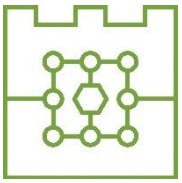
Politechnika Krakowska

im. Tadeusza Kościuszki

Wydział Informatyki i Telekomunikacji



Adrian Chołody

Numer albumu: 144755

Projekt i implementacja programowalnego makro klawiatury wraz z oprogramowaniem

Design and implementation of a programmable macro pad with configuration software

Praca inżynierska

na kierunku Informatyka

Praca wykonana pod kierunkiem:

dr Radosław Kycia

Kraków, 2025

Spis treści

1 Wstęp 4

2 Cel i zakres pracy 5

2.1 Cel pracy 5

2.2 Zakres pracy 5

2.3 Metodyka pracy 5

3 Część teoretyczna 6

3.1 Multiplexing 6

3.2 Przerwania 6

3.3 Deskryptory i raporty 6

3.4 Firmware 8

3.5 Aplikacja desktopowa 8

4 Projekt PCB i obudowy 9

4.1 Wymagania 9

4.2 Komponenty elektroniczne 9

4.3 PCB 10

4.4 Obudowa 14

5 Firmware 17

5.1 Wymagania 17

5.2 Obsługa protokołu HID 17

5.3 Odwzorowanie przycisków i innych elementów wejścia 22

5.4 Odwzorowanie wywoływanych akcji 26

5.5 Główna pętla programu 29

5.6 Serializacja i deserializacja 29

5.7 Komunikacja z aplikacją 32

5.8 Zapis i odczyt z pamięci 33

6 Aplikacja desktopowa 34
6.1 Wymagania 34
6.2 Połączenie z urządzeniem 34
6.3 Okno główne 36
6.4 Dodawanie, edycja i usuwanie akcji 38
6.5 Zapis do urządzenia 39
6.6 Serializacja i deserializacja 39
7 Testy 42
8 Podsumowanie 43
9 Bibliografia 44
10 Spis rysunków 46
11 Spis listingów 47

1 Wstęp

Rynek klawiatur znacząco rozwinął się w ciągu 20 lat. Wraz ze wzrostem czasu spędzanego przed komputerem i ilości wykonywanej na nim pracy, wymagania pewnej grupy użytkowników wzrosły wobec tego urządzenia peryferyjnego. Tą grupą są głównie programiści, graficy 2D i 3D, muzycy i gracze (zarówno zwykli pasjonaci jak i również profesjonalni zawodnicy).

Oczekują oni od tego sprzętu odpowiedniej ergonomii, by nie męczyć rąk podczas wielogodzinnego użytkowania. Chcą by ich klawiatury były niezawodne i przyjemne w korzystaniu. Z tego powodu od kilku lat święcą triumfy rozwiązania wykorzystujące przełączniki ze względu na ich wytrzymałość, responsywność oraz przyjemne dla użytkowników wrażenia związane ze wciskaniem klawiszy. Oprócz tego te grupy oczekują jak największej możliwości dostosowania urządzenia do swoich indywidualnych potrzeb poprzez przypisanie nowych akcji pod dane klawisze.

Standardowa klawiatura posiada określoną liczbę przycisków, gdzie każdy ma określoną rolę. Nie każdy chce jednak pozbywać się jednej funkcjonalności na rzecz drugiej. Z tego powodu w niektórych modelach zaczęto dodawać dodatkowe klawisze z nieprzypisaną na stałe rolą, dedykowane pod przypisanie zupełnie nowych czynności pod ich wciśnięciem. Jednak dla niektórych to i tak było mało. Przez tę potrzebę narodziła nowa kategoria urządzeń, zwanymi makro klawiaturami (ang. macro pad).

Jest to urządzenie znacznie mniejsze od pełnowymiarowej klawiatury, często swoim rozmiarem i ilością przycisków przypominające sekcję klawiszy numerycznych (tzw. numpad). Wszystkie klawisze nie mają określonej akcji na wciśnięcie. Użytkownik może dowolnie je dostosować. Często oprócz samych klawiszy posiadają również takie elementy jak gałki obrotowe, suwaki, czy małe wyświetlacze OLED wyświetlające wybrane informacje z komputera.

Urządzenia i rozwiązania dostępne na rynku nie są pozbawione wad. Największą z nich jest cena, nierzadko przekraczająca 500 zł [1, 2, 3], czyli tyle ile potrafi kosztować wysokiej jakości pełnowymiarowa klawiatura. Część urządzeń nie pozwala na aż tak swobodną konfigurację, jakiej oczekują użytkownicy.

Opisane wady stanowiły motywację do realizacji projektu opisanego w dalszej części dokumentu

2 Cel i zakres pracy

2.1 Cel pracy

Celem niniejszej pracy inżynierskiej jest zaprojektowanie i wdrożenie programowalnej makro klawiatury, która umożliwia przypisanie różnych akcji do poszczególnych przycisków. Projekt obejmuje zarówno opracowanie sprzętowe urządzenia, jak i implementację dedykowanego oprogramowania wbudowanego (ang. firmware) oraz aplikację desktopową, pozwalającą na intuicyjną konfigurację akcji, takich jak skróty klawiszowe, uruchamianie programów, czy skrypty automatyzujące pracę. Makro klawiatura komunikuje się z komputerem poprzez interfejs USB, działając jako urządzenie wejściowe, które pozwala na personalizację akcji przypisanych do każdego przycisku przez użytkownika. Opracowane rozwiązanie znajduje zastosowanie w wielu obszarach, m.in. w programowaniu, obróbce multimediów, modelowaniu 3D, grach komputerowych oraz obróbce audio, gdzie automatyzacja i optymalizacja procesów są kluczowe dla efektywności pracy.

2.2 Zakres pracy

Zakres pracy obejmuje zaprojektowanie fizycznego urządzenia, co wiąże się z przygotowaniem schematu PCB z elementami uwzględnionymi w wymaganiach oraz modelu 3D obudowy, który posłuży do jej wyprodukowania. Następnym etapem jest przygotowanie firmware'u dla urządzenia, który umożliwi komunikację za pomocą protokołu HID oraz konfigurację w locie, bez konieczności wgrywania nowego programu. Ostatnim etapem jest opracowanie aplikacji na komputery z systemem Windows, która pozwoli na konfigurację klawiatury oraz integrację obu programów, aby można było płynnie przypisywać nowe ustawienia

2.3 Metodyka pracy

Metodyka pracy opiera się na podzieleniu projektu na 3 części: zaprojektowanie i konstrukcja fizycznego urządzenia, stworzenie firmware'u opracowanie aplikacji komputerowej do konfiguracji. Dla każdego z etapów zostaną określone wymagania jakie muszą spełnić, by uznać projekt za skończony. Następnie na podstawie wymagań każdy etap zostanie opracowany. Na koniec zostaną przeprowadzone testy sprawdzające integrację wszystkich komponentów.

3 Część teoretyczna

3.1 Multiplexing

Większość mikrokontrolerów ma ograniczoną liczbę pinów możliwych do wykorzystania, a jeśli mają dużo złącz wejść/wyjść (np. powyżej 20), to ich cena również jest wyższa. Dlatego przy projektowaniu płytki należy wziąć pod uwagę to ograniczenie. Jednym z rozwiązań jest multiplexing, czyli przesył danych za pomocą mniejszej liczby kanałów. Zwykle do multiplexingu cyfrowego stosuje się specjalne układy scalone, nazywane multiplekserami. Zamieniają one kilka równoległych linii sygnałów na jedną, która przesyła zakodowany zestaw stanów. Następnie przesyłane są do demultipleksera, który odwraca proces. W przypadku tej pracy zostanie wykorzystana podobna technika, nazywana charlieplexingiem. Polega ona na połączeniu elementów, w tym przypadku przycisków, w siatkę ścieżek, a następnie umiejętne zarządzanie

3.2 Przerwania

Do implementacji obsługi enkodera należy wykorzystać przerwania sprzętowe. Gdy zostanie wykryte przerwanie, zostanie wywołana określona funkcja w kodzie, aby sprawdzić czy doszło do obrotu i w którą stronę. Dlatego mikrokontroler musi wspierać przerwania dla pinów. Wybrany mikrokontroler obsługuje je na wszystkich pinach, jednak nie mogą być włączone przerwania jednocześnie dla pinu 5 i 7. Należy wziąć to pod uwagę podczas projektowania PCB [4]

3.3 Deskryptory i raporty

Jedną z pierwszych rzeczy, jakie mikrokontroler musi zrobić po podłączeniu do komputera, jest poinformowanie hosta o tym że jest urządzeniem wejścia. Ale musi również poinformować jakim urządzeniem wejścia jest. Czy jest myszką, czy jest klawiaturą. Musi przekazać komputerowi informację jaki rodzaj danych przekazuje i w jakiej kolejności. Do tego służą deskryptory

W urządzeniach klasy HID wyróżniamy 3 rodzaje deskryptorów:

- Deskryptor HID – określa długość i typy deskryptorów używanych przez urządzenie,
- Deskryptor raportu – opisuje strukturę przesyłanych danych z akcjami wykonanymi na urządzeniu,
- Deskryptor fizyczny – informuje która część ludzkiego ciała aktywuje daną kontrolkę. Jest on opcjonalny [5, p. 22]

Dla tego projektu najistotniejszy jest deskryptor raportu. To on opisuje jakie rodzaje sygnałów wejścia i wyjścia obsługuje klawiatura. Definiuje również strukturę danych wysyłanych do komputera.

```
const uint8_t keyboardDescriptorReport[] PROGMEM = {
    0x05, 0x01,                     /* USAGE_PAGE (Generic Desktop)      47
*/
    0x09, 0x06,                     /* USAGE (Keyboard) */
    0xa1, 0x01,                     /* COLLECTION (Application) */
    0x85, KEYBOARD_REPORT_ID,    /*   REPORT_ID */
    0x05, 0x07,                     /*   USAGE_PAGE (Keyboard) */

    /* Keyboard Modifiers (shift, alt, ...) */
    0x19, 0xe0,                     /*   USAGE_MINIMUM (Keyboard LeftControl)
*/
    0x29, 0xe7,                     /*   USAGE_MAXIMUM (Keyboard Right GUI)
*/
    0x15, 0x00,                     /*   LOGICAL_MINIMUM (0) */
    0x25, 0x01,                     /*   LOGICAL_MAXIMUM (1) */
    0x75, 0x01,                     /*   REPORT_SIZE (1) */
    0x95, 0x08,                     /*   REPORT_COUNT (8) */
    0x81, 0x02,                     /*   INPUT (Data,Var,Abs) */

    /* 104 Keys as bitmap */
    0x19, 0x00,                    /*   Usage Minimum (0) */
    0x29, KEYBOARD_KEY_COUNT - 1,   /*   Usage Maximum (103) */
    0x15, 0x00,                    /*   Logical Minimum (0) */
    0x25, 0x01,                    /*   Logical Maximum (1) */
    0x75, 0x01,                    /*   Report Size (1) */
    0x95, KEYBOARD_KEY_COUNT,       /*   Report Count (104) */
    0x81, 0x02,                    /*   Input (Data, Variable, Absolute) */

    /* End */
    0xC0                            /*   End Collection */
};
```

Listing 3.1 Deskryptor raportu klawiatury [6]Listing 3.2 Struktura przechowująca raport dla klawiatury [6]

Deskryptor zawiera informacje z jakich stron użycia korzysta urządzenie, z których konkretnie identyfikator użycia oraz opisuje w jaki sposób ustrukturyzowane są dane wysyłane w raporcie. W przykładzie Listing 3.1 informuje że korzysta ze strony użycia typu Generic Desktop, który również jest kolekcją zastosowań. Wykorzystywanym zastosowaniem jest użycie typu Keyboard. Następnie tworzona jest kolekcja, która zawiera różne zakresy identyfikatorów użycia oraz opisuje sposób zapisu danych. W tym przypadku oba podzbiory są przekazywane jako bitmapy. Jest przypisany identyfikator raportu, który musi być unikalny dla każdego raportu w urządzeniu. [5, p. 15]

Tabele użycia określają również typy danych jakie są przesyłane przy użyciu HID. W zależności, czy celem jest przesłanie wciśnięcia klawisza, przesunięcia myszki czy odczyt z żyroskopu w kontrolerze wirtualnej rzeczywistości, wysłany zostanie inny rodzaj zmiennej. Tabele opisują również kody użycia, określające, co jest wysyłane [7]

3.4 Firmware

Firmware zostanie napisany w języku C++. Wybór jest podyktowany popularnością tego języka w rozwiązaniach typu embedded, wieloma dostępnymi bibliotekami i szybkością działania. Dodatkowo zostanie użyty framework Arduino, który znacznie ulatwia pracę z mikrokontrolerami, dostarczając wiele narzędzi oraz upraszczając wiele powtarzalnych elementów. Pozwoli to na wykorzystanie biblioteki „hid.h”, która jest kluczowa dla działania programu. W skład Arduino wchodzi również gotowa klasa do komunikacji przy użyciu protokołu Serial. Zostanie on użyty do komunikacji z aplikacją desktopową [8]. Dopełnieniem tego będzie biblioteka „ArduinoJson”, która uprości proces serializacji obiektów w celu wczytywania i udostępniania konfiguracji urządzenia. Jest ona stworzona specjalnie pod mikrokontrolery, co pozwala na dobre wykorzystanie ograniczonych zasobów. [9]

3.5 Aplikacja desktopowa

Do stworzenia aplikacji desktopowej zostanie wykorzystany język C# oraz .NET8. Wybór jest podyktowany znajomością tego języka przez autora, posiadaniem rozbudowanego modułu tworzenia interfejsów użytkownika, jakim jest WPF (Windows Presentation Foundation) oraz mnogością dostępnych bibliotek z repozytorium NuGet. Ponadto język ten natywnie zawiera takie moduły jak moduł komunikacji Serial oraz moduł serializacji do pliku JSON.

