buttonText.startsWith("KEY_")) {
322.           buttonMapping(buttonText);
323.           digitalWrite(LED_BUILTIN, HIGH);
324.           delay(200);
325.           digitalWrite(LED_BUILTIN, LOW);
326.         } else {
327.           // Otherwise, type the text

```

```

328.         Keyboard.print(buttonText.c_str());
329.         Serial.println(buttonText);
330.         digitalWrite(LED_BUILTIN, HIGH);
331.         delay(200);
332.         digitalWrite(LED_BUILTIN, LOW);
333.     }
334. }
335. } else if (v[4] == '0' and currently_pressed == true){
336.     currently_pressed = false;
337.     Serial.println("Not Pressed");
338. } else {
339.     Serial.println("Something Strange has happened");
340. }
341. }
342. }
343.
344. // ----- Key Kodes -----
345. #define KEY_ENTER 176 // Keyboard Return (ENTER)
346. #define KEY_KP_DOT 235 // Keyboard Spacebar
347. #define KEY_RIGHT 215 // Keyboard Right Arrow
348. #define KEY_LEFT 216 // Keyboard Left Arrow
349. #define KEY_DOWN 217 // Keyboard Down Arrow
350. #define KEY_UP 218 // Keyboard Up Arrow
351. #define KEY_F5 198 // Keyboard F5
352.
353. // ----- Function to map the button actions -----
354. void buttonMapping(String txt) {
355.     if(txt == "KEY_ENTER"){
356.         Keyboard.press(KEY_ENTER);
357.         delay(10);
358.         Keyboard.release(KEY_ENTER);
359.     } else if (txt == "KEY_KP_DOT"){
360.         Keyboard.press(KEY_KP_DOT);
361.         delay(10);
362.         Keyboard.release(KEY_KP_DOT);
363.     } else if (txt == "KEY_RIGHT"){
364.         Keyboard.press(KEY_RIGHT);
365.         delay(10);
366.         Keyboard.release(KEY_RIGHT);
367.     } else if (txt == "KEY_LEFT"){
368.         Keyboard.press(KEY_LEFT);
369.         delay(10);
370.         Keyboard.release(KEY_LEFT);
371.     } else if (txt == "KEY_DOWN"){
372.         Keyboard.press(KEY_DOWN);
373.         delay(10);
374.         Keyboard.release(KEY_DOWN);
375.     } else if (txt == "KEY_UP"){
376.         Keyboard.press(KEY_UP);
377.         delay(10);
378.         Keyboard.release(KEY_UP);
379.     } else if (txt == "KEY_F5"){
380.         Keyboard.press(KEY_F5);
381.         delay(10);
382.         Keyboard.release(KEY_F5);
383.     }
384. }
385.
386. void setup_Wifi() {
387.     Serial.println("Access Point Web Server");
388.     digitalWrite(LED_BUILTIN, HIGH);
389.     // Check for the WiFi module:
390.     if (WiFi.status() == WL_NO_MODULE) {
391.         Serial.println("Communication with WiFi module failed!");
392.         while (true);
393.     }
394. }

```

```

395. // Create access point
396. Serial.print("Creating access point named: ");
397. Serial.println(ssid);
398. WiFi.mode(WIFI_OFF); // Reset WiFi mode
399. WiFi.disconnect(true); // Clear previous WiFi settings
400.
401. delay(100);
402. WiFi.mode(WIFI_AP); // Set mode to Access Point
403. status = WiFi.beginAP(ssid, pass);
404. WiFi.setHostname(ssid);
405. delay(5000); // Wait for the AP to start
406. server.begin(); // Start the web server
407. printWiFiStatus(); // Print the connection status
408. }
409.
410. void loop_Wifi() {
411.   WiFiClient client = server.available(); // Check for incoming client
412.   if (client) {
413.     Serial.println("New client connected");
414.     String currentLine = "";
415.     String request = "";
416.     bool isRequestLine = true;
417.     // wait for data to be available
418.     unsigned long timeout = millis();
419.
420.     while (client.connected()) {
421.       if (client.available()) {
422.         char c = client.read();
423.         Serial.write(c);
424.
425.         // Accumulate the request line
426.         if (c != '\r' && c != '\n') {
427.           currentLine += c;
428.         }
429.
430.         if (c == '\n') {
431.           if (isRequestLine) {
432.             request = currentLine;
433.             Serial.print("Request: ");
434.             Serial.println(request);
435.             isRequestLine = false;
436.           }
437.
438.           if (currentLine.length() == 0) {
439.             // Send HTTP response
440.             client.println("HTTP/1.1 200 OK");
441.             client.println("Content-type:text/html");
442.             client.println();
443.             client.println("<!DOCTYPE HTML>");
444.             client.println("<html><body><h1>Configure Bluetooth Buttons</h1>");
445.             client.println("<form action=\"/save\" method=\"GET\">");
446.             client.println("<label for=\"maxButtons\">Max Buttons: </label>");
447.             client.print("<input type=\"number\" id=\"maxButtons\" name=\"maxButtons\" "
value=""");
448.             client.print(maxButtons);
449.             client.println("<input type=\"text\" id=\"button\" name=\"button\">");
450.
451.             for (int i = 0; i < maxButtons; i++) {
452.               client.print("<label for=\"button\">");
453.               client.print(i);
454.               client.print(">Button ");
455.               client.print(i + 1);
456.               client.print(" Action: </label>");
457.               client.print("<input type=\"text\" id=\"button\">");
458.               client.print(i);
459.               client.print(" name=\"button\">");
460.               client.print(i);

```

