AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki, Elektroniki i Telekomunikacji

Instytut Elektroniki

Projekt dyplomowy

Asystent do nauki gry na keyboardzie

Autor: *Nikodem Szafran*

Kierunek studiów: *Elektronika*

Opiekun pracy: *dr. inż. Jacek Kołodziej*

Kraków, *2026*

Spis treści

Wstęp

Nauka gry na instrumentach klawiszowych nie jest prostą rzeczą i często wymaga ona bardzo dużej determinacji lub nakładów finansowych, które umożliwią łatwiejszy dostęp do wiedzy, np. w postaci różnych aplikacji edukacyjnych lub pomocy osób bardziej doświadczonych, dzięki którym początkujący mogą szybciej rozwijać i doskonalić swoje umiejętności.

Bardzo często pierwszym impulsem u osób, które zaczynają naukę jest chęć zagrania swoich ulubionych utworów - np. piosenek, które często nie są ciężkie do zagrania technicznie, nawet dla osób zaczynających przygodę z instrumentem.

Nierzadko jednak przytłaczająca potrafi być wtedy ilość czynności, które należy wykonywać równocześnie: koordynacja pracy rąk, odczyt zapisu nutowego oraz mapowanie go na właściwe klawisze, a także samodzielna ocena poprawności wykonywanych dźwięków.

Popularność lekcji z osobami doświadczonymi, a także aplikacji wspierających naukę pokazuje, że możliwość odciążenia użytkownika choćby z jednej z tych czynności ułatwia naukę i zmniejsza frustrację użytkownika podczas nauki.

Upowszechnienie elektronicznych instrumentów klawiszowych, zwykle tańszych od pianin akustycznych, obniżyło próg wejścia dla początkujących – pojawia się zatem coraz większa grupa osób, która chce zacząć swoją przygodę z instrumentami. Dodatkowo, wraz z pojawieniem się takiej grupy instrumentów dużo większą rolę zaczął odgrywać standard MIDI (Musical Instrument Digital Interface), dzięki któremu jesteśmy w stanie analizować zdarzenia wykonawcze (np. naciśnięcia klawiszy i ich dynamikę) generowane przez instrument. A ogromna popularność protokołu USB w ostatnich latach spowodowała, że większość takich urządzeń komunikuje się wykorzystując te dwa standardy.

Motywacją do podjęcia tematu pracy była chęć stworzenia rozwiązania, które ułatwi naukę gry na elektronicznych instrumentach klawiszowych (potocznie nazywanych keyboardami) poprzez uproszczenie konieczności ciągłego czytania zapisu nutowego z parytury oraz zapewnienie natychmiastowej informacji zwrotnej o poprawności zagranych dźwięków.

Głównym celem pracy jest zaprojektowanie i implementacja prototypu asystenta wspomagającego naukę gry na elektronicznym instrumencie klawiszowym. Praca koncentruje się na warstwie aplikacyjnej, w szczególności na logice prowadzenia lekcji krok po kroku dla utworów oraz akordów, z możliwością cofania i pomijania kroków w celu ćwiczenia wybranych fragmentów. Po zakończeniu lekcji system prezentuje podsumowanie w postaci odsetka poprawnych zagrań. Dzięki komunikacji USB-MIDI możliwa jest jednoznaczna rejestracja naciśnięć klawiszy i ich weryfikacja względem oczekiwanych nut.

Rozdział 1

Protokół MIDI

1.1 Wprowadzenie do protokołu MIDI

Protokół MIDI, czyli Musical Instrument Digital Interface, to powstały w sierpniu 1983 roku [1] protokół komunikacyjny stworzony do przesyłania zdarzeń muzycznych, czyli informacji, które reprezentują dany dźwięk.

Stosowany jest on w rozmaitych elektronicznych urządzeniach muzycznych, stosowany przez artystów i kompozytorów, a także np. w grach wideo, gdzie znajduje swoje zainteresowanie poprzez bardzo małą ilość miejsca jakie zajmują pliki w tym formacie.

Zresztą powód, dlaczego pliki formatu Standard MIDI File (*.mid) zajmują tak mało miejsca, to jedna z najważniejszych cech i zalet tego protokołu – ponieważ nie zawiera on dźwięków, a tylko instrukcję, jak dany dźwięk odtworzyć [2].

Dzięki temu pliki w tym formacie, poza lekkością danych, są bardzo łatwe do edycji i zmiany parametrów (takich jak tempo, tonacja czy nawet podmiana instrumentów na inne), ponieważ urządzenie generujące dźwięki (np. syntezator) robi to na podstawie tych właśnie instrukcji [4]. Zapewnia to także ogromną kompatybilność pomiędzy różnymi elektronicznymi sprzętami muzycznymi, gdzie generacja dźwięków może odrobinkę się różnić w zależności od hardware'u, natomiast software'owo przesyłane są te same instrukcje do wszystkich urządzeń.

1.2 Dziennik definicji

Wiadomość/komunikat/ramka/zdarzenia MIDI – zbiór instrukcji, które przechowują informacje dla syntezatora jak ma odtworzyć dany dźwięk [3]. W dalszej części pracy wszystkie te terminy są używane zamiennie.

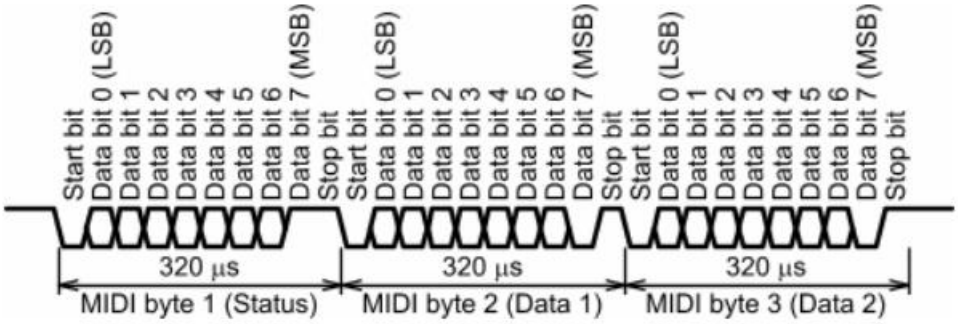
Kontroler MIDI – to urządzenie, które jest używane jako instrument (np. klawiatura elektronicznego instrumentu elektronicznego), generuje wiadomości MIDI, które są następnie transmitowane do syntezatora [3].

Syntezator MIDI – urządzenie, które zajmuje się przechwytywaniem, przechowywaniem, edycją, a przede wszystkim odtwarzaniem dźwięków wysłanych w wiadomościach MIDI [3].

Kanał MIDI – mimo połączenia urządzeń jednym fizycznym przewodem, to mamy do wyboru aż 16 kanałów, na których możemy przesyłać ramki MIDI. Jest to możliwe dzięki wariacjom 4 bitowego nibble'a danych, który znajduje się w każdej przesyłanej ramce MIDI. Często na osobnych kanałach przesyłamy osobne linie melodyczne instrumentów np. na jednym pianino, na drugim perkusje, a na trzecim bas [3].

1.3 Budowa ramki/komunikatu MIDI

Ramka MIDI jest jednokierunkowym, asynchronicznym strumieniem bitów. Są one przesyłane z przepustowością 31.25 kb/s. Każdy bajt składa się z 10 bitów, tj. bitu startu (niski stan logiczny), 8 bitów danych i bitu stopu (wysoki stan logiczny). Typowa ramka MIDI składa się natomiast z 3 takich bajtów, gdzie pierwszy z nich to tzw. Status byte, który opisuje typ przesyłanej ramki (np. co robi dana ramka, czy na który kanał przesyłane są informacje), pozostałe dwa to bajty danych , które opisują różne parametry w zależności od tego, jaki jest typ przesyłanej ramki [3].



Rys. 1.: Budowa ramki MIDI, ***źródło: [5]***

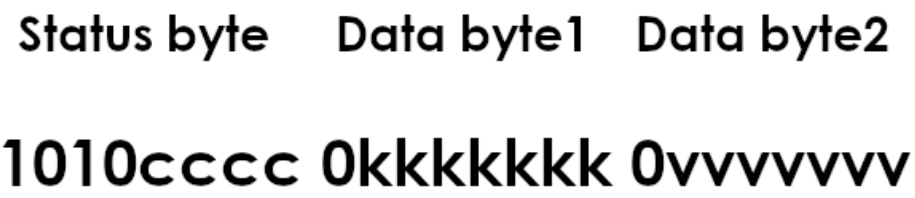
1.4 Typy komunikatów MIDI

Typów ramek MIDI standard wyróżnia bardzo dużo, postanowiono wyróżnić trzy, które mają największe znaczenie dla warstwy aplikacyjnej.

Channel Voice Message – najczęściej występujący w MIDI typ komunikatu. Przenosi on kluczowe instrukcje dotyczące zdarzeń muzycznych, takich jak naciśnięcie/puszczenie klawisza, siłę naciśnięcia/puszczenia, wysokość tonu czy typ instrumentu w jakim dany dźwięk ma być wygenerowany [3].