4 Projekt PCB i obudowy

4.1 Wymagania

Wymaganie funkcjonalne:

- Obudowa urządzenia powinna być możliwa do stworzenia za pomocą drukarki 3D
- Urządzenie powinno posiadać 16 przycisków i 1 enkoder obrotowy
- Urządzenie powinno łączyć się z komputerem za pomocą USB

Wymaganie niefunkcjonalne

- Koszt stworzenia pojedynczego urządzenia powinien być znacząco niższy od alternatywnych, komercyjnych rozwiązań na rynku
- Konstrukcja urządzenia powinna być jak najprostsza, co pozwoli na stworzenie go bez posiadania drogiego, specjalistycznego sprzętu
- Komponenty elektroniczne konieczne do stworzenia urządzenia powinny być łatwo dostępne do kupienia w sklepach poświęconych elektronice dla hobbystów oraz pasjonatów klawiatur
- Obudowa powinna być minimalistyczna

4.2 Komponenty elektroniczne

Do wykonania klawiatury wybrano następujące komponenty:

- Mikrokontroler – Seeeduino Xiao SAMD21 [4]
- Przełączniki – Cherry MX lub inne pasujące kształtem oraz sposobem wykrywania wciśnięcia (stykowe) [10]
- Diody Shotky'ego BAT85 PHI 0,2A / 30V 4NS
- Enkoder obrotowy EC11

Seeeduino Xiao SAMD21 to kompaktowa płytka rozwojowa, która obsługuje przerwania na każdym z pinów, ma wbudowany port USB-C oraz jest wspierana przez framework Arduino. To ostatnie jest ważne, ponieważ firmware jest oparty o ten framework, w szczególności o bibliotekę „hid.h". Dla przykładu podobna płytka rozwojowa, Seeed Studio ESP32-S3 teoretycznie też jest wspierana przez Arduino, ale wspomniana wyżej biblioteka nie działa dla

mikrokontrolerów z rodziny ESP32. Oprócz tego Seeeduino Xiao SAMD21 posiada sporą jak na mikrokontrolery pamięć i moc obliczeniową, co jest kluczowe dla pomieszczenia programu sterującego oraz by reakcja na wciśnięcia użytkownika była jak najszybsza.

Diody Shotky'ego zostały zastosowane z dwóch powodów. Po pierwsze, chronią urządzenie przed tzw. ghostingiem, czyli sytuacją w której urządzenie źle rejestruje które klawisze są wciśnięte w przypadku wciśnięcia wielu na raz. Po drugie diody pozwalają na ograniczenie ilości pinów potrzebnych to obsługi macierzy przycisków, co jest dokładniej wyjaśnione dalej.

Przełączniki Cherry MX są jednymi z najpopularniejszych na rynku. Uwzględnienie ich w projekcie pozwala na dostosowanie typów klawiszy pod gusta użytkownika oraz wykorzystanie zamienników w tym samym formacie

4.3 PCB

Projekt zakładał by PCB było tanie w produkcji i proste do montażu w domu. Dlatego też płytka zawiera jak najmniej przelotek, powierzchnia płytki jest dobrze wykorzystana pod względem rozmieszczenia elementów montażowych. Projekt nie wykorzystuje żadnych komponentów typu SMD lub LGA, które wymagałyby specjalistycznych narzędzi jak stół grzewczy oraz by uprościć proces składania urządzenia dla majsterkowiczów.

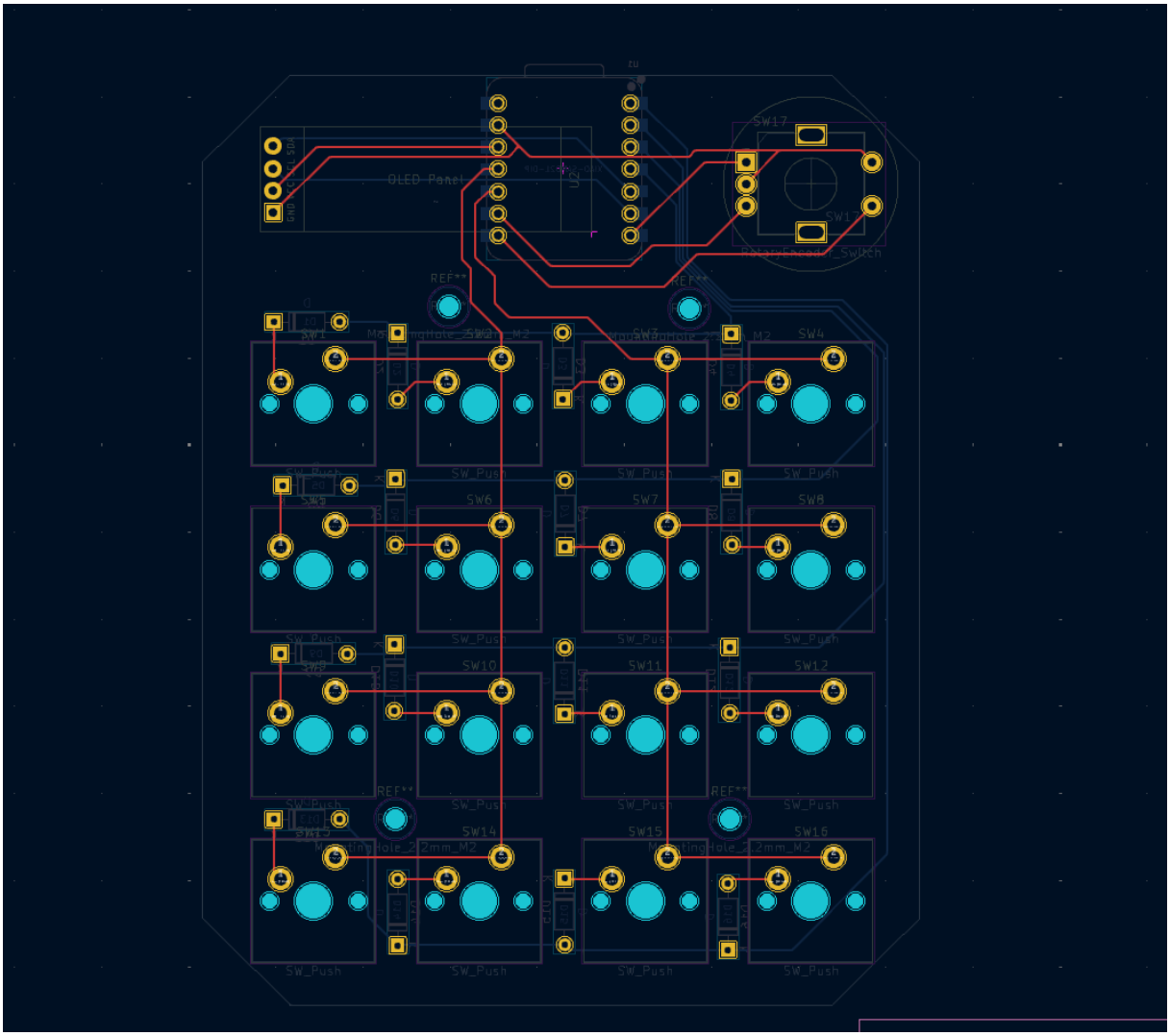
Odstęp pomiędzy przełącznikami wynosi 19.05 mm względem wspólnego punktu odniesienia, zarówno w osi X oraz Y.

Mikrokontroler został umieszczony na brzegu płytki, aby złącza USB-C wystawało poza obrys PCB i można było z jego korzystać. Ten zabieg upraszcza złożoność płytki poprzez brak konieczności umieszczenia osobnego złącza USB-C. Przez to nie trzeba poświęcać pinów w mikrokontrolerze na obsługę dodatkowego portu. Również montaż zewnętrznego portu wymagałby hot-aira z regulacją temperatury i wprawy w lutowaniu drobnych elementów.

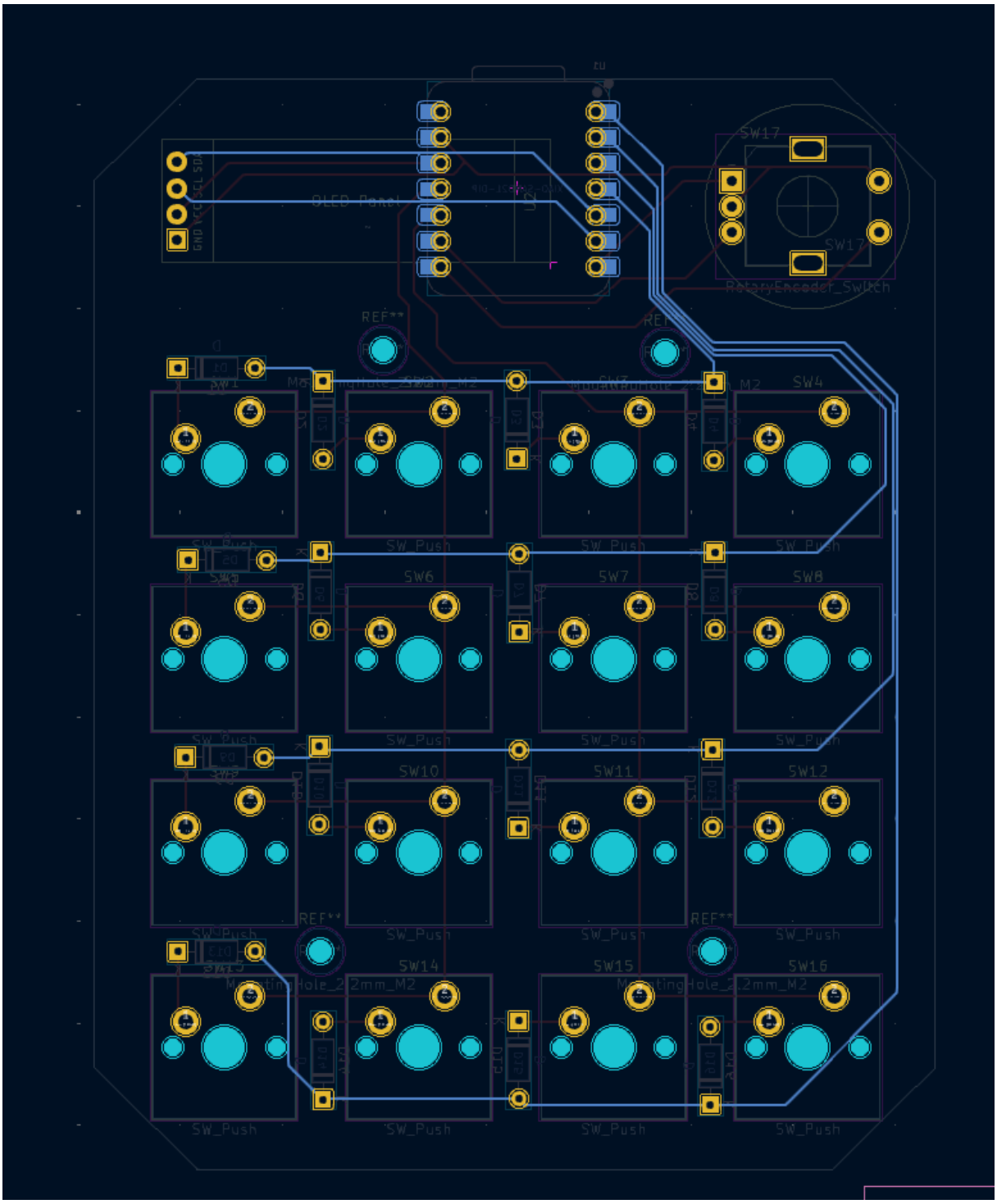
Na górnej stronie płytki zostały osadzone przełączniki oraz enkoder. Na dolnej stronie umieszczono diody oraz mikrokotroler. Ta decyzja jest podyktowana tym, by diody nie kolidowały z nakładką przycisku podczas kliknięcia klawisza. Umieszczenie mikrokontrolera na dolnej stronie pozwoliło na lepsze umieszczenie złącza USB w obudowie oraz lepszego poprowadzenia ścieżek między komponentami.