```

461.         client.print("\n value=\"");
462.         client.print(buttonActions[i]);
463.         client.println("\n/><br/><br/>");
464.     }
465.
466.     client.println("<input type=\"submit\" value=\"Save Configuration\"/></form>");
467.     client.println("<form action=\"/startBLE\" method=\"GET\">");
468.     client.println("<input type=\"submit\" value=\"Start FABI\"/></form>");
469.     client.println("</body></html>");
470.     client.println();
471.     break;
472. }
473.     currentLine = "";
474. }
475. }
476. }
477. Serial.println("Client disconnected");
478.
479. // Process request
480. if (request.startsWith("GET /save?")) {
481.     Serial.println("Saving configuration");
482.     parseConfig(request);
483.     delay(1000);
484.     Keyboard.consumerPress(KEY_F5);
485. } else if (request.startsWith("GET /startBLE")) {
486.     Serial.println("Starting FABI (BLE)");
487.     startFABI();
488. }
489. }
490. }
491.
492.
493. void parseConfig(String line) {
494.     // Extract the maximum number of buttons
495.     int idx = line.indexOf("maxButtons=");
496.     if (idx != -1) {
497.         int endIdx = line.indexOf('&', idx);
498.         if (endIdx == -1) endIdx = line.length();
499.         maxButtons = line.substring(idx + 11, endIdx).toInt();
500.     }
501.
502.     // Ensure maxButtons is within a valid range
503.     maxButtons = constrain(maxButtons, 1, 20); // Limit maxButtons
504.
505.     // Extract actions for each button
506.     for (int i = 0; i < maxButtons; i++) {
507.         String buttonParam = "button" + String(i) + "=";
508.         idx = line.indexOf(buttonParam);
509.         if (idx != -1) {
510.             int endIdx = line.indexOf('&', idx);
511.             if (endIdx == -1) endIdx = line.length();
512.             String action = line.substring(idx + buttonParam.length(), endIdx);
513.             action.replace("+", " "); // Replace '+' with space
514.             action.trim(); // Remove leading/trailing whitespace
515.
516.             // Remove anything starting with "HTTP"
517.             int httpIdx = action.indexOf("HTTP");
518.             if (httpIdx != -1) {
519.                 action = action.substring(0, httpIdx); // Truncate before "HTTP"
520.             }
521.             action.trim();
522.             // Store the cleaned action
523.             buttonActions[i] = action; // Save the action for the button
524.             Serial.println("Button " + String(i + 1) + " Action: " + buttonActions[i]);
525.         }
526.     }
527. }

```

```

528. // Reset the buttonActions array if necessary
529. for (int i = maxButtons; i < 20; i++) {
530.     buttonActions[i] = ""; // Clear any unused actions
531. }
532. }
533.
534. void startFABI() {
535.     // Close Wifi Connection so BLE can start
536.     WiFi.end();
537.     Serial.println("WiFi off, BLE started");
538.     setup_BLE();
539. }
540.
541. void printWiFiStatus() {
542.     Serial.print("SSID: ");
543.     Serial.println(WiFi.SSID());
544.
545.     IPAddress ip = WiFi.localIP();
546.     Serial.print("IP Address: ");
547.     Serial.println(ip);
548.
549.     Serial.print("To see this page in action, open a browser to http://");
550.     Serial.println(ip);
551. }

```

Source: [42]

Appendix C: Code for Arduino Nano RP2040 Connect