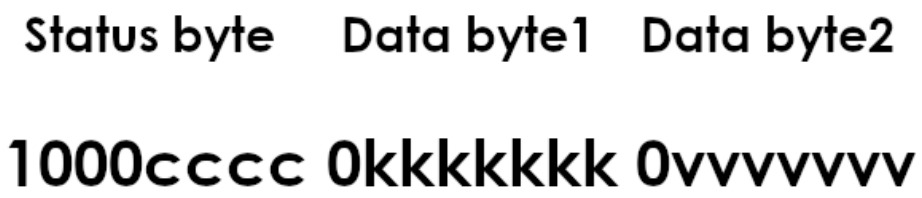
Najważniejszymi z punktu widzenia założeń projektu są oczywiście informacje o naciśnięciu danego klawisza (Note On), puszczeniu tego klawisza (Note Off) oraz moc z jaką dany klawisz został naciśnięty (Velocity). W formacie MIDI zdarzenia Note On i Note Off są dwiema osobno wysyłanymi ramkami, chociaż czasami zdarza się, że wiadomość Note On z Velocity równym 0, jest traktowana jako Note Off [3].

Komunikat Note On jest wysyłany, w początkowym momencie naciśnięcia klawisza. Zaczyna się od bajtu statusu, którego wartość to 0x9n (gdzie n – numer kanału), następnie są dwa bajty danych: pierwszy informuje o klawiszu, który został naciśnięty, a drugi bajt o mocy tego naciśnięcia (Velocity) [3].



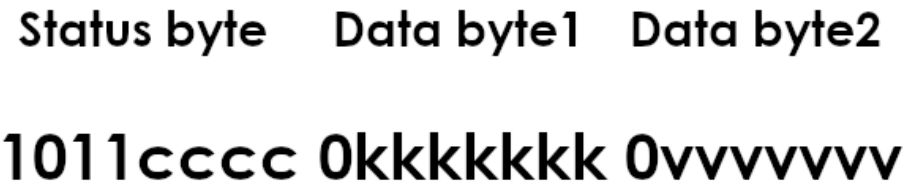
Rys. 2.: Budowa ramki Note On, c – numer kanału, k – klawisz, v – wartość velocity (siła naciśnięcia), ***źródło: opracowanie własne na podstawie [3]***

Ramka Note Off jest wysyłana w momencie końcowego, zakończonego puszczenia klawisza. Zaczyna się od bajtu statusu, którego wartość to 0x8n, kolejne dwa bajty są identyczne jak w ramce Note On. W tym wypadku natomiast bajt danych Velocity jest bardzo często ignorowany [3].



Rys. 3.: Budowa ramki Note Off, c – numer kanału, k – klawisz, v – wartość velocity (siła puszczenia), ***źródło: opracowanie własne na podstawie [3]***

Channel Mode Messages – są to instrukcje przesyłane do kanałów MIDI, które np. wyłączają generację dźwięków z tego kanału, resetują podłączone kontrolery MIDI do parametrów ustawionych domyślnie czy zmieniają tryby generacji dźwięków przez syntezator (np. tylko przez selektywne kanały lub wszystkie, granie wielu nut na raz, lub tylko jednej). Ich bajt statusu zaczyna się od 0xBn [3].



Rys. 4,: Budowa ramki Channel Mode Message, c – numer kanału, k – komenda, v – wartość o różnym znaczeniu w zależności od komendy, ***źródło: opracowanie własne na podstawie [3]***

Ramki Channel Mode operują na komendach w zakresie (czyli wartości k) równej [120,127].

Rozdział 2

Obecne rozwiązania rynkowe do nauki gry na keyboardzie

2.1 Standardy na rynku

Aktualnie na rynku znajdziemy kilka pozycji, które oferują różne podejście do nauki gry na elektronicznych instrumentach klawiszowych dla użytkowników:

- płatne, zazwyczaj w modelu subskrypcji czasowej, aplikacje zewnętrzne na komputer lub telefon nieoferujące możliwości wgrywania własnych utworów MIDI (grupa 1.).

- płatne aplikacje zewnętrzne na komputer oferujące możliwość wgrywania własnych utworów MIDI (grupa 2.), wspierające systemy podświetlania klawiszy.

- wbudowane w keyboardy systemy nauki podświetlające klawisze dla ustalonej przez producenta bazy piosenek, czasami z możliwością wgrywania własnych utworów MIDI (grupa 3.).

2.2 Grupa pierwsza, reprezentant: aplikacja Flowkey

Jest to aplikacja, odpalana na zewnętrznym urządzeniu, np. komputerze lub telefonie, która posiada wbudowaną przez deweloperów bazę piosenek – nie ma w niej możliwości importowania własnych utworów w formacie MIDI [6].

Flowkey wyświetla na ekranie urządzenia zewnętrznego (tj. komputera lub telefonu) paryturę oraz widok z góry na pianistę grającego wybrany przez użytkownika utwór, dzięki czemu może on zaobserwować jego ułożenie rąk – co potrafi przyśpieszyć naukę i zwłaszcza na początku – pomóc w doskonaleniu poprawnej techniki gry. Podczas grania utworu nad klawiszami wyświetlane są jedynie podstawowe oznaczenia nutowe (tzn. B, a nie B3 czy B4). Użytkownik ma także możliwość zapętlenia interesującego go fragmentu utworu, w celu skupienia się na jego doskonaleniu [6].

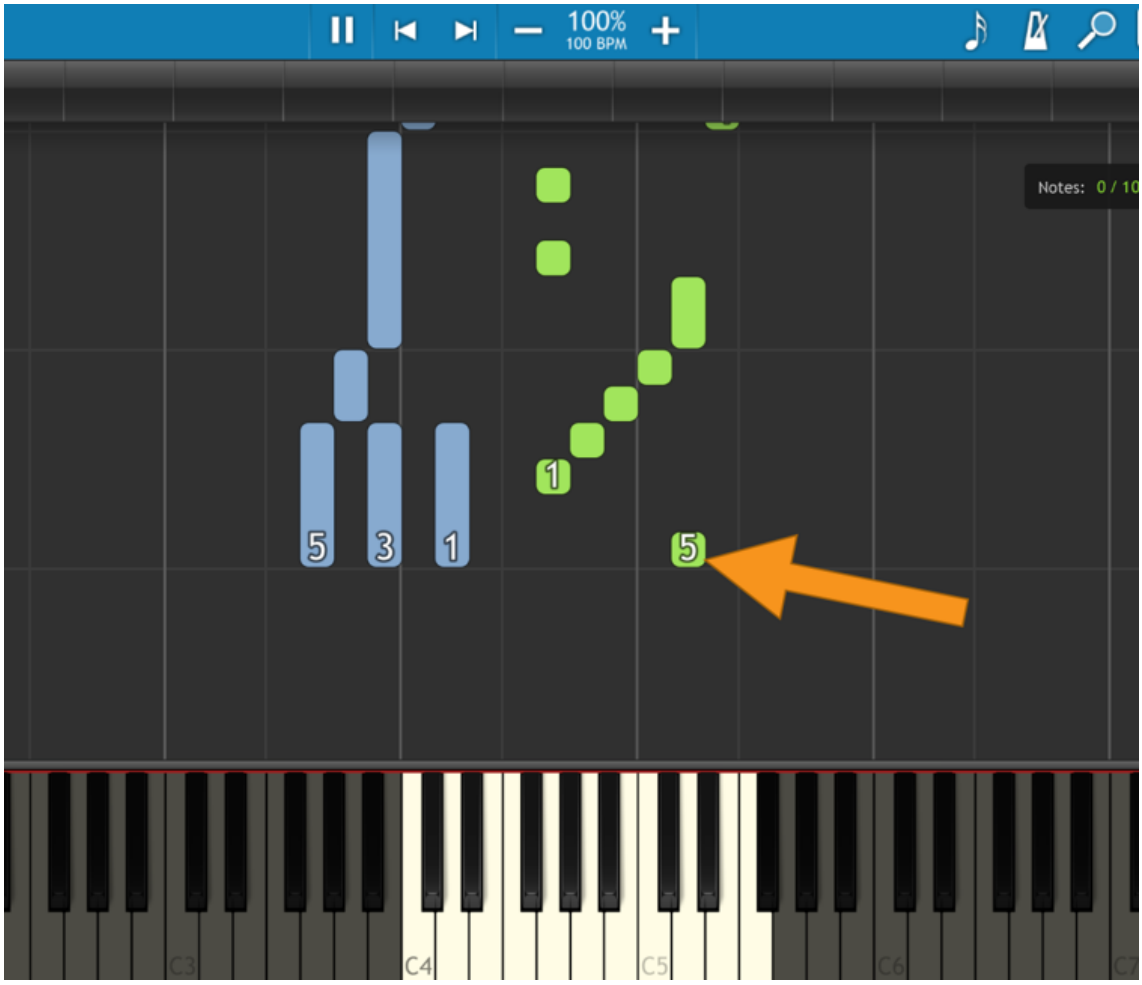
Rys. 5.: Interfejs nauki, dostępny na stronie, *źródło: [6]*

Program posiada tzw. „Wait Mode", który czeka z zagraniem kolejnego fragmentu utworu (np. nuty), do momentu naciśnięcia klawisza przez użytkownika [6]. Wyłapywanie, które nuty zagrał użytkownik jest realizowane domyślnie przez mikrofon zewnętrznego urządzenia, jest to jednak niedokładna metoda, prowadząca do zafałszowań na korzyść lub niekorzyść użytkownika – co wpływa negatywnie na proces nauki. Można jednak to wyeliminować podłączając instrument do urządzenia zewnętrznego (komputer lub telefon), bezprzewodowo lub poprzez kabel, dzięki czemu aplikacja komunikując się po protokole MIDI zbiera dane bezpośrednio z keyboarda.