Do każdego z wierszy przycisków idzie ścieżka, co przedstawia Rys. 4.2. Każdy przycisk jest podłączony do ścieżki poziomej poprzez diodę Shotky'ego, o odpowiedniej polaryzacji, obróconej o 180 stopni co każdą kolumnę. Następnie każda para kolumn przycisków jest połączona pojedynczą ścieżką, która wraca do mikrokontrolera. Widać to na Rys. 4.1.

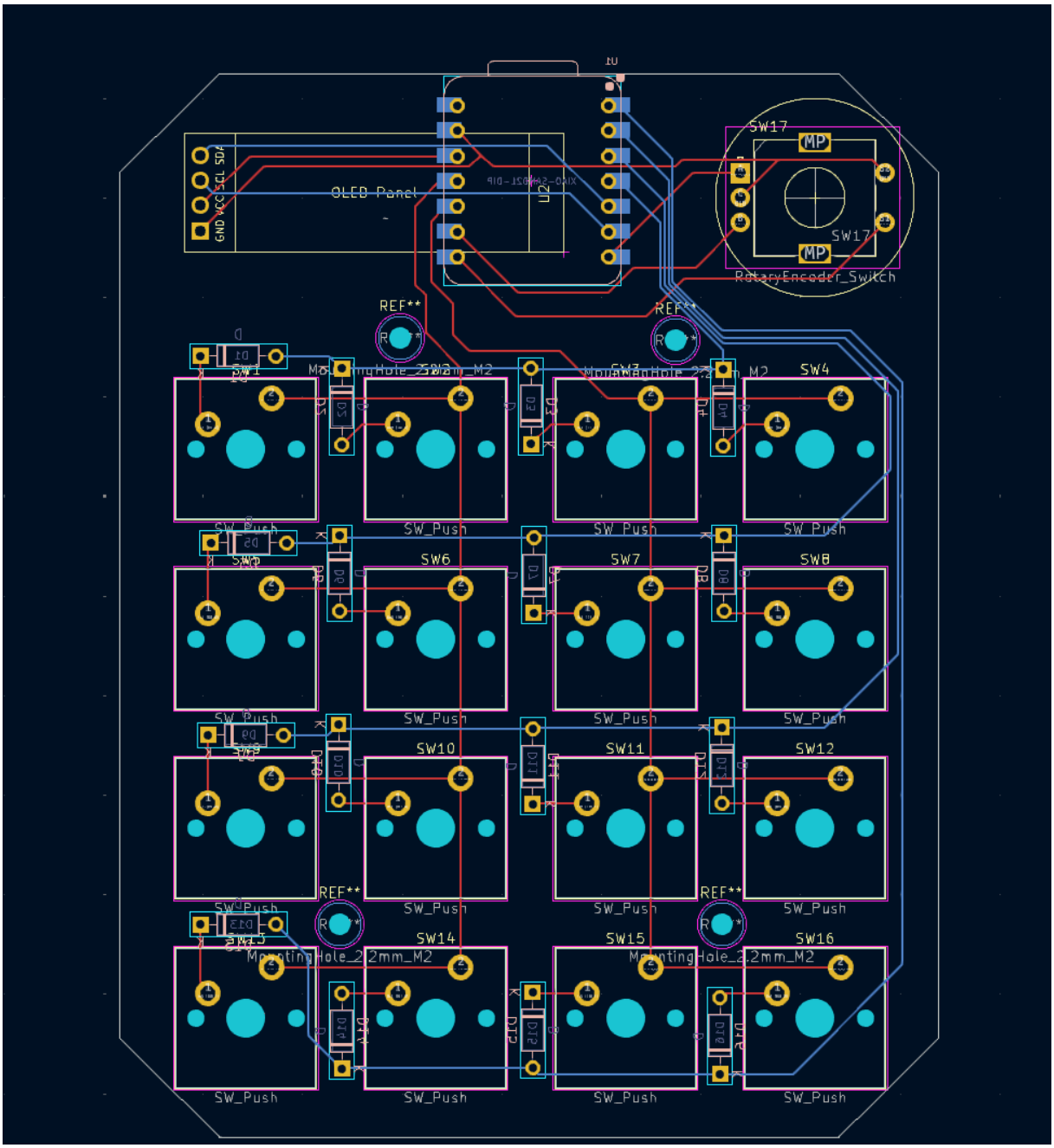
Zastosowany na Rys. 4.3 układ diod Shotky'ego pozwolił na ograniczenie wymaganych pinów do połączenia kolumn przycisków z 4 do 2. Oszczędność ta pozwoliła na wygospodarowanie 2 wolnych pinów które mogą posłużyć do komunikacji I^2C lub do debugowania oprogramowania sprzętowego.



Rys. 4.1 PCB, uwydatnione ścieżki górne [opracowanie własne]

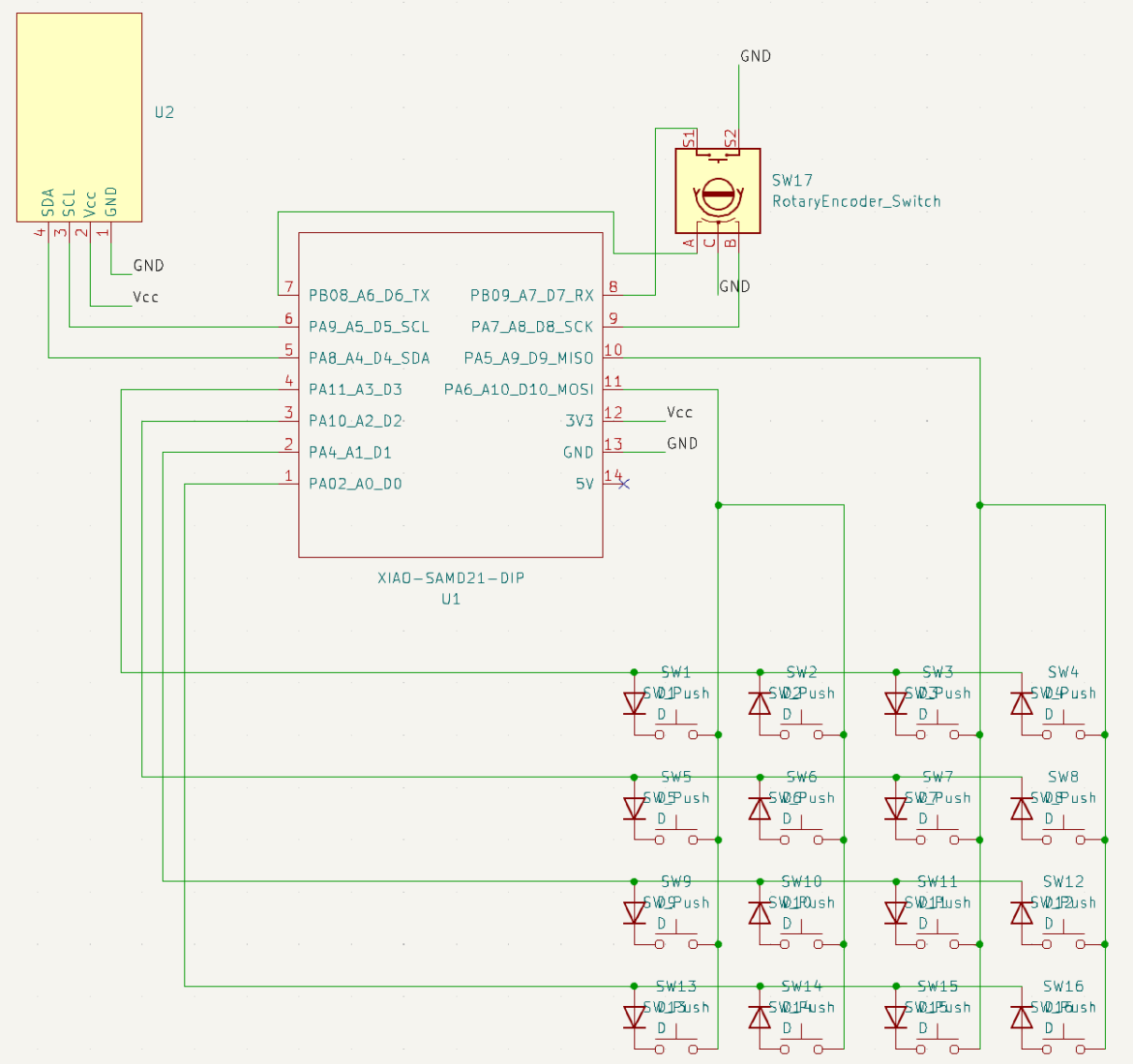


Rys. 4.2 PCB, uwydatnione ścieżki dolne [opracowanie własne]



Rys. 4.3 PCB, wszystkie elementy widoczne [opracowanie własne]

Rys. 4.4 przedstawia schemat połączeń poszczególnych komponentów.



Rys. 4.4 Schemat elektroniczny urządzenia

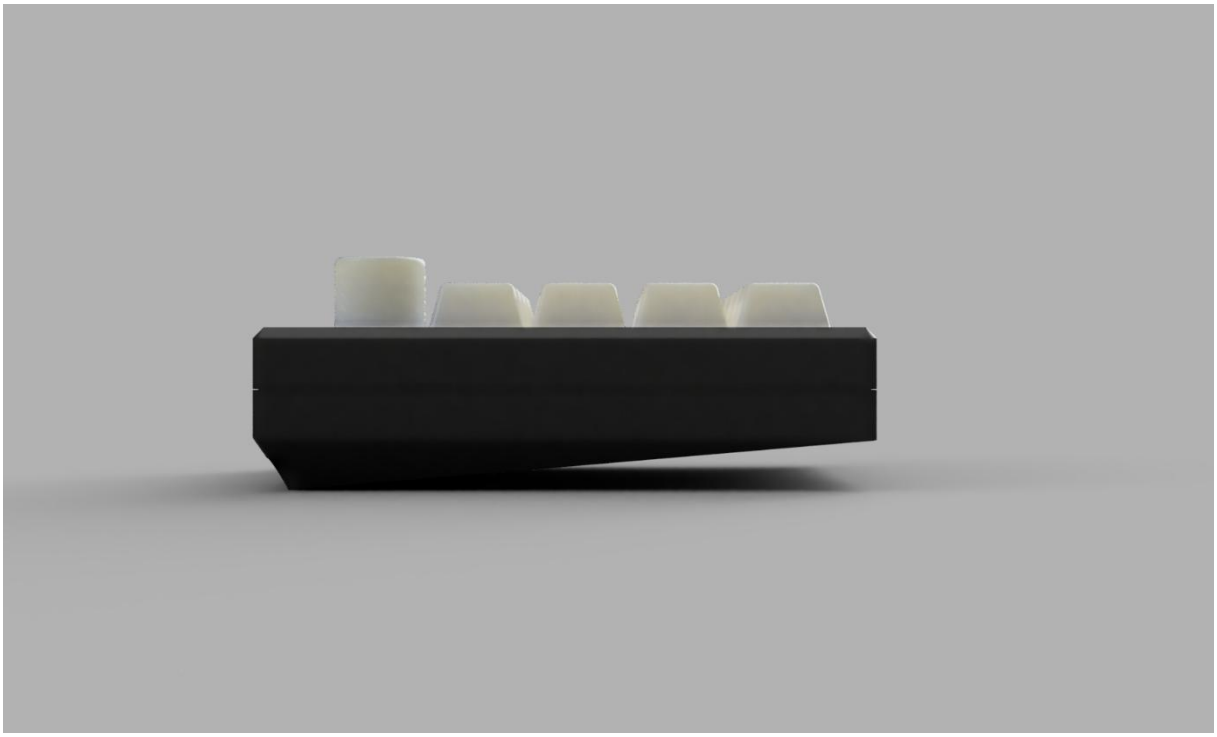
4.4 Obudowa

Obudowa została zaprojektowana wokół PCB. Głównym założeniem było to, aby można było ją stworzyć przy użyciu drukarki 3D. Dlatego też została zaprojektowana dla tej metody produkcyjnej. Obudowa została podzielona na 2 części.