```
1. #include "PluggableUSBHID.h"
2. #include "USBKeyboard.h"
3. #include <SPI.h>
4. #include <WiFiNINA.h>
5. #include <ArduinoBLE.h>
6.
7. //Keyboard
8. USBKeyboard Keyboard;
9.
10. //Wifi
11. bool wifiActive = true;      // To track if WiFi is active
12.
13. char ssid[] = "FABI";        //network SSID name
14. char pass[] = "asterics";    //network password (use for WPA, or use as key for WEP)
15.
16. int status = WL_IDLE_STATUS;
17. WiFiServer server(80);
18.
19. int maxButtons = 2;          // Default value for maximum number of buttons
20. String buttonActions[100];   // Array to store actions for each button
21.
22. //BLE
23. // UUIDs for the service and characteristic
24. const char* PUCK_SERVICE_UUID = "6e400001-b5a3-f393-e0a9-e50e24dcca9e";
25. const char* PUCK_CHARACTERISTIC_UUID = "6e400003-b5a3-f393-e0a9-e50e24dcca9e";
26.
27. BLEDevice puckDevices[100]; //Store up to 100 Pucks
28. BLECharacteristic buttonCharacteristics[100];
29. String puckNames[100];      // Store Puck.js names
30.
31. int puckCount = 0; //How many Puck.js were already found
32.
33. void setup() {
34.   Serial.begin(9600);
35.   while (!Serial);
36.   // put your setup code here, to run once:
37.   pinMode(LED_BUILTIN, OUTPUT);
38.   pinMode(LED_R, OUTPUT);
39.   pinMode(LED_G, OUTPUT);
40.   pinMode(LED_B, OUTPUT);
41.
42.   digitalWrite(LED_R, HIGH);
43.   delay(200);
44.   digitalWrite(LED_R, LOW);
45.   digitalWrite(LED_G, HIGH);
46.   delay(200);
47.   digitalWrite(LED_G, LOW);
48.   digitalWrite(LED_B, HIGH);
49.   delay(200);
50.   digitalWrite(LED_B, LOW);
51.   setup_Wifi();
52. }
53.
54. void loop() {
55.   // put your main code here, to run repeatedly:
56.   while(wifiActive){
57.     loop_Wifi();
58.   }
59.   while(!wifiActive){
60.     loop_BLE();
61.     Serial.println("!");
62.   }
63. }
```



```

64.
65. void setup_BLE(){
66.   digitalWrite(LED_BUILTIN, LOW);
67.   // Start BLE central mode
68.   if (!BLE.begin()) {
69.     Serial.println("Failed to start BLE!");
70.     while (1);
71.   }
72.
73.   Serial.println("BLE Central - Arduino RP2040 Connect");
74.   BLE.scan(); // Start scanning for peripherals
75. }
76.
77. void loop_BLE(){
78.   // If less than maxButtons connected, keep scanning
79.   if (puckCount < maxButtons) {
80.     BLEDevice peripheral = BLE.available();
81.
82.     if (peripheral) {
83.       String deviceName = peripheral.localName();
84.
85.       Serial.print("Found device: ");
86.       Serial.println(deviceName);
87.
88.       // Check if it's advertising the correct service UUID
89.       if (peripheral.advertisedServiceUuid() == PUCK_SERVICE_UUID) {
90.         Serial.println("Found a Puck.js device!");
91.         BLE.stopScan(); // Stop scanning to connect
92.
93.         if (peripheral.connect()) {
94.           Serial.println("Connected to Puck.js");
95.
96.           // Discover the service and characteristic
97.           BLEService puckService = peripheral.service(PUCK_SERVICE_UUID);
98.           if (peripheral.discoverService(PUCK_SERVICE_UUID)) {
99.             BLECharacteristic buttonCharacteristic = peripheral.characteristic(PUCK_CHARAC-
TERISTIC_UUID);
100.
101.             if (!buttonCharacteristic || !buttonCharacteristic.canSubscribe() || !button-
Characteristic.subscribe()) {
102.               Serial.println("Failed to subscribe!");
103.               peripheral.disconnect();
104.               return;
105.             } else {
106.               Serial.println("Subscribed to button notifications");
107.
108.               // Store device and characteristic
109.               puckDevices[puckCount] = peripheral;
110.               buttonCharacteristics[puckCount] = buttonCharacteristic;
111.               puckNames[puckCount] = deviceName;
112.
113.               puckCount++;
114.               Serial.print("Connected to ");
115.               Serial.println(deviceName);
116.             }
117.           } else {
118.             Serial.println("Failed to connect!");
119.           }
120.         }
121.
122.         BLE.scan(); // Continue scanning for more devices
123.       }
124.     }
125.   }
126.   checkForButtonPresses(); //The Keyboard Action
127. }
128.

```