Użytkownik zyskuje dostęp do aplikacji opłacając cykliczny abonament, który występuje w dwóch wersjach – podstawowej bazy piosnek, lub rozszerzonej. Dla tej pierwszej ceny zaczynają się od 10 do 20 dolarów amerykańskich za miesiąc (w zależności od płatności z góry za cały rok, czy opłaty pojedynczego miesiąca), a za pełen dostęp użytkownik jest zobligowany do zapłaty od 15 do 30 dolarów miesięcznie [6]. Wszystkie ceny są podawane na stan dnia 21.12.2025.

2.3 *Grupa druga, reprezentant: aplikacja Synthesia*

Do korzystania z Synthesii również potrzeba zewnętrznego urządzenia, takiego jak np. komputer. Oferowany przez Synthesie tryb nauki działa analogicznie jak w poprzedniej aplikacji – dopóki użytkownik nie naciśnie właściwej nuty – utwór jest zatrzymany. Domyślną opcją jest nasłuchiwanie zagranych dźwięków przez mikrofon urządzenia, na którym odpalony jest program. Mamy także możliwość podłączenia instrumentu klawiszowego i komunikacji z urządzeniem zewnętrznym za pomocą protokołu MIDI [7].



Rys. 6.: Interfejs programu Synthesia, ***źródło: [7]***

Bardzo dużą zaletą aplikacji Synthesia, jest to że oferuje ona możliwość wgrywania zewnętrznych utworów w formacie MIDI [7] – co pozwala użytkownikom na naukę ogromnej ilości utworów. Program posiada także bardziej zaawansowane możliwości

nauki gry utworów, np. tylko na jednej, konkretnej ręce [7], jednak jest to opcja, która nie zawsze działa poprawnie, przy każdym wgranym przez użytkownika utworze. Aplikacja wspiera także część keyboardów z podświetlanymi klawiszami, np. z podanym jako przykład w grupie 3. CASIO LK-S250 [7].

Aplikacja jest ciągle rozwijana przez twórców oraz dużą społeczność osób z niej korzystających. Użytkownik dostaje dożywotni dostęp do programu po jednorazowej opłacie w wysokości 40 dolarów amerykańskich [7] – cena podana ze stanu na dzień 21.12.2025.

2.4 Grupa trzecia, reprezentant: rodzina instrumentów klawiszowych CASIO LK

Są to elektroniczne instrumenty klawiszowe nazywane jako Key Lighting Keyboards. Jest to najbardziej rozbudowany w stronę komfortu użytkownika system i najbardziej przypominający proponowane w tej pracy dyplomowej rozwiązanie.

CASIO LK, to rodzina keyboardów posiadający wbudowany przez producenta system podświetlania klawiszy. Obecnie, według producenta, w tej rodzinie znajdują się jedynie dwa dalej produkowane, dostępne do kupienia modele urządzeń [8], a funkcja nie jest dostępna w żadnych innych instrumentach firmy.

Ceny urządzeń wahają się od 200 do 300 dolarów amerykańskich (stan na dzień 21.12.2025), w modelu tańszym mamy dostępne 60 wbudowanych utworów, a droższy CASIO LK-S450 posiada ich 160 [8].

Wbudowany w keyboardy LK system pozwala na naukę w kilku trybach [9]:

- EASY, gdzie podświetlane są klawisze aktualnie grane w utworze, utwór zatrzymuje się przy każdej nucie, do czasu zagrania przez użytkownika dowolnego klawisza (służy do nauki rytmu utworu).

- LISTEN, gdzie po prostu zagrany jest cały utwór z podświetlaniem klawiszy (do wzrokowej nauki utworu).

- WATCH, gdzie podświetlane są klawisze aktualnie grane w utworze, a utwór jest zatrzymany dopóki użytkownik nie naciśnie aktualnie wymaganego klawisza, kolejne do grania klawisze są reprezentowane przez mruganie.

- REMEMBER, gdzie użytkownik gra utwór bez podświetlania klawiszy, jednak utwór jest zatrzymany do momentu zagrania właściwego klawisza.

Przy wbudowanych w instrument utworach mamy także wyświetlany na ekranie poprawny sposób ułożenia rąk do gry konkretnych nut [9].

Keyboardy z tej rodziny umożliwiają też wgrywania własnych utworów w formacie MIDI, poprzez połączenie instrumentu z aplikacją producenta *Chordana Play* [10]. Co ważne – urządzenie dalej obsługuje wtedy system nauki gry.

Rozdział 3

Standard USB

3.1 Topologia USB i zasilanie

Standard USB jest obecnie jednym z najpopularniejszych protokołów w zastosowaniach konsumenckich i przemysłowych, opracowany w połowie lat 90. [12], który miał na celu ułatwienie komunikacji urządzeń elektronicznych, dzięki zastąpieniu wolniejszych portów szeregowych i równoległych [12]. Magistrala USB cieszy się bardzo dużą popularnością, dzięki komfortowi użytkownika, który oferuje – z racji tego, że jest ona plug&play [12]. Standard USB wyróżnia dwa typy urządzeń w komunikacji:

- Host (urządzenie główne)

- Device (urządzenie peryferyjne)

Na magistrali USB to Host zarządza całą komunikacją: inicjuje transfery danych, przydziela czas na transmisję i koordynuje komunikację z urządzeniami peryferyjnymi [11]. W konsekwencji tego urządzenie USB typu Device nie inicjuje transmisji samodzielnie – dane są przesyłane w ramach transakcji inicjowanych przez hosta [11].

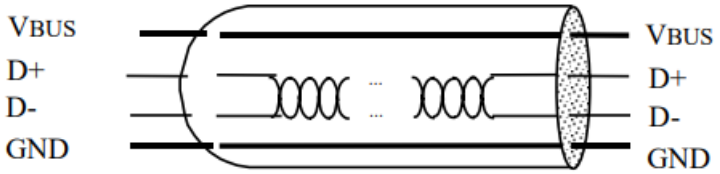
Najprostszym przykładem takiej komunikacji jest układ komputer-klawiatura. Komputer tutaj pełni rolę Hosta – zarządzając całą transmisją, a także dostarczając do niej zasilanie. Rolą klawiatury jest zbieranie i buforowanie danych, w tym wypadku klawiszy które naciska użytkownik, które następnie przekaże do Hosta, przy cyklicznym jej odpytaniu.

Istotną rolę przy rozróżnianiu urządzeń Host od Device pełni linia zasilająca VBUS, dzięki której pobieramy zasilania z magistrali dla urządzeń bus-powered (takich jak typowa podłączona przez kabel klawiatura) oraz, co ważniejsze, wykrywamy sygnał obecności aktywnego połączenia [11]. To dopiero po wykryciu linii VBUS urządzenie typu Device sygnalizuje swoją obecność poprzez dołączenie rezystora pull-up do odpowiedniej linii danych, co pozwala Hostowi wykryć podłączenie oraz określić prędkość pracy urządzenia, a następnie rozpocząć inicjalizację połączenia (enumerację) [11][12].

Wraz z rozwojem systemów wbudowanych coraz częściej to właśnie mikrokontrolery przyjmują rolę Hosta w komunikacji USB [13]. W proponowanym w pracy rozwiązaniu urządzeniem typu Host jest płytka STM32 NUCLEO-L476RG, a typu Device będzie klawiszowy instrument elektroniczny. Oznacza to, że cała komunikacja będzie zarządzana przez płytkę, która będzie odbierać dane, a następnie je przetwarzać.

3.2 Warstwy fizyczna i prędkości standardu

Standard USB 2.0 składa się z czterech przewodów: VBUS (zasilania), GND (masy) oraz pary różnicowej przesyłającej dane D+ oraz D- [11][12].