Część górna zawiera wycięcia na klawisze oraz enkoder. Wewnątrz, w jej rogach znajdują się otwory do wkręcenia śrub łączących obie części

Część dolna zawiera otwory do łączenia obu stron. W niej również jest osadzona płytka drukowana. Z tyłu dolnej części znajduje się otwór przelotowy dla USB-C. W środku znajdują się 4 otwory na podwyższeniu do umocowania PCB

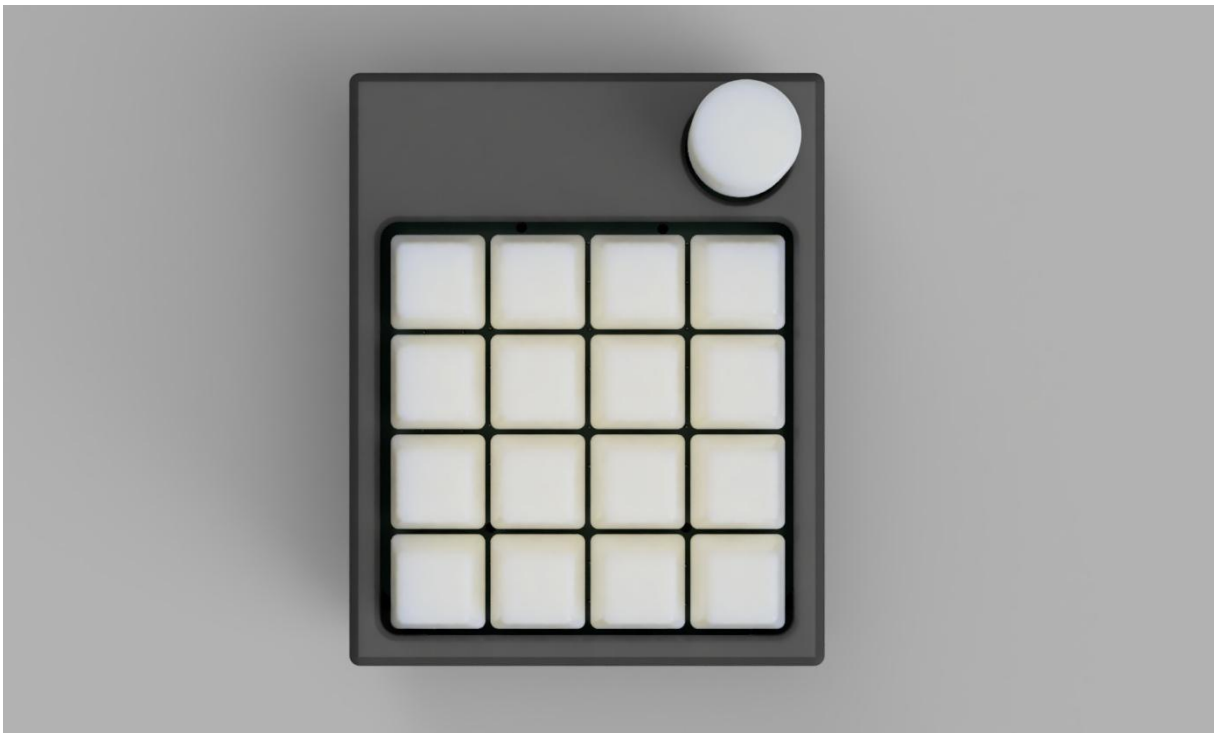
Podstawa została ścięta u spodu, przez co klawisze są pod kątem, co jest bardziej ergonomiczne dla użytkownika. Widać to na Rys. 4.5



Rys. 4.5 Urządzenie z boku [opracowanie własne]

Krawędzie zewnętrzne obudowy zostały zaokrąglone lub ścięte, co ułatwia proces drukowania oraz sprawia, że urządzenie nie rani użytkownika.

Rys. 4.6 przedstawia, jak urządzenie wygląda z góry. [10]



Rys. 4.6 Render urządzenia, widok z góry [opracowanie własne]

5 Firmware

5.1 Wymagania

Wymaganie funkcjonalne:

- Urządzenie powinno zapisywać konfigurację w pamięci nieulotnej, aby nie wymagało ciągłego połączenia z aplikacją desktopową.
- Urządzenie po kliknięciu przycisku lub obrocie gałki enkodera powinno wykonywać akcję lub ciąg akcji na komputerze.
- Urządzenie powinno obsługiwać takie akcje jak kliknięcia przycisków standardowej klawiatury, sterowanie mediami i systemem.

Wymaganie niefunkcjonalne

- Biblioteka pozwalająca stworzyć firmware dla urządzenia, powinna być jak najbardziej elastyczna pod względem konfiguracji oraz pozwalać na łatwe zaimplementowanie dodatkowych funkcjonalności poprzez rozszerzenie istniejących klas.
- Firmware powinien bezpiecznie obsłużyć przypadki nieudanego transferu konfiguracji.
- Firmware powinien pozwalać na korzystanie z urządzenia bez ciągłego połączenia z aplikacją desktopową.

Firmware składa się z 3 rodzajów klas

- Klasy odpowiedzialne za komunikację przy użyciu protokołu HID
- Klasy opisujące akcje
- Klasy opisujące fizyczne kontrolki w urządzeniu

5.2 Obsługa protokołu HID

W projekcie zastosowano 3 deskryptory. Pierwszy z nich odpowiada za wysyłanie sygnałów powiązanych z klawiaturą, takie jak litery, klawisze funkcyjne itp. Drugi odpowiada za sygnały takie jak kontrola głośności, jasności ekranu i usypianie. Trzeci odpowiada korzysta z usage page typu System, która odpowiada m.in. za wyłączanie i usypianie urządzenia.

Do obsługi wysyłania deskryptora oraz raportów została stworzona klasa abstrakcyjna „HIDDevice_”, widoczna na Listing 5.1. Na jej podstawie wszelkie inne klasy które

odpowiadają za wysyłanie. Klasy pochodne zawierają deskryptor oraz strukturę opisującą wysyłane dane dla tego deskryptora

```
class HIDDevice_ {
public:
    uint8_t* report;

    HIDDevice_(const uint8_t* descriptorReport, int descriptorSize, unsigned
int reportID, uint8_t* report, int reportSize);
    virtual void clearReport();

    //sends report to host
    int send();
protected:
    unsigned int reportID;

    // need to be assigned in child class
    int reportSize;

    const uint8_t* descriptorReport;

    int descriptorSize;
};
HIDDevice_::HIDDevice_(const uint8_t* descriptorReport, int descriptorSize,
unsigned int reportID, uint8_t* report, int reportSize) {
    this->descriptorReport = descriptorReport;
    this->descriptorSize = descriptorSize;
    this->reportID = reportID;
    this->report = report;
    this->reportSize = reportSize;
    static HIDSubDescriptor node(descriptorReport, descriptorSize);
    HID().AppendDescriptor(&node);
    this->clearReport();
}

int HIDDevice_::send() {
    return HID().SendReport(reportID, report, reportSize);
}

void HIDDevice_::clearReport() {
    // empty implementation
}
```

Listing 5.1 Klasa "HIDDevice_" [opracowanie własne]

Klasa w konstruktorze zapisuje wskaźnik do desktryptora, jego rozmiar, identyfikator reportu, wskaźnik do zmiennej przechowującej raport oraz jego rozmiar. W metodzie „send” następuje

wysłanie raportu przy użyciu klasy z biblioteki „hid.h”. Metoda wirtualna „clearReport” służy do wyczyszczenia raportu.

Listing 5.2 przedstawia implementację dziedziczenia klasy „HIDDevive_” na przykładzie klasy „Keyboard_”

```
struct ATTRIBUTE_PACKED {
    uint8_t modifiers;
    uint8_t keys[KEYBOARD_KEY_COUNT / 8];
} KeyboardReport;

const uint8_t keyboardDescriptorReport[] PROGMEM = {
    0x05, 0x01,                     /* USAGE_PAGE (Generic Desktop)      47
*/
    0x09, 0x06,                     /* USAGE (Keyboard) */
    0xa1, 0x01,                     /* COLLECTION (Application) */
    0x85, KEYBOARD_REPORT_ID,    /*   REPORT_ID */
    0x05, 0x07,                     /*   USAGE_PAGE (Keyboard) */

    /* Keyboard Modifiers (shift, alt, ...) */
    0x19, 0xe0,                     /*   USAGE_MINIMUM (Keyboard LeftControl)
*/
    0x29, 0xe7,                     /*   USAGE_MAXIMUM (Keyboard Right GUI)
*/
    0x15, 0x00,                     /*   LOGICAL_MINIMUM (0) */
    0x25, 0x01,                     /*   LOGICAL_MAXIMUM (1) */
    0x75, 0x01,                     /*   REPORT_SIZE (1) */
    0x95, 0x08,                     /*   REPORT_COUNT (8) */
    0x81, 0x02,                     /*   INPUT (Data,Var,Abs) */

    /* 104 Keys as bitmap */
    0x19, 0x00,                    /*   Usage Minimum (0) */
    0x29, KEYBOARD_KEY_COUNT - 1,   /*   Usage Maximum (103) */
    0x15, 0x00,                    /*   Logical Minimum (0) */
    0x25, 0x01,                    /*   Logical Maximum (1) */
    0x75, 0x01,                    /*   Report Size (1) */
    0x95, KEYBOARD_KEY_COUNT,      /*   Report Count (104) */
    0x81, 0x02,                    /*   Input (Data, Variable, Absolute) */

    /* End */
    0xC0                            /*   End Collection */
};

Keyboard_::Keyboard_(void) : HIDDevice_(keyboardDescriptorReport,
sizeof(keyboardDescriptorReport), (unsigned int)KEYBOARD_REPORT_ID,
reinterpret_cast<uint8_t*>(&KeyboardReport), sizeof(KeyboardReport))
```

```
{
    report = reinterpret_cast<uint8_t*>(&KeyboardReport);
    removeAll();
    send();
}

void Keyboard_::clearReport(void)
{
    removeAll();
}

int Keyboard_::add(KeyboardKeycode key)
{
    // Add key to report
    return set(key, true);
}
int Keyboard_::remove(KeyboardKeycode key)
{
    // Remove key from report
    return set(key, false);
}

int Keyboard_::press(KeyboardKeycode key)
{
    int ret = add(key);
    if(ret){
        send();
    }
    return ret;
}

int Keyboard_::release(KeyboardKeycode key)
{
    int ret = remove(key);
    if(ret){
        send();
    }
    return ret;
}

int Keyboard_::releaseAll()
{
    for (uint8_t i = 0; i < sizeof(KeyboardReport.keys); i++)
    {
        KeyboardReport.keys[i] = 0;
    }
    return 1;
```

```
}

int Keyboard_::click(KeyboardKeycode key)
{
    int ret = press(key);
    if(ret){
        release(key);
    }
    return ret;
}

int Keyboard_::set(KeyboardKeycode key, bool s)
{
    if (key < KEYBOARD_KEY_COUNT){
        uint8_t bit = 1 << (uint8_t(key) % 8);
        if(s){
            KeyboardReport.keys[key / 8] |= bit;
        }
        else{
            KeyboardReport.keys[key / 8] &= ~bit;
        }
        return 1;
    }

    // It's a modifier key
    if(key >= KEY_LEFT_CTRL && key <= KEY_RIGHT_GUI)
    {
        // Convert key into bitfield (0 - 7)
        key = KeyboardKeycode(uint8_t(key) - uint8_t(KEY_LEFT_CTRL));
        if(s){
            KeyboardReport.modifiers |= (1 << key);
        }
        else{
            KeyboardReport.modifiers &= ~(1 << key);
        }
        return 1;
    }

    // No empty/pressed key was found
    return 0;
}

int Keyboard_::removeAll(void)
{
    // Release all keys
    for (uint8_t i = 0; i < sizeof(KeyboardReport.keys); i++)
    {
```

```
        KeyboardReport.keys[i] = 0;
    }
    KeyboardReport.modifiers = 0;
    return 1;
}

Keyboard_ Keyboard;
```