```

129. //Examples can be adjusted if needed
130. #define KEY_ENTER 0x28 // Keyboard Return (ENTER)
131. //define KEY_ENTER 0x1c // Keyboard Return (ENTER)
132. #define KEY_SPACE 0x2c // Keyboard Spacebar
133. #define KEY_RIGHT 0x4f // Keyboard Right Arrow
134. #define KEY_LEFT 0x50 // Keyboard Left Arrow
135. #define KEY_DOWN 0x51 // Keyboard Down Arrow
136. #define KEY_UP 0x52 // Keyboard Up Arrow
137.
138. void buttonMapping(String txt){
139.     if(txt == "KEY_ENTER"){
140.         Keyboard.key_code_raw(KEY_ENTER);
141.         Serial.println("HENLOO");
142.     } else if (txt == "KEY_SPACE"){
143.         Keyboard.key_code_raw(KEY_SPACE);
144.     } else if (txt == "KEY_RIGHT"){
145.         Keyboard.key_code_raw(KEY_RIGHT);
146.     } else if (txt == "KEY_LEFT"){
147.         Keyboard.key_code_raw(KEY_LEFT);
148.     } else if (txt == "KEY_DOWN"){
149.         Keyboard.key_code_raw(KEY_DOWN);
150.     } else if (txt == "KEY_UP"){
151.         Keyboard.key_code_raw(KEY_UP);
152.     }
153. }
154.
155. void checkForButtonPresses() {
156.     // Check for updates from each connected Puck.js device
157.     for (int i = 0; i < puckCount; i++) {
158.         if (puckDevices[i].connected() && buttonCharacteristics[i].valueUpdated()) {
159.             byte value = 0;
160.             buttonCharacteristics[i].readValue(value);
161.
162.             // Check if the left button (second bit) was pressed
163.             if (value & 0x02) {
164.                 Serial.print(puckNames[i]);
165.                 Serial.println(" button pressed");
166.
167.                 // Print the corresponding button action to USB using c_str()
168.                 String buttonText = buttonActions[i].c_str();
169.                 if (buttonText.startsWith("KEY_")){
170.                     buttonMapping(buttonText);
171.                     digitalWrite(LED_BUILTIN, HIGH); //RGB LED not compatibl with BLE
172.                     delay(200);
173.                     digitalWrite(LED_BUILTIN, LOW);
174.                 } else {
175.                     Keyboard.printf(buttonActions[i].c_str()); // Convert String to const char*
176.                     Serial.println(buttonText);
177.                     digitalWrite(LED_BUILTIN, HIGH);
178.                     delay(200);
179.                     digitalWrite(LED_BUILTIN, LOW);
180.                 }
181.             }
182.         }
183.     }
184. }
185.
186. void setup_Wifi() {
187.     Serial.println("Access Point Web Server");
188.     digitalWrite(LED_BUILTIN, HIGH);
189.     // Check for the WiFi module:
190.     if (WiFi.status() == WL_NO_MODULE) {
191.         Serial.println("Communication with WiFi module failed!");
192.         while (true);
193.     }
194.
195.     // Create access point

```