Rys. 7.: Kabel w standardzie USB, ***źródło: [12]***

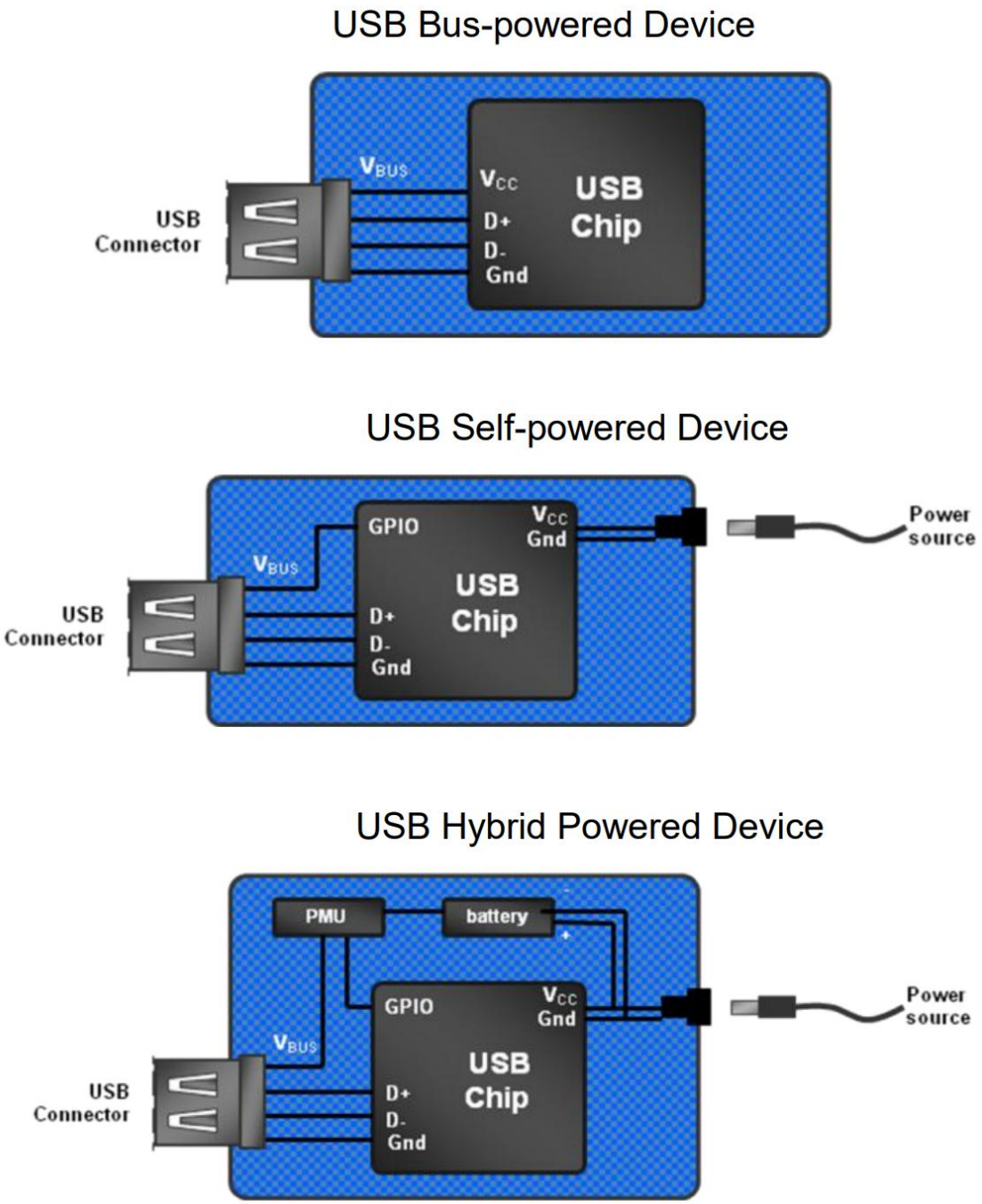
Dzięki wykorzystaniu pary różnicowej do przesyłania danych jesteśmy w stanie uzyskać lepsze warunki transmisji i tłumienie zakłóceń wspólnych, takich jak zakłócenia z innych elementów czy emisji EMI. Dodatkowo poprawia to jakość sygnału przy większych prędkościach [12].

Celem linii masy jest posiadanie wspólnego punktu odniesienia pomiędzy Hostem, a urządzeniem typu Device.

Linia VBUS spełnia dwie funkcje:

1. Pomaga Hostowi wykryć urządzenie typu Device – po podłączeniu, na linię VBUS jest podawane napięcie 5V, które jest odczytywane przez urządzenie zewnętrzne, dzięki czemu w kolejnym kroku mogą one ustalić prędkość transmisji i kontynuować inicjalizację transmisji [11][12].
2. W zależności od typu urządzenia zewnętrznego linia VBUS może także służyć jako linia zasilająca. Wyróżniamy dwa podstawowe typy urządzeń: bus-powered device i self-powered device, jednak często wyróżnia się też hybrid-powered device, które są deklarowane w deskryptorach jako self-powered, tylko z układami bateryjnymi [12]. Pierwszy typ jest zasilany bezpośrednio przez Hosta (przykład:

myszka podłączana kablem do komputera), urządzenia self-powered potrzebują ciągłego podłączenia do zewnętrznego źródła zasilania (np. drukarka), urządzenia zasilane hybrydowo domyślnie powinny być zasilane zewnętrznie, posiadają one jednak też własny układ bateryjny, który przy braku zewnętrznego zasilania, może być ładowany z wykorzystaniem linii VBUS [11].



Rys. 8.: Schematy połączeń przy różnych typach zasilania, ***źródło: [12]***

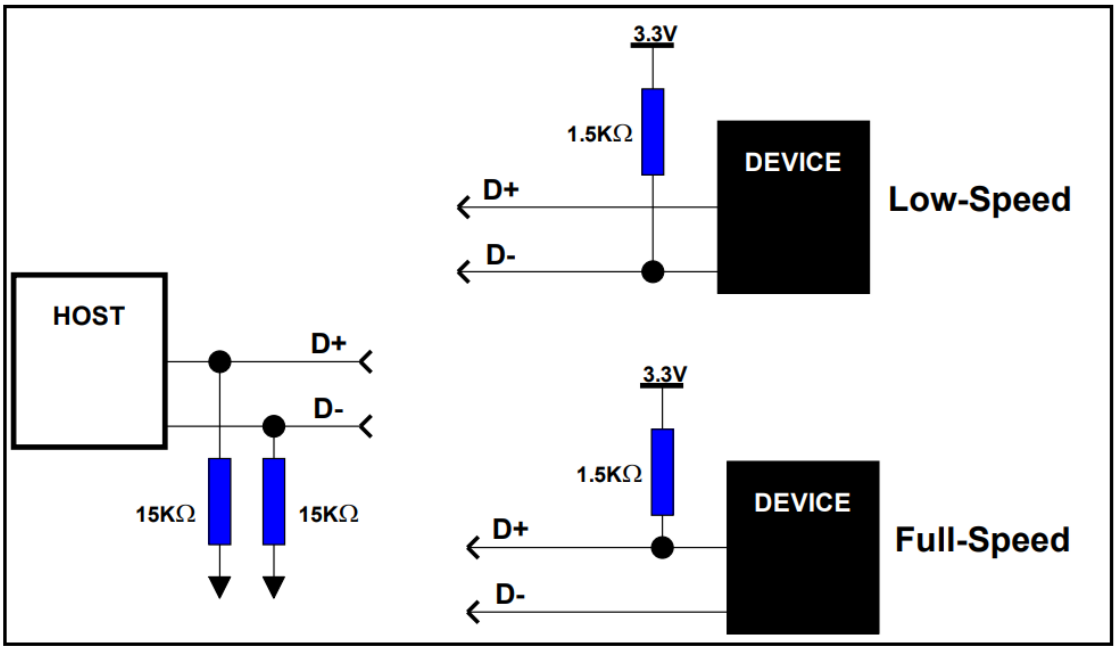
W przypadku klawiszowych instrumentów elektronicznych potrzebują one dużej ilości mocy, zatem zazwyczaj są urządzeniami typu self-powered.

W standardzie USB 2.0 wyróżniamy trzy różne prędkości transmisji, są to:

- Low Speed (1.5 Mb/s)
- Full Speed (12 Mb/s)
- High Speed (480 Mb/s)

Tutaj skupimy się tylko na inicjalizacji w przypadku dwóch pierwszych, ponieważ nasza transmisja z instrumentem klawiszowym będzie realizowana w trybie Full-Speed.

Po podaniu 5V linią VBUS na urządzenie typu Device urządzenie zgłasza prędkość transmisji jaką jest w stanie obsłużyć. Jest to realizowane poprzez ustawienie konkretnej linii danych na stan wysoki, dzięki połączeniu typu pull-up: D- dla transmisji Low-Speed, oraz D+ dla transmisji typu Full-Speed [11]. Host domyślnie posiada obie linie danych w połączeniu pull-down [12], dzięki czemu jest w stanie zidentyfikować, który tryb transmisji wskazuje urządzenie zewnętrzne.



Rys. 9.: Wskazanie trybu transmisji przez urządzenie typu Device, *źródło: [12]*

3.3 Pakiety i transakcje USB

Komunikacja w standardzie USB nie odbywa się w sposób ciągły, jest natomiast przesyłana w krótkich partiach danych, które nazywamy pakietami. Wyróżniamy trzy najważniejsze: Token Packet, Data Packet oraz Handshake Packet [11][12].

Pakiety typu Token służą do inicjowania komunikacji – niosą informację o tym z jakim urządzeniem chcemy nawiązać komunikację, z którym konkretnie endpointem (końcowym punktem poboru informacji) oraz kierunek transmisji, które dzielimy na IN, który informuje że host chce odczytać dane z urządzenia lub OUT, czyli host chce wysłać dane do urządzenia. Dodatkowo wyróżnia się specjalny typ Token Packetu - SETUP, który jest wykorzystywany w transferach kontrolnych do inicjowania komunikacji (np. podczas enumeracji) [11].

Data Packet zawiera dane użytkowe dla danego typu Token Packet. Maksymalny rozmiar pakietu danych jest zależny od szybkości transmisji, dla Low Speed jest to 8 bajtów. Przy transmisji Full Speed rozmiar pakietu jest zależny od operacji, często spotykany to 64 bajty [11].