Listing 5.2 Klasa "Keyboard_" [6]

Klasa ta zawiera tablicę będącą deskryptorem oraz strukturę opisującą raport. Te są przekazane do konstruktora klasy bazowej. Zaimplementowano metodę wirtualną „clearReport" czyszczącą strukturę i wysyłającą czysty raport. Dodatkowo zaimplementowano metody potrzebne do obsługi danych opisanych w deskryptorze. Na końcu została zadeklarowany obiekt globalny „Keyboard" aby nie musieć przekazywać wskaźnika do akcji. Wartość „reportID" musi być unikalna w obrębie całego firmware'u.

5.3 Odwzorowanie przycisków i innych elementów wejścia

Klasy opisujące fizyczne kontrolki dziedziczą po klasie abstrakcyjnej „PhysicalInput". Klasa ta zawiera metodę wirtualną „invoke" wywołującą przypisane do obiektu akcje.

Klasa „Button" odpowiada za sprawdzenie czy dany przycisk został wciśnięty. Przechowuje dane o pinach które mają być użyte do sprawdzenia przycisku oraz stan który ma być brany jako wciśnięcie. Po wykryciu wciśnięcia wywołuje akcję.

```
Button::Button(int pinA, int pinB, int type, Action* action)
{
    this->pinA = pinA;
    this->pinB = pinB;
    this->action = action;
    this->type = type;
    if (type == INPUT_PULLUP)
    {
        this->pressedState = LOW;
    }
    else
    {
        this->pressedState = HIGH;
    }
}
Button::~Button()
{
    delete action;
```

```
}

int Button::invoke()
{
    pinMode(pinA, type);
    pinMode(pinB, OUTPUT);
    digitalWrite(pinB, pressedState);
    int result = 0;
    int pinRead = digitalRead(pinA);
    if (pinRead == pressedState && wasPressed == 0)
    {
        result =  action->invoke();

        wasPressed = 1;
    }
    else if (pinRead != pressedState && wasPressed == 1)
    {
        wasPressed = 0;
    }
    pinMode(pinA, OUTPUT);
    digitalWrite(pinA, !pressedState);
    digitalWrite(pinB, !pressedState);

    return result;
}
```

Listing 5.3 Klasa "Button" [opracowanie własne]

Klasa „SecuredButton”, widoczna na Listing 5.4, jest dekoratorem dla klasy „Button”. Rozwija ona klasę bazową o to, że metoda „invoke” przed sprawdzeniem wciśnięcia wyłącza inne piny, które mogłyby wpłynąć na błędne odczytanie stanu wciśnięcia. Jest to kluczowe przy używanej w urządzeniu macierzy przycisków.

```
class ButtonSecured : public PhysicalInput
{
    private:
        Button* button;
        int* turnedOffPins;
        int turnedOffPinsSize;

    public:
        int invoke() override;
        ButtonSecured(int pinA, int pinB, int type, Action* action, int*
turnedOffPins, int turnedOffPinsSize);
        ~ButtonSecured();
        std::string serialize();
};
```

```
int ButtonSecured::invoke()
{
    for (int i = 0; i < turnedOffPinsSize; i++)
    {
        pinMode(turnedOffPins[i], OUTPUT);
        digitalWrite(turnedOffPins[i], !button->getPressedState());
    }

    return this->button->invoke();
}
ButtonSecured::~ButtonSecured()
{
    delete button;
}

ButtonSecured::ButtonSecured(int pinA, int pinB, int type, Action* action,
int* turnedOffPins, int turnedOffPinsSize)
{
    button = new Button(pinA, pinB, type, action);
    this->turnedOffPins = turnedOffPins;
    this->turnedOffPinsSize = turnedOffPinsSize;
}
```

Listing 5.4 Klasa "ButtonSecured" [opracowanie własne]

Klasa „Encoder”, przedstawiona na Listing 5.5, odpowiada poprawne interpretowanie czynności wykonywanych na enkoderze. Przechowuje 3 akcje, dla przekręcenia w lewo, w prawo oraz wciśnięcia. Do sprawdzenia czy został wykonany obrót, jest wywoływana metoda „checkRotation” podczas wykrycia przerwania na pinach A oraz B. Ponieważ do przerwania można przypisać jedynie metodę statyczną, należało stworzyć dodatkową metodę „globalCheckRotation” która sprawdza czy został wykonany obrót dla obiektu z tablicy statycznej „Encoder::instances”. Jeżeli tak, w obiekcie jest zapisywana informacja o wykonanym obrocie oraz kierunku i przy następnym wywołaniu „invoke” na obiekcie, zostanie wykonana odpowiednia akcja, w zależności od kierunku. Wciśnięcie przycisku enkodera jest niezależne od przerwań

```
Encoder* Encoder::instances[MAX_ENCODERS] = {nullptr};
int Encoder::instanceCount = 0;

Encoder::Encoder(int pinEncA, int pinEncB, int pinButton, Action* actionLeft,
Action* actionRight, Action* actionButton)
{
    this->pinEncA = pinEncA;
    this->pinEncB = pinEncB;
    this->pinButton = pinButton;
```

```
    this->actionLeft = actionLeft;
    this->actionRight = actionRight;
    this->actionButton = actionButton;

    pinMode(this->pinEncA, INPUT_PULLUP);
    pinMode(this->pinEncB, INPUT_PULLUP);
    pinMode(this->pinButton, INPUT_PULLUP);

    attachInterrupt(this->pinEncA, globalCheckRotation, CHANGE);
    attachInterrupt(this->pinEncB, globalCheckRotation, CHANGE);
    if(instanceCount < MAX_ENCODERS)
    {
        instances[instanceCount] = this;
        instanceCount++;
    }
}

void Encoder::checkRotation()
{

    old_AB <<= 2; // Remember previous state

    if (digitalRead(pinEncA))
        old_AB |= 0x02; // Add current state of pin A
    if (digitalRead(pinEncB))
        old_AB |= 0x01; // Add current state of pin B

    encval += enc_states[(old_AB & 0x0f)];

    if (encval > 3)
    { // Four steps forward
        direction = 1;
        detected = 1;
        encval = 0;
    }
    else if (encval < -3)
    { // Four steps backward
        direction = 0;
        detected = 1;
        encval = 0;
    }
}

void Encoder::globalCheckRotation()
{
    for (int i = 0; i < instanceCount; i++)
```

```
    {
        instances[i]->checkRotation();
    }
}

int Encoder::invoke()
{
    if (detected)
    {
        if (direction)
        {
            actionRight->invoke();
        }
        else
        {
            actionLeft->invoke();
        }
        detected = 0;
    }

    if (!digitalRead(pinButton))
    {
        actionButton->invoke();
    }
}
```

Listing 5.5 Klasa "Encoder" [11]

5.4 Odwzorowanie wywoływanych akcji

Każda z akcji dziedziczy po klasie abstrakcyjnej Action, która pełni rolę interfejsu. Klasa ta zawiera jedną metodę wirtualną się invoke.

Klasa „KeyboardAction” przechowuje informację o tym jaki identyfikator użycia ma zostać wysłany oraz czy ma być to wciśnięcie, puszczenie czy kliknięcie przycisku. W widocznym na Listing 5.6 konstruktorze jest zapisywany wskaźnik do metody, która ma zostać wywołana podczas metody „invoke”. Klasa ta wykorzystuje obiekt globalny „Keyboard” do wysyłania

```
class KeyboardAction : public Action
{
private:
    using ActionMethod = int (KeyboardAction::*)();
    KeyState state;
    ActionMethod actionMethod;
    KeyboardKeycode key;
```

```
    int pressKey();
    int releaseKey();
    int clickKey();

public:

    KeyboardAction(KeyboardKeycode key, KeyState state);
    int invoke();

    static KeyboardAction* deserialize(std::string json);
    std::string serialize();
};
KeyboardAction::KeyboardAction(KeyboardKeycode key, KeyState state)
{
    this->key = key;
    this->state = state;
    if (state == KeyState::Press)
    {
        actionMethod = &KeyboardAction::pressKey;
    }
    else if (state == KeyState::Release)
    {
        actionMethod = &KeyboardAction::releaseKey;
    }
    else if (state == KeyState::Click)
    {
        actionMethod = &KeyboardAction::clickKey;
    }
    else
    {
    }
}

int KeyboardAction::invoke()
{
    if (actionMethod)
    {
        return (this->*actionMethod)();
    }
    else
    {
        return -1;
    }
}

int KeyboardAction::pressKey()
{
```

```
    return Keyboard.press(key);
}

int KeyboardAction::releaseKey()
{
    return Keyboard.release(key);
}

int KeyboardAction::clickKey()
{
    return Keyboard.click(key);
}
```

Listing 5.6 Klasa "KeyboardAction" [opracowanie własne]

Klasa “DelayAction” służy jako opóźnienie między akcjami. W obiekcie przechowywany jest czas który ma czekać metoda, podany w milisekundach. Metoda „invoke”, widoczna na Listing 5.7, przy pierwszym wywołaniu zapisuje obecny czas mikrokontrolera i zmienia stan zmiennej „isStarted” na true. Następnie sprawdza czy obecny czas jest mniejszy od czasu startowego plus czas opóźnienia. Jeżeli tak, metoda zwraca 0. W przeciwnym wypadku zmienna „isStarted” jest ustawiana na false i metoda zwraca 1. Mechanizm ten wykorzystuje klasa „MacroAction” która wykonuje pętlę dopóki nie otrzyma wartości 1.

```
class DelayAction : public Action
{
    private:
    unsigned int start;
    bool isStarted = false;
    int duration;
    public:
    DelayAction(int duration);
    int invoke() override;
    static DelayAction* deserialize(std::string json);
    std::string serialize() override;
};
DelayAction::DelayAction(int duration)
{
    this->duration = duration;
}

int DelayAction::invoke()
{
    if (!isStarted)
    {
        start = millis();
        isStarted = true;
```

```
    }
    if (millis() - start < duration)
    {
        return 0;
    }
    isStarted = false;
    return 1;
}
```

Listing 5.7 Klasa "DelayAction" [opracowanie własne]

Klasy „SystemAction” oraz „ConsumerAction” są podobne do „KeyboardAction” z tą różnicą że nie korzystają z obiektu klasy „Keyboard_”, tylko z odpowiadających im obiektów klas dziedziczących po „HIDDevice”.

Klasa „MacroAction” zawiera zbiór obiektów dziedziczących po „Action”. W metodzie „invoke” jest pętla, która wywołuje funkcję o tej samej nazwie dla każdego z obiektów w zbiorze. Pętla przechodzi do następnego obiektu, jeśli „invoke” dla obecnego obiektu zwróci wartość 1.

5.5 Główna pętla programu

Główna pętla programu polega na tym, że jest wywoływana metoda „invoke” dla wszystkich obiektów typu „PhysicalInput”. Do tego celu została utworzona klasa „MacroPadRunner” która przechowuje obiekty typu „ButtonSecured” i „Encoder” oraz posiada metodę „run”, która wywołuje metodę „invoke” dla każdego zapisanego obiektu typu „PhysicalInput”. Klasa używa obiektów klasy „ButtonSecured”, ponieważ urządzenie korzysta z macierzy przycisków. Do sprawdzenia stanu pojedynczego, odpowiednie dwa piny muszą zostać ustawione w tryb „INPUT_PULLUP” i „OUTPUT” o stanie „LOW”. Reszta nieużywanych pinów powinna być ustawiona w tryb „OUTPUT” o stanie „HIGH” by nie zaburzać sprawdzania wciśnięcia.

5.6 Serializacja i deserializacja

Każda z klas dziedziczących po interfejsach „Action” i „PhysicalInput” oraz klasa „MacroPadRunner” zawierają metody do serializacji i deserializacji. Serializacja to proces zapisu stanu obiektu do postaci trwałej, np. pliku formatu JSON lub XML. Deserializacja to proces odwrotny, odtwarza z pliku obiekt do pamięci programu. Oba procesy są wykorzystywane do przesyłania złożonych danych między urządzeniami oraz zapisu stanu

pamięci programu. Listing 5.8 przedstawia przykładową implementację serializacji i deserializacji

```
class KeyboardAction : public Action
{
public:
    static KeyboardAction* deserialize(std::string json);
    std::string serialize();
};
KeyboardAction* KeyboardAction::deserialize(std::string json)
{
    JsonDocument doc;
    deserializeJson(doc, json.c_str());
    return new
KeyboardAction(static_cast<KeyboardKeycode>(static_cast<uint8_t>(doc["key"])),
static_cast<KeyState>(static_cast<uint8_t>(doc["state"])));
}

std::string KeyboardAction::serialize()
{
    JsonDocument doc;
    doc["type"] = "KeyboardAction";
    JsonObject details = doc.createNestedObject("details");
    details["key"] = static_cast<uint8_t>(key);
    details["state"] = static_cast<uint8_t>(state);
    std::string output;
    serializeJson(doc, output);
    return output;
}
```

Listing 5.8 Serializacja i deserializacja klasy "KeyboardAction" [opracowanie własne]

Metoda „deserialize” jest metodą statyczną, która zwraca obiekt na podstawie zawartości pliku JSON. Metoda „serialize” tworzy na podstawie obiektu, na którym została wywołana, ciąg znaków z obiektem zapisanym w JSON. Przykładowy wynik serializacji można zobaczyć na Listing 5.9. Plik ten powstał na podstawie deserializacji obiektu typu „MacroPadRunner”. Obie metody korzystają z obiektów i klas pochodzących z biblioteki ArduinoJson. Została ona stworzona pod efektywne wykorzystanie zasobów przy tworzeniu rozwiązań dla systemów embedded.

```
{
  "buttons": [
    {
      "id": 0,
      "action": {
        "type": "MacroAction",
```

```
      "actions": [
        {
          "type": "KeyboardAction",
          "details": { "state": 3, "key": 4, "modifiers": 0 }
        },
        { "type": "DelayAction", "details": { "duration": 1000 } },
        {
          "type": "KeyboardAction",
          "details": { "state": 3, "key": 5, "modifiers": 0 }
        }
      ]
    }
  },
  {
    "id": 1,
    "action": {
      "type": "SystemAction",
      "details": { "state": 3, "code": 130 }
    }
  },
  { "id": 2, "action": null },
  { "id": 3, "action": null },
  { "id": 4, "action": null },
  { "id": 5, "action": null },
  { "id": 6, "action": null },
  { "id": 7, "action": null },
  { "id": 8, "action": null },
  { "id": 9, "action": null },
  { "id": 10, "action": null },
  { "id": 11, "action": null },
  { "id": 12, "action": null },
  { "id": 13, "action": null },
  { "id": 14, "action": null },
  { "id": 15, "action": null }

],
"encoders": [
  {
    "id": 0,
    "actionLeft": {
      "type": "ConsumerAction",
      "details": { "state": 3, "key": 234 }
    },
    "actionRight": {
      "type": "ConsumerAction",
      "details": { "state": 3, "key": 233 }
    },
```

```
      "actionButton": {
        "type": "ConsumerAction",
        "details": { "state": 3, "key": 226 }
      }
    }
  ]
}
```

Listing 5.9 Przykładowy plik JSON z konfiguracją urządzenia [opracowanie własne]

5.7 Komunikacja z aplikacją

Komunikacja występuje poprzez protokół Serial. Firmware obsługuje kilka komend wykonujących określone akcje. Ich implementacja znajduje się na Listing 5.10. „isCompatible" sprawdza, czy urządzenie z którym próbuję skomunikować się aplikacja desktopowa jest kompatybilna. „getName" zwraca nazwę urządzenia, co pozwala na dopasowanie odpowiedniego okna konfiguracji. „getConfiguration" zwraca zapisany w pamięci urządzenia plik json opisujący kontrolki i akcje. „setConfiguration" uruchamia procedurę zapisu nowej konfiguracji.

```
if (Serial.available())
    {
        String receivedMessage = Serial.readString();
        if (receivedMessage == "isCompatible")
        {
            Serial.println("true");
        }
        else if (receivedMessage == "getName")
        {
            Serial.println(deviceName.c_str());
        }
        else if (receivedMessage == "getConfiguration")
        {
            Serial.println(flashStorage.read().json);
        }
        else if (receivedMessage == "setConfiguration")
        {
            bool lastState = runnerRun;
            runnerRun = false;
            Serial.println("ready");

            StoredData dataToSave;
            memset(dataToSave.json, 0, sizeof(dataToSave.json));
```

```
            strncpy(dataToSave.json, Serial.readString().c_str(),
sizeof(dataToSave.json) - 1);
            dataToSave.json[sizeof(dataToSave.json) - 1] = '\0';
            if (dataToSave.json != "")
            {
                delete runner;

                flashStorage.write(dataToSave);
                Serial.println("completed");
                delay(500);
                NVIC_SystemReset();
            }
            else
            {
                Serial.println("error");
                runnerRun = lastState;
            }
        }
        Serial.flush();
    }
```

Listing 5.10 System komend dla komunikacji protokołem Serial [opracowanie własne]

5.8 Zapis i odczyt z pamięci

Konfiguracja urządzenia w formacie JSON jest zapisywana do nieulotnej pamięci flash mikrokontrolera. Do tego celu została wykorzystana biblioteka FlashStorage. Na podstawie zadeklarowanej w programie struktury, rezerwuje obszar pamięci w taki sposób by nie nadpisać zapisanego kodu. Następnie można tak zapisane dane odczytać po restarcie urządzania. Uwalnia to firmware od konieczności ciągłej komunikacji z aplikacją w celu przesyłania pliku konfigurującego. Po otrzymaniu tej komendy program wysyła sygnał „ready” aby poinformować aplikację, że urządzenie jest gotowe do transferu danych. Po udanym przesłaniu pliku urządzenie przesyła sygnał „completed”, a następnie restartuje się, by załadować nową konfigurację. Jeżeli plik nie zostanie przechwycony, zwróci komunikat „error”.

6 Aplikacja desktopowa

6.1 Wymagania

Wymaganie funkcjonalne:

- Aplikacja powinna umożliwiać użytkownikowi na konfigurację każdego klawisza oraz enkodera
- Urządzenie powinno obsługiwać takie akcje jak kliknięcia przycisków standardowej klawiatury, sterowanie mediami i systemem

Wymaganie niefunkcjonalne

- Aplikacja do konfiguracji powinna umożliwiać łatwą konfigurację urządzenia
- Aplikacja powinna posiadać przejrzysty, prosty interfejs

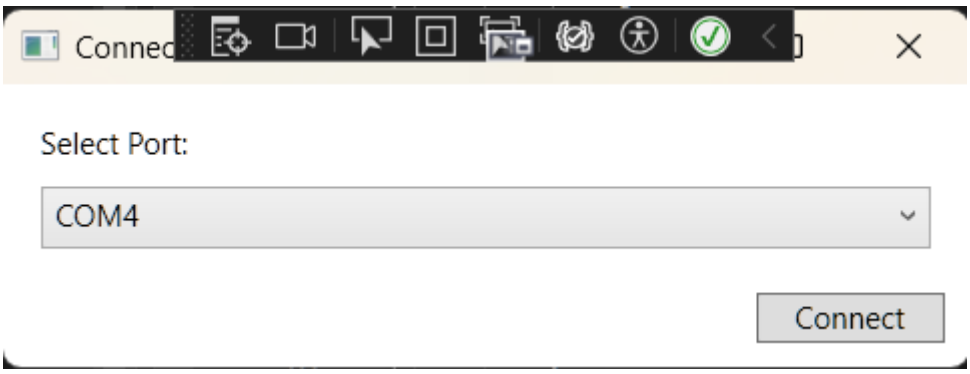
6.2 Połączenie z urządzeniem

Po włączeniu aplikacji ukazuje się okno do połączenia z urządzeniem, widoczne na Rys. 6.1. Na rozwijanej liście znajdują się dostępne porty COM. Po wybraniu i kliknięciu przycisku „Connect” następuje próba połączenia z urządzeniem. Jeżeli próba połączenia przebiegnie pomyślnie, okno połączenia zostanie ukryte i zostanie otwarte okno główne. Listing 6.1 przedstawia implementację tego rozwiązania.

```
private void ConnectButton_Click(object sender, RoutedEventArgs e)
{
    if (_serialPort.IsOpen)
    {
        MessageBox.Show("Port is already open.", "Information",
MessageBoxButton.OK, MessageBoxImage.Information);
        return;
    }

    string selectedPort = PortComboBox.SelectedItem?.ToString();
    if (string.IsNullOrEmpty(selectedPort))
    {
        MessageBox.Show("Please select a port before connecting.", "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
        return;
    }

    try
    {
```

```
        _serialPort.PortName = selectedPort;
        _serialPort.Open();

        _serialPort.Write("isCompatible");
        string response = _serialPort.ReadLine();
        if (response != "true")
        {
            MessageBox.Show("Device is not compatible with this application.",
"Error", MessageBoxButton.OK, MessageBoxImage.Error);
            _serialPort.Close();
            return;
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show($"Failed to connect: {ex.Message}", "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
    }

    if (_serialPort.IsOpen)
    {
        _mainWindow = new MainWindow(_serialPort, this);
        _mainWindow.Show();
        this.Hide();
    }
}
```

Listing 6.1 Funkcja łądzenia aplikacji z urządzeniem [opracowanie własne]



Rys. 6.1 Okno do połączenia z urządzeniem [opracowanie własne]

W trakcie działania głównego okna, za pomocą funkcji z Listing 6.2, w tle jest sprawdzane, czy połączenie z urządzeniem nadal trwa. Jeśli nie, następuje próba odnowienia połączenia. Jeżeli próba zakończy się niepowodzeniem, okno główne zostanie zamknięte, a okno połączenia zostanie odkryte.

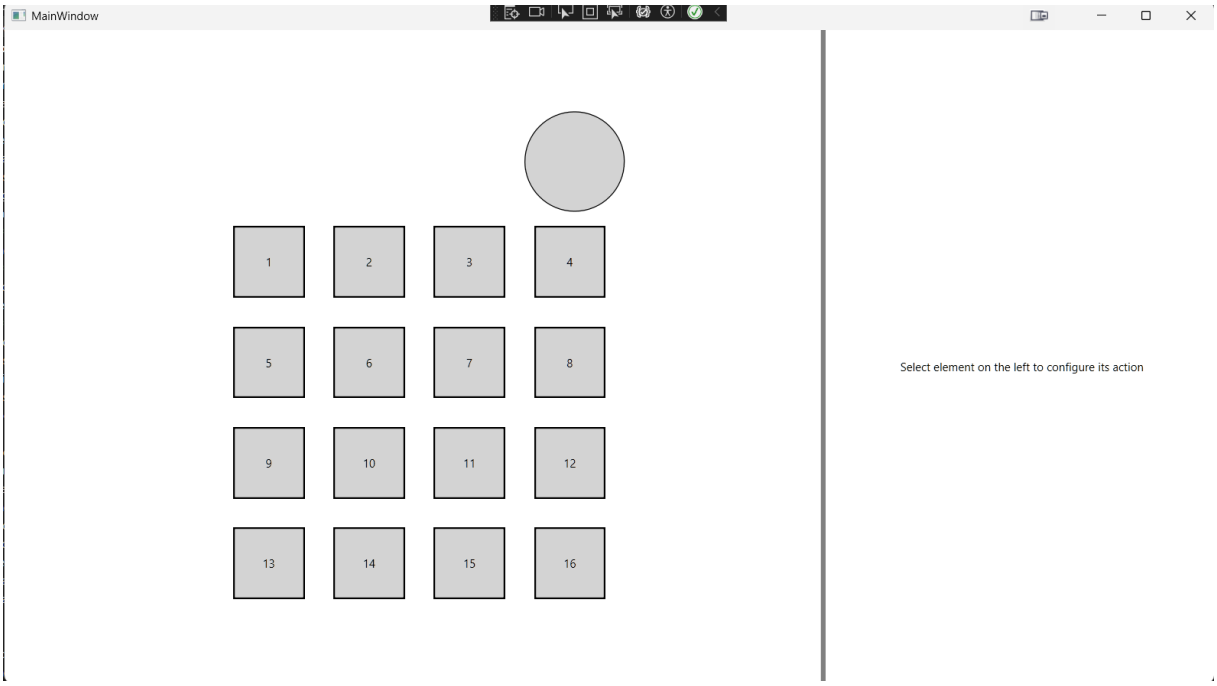
```
private void ConnectionCheckTimer_Elapsed(object? sender, EventArgs e)
{
    if (!_serialPort.IsOpen)
    {
        ConnectivityCheck = false;
        try
        {
            _serialPort.Open();
```

```
            ConnectivityCheck = true;
            return;

        }
        catch (Exception)
        {
            ConnectivityCheck = false;
            Dispatcher.Invoke(() =>
            {
                MessageBox.Show("Connection lost: Cannot reconnect.", "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
                this.Show();
                _mainWindow.Close();
            });

        }
    }
}
```

Listing 6.2 Funkcja sprawdzająca stan połączenia [opracowanie własne]

6.3 Okno główne

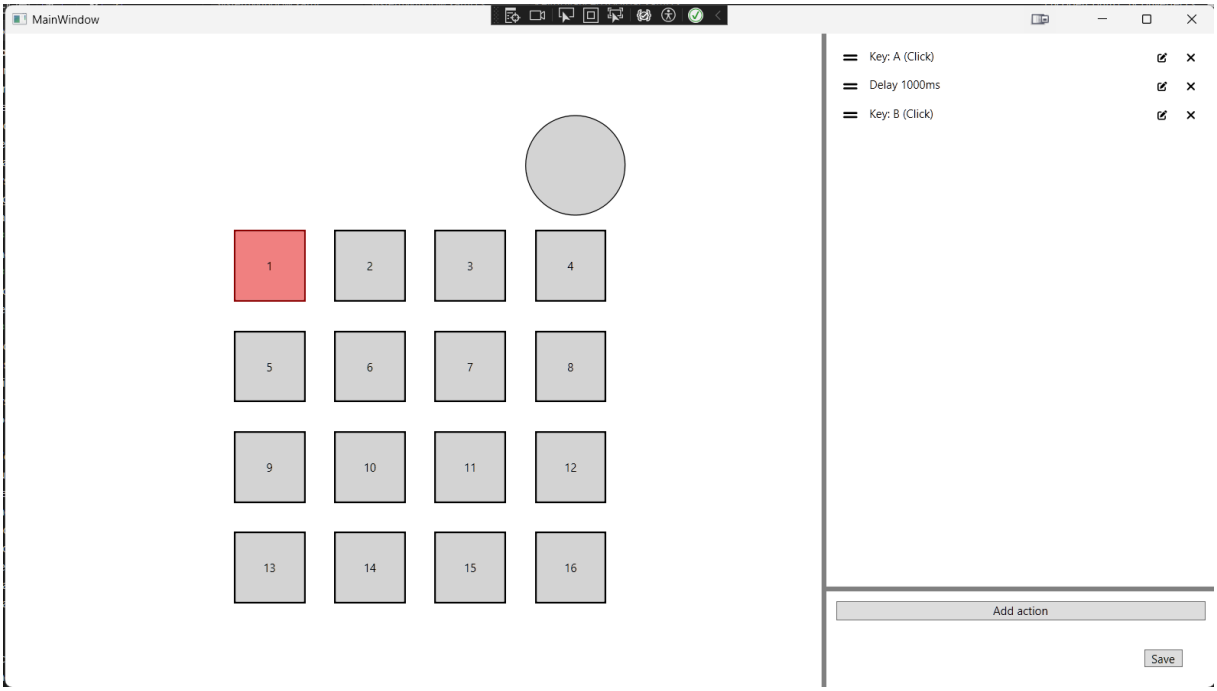
Okno główne dzieli się na 2 części: graficzny interfejs do wybrania kontrolki, do której użytkownik chce przypisać akcje oraz zakładka z konfiguracją dla wybranej kontrolki. Rys. 6.2 przedstawia jak wygląda okno główne.



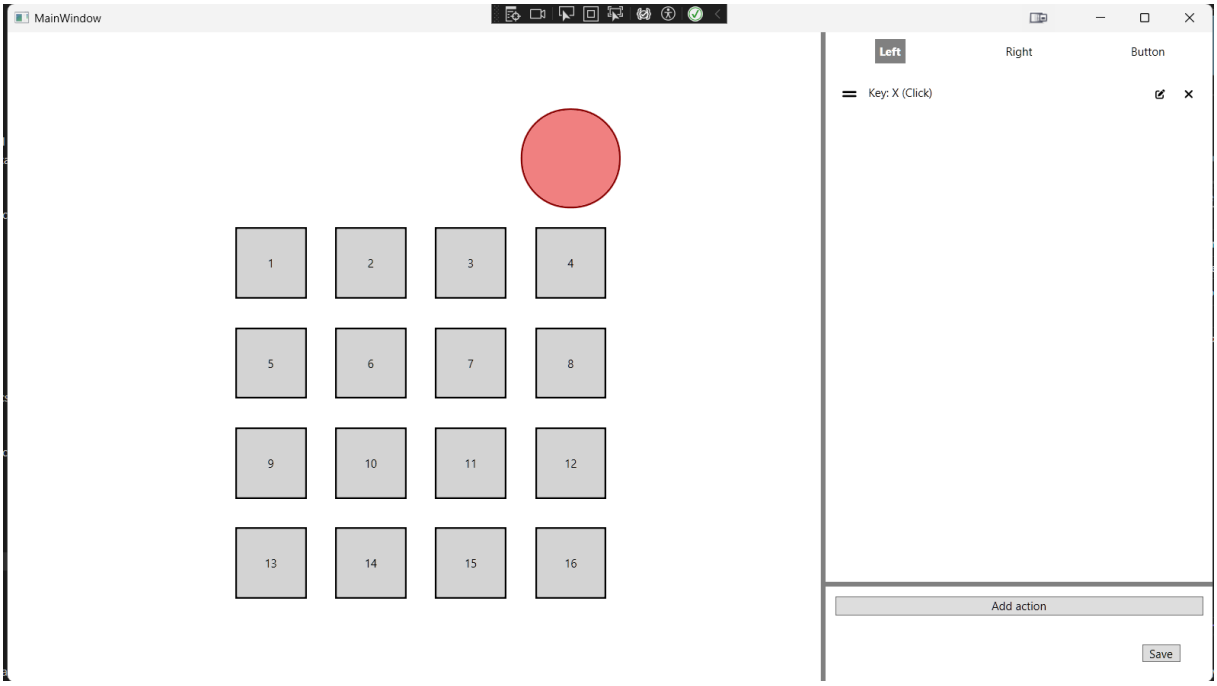
Rys. 6.2 Okno główne aplikacji [opracowanie własne]

Po wybraniu kontrolki na lewym panelu, po prawej pojawia panel do konfiguracji. Prawy panel dostosowuje swój wygląd na podstawie wybranej kontrolki. Dla przycisku jest to lista

wszystkich akcji mających zostać wywołane, co widać na Rys. 6.3. Dla wybranego enkodera prawy panel wyświetla jedną z trzech możliwych do wybrania zakładek, które reprezentują obrót w lewo, w prawo oraz wciśnięcie przycisku, co widać na Rys. 6.4.



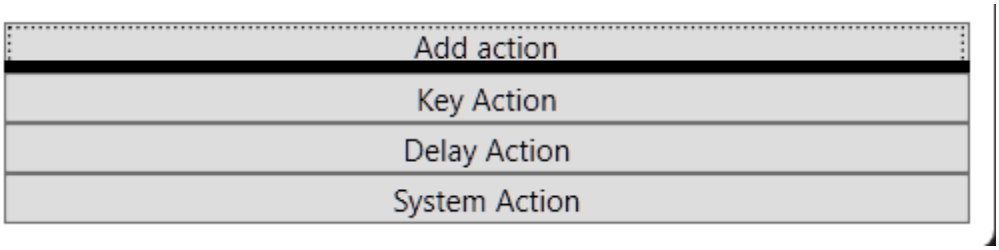
Rys. 6.3 Okno główne, wybrany przycisk [opracowanie własne]



Rys. 6.4 Okno główne, wybrany enkoder [opracowanie własne]

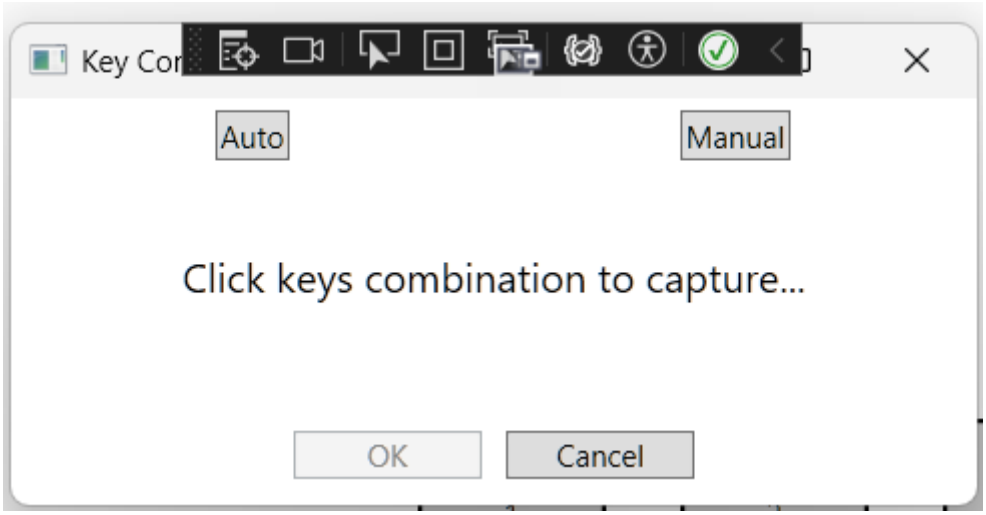
6.4 Dodawanie, edycja i usuwanie akcji

Do dodawania akcji służy przycisk „Add action” widoczny na Rys. 6.5. Po kliknięciu rozwija się lista z akcjami do dodania. Po wybraniu akcji pojawia się okno dialogowe, w którym konfiguruje się akcję. Po zatwierdzeniu akcja pojawia się na końcu listy. Klikając przycisk „X” obok niej, użytkownik usunie ją, a klikając okienko z ołówkiem, edytuje ją, ponownie otwierając okno dialogowe. Ikona dwóch poziomych kresek po lewej stronie akcji służy do zmiany kolejności akcji na liście.

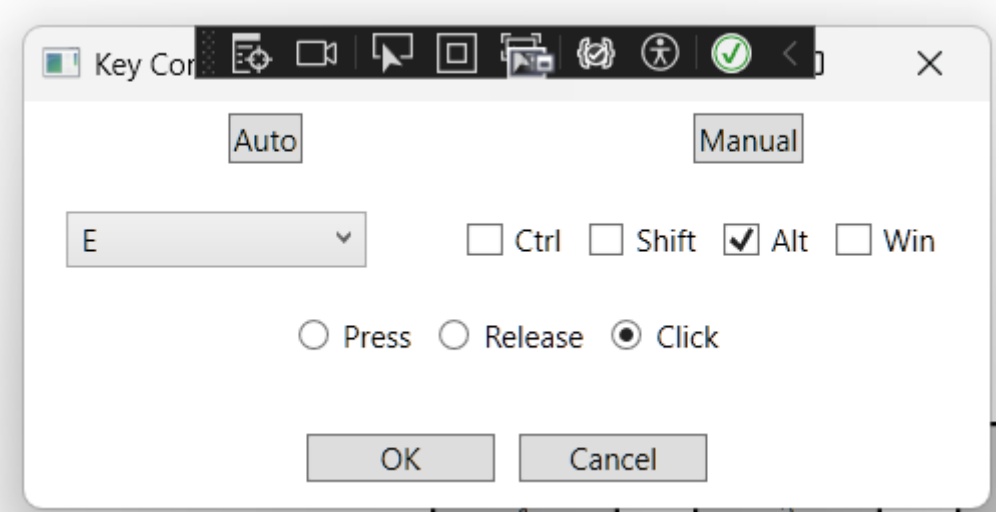


Rys. 6.5 Przycisk "Add action" [opracowanie własne]

Okno dialogowe dla „Key Action” zawiera 2 zakładki. Pierwsza z nich, widoczna na Rys. 6.6 przechwytuje wciśniętą przez użytkownika kombinację klawiszy. Druga zakładka, przedstawiona na Rys. 6.7 pozwala na ręczne ustawienie kombinacji. Jest to szczególnie konieczne w przypadku gdy klawiatura użytkownika nie posiada danego przycisku lub zdarzenia z innych aplikacji użytkownika nie pozwalają na poprawne automatyczne przechwycenie kombinacji klawiszy.



Rys. 6.6 Okno dialogowe "Key Action", zakładka "Auto" [opracowanie własne]



Rys. 6.7 Okno dialogowe "Key Action", zakładka "Manual" [opracowanie własne]

Okno dialogowe dla „Delay Action” zawiera pole tekstowe, które przyjmuje wartość liczbową będącą czasem który akcja ma odczekać, w milisekundach.

Okno dialogowe dla „System Action” jest podobne do okna dla „Key Action” tyle, że zamiast akcji typowych klawiatury, zawiera listę takich akcji jak sterowanie głośnością, jasnością ekranu lub usypianie.

6.5 Zapis do urządzenia

Po kliknięciu przycisku „Save” akcja lub zbiór akcji są zapisywane w pamięci programu. Po tym następuje serializacja zapisanych akcji i rozpoczęcie procedury przesyłania pliku JSON do urządzenia. Aplikacja wysyła komendę „setConfiguration”, a następnie czeka na odpowiedź „ready”. Po tym wysyła plik z konfiguracją i czeka na odpowiedź zwrotną „completed” w celu upewnienia się że transfer przebiegł pomyślnie. Na koniec aplikacja wyświetla komunikat o udanym zapisie. W razie niepowodzenia aplikacja informuje użytkownika o błędzie.

6.6 Serializacja i deserializacja

Do serializacji i deserializacji zostały utworzone klasy dziedziczące po klasie „JsonConverter” z przestrzeni nazw „System.Text.Json”. Metoda „Write” odpowiada za serializację, a „Read” za deserializację. Listing 6.3 jest przykładem implementacji serializacji i deserializacji.

```
namespace configApp.JsonConverters
{
    class EncoderControlJsonConverter : JsonConverter<EncoderControl>
    {
        public override EncoderControl? Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options)
        {
            var encoder = new EncoderControl();

            using (var document = JsonDocument.ParseValue(ref reader))
            {
                var root = document.RootElement;

                if (root.TryGetProperty("id", out var idElement))
                {
                    encoder.Id = idElement.GetInt32();
                }

                if (root.TryGetProperty("actionLeft", out var actionLeftElement))
                {
                    encoder.ActionLeft =
JsonSerializer.Deserialize<IAction>(actionLeftElement.GetRawText(), options);
                }

                if (root.TryGetProperty("actionRight", out var
actionRightElement))
                {
                    encoder.ActionRight =
JsonSerializer.Deserialize<IAction>(actionRightElement.GetRawText(), options);
                }

                if (root.TryGetProperty("actionButton", out var
actionButtonElement))
                {
                    encoder.ActionButton =
JsonSerializer.Deserialize<IAction>(actionButtonElement.GetRawText(), options);
                }
            }
            return encoder;
        }

        public override void Write(Utf8JsonWriter writer, EncoderControl value,
JsonSerializerOptions options)
        {
            writer.WriteStartObject();

            writer.WriteNumber("id", value.Id);

            writer.WritePropertyName("actionLeft");
            JsonSerializer.Serialize(writer, value.ActionLeft, options);
            writer.WritePropertyName("actionRight");
            JsonSerializer.Serialize(writer, value.ActionRight, options);
            writer.WritePropertyName("actionButton");
            JsonSerializer.Serialize(writer, value.ActionButton, options);

            writer.WriteEndObject();
        }
    }
}
```

Listing 6.3 Serializacja i deserializacja dla klasy "EncoderControl" [opracowanie własne]

Następnie każda z tego typu klas jest dołączana do opcji obiektu „JsonSerializer” przy każdorazowym wywołaniu serializacji lub deserializacji. Jest to widoczne na Listing 6.4.

```
JsonSerializerOptions options = new JsonSerializerOptions
{
    PropertyNameCaseInsensitive = true,
};

options.Converters.Add(new IControlListJsonConverter());
options.Converters.Add(new MacroActionJsonConverter());
options.Converters.Add(new KeyboardActionJsonConverter());
options.Converters.Add(new ButtonControlJsonConverter());
options.Converters.Add(new DelayActionJsonConverter());
options.Converters.Add(new IActionJsonConverter());
options.Converters.Add(new EncoderControlJsonConverter());

JsonSerializer.Deserialize<List<IControl>>(json, options);
```

Listing 6.4 Dodawanie niestandardowych konwerterów do opcji serializera [opracowanie własne]

7 Testy

Testy polegały sprawdzeniu czy wszystkie wymagania zostały spełnione oraz czy wszystkie elementy projektu współgrają ze sobą. W tym celu przypisano w aplikacji komputerowej akcje do każdego klawisza i enkodera, zapisano konfigurację oraz użyto każdej kontrolki na fizycznym urządzeniu w celu sprawdzenia, czy akcja została wywołana. Z wyników testów okazało się, że wszystkie wymagania projektu zostały spełnione oraz wszystkie części projektu współgrają ze sobą tworząc działający system.

8 Podsumowanie

W wyniku realizacji pracy inżynierskiej spełniono wszystkie wymagania projektu. Zaprojektowano fizyczne urządzenie, w tym PCB oraz model 3D obudowy. Zaprogramowano firmware obsługujący urządzenie, zapewniający wszelkie potrzebne funkcje. Stworzono aplikację desktopową, pozwalającą na konfigurację urządzenia wedle potrzeb użytkownika. Przeprowadzono testy sprawdzające spełnienie postawionych wymagań oraz integrację poszczególnych etapów projektu. Testy przebiegły pomyślnie. Postawiony cel pracy został zrealizowany.

9 Bibliografia

[1] X-Kom, „Sklep - Glorious GMMK Wireless Numpad,” 2025, 1 1. [Online]. Available: https://www.x-kom.pl/p/1107536-klawiatura-bezprzewodowa-glorious-gmmk-wireless-numpad-black.html.

[2] Thomann, „Sklep - Elgato Stream Deck Dial Set,” 1 1 2025. [Online]. Available: https://www.thomann.pl/elgato_stream_deck_dial_set_gold.htm.

[3] Keychron, „Sklep - Keychron Q0 Max QMK Custom Number Pad,” 1 1 2025. [Online]. Available: https://keychronpoland.com/products/keychron-q0-max-qmk-custom-number-pad.

[4] Seeeduino , „Dokumentacja płytki prototypowej Seeeduino XIAO,” 1 1 2025. [Online]. Available: https://wiki.seeedstudio.com/Seeeduino-XIAO/.

[5] USB Implementers Forum, „Device Class Definition for Human Interface Devices ver. 1.11,” 27 1 2005. [Online]. Available: https://www.usb.org/hid.

[6] NicoHood, 1 1 2025. [Online]. Available: https://github.com/NicoHood/HID.

[7] USB Implementers Forum, „HID Usage Tables ver.1.12,” 28 10 2004. [Online]. Available https://www.usb.org/hidhttps://www.usb.org/hid.

[8] Arduino, „Dokumentacja frameworka Arduino,” 1 1 2025. [Online]. Available: https://docs.arduino.cc/programming/.

[9] B. Blanchon, „Dokumentacja biblioteki ArduinoJson,” 1 1 2025. [Online]. Available: https://arduinojson.org/v7/.

[10] Cherry, „SparkFun - Cherry MX Switch Documentation,” 1 1 2025. [Online]. Available: https://www.sparkfun.com/cherry-mx-switch.html#documentation.

[11] mo-thunderz, „Repozytorium o nazwie RotaryEncoder,” 1 1 2025. [Online]. Available: https://github.com/mo-thunderz/RotaryEncoder/blob/main/Arduino/ArduinoRotaryEncoder/ArduinoRotaryEncoder.ino.

10 Spis rysunków

Rys. 4.1 PCB, uwydatnione ścieżki górne [opracowanie własne] 11
Rys. 4.2 PCB, uwydatnione ścieżki dolne [opracowanie własne] 12
Rys. 4.3 PCB, wszystkie elementy widoczne [opracowanie własne] 13
Rys. 4.4 Schemat elektroniczny urządzenia 14
Rys. 4.5 Urządzenie z boku [opracowanie własne] 15
Rys. 4.6 Render urządzenia, widok z góry [opracowanie własne] 16
Rys. 6.1 Okno do połączenia z urządzeniem [opracowanie własne] 35
Rys. 6.2 Okno główne aplikacji [opracowanie własne] 36
Rys. 6.3 Okno główne, wybrany przycisk [opracowanie własne] 37
Rys. 6.4 Okno główne, wybrany enkoder [opracowanie własne] 37
Rys. 6.5 Przycisk "Add action" [opracowanie własne] 38
Rys. 6.6 Okno dialogowe "Key Action", zakładka "Auto" [opracowanie własne] 38
Rys. 6.7 Okno dialogowe "Key Action", zakładka "Manual" [opracowanie własne] 39

11 Spis listingów

Listing 3.1 Deskryptor raportu klawiatury [6]Listing 3.2 Struktura przechowująca raport dla klawiatury [6] 7
Listing 5.1 Klasa "HIDDevice_" [opracowanie własne] 18
Listing 5.2 Klasa "Keyboard_" [6] 22
Listing 5.3 Klasa "Button" [opracowanie własne] 23
Listing 5.4 Klasa "ButtonSecured" [opracowanie własne] 24
Listing 5.5 Klasa "Encoder" [11] 26
Listing 5.6 Klasa "KeyboardAction" [opracowanie własne] 28
Listing 5.7 Klasa "DelayAction" [opracowanie własne] 29
Listing 5.8 Serializacja i deserializacja klasy "KeyboardAction" [opracowanie własne] 30
Listing 5.9 Przykładowy plik JSON z konfiguracją urządzenia [opracowanie własne] 32
Listing 5.10 System komend dla komunikacji protokołem Serial [opracowanie własne] 33
Listing 6.1 Funkcja łądzenia aplikacji z urządzeniem [opracowanie własne] 35
Listing 6.2 Funkcja sprawdzająca stan połączenia [opracowanie własne] 36
Listing 6.3 Serializacja i deserializacja dla klasy "EncoderControl" [opracowanie własne] ... 40
Listing 6.4 Dodawanie niestandardowych konwerterów do opcji serializera [opracowanie własne] 41