```

196.     Serial.print("Creating access point named: ");
197.     Serial.println(ssid);
198.     status = WiFi.beginAP(ssid, pass);
199.     if (status != WL_AP_LISTENING) {
200.         Serial.println("Creating access point failed");
201.         while (true);
202.     }
203.
204.     delay(10000); // Wait for connection
205.
206.     // Start the web server on port 80
207.     server.begin();
208.
209.     // You're connected now, so print out the status
210.     printWiFiStatus();
211. }
212.
213. void loop_Wifi() {
214.     WiFiClient client = server.available(); // listen for incoming clients
215.     if (client) { // if you get a client
216.         Serial.println("New client connected");
217.         String currentLine = ""; // Stores current line of HTTP request
218.         String request = ""; // Stores full HTTP request line
219.         bool isRequestLine = true; // Flag for first request line
220.
221.         while (client.connected()) { // loop while the client is connected
222.             if (client.available()) { // if there's bytes to read from the cli-
ent
223.                 char c = client.read(); // read a byte
224.                 Serial.write(c); // print it out to the serial monitor
225.
226.                 // Accumulate the current line
227.                 if (c != '\r' && c != '\n') {
228.                     currentLine += c; // Accumulate the HTTP request line
229.                 }
230.
231.                 // If the line is complete (newline), process it
232.                 if (c == '\n') {
233.                     // If this is the request line (first line), store it
234.                     if (isRequestLine) {
235.                         request = currentLine;
236.                         Serial.print("Received Request: ");
237.                         Serial.println(request);
238.                         isRequestLine = false; // Ensure this is only done once
239.                     }
240.
241.                     // If the line is blank, this means the headers are done
242.                     if (currentLine.length() == 0) {
243.                         // Send the HTML response (this part happens for every request)
244.                         client.println("HTTP/1.1 200 OK");
245.                         client.println("Content-type:text/html");
246.                         client.println();
247.
248.                         // Generate the HTML form response
249.                         client.print("<html><body><h1>Configure Bluetooth Buttons</h1>");
250.                         client.print("<form action=\"/save\" method=\"GET\">");
251.                         client.print("<label for=\"maxButtons\">Max Buttons: </label>");
252.                         client.print("<input type=\"number\" id=\"maxButtons\" name=\"max-
Buttons\" value=\"\" + String(maxButtons) + \"\" min=\"1\" max=\"20\"/><br/><br/>");
253.
254.                         // Generate input fields for each button action
255.                         for (int i = 0; i < maxButtons; i++) {
256.                             client.print("<label for=\"button\" + String(i) + \">Button " +
String(i + 1) + " Action: </label>");
257.                             client.print("<input type=\"text\" id=\"button\" + String(i) +
\"\" name=\"button\" + String(i) + \"\" value=\"\" + buttonActions[i] + \"\"/><br/><br/>");
258.                         }

```

```

259.
260.                 client.print("<input type=\"submit\" value=\"Save Configura-
tion\"/></form>");
261.                 client.print("<form action=\"/startBLE\" method=\"GET\">");
262.                 client.print("<input type=\"submit\" value=\"Start
FABI\"/></form>");
263.                 client.println("</body></html>");
264.
265.                 client.println(); // End the response
266.                 break;           // Exit the client loop after sending the re-
sponse
267.             }
268.
269.             // Clear the current line for the next line
270.             currentLine = "";
271.         }
272.     }
273. }
274.
275. // Now that the response is sent, handle the request logic
276. client.stop(); // Close the connection after serving the page
277. Serial.println("Client disconnected");
278.
279. // Handle the GET request (check after the client is disconnected)
280. if (request.startsWith("GET /save?")) {
281.     Serial.println("Processing save request");
282.     parseConfig(request); // Process the configuration form data
283.     Keyboard.key_code_raw(KEY_F5);
284. } else if (request.startsWith("GET /startBLE")) {
285.     Serial.println("Processing Start FABI request");
286.     startFABI();           // Start BLE and disable WiFi
287.     wifiActive = false;    // Mark WiFi as inactive
288.     setup_BLE();          // Setup BLE
289. }
290.
291. // Reset the request variable after processing
292. request = ""; // Clear request to prevent old data from affecting future requests
293. }
294. }
295.
296. void parseConfig(String line) {
297.     // Extract the maximum number of buttons
298.     int idx = line.indexOf("maxButtons=");
299.     if (idx != -1) {
300.         int endIdx = line.indexOf('&', idx);
301.         if (endIdx == -1) endIdx = line.length();
302.         maxButtons = line.substring(idx + 11, endIdx).toInt();
303.     }
304.
305.     // Ensure maxButtons is within a valid range
306.     maxButtons = constrain(maxButtons, 1, 20); // Limit maxButtons
307.
308.     // Extract actions for each button
309.     for (int i = 0; i < maxButtons; i++) {
310.         String buttonParam = "button" + String(i) + "=";
311.         idx = line.indexOf(buttonParam);
312.         if (idx != -1) {
313.             int endIdx = line.indexOf('&', idx);
314.             if (endIdx == -1) endIdx = line.length();
315.             String action = line.substring(idx + buttonParam.length(), endIdx);
316.             action.replace("+", " "); // Replace '+' with space
317.             action.trim(); // Remove leading/trailing whitespace
318.
319.             // Remove anything starting with "HTTP"
320.             int httpIdx = action.indexOf("HTTP");
321.             if (httpIdx != -1) {
322.                 action = action.substring(0, httpIdx); // Truncate before "HTTP"

```