Ostatni z pakietów, czyli Handshake Packet informuje o wyniku całej operacji. Występuje on w trzech stanach, z czego dla różnych pakietów typu Token Packet mają one różne znacznie, tutaj ogólnikowo: ACK – czyli potwierdzenie o poprawności wykonanej operacji, NAK – informujący o braku możliwości (poprzez np. brak danych na urządzeniu przy próbie odczytu) oraz STALL, który zgłasza błąd lub brak dostępu do danego endpointu. Jest to więc mechanizm dający informację zwrotną o statusie naszej operacji [11].

Wszystkie te trzy pakiety razem składają się na jedną transakcję USB – czyli kompletną wymianę informacji. Jak było wspomniane wcześniej – w standardzie USB to Host inicjuje całą komunikację, zatem w przypadku chęci odczytania informacji z urządzenia typu Device wysyłamy na odpowiedni adres pakiet Token Packet, który sygnalizuje kierunek transmisji IN, po odebraniu którego to urządzenie zewnętrzne nadaje Data Packet z właściwymi danymi użytkowymi. Na samym końcu Host kończy odbiór informacji wysyłając Handshake Packet. Oznacza to więc, że jeśli urządzenie posiada dane do wysłania nie może ich wysłać dopóki nie pojawi się odpowiednie żądanie od Hosta [11][12].

Przy wysyłaniu danych do urządzenia Host najpierw wysyła odpowiedni Token Packet, później Data Packet, a następnie czeka na odpowiedź od urządzenia zewnętrznego w postaci Handshake Packet, który poinformuje go o statusie operacji [11][12].

Komunikacja w USB jest zorganizowana czasowo, dzięki podziałowi na ramki czasowe. Host może w ten sposób planować cykliczne odpytywanie konkretnego urządzenia zewnętrznego w celu odbioru danych [11]. Przykładem może być tutaj cykliczny odbiór informacji z klawiatury.

Sama wymiana danych następuje jednak za pośrednictwem endpointów, czyli logicznych kanałów komunikacyjnych, które są zdefiniowane w urządzeniu peryferyjnym. Każdy z endpointów posiada kierunek transmisji (IN/OUT) określający do którego typu operacji może zostać wykorzystany, a także maksymalny rozmiar pojedynczego pakietu danych jaki może obsłużyć [11]. Aby nawiązać połączenie Host otwiera po swojej stronie pipe, który pozwala mu połączyć się z konkretnym endpointem. Dzięki temu transmisja odbywa się zawsze w konkretnie zdefiniowanych parametrach oraz endpointach [11].

W standardzie USB wyróżniamy także pojęcie transferu – jest to po prostu wiele transakcji USB, których celem jest przesłanie większej ilości danych lub wykonanie konkretnej operacji, np. inicjalizacji połączenia pomiędzy Hostem a urządzeniem zewnętrznych [11].

3.4 Enumeracja

Enumeracja to procedura inicjalizująca urządzenie USB typu Device, która jest wykonywana przez Hosta po podłączeniu urządzenia zewnętrznego do magistrali. Cel enumeracji to nawiązanie komunikacji, odczytanie możliwości podłączonego urządzenia oraz wybranie takiego typu konfiguracji, który pozwoli na dalszą wymianę informacji, co umożliwia poprawną pracę urządzenia peryferyjnego w ramach magistrali USB. Proces realizowany jest z wykorzystaniem transferów typu Control, a wymiana informacji następuje poprzez specjalny endpoint 0, który jest obowiązkowym endpointem kontrolnym w każdym urządzeniu USB [11].

Pierwszy etap enumeracji zaczyna się w momencie wykrycia podłączenia urządzenia zewnętrznego, kiedy to Host rozpoznaje obecność urządzenia na podstawie zmiany stanu linii danych D+ i D-, jak również określa tryb prędkości na podstawie sygnalizacji realizowanej przez rezystor pull-up po stronie urządzenia peryferyjnego [11][12]. Następnie Host resetuje magistralę do stanu początkowego, w celu doprowadzenia podłączonego urządzenia do stanu domyślnego, co przygotowuje je do dalszej komunikacji [11].

Każde urządzenie USB otrzymuje 7-bitowy adres nadawany przez Hosta, unikalny w ramach danej magistrali dzięki któremu Host jest w stanie identyfikować urządzenia [11]. W ustawieniach domyślnych każde urządzenie ma początkowo przypisany adres 0. Zatem pierwszą czynnością po resecie magistrali jaką wykonuje Host jest zmiana adresu urządzenia na unikalny, z wykorzystaniem komendy SET_ADDRESS [11].

W kolejnym kroku Host odczytuje z EP0 (skrót od endpoint 0) deskryptory urządzenia za pomocą Control Transfer, dzięki czemu dowiaduje się on o funkcjach jakie realizuje urządzenie, możliwościach konfiguracji oraz zasilania [11]. Same deskryptory są opisane bardziej szczegółowo w podnagłówku „3.5 – Deskryptory i ich hierarchia".

Gdy Host pobierze wszystkie informacje o urządzeniu zewnętrznym wybiera on konfigurację urządzenia, która określa sposób jego pracy oraz parametry zasilania (w praktyce wiele urządzeń udostępnia tylko jedną konfigurację, jednak standard dopuszcza ich większą liczbę [12]). Wybór konfiguracji jest realizowany poprzez komendę SET_CONFIGURATION. Po tym możliwa już jest standardowa komunikacja z wykorzystaniem wszystkich endpointów, a nie tylko EP0 [11].

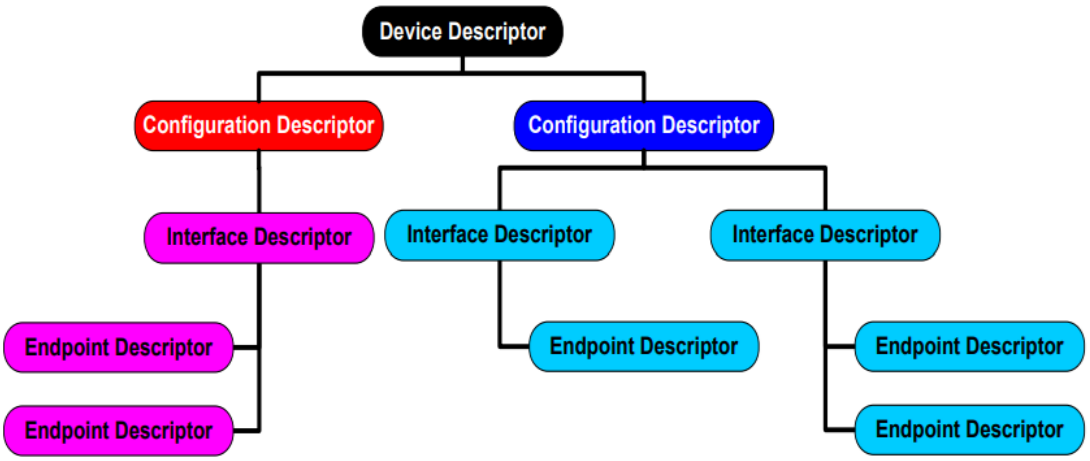
Host konfiguruje więc po swojej stronie kanały komunikacyjne, czyli pipe'y, które będą powiązane z odpowiednimi endpointami w celu wymiany informacji. Uruchamia on także odpowiednią obsługę klasy urządzenia (np. HID, Audio czy Mass Storage), co stanowi ostatni etap umożliwiający rozpoczęcie docelowej wymiany danych [11][13].

3.5 Deskryptory i ich hierarchia

Deskryptory w standardzie USB są strukturami danych, które opisują urządzenie peryferyjne oraz sposób komunikacji z nim. Odczytywane są one w procesie enumeracji,

dzięki czemu Host jest później w stanie właściwie skonfigurować połączenie z urządzeniem [11][12].

Deskryptory tworzą hierarchię, w której podstawową strukturą jest Device Descriptor, czyli deskryptor urządzenia. Kolejne deskryptory opisują możliwe do wyboru konfiguracje, interfejsy oraz dostępne w urządzeniu endpointy [11].



Rys. 10.: Hierarchiczna struktura deskryptorów, ***źródło: [12]***

Device Descriptor, czyli deskryptor urządzenia, zawiera podstawowe informacje, które identyfikują urządzenie i opisują jego ogólne parametry, takie jak identyfikator producenta czy produktu, wersje obsługiwanego standardu USB, rozmiar pakietu obsługiwanego przez EP0 oraz liczbę dostępnych konfiguracji, które oferuje urządzenie [11].

Następnie odczytywany jest Configuration Descriptor, czyli deskryptor konfiguracji. Opisuje on konkretny tryb pracy urządzenia, ponieważ standard dopuszcza istnienie wielu konfiguracji w ramach jednego urządzenia [12]. Do najważniejszych parametrów opisywanych przez ten deskryptor możemy zaliczyć: liczbę dostępnych intefejsow w tej konkretnej konfiguracji, informacje o trybie zasilania (bus-powered czy self-powered), a także deklarowany przez urządzenie pobór mocy (prąd podany tu jako 8 bitowa liczba w jednostce 2 mA). Configuration Descriptor zawiera również pole określające jaka jest całkowita długość zestawu deskryptorów należących do tej konkretnej konfiguracji, dzięki czemu Host może pobrać kompletną strukturę opisującą dalsze deskryptory (interfejsy oraz endpointy) [11].

Interface Descriptor, czyli deskryptor interfejsu, opisuje daną logiczną funkcję urządzenia peryferyjnego w ramach wybranej konfiguracji (np. interfejsy klasy HID, Audio, Mass Storage). Do jego najważniejszych parametrów należą pola opisujące klasę, podklasę i protokół interfejsu, na podstawie których Host dobiera odpowiedni sterownik klasy (class driver) [11]. Ma to bardzo duże znaczenie, ponieważ o ile protokół USB jest w stanie wykonać enumerację i zebrać informację o urządzeniu, to dopiero właściwie dobrany sterownik klasy pozwala realizować transmisję na endpointach oraz interpretować dane zgodnie ze specyfikacją danej klasy [11][12]. Deskryptor interfejsu informuje także o ilości endpointów, które są przypisane do konkretnego interfejsu [11].

Endpoint Descriptor, nazywany deskryptorem endpointu, opisuje nam pojedynczy logiczny kanał transmisji danych. Zawiera on wszystkie informacje potrzebne do wymiany danych, czyli adres endpointu, kierunek obsługiwanej transmisji, typ transferu (np. Bulk, Interrupt) oraz maksymalny rozmiar pakietu danych jaki potrafi obsłużyć. W przypadku niektórych typów transferu zawiera także informację o parametrach czasowych. Na podstawie tych informacji Host jest w stanie skonfigurować pipe, czyli kanał komunikacyjny po swojej stronie oraz rozpocząć wymianę danych z urządzeniem [11].

Poza obecnymi na „Rys. 10.” deskryptorami podstawowymi standard USB przewiduje również deskryptory dodatkowe [11], między innymi:

- String Descriptors, które przenoszą informacje tekstowe, które niosą wartość dodaną dla użytkownika (np. nazwa w menedżerze urządzeń na komputerze) [11][12].

- Class-specific Descriptors, które są deskryptorami specyficznymi dla klasy urządzenia. Są one wykorzystywane przez wybrane klasy czy też podklasy, takie jak np. Audio/MIDI, do opisu funkcji charakterystycznych dla danej klasy. Nie zawsze wymaganym jest ich obecność do zestawiania komunikacji, ale mogą one dostarczać dodatkowych informacji koniecznych do pełnej obsługi funkcji danej klasy/podklasy [11].

3.6 Typy transferów

W standardzie USB definiujemy cztery podstawowe typy transferów danych: Control, Bulk, Interrupt oraz Isochronous. Każdy z nich ma inaczej określone parametry

transmisji, do których zaliczamy: sposób planowania transmisji przez Hosta, poziom niezawodności (kontrola poprawności danych CRC, potwierdzenia i retransmisje) oraz wymagania czasowe (takie jak gwarancja pasma czy opóźnienia) [11][12].

Transfer typu Control jest wykorzystywany głównie w procesie enumeracji [11], więcej informacji można znaleźć w podnagłówku „3.4 Enumeracja".

Najważniejszym dla proponowanego w pracy rozwiązania jest transfer typu Bulk. Jest on wykorzystywany do przesyłania danych bez gwarancji czasowych (nie ma stałego czasu dostarczania), ponieważ jest on obsługiwany przez Hosta w ramach dostępnego czasu magistrali, dopiero po zrealizowaniu innych transferów o wyższych wymaganiach czasowych. Kluczową zaletą transferu typu Bulk jest wysoka niezawodność transmisji, która jest realizowana poprzez mechanizmy potwierdzeń, kontroli poprawności przesłanych danych CRC oraz retransmisji w przypadku wykrycia błędów [11]. W związku z tym jest on najczęściej używany w zastosowaniach, gdzie ważniejsza jest poprawność przesyłanych danych, a nie gwarancja czasowa ich przesłania. W trybie Full Speed bardzo często spotykanym rozmiarem pakietu dla endpointów tego typu są 64 bajty [11][13]. Przykładem urządzeń peryferyjnego wykorzystującego ten typ transmisji są pamięci masowe klasy USB Mass Storage, takie jak np. pendrive'y.

Kolejnym typem transferu jest Interrupt Transfer, który jest przeznaczony do cyklicznego przesyłania niewielkich porcji danych, które zwykle powinny być dostarczone jak najszybciej [11], typowym przykładem są urządzenia klasy HID, czyli między innymi klawiatury i myszki. Host planuje z określonym, zazwyczaj cyklicznym interwałem, odpytania urządzeń wykorzystujących ten typ transferu co zapewnia przewidywalność czasową. Interrupt Transfer jednocześnie zachowuje niezawodność transmisji, dzięki wykorzystaniu mechanizmów potwierdzeń oraz retransmisji [11].

Ostatnim typem transferu jest Isochronous Transfer, który jest stosowany w transmisjach strumieniujących dane w czasie rzeczywistym, takich jak np. audio czy wideo. W przypadku tego transferu Host przydziela konkretne, stałe pasmo w harmonogramie, co ma na celu zapewnienie stałego przepływu danych oraz ograniczenie opóźnień. W Isochronous Transfer nie robimy retransmisji w przypadku błędów, gdyż w przypadku takiej transmisji ważniejsze jest utrzymanie ciągłości danych – w związku z tym ubytki danych są akceptowalne [11].

W proponowanym w tej pracy rozwiązaniu zdecydowano się na realizację odbioru danych z elektronicznego instrumentu klawiszowego z wykorzystaniem endpointu typu Bulk IN, dzięki czemu zachowano poprawność przesyłanych zdarzeń MIDI oraz uzyskano prostszą integrację z implementacją Hosta po stronie aplikacyjnej mikrokontrolera. Brak gwarancji czasowej nie stanowi ograniczenia w tym zastosowaniu, ponieważ opóźnienia rzędu pojedynczych ramek są akceptowalne dla zdarzeń MIDI, a Host obsługuje tylko jedno urządzenie peryferyjne.

Rozdział 4

Komunikacja MIDI przez USB (USB-MIDI)

USB-MIDI to standard, który definiuje sposób przesyłania komunikatów MIDI z wykorzystaniem magistrali USB. Dzięki takiemu rozwiązaniu można traktować urządzenia muzyczne, takie jak elektroniczne instrumenty klawiszowe, jako urządzenia peryferyjne USB i przekazywać do Hosta zdarzenia MIDI [14]. W standardzie USB-MIDI komunikaty MIDI są przesyłane w ramach mechanizmów transferu danych USB, czyli to Host inicjuje i zarządza transmisją, co odróżnia ten typ wymiany danych od klasycznej transmisji szeregowej interfejsu MIDI [14].

Należy pamiętać, że USB-MIDI jest standardem definiującym klasę urządzeń MIDI oraz ich obsługi po stronie Hosta, posiada więc konkretną strukturę interfejsu oraz format przesyłania danych [14]. Po skończonej inicjalizacji transmisji i nawiązaniu komunikacji, czyli zakończeniu enumeracji zgodnej ze standardem USB, kiedy Host zna już możliwe konfiguracje, interfejsy oraz endpointy urządzenia, konieczne jest zastosowanie odpowiedniego sterownika klasy – czyli implementacja kodu obsługującego dane zgodnie ze standardem USB-MIDI. Dzięki temu Host jest w stanie poprawnie zinterpretować dane zgodnie ze specyfikacją USB-MIDI. Proces enumeracji umożliwia Hostowi uzyskanie dostępu do konfiguracji, interfejsów i endpointów urządzenia, natomiast interpretacja przesyłanych danych wynika ze specyfikacji USB-MIDI i jest realizowana przez sterownik klasy [14].

W ramach USB-MIDI dane są przesyłane jako zdarzenia w postaci 4-bajtowych pakietów (USB-MIDI Event Packets). Zawierają one bajty komunikatów MIDI (np. Note On czy Note Off) oraz informacje pozwalające określić ich typ i sposób interpretacji [14].

W dalszej części rozdziału przedstawiono bardziej szczegółowe informacje nt. standardu USB-MIDI – jego umiejscowienie w strukturze klas USB, deskryptory klasowe wykorzystywane do opisu funkcji MIDI oraz format przesyłania danych i sposób ich wykorzystania w aplikacji.

4.1 Umiejscowienie USB-MIDI w standardzie USB

Standard USB-MIDI nie jest zdefiniowany jako osobna klasa interfejsów, tak jak np. HID czy Mass Storage. Zamiast tego jest on klasyfikowany jako subklasa interfejsu Audio. W praktyce oznacza to, że interfejs MIDI jest identyfikowany na podstawie dwóch pól deskryptora interfejsu (Interface Descriptor) [14]:

- pole *bInterfaceClass* = 0x01 - jest to identyfikator interfejsu klasy Audio

- pole *bInterfaceSubClass* = 0x03 – jako subklasa MIDI Streaming

Dzieje się tak, ponieważ specyfikacja USB-MIDI przewiduje rozdzielenie funkcji kontrolnych oraz przesyłania danych, dzięki czemu zarządzanie i konfiguracja są oddzielone od właściwej transmisji danych. Interfejs kontrolny, czyli Interfejs Audio Control zapewnia mechanizmy sterowania i konfiguracji funkcji klasy Audio (w tym także funkcji MIDI). Subklasa MIDI Streaming jest za to interfejsem przesyłania danych i odpowiada za właściwy transport danych [14].

Dla rozwiązania proponowanego w pracy oznacza to, że konieczne jest zastosowanie po stronie Hosta (czyli mikrokontrolera) sterownika klasy, który na podstawie standardu USB-MIDI będzie umożliwiał właściwą komunikację z elektronicznym instrumentem klawiszowym [14].

4.2 Deskryptory USB-MIDI i elementy klasowe

Każde urządzenie peryferyjne USB jest opisane z wykorzystaniem deskryptorów podstawowych – Device, Configuration, Interface oraz Endpoint. Standard USB-MIDI wykorzystuje dodatkowo deskryptory specyficzne dla klasy (Class-Specific Descriptors), które to są przypisane do podklasy MIDI Streaming [14]. Deskryptory te doprecyzowują, w jaki sposób funkcja MIDI jest zorganizowana w urządzeniu – wskazują dostępne porty MIDI oraz ich logiczne połączenia [14].

Kluczowym elementem USB-MIDI są tzw. gniazda MIDI (MIDI Jacks) – czyli opisane w deskryptorach wejścia i wyjścia MIDI, które Host widzi jako osobne porty MIDI, nawet jeśli nie są one fizycznymi złączami na obudowie [14]. Standard zachowuje podział zgodny z USB jeśli chodzi o kierunku transmisji – definiujemy zatem gniazda MIDI IN Jack oraz MIDI OUT Jack [14]. Dodatkowo wprowadza on podział gniazd na Embedded, czyli takie które są osadzone w urządzeniu peryferyjnym USB i reprezentują

wewnętrzne funkcje MIDI, oraz gniazda typu External, które reprezentują porty widziane przez Hosta jako zewnętrzne wejścia lub wyjścia MIDI [14].

Dzięki wykorzystaniu gniazd MIDI Host jest w stanie zidentyfikować i opisać udostępniane przez urządzenie elementy funkcji MIDI na poziomie logicznym portów [14].

4.3 Endpointy i tryb transmisji w USB-MIDI

W standardzie USB-MIDI przesyłanie danych jest realizowane w ramach interfejsu MIDI Streaming (klasa Audio, podklasa MIDI Streaming), który posiada endpointy przeznaczone do transportu zdarzeń MIDI [14]. Komunikacja z urządzenia peryferyjnego do Hosta odbywa się z wykorzystaniem endpointu o kierunku IN. Możliwe jest też wysyłanie danych do urządzenia peryferyjnego przez Hosta, poprzez endpoint o kierunku OUT [14], natomiast w tej pracy takie rozwiązanie nie jest wspierane. Endpointy te są opisane w odpowiednich deskryptorach (Endpoint Descriptors), a deskryptory klasowe MIDI Streaming uzupełniają opis o elementy MIDI (gniazda MIDI) i ich powiązanie z endpointami [14].

Transmisja danych w USB-MIDI jest realizowana zgodnie ze standardem USB, czyli to Host inicjuje transakcje oraz planuje ich wykonanie w czasie. Dane są więc wysyłane do Hosta dopiero kiedy ten wyśle na urządzenie peryferyjne odpowiednie żądanie. Urządzenie buforuje więc dane do czasu kolejnego żądania IN, do maksymalnego rozmiaru pakietu endpointu, typowo 64 bajty w Full Speed [14].

W ramach USB-MIDI wykorzystujemy w endpointach transfer typu Bulk, który gwarantuje niezawodny odbiór zdarzeń MIDI, dzięki mechanizmom CRC oraz retransmisji w przypadku błędów [14].

4.4 Format danych: USB-MIDI Event Packet

Standard USB-MIDI przesyła dane inaczej niż w klasycznym interfejsie MIDI, w którym komunikaty mają zmienną długość. Tutaj przesyłane są one w postaci stałej struktury o długości 4 bajtów, nazywanej USB-MIDI Event Packet [14]. W każdym pakiecie przesyłane jest pojedyncze zdarzenie MIDI lub ich fragmenty (w przypadku

pakietów systemowych SysEx), dzięki czemu Host zawsze odbiera dane jako sekwencje o stałej długości [14].

Rys. 11.: Struktura pakietu USB-MIDI Event Packet, *źródło: [14]*

Pierwszy bajt pakietu składa się z dwóch pól: Cable Number (CN) oraz Code Index Number (CIN).

Pole CN zajmuje cztery starsze bity i identyfikuje ono tzw. wirtualny kabel, czyli logiczny port MIDI w ramach strumienia USB. Dzięki temu możliwe jest przesyłanie jednocześnie danych z wielu portów MIDI w ramach jednego endpointu. W praktyce prostsze urządzenia zawierają tylko jeden port o wartości CN = 0 [14].

Pole CIN obejmuje cztery młodsze bity i określa nam typ przenoszonego w tym pakiecie komunikatu (np. 0x8 dla komunikatów Note Off oraz 0x9 dla komunikatów Note On), dzięki czemu możliwa jest właściwa interpretacja przenoszonych przez pakiet bajtów danych i tym samym wskazuje, ile bajtów w polach danych jest istotnych (np. w przypadku komunikatów krótszych lub fragmentów SysEx) [14].

Ostatnie trzy bajty pakietu zawierają już właściwe dane MIDI, zgodne ze standardem MIDI [14], opisanym w podnagłówku „1.4 – Typy komunikatów MIDI". Oznacza to, że dla proponowanego w pracy rozwiązania najważniejsze będą komunikaty Note On oraz Note Off. Wykorzystują one trzy bajty – pierwszy to zawsze bajt statusu, następnie mamy dwa bajty danych. Szczegółowy opis tych komunikatów przedstawiono we wspomnianym powyżej nagłówku.

Z perspektywy aplikacyjnej implementacji sterownika klasy dla Hosta, pakiety USB-MIDI Event Packet upraszczają odbiór danych, gdyż odbiór danych z endpointu można przetwarzać w odpowiednim buforze jako sekwencje o stałej długości czterech bajtów [14].

4.5 Przepływ danych w projekcie

W proponowanym przez prace rozwiązaniu elektroniczny instrument klawiszowy (keyboard) pełni rolę urządzenia peryferyjnego wykorzystującego standard USB-MIDI. Rolę Hosta pełni mikrokontroler STM32 NUCLEO-L476RG, który inicjuje odczyt danych z endpointu typu Bulk IN, z wykorzystaniem interfejsu MIDI Streaming.

Odbierane dane mają postać sekwencji 4-bajtowych pakietów USB-MIDI Event Packet, w których zawierają się komunikaty MIDI (Note On/Off) wraz z danymi. Po stronie aplikacji odbierane zdarzenia są interpretowane i wykorzystywane do dalszej logiki działania systemu.

Dokładne szczegóły implementacji są przedstawione w części praktycznej.

Rozdział 5

Część praktyczna

5.1 Wybrane środowiska i narzędzia

W realizacji projektu wykorzystano narzędzia wspomagające poszczególne etapy pracy:

- konfiguracji mikrokontrolera i generacji kodu inicjalizacyjnego (program STM32CubeMX)

- implementację oraz debugowanie oprogramowania (STM32CubeIDE)

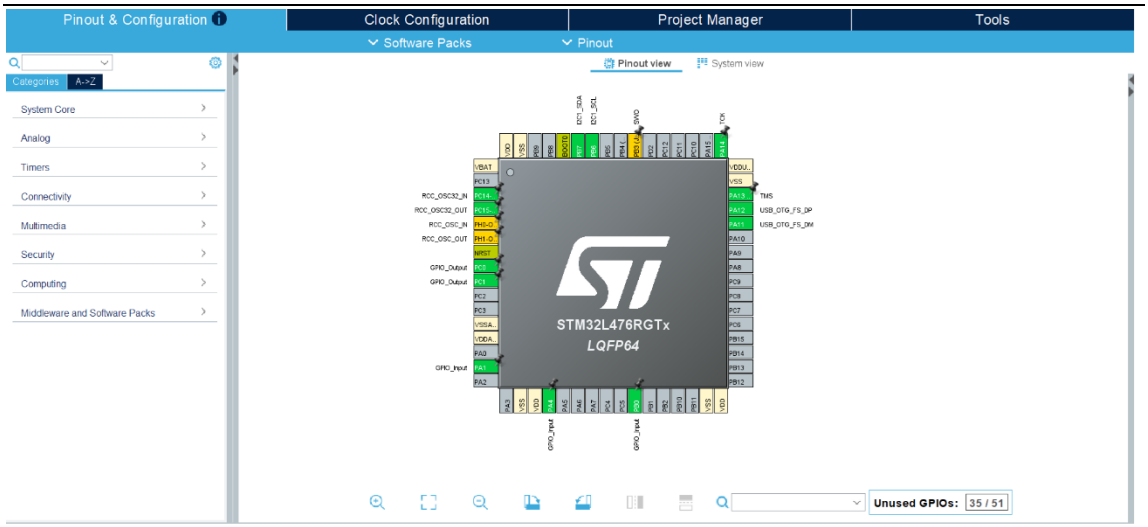
- opracowanie schematu elektrycznego układu (Altium Designer)

Dobór dwóch pierwszych narzędzi został podyktowany ich szerokim wsparciem dla mikrokontrolerów STM32.

5.1.1 STM32CubeMX

STM32CubeMX to narzędzie firmy STMicroelectronics, które służy do wstępnej konfiguracji mikrokontrolerów z rodziny STM32 oraz generowania kodu inicjalizacyjnego w języku C. Program ten umożliwia wybór konkretnego układu lub płytki, a następnie jej konfiguracje zasobów sprzętowych poprzez mapowanie pinów (np. włączanie konkretnych funkcji), ustawiania parametrów peryferiów czy częstotliwości zegarów. Na podstawie konfiguracji CubeMX jest w stanie wygenerować szkielet projektu oraz kod inicjalizacyjny, który jest zgodny z środowiskiem STM32Cube [15]. Pozwala to ograniczyć liczbę błędów na etapie wstępnej konfiguracji mikrokontrolera. Zawiera on także wiele przydatnych middleware'ów, takich jak FreeRTOS, czy wykorzystane w tej pracy rozszerzenie USB Host.

Ogromną zaletą tego narzędzia jest również możliwość łatwej i szybkiej zmiany konfiguracji płytki – program pozwala na ponowną generację kodu po zmianie konfiguracji, która nie wpływa na kod napisany przez użytkownika, dopóki jest on pisany w odpowiednich polach dla niego przeznaczonych [15].



Rys. 12.: Przykładowa konfiguracja w środowisku STM32CubeMX, *źródło: opracowanie własne*

W proponowanym w tej pracy rozwiązaniu wykorzystano STM32CubeMX do utworzenia pliku konfiguracyjnego projektu (.ioc) oraz do wygenerowania kodu startowego, który był fundamentem dla dalszego rozwoju implementacji aplikacji. Konieczne było skonfigurowanie w odpowiedni sposób interfejsu USB, I2C oraz linii GPIO.

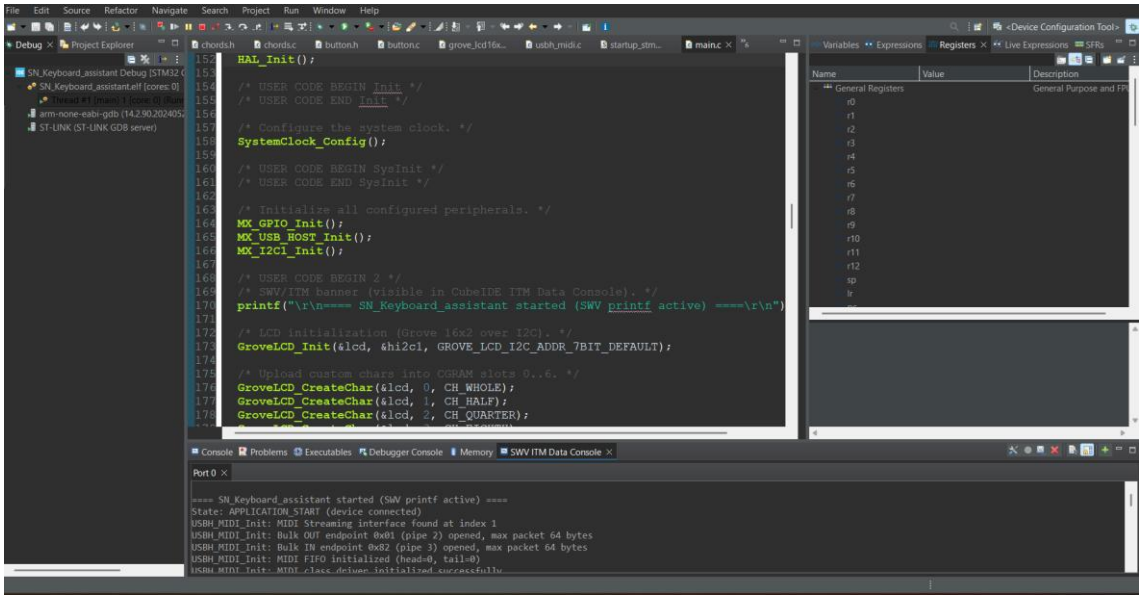
5.1.2 STM32CubeIDE – dopisz tutaj o HAL jeszcze (1 zdanie)

STM32CubeIDE to zintegrowane środowisko programistyczne (IDE) firmy STMicroelectronics, które jest przeznaczone do tworzenia aplikacji dla mikrokontrolerów rodziny STM32 w językach C/C++. Środowisko wykorzystuje toolchain GCC do kompilacji oraz debugger GDB do uruchamiania sesji debugowania [16].

STM32CubeIDE integruje bibliotekę HAL (Hardware Abstraction Layer), która udostępnia wysokopoziomowe funkcje do konfiguracji i obsługi peryferiów, upraszczając przenoszenie kodu i skracając czas implementacji.

W niniejszym projekcie kluczowe znaczenie miały możliwości debuggera dostępnego w STM32CubeIDE. Środowisko udostępnia bowiem podstawowe narzędzia debugowania, takie jak punkty przerwań oraz kroki wykonywania programu, jak i bardziej zaawansowane, do których należą między innymi możliwość podglądu rejestrów

CPU, pamięci czy peryferiów [16], co ułatwia weryfikację działania programu i diagnozowanie błędów.

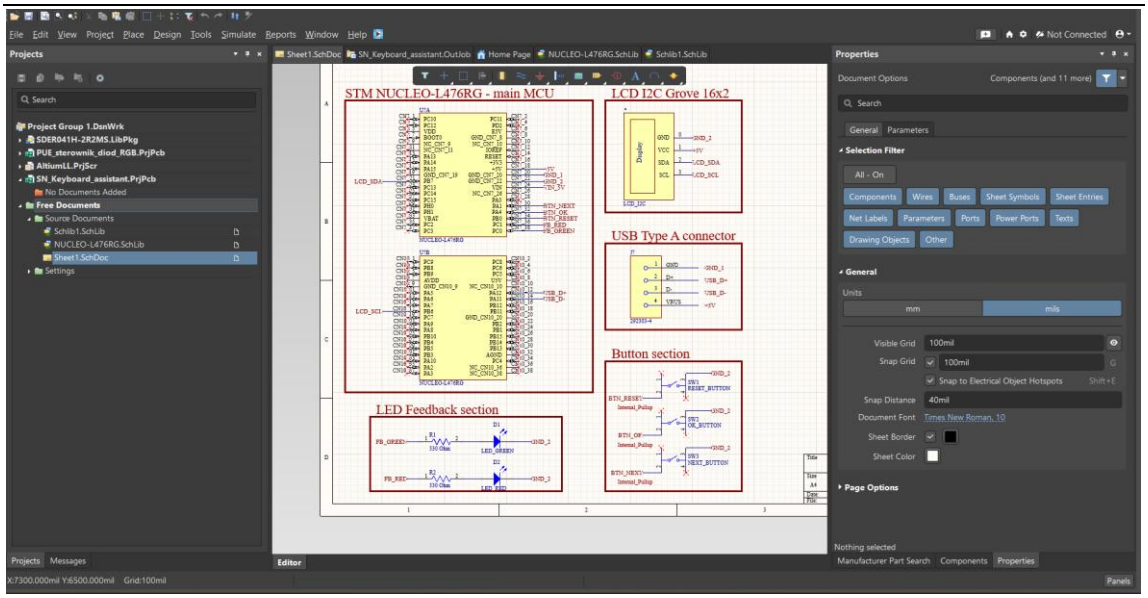


Rys. 13.: Środowisko STM32CubeIDE z włączonym debugowaniem, *źródło: opracowanie własne*

Dodatkową pomocą są także mechanizmy umożliwiające podgląd zmiennych w czasie działania programu (mechanizmy typu Live Watch) i wsparcie dla interfejsu Serial Wire Viewer (SWV) [16].

5.1.3 Altium Designer

Altium Designer jest środowiskiem EDA (Electronic Design Automation) przeznaczonym do projektowania elektroniki, które obejmuje przygotowanie schematów elektrycznych oraz projektowanie obwodów drukowanych (PCB) w ramach jednego, spójnego projektu. Do możliwości tego narzędzia należą m.in.: tworzenie dokumentów schematowych, korzystanie z bibliotek komponentów przygotowanych przez producentów lub społeczność, tworzenie własnych bibliotek, definiowanie połączeń pomiędzy elementami oraz weryfikację poprawności działania projektu na poziomie zarówno logicznym, jak i elektrycznym [17].



Rys. 14.: Środowisko Altium Designer, *źródło: opracowanie własne*

W niniejszej pracy narzędzie wykorzystano jedynie do opracowania schematu elektrycznego układu w celu przedstawienia jego struktury sprzętowej – pokazania połączeń pomiędzy konkretnymi elementami oraz interfejsami. Wykorzystanie w tym celu programu Altium Designer pozwala jednak na łatwą rozbudowę projektu o etap projektowania płytki PCB w przyszłości, ponieważ schemat wraz z listą połączeń stanowi podstawę do utworzenia PCB w ramach tego samego projektu [17].

5.2 Schemat projektu oraz elementy składowe