```

323.         }
324.         action.trim();
325.         // Store the cleaned action
326.         buttonActions[i] = action; // Save the action for the button
327.         Serial.println("Button " + String(i + 1) + " Action: " + buttonActions[i]);
328.     }
329. }
330.
331. // Reset the buttonActions array if necessary
332. for (int i = maxButtons; i < 20; i++) {
333.     buttonActions[i] = ""; // Clear any unused actions
334. }
335. }
336.
337. void startFABI() {
338.     // Close Wifi Connection so BLE can start
339.     WiFi.end();
340.     Serial.println("WiFi off, BLE started");
341. }
342.
343. void printWiFiStatus() {
344.     Serial.print("SSID: ");
345.     Serial.println(WiFi.SSID());
346.
347.     IPAddress ip = WiFi.localIP();
348.     Serial.print("IP Address: ");
349.     Serial.println(ip);
350.
351.     Serial.print("To see this page in action, open a browser to http://");
352.     Serial.println(ip);
353. }

```

Source: [42]

Appendix D: Code for Puck.js (for the Connection with Arduino Nano RP2040 Connect)

```
1. // Service UUID and Characteristic UUID (using a valid 16-bit alias)
2. const SERVICE_UUID = '6e400001-b5a3-f393-e0a9-e50e24dcca9e';
3. const CHARACTERISTIC_UUID = '6e400003-b5a3-f393-e0a9-e50e24dcca9e';
4.
5. var state = 1;
6. function press() {
7.   /* switch(state) {
8.     case 0:
9.       state = 1;
10.      LED1.write(1);
11.      break;
12.     case 1:
13.       state = 0;
14.       LED1.write(0);
15.       break;
16.   } */
17.
18.   LED1.write(1);
19.   setTimeout(function(){
20.     LED1.write(0);
21.   }, 100);
22.   sendButtonState();
23. }
24.
25. // Detect button press and release
26. setWatch(press, BTN1, { edge: 'rising', debounce: 50, repeat: true });
27.
28. // Start advertising with specified service
29. NRF.setAdvertising({}, {
30.   name: "Puck.js Button 1",
31.   connectable: true,
32.   services: [SERVICE_UUID] // Ensure the service UUID is included
33. });
34.
35. // Function to send button state over BLE
36. function sendButtonState(state) {
37.   // Advertise with services that notify button press state
38.   NRF.setServices({
39.     '6e400001-b5a3-f393-e0a9-e50e24dcca9e': {
40.       '6e400003-b5a3-f393-e0a9-e50e24dcca9e': {
41.         value: [state],
42.         readable: true,
43.         notify: true
44.       }
45.     }
46.   });
47.   NRF.setAdvertising({}, {
48.     name: "Puck.js Button 2",
49.     connectable: true,
50.     services: [SERVICE_UUID], // Ensure the service UUID is included
51.     manufacturer: 0x0590,
52.     manufacturerData: JSON.stringify({state}),
53.   });
54. }
```

Source: [42]

Appendix E: Mermaid Code for the Flowchart of Figure 3

flowchart TD

```
A[Setup: Serial, Keyboard, LED] --> B[Setup WiFi AP & Web Server]
B --> C[Main Loop]
C --> D{WiFi Active?}
D -- Yes --> E[Handle HTTP Requests]
E --> F{Request: Save Config or Start BLE?}
F -- Save Config --> G[Update Configuration]
F -- Start BLE --> H[Disable WiFi & Setup BLE]
F -- Other --> E
D -- No --> I[Run BLE Loop]

I --> J[Scan for BLE Devices]
J --> K[Connect to BLE Device]
K --> L[Discover Services & Characteristics]
L --> M[Subscribe for Notifications]
M --> N[Receive BLE Notification]
N --> O[Execute Button Action]
```

Appendix F: Link to the Github Repository

<https://github.com/FriedrichKoenig/WirelessButtonforFABI> [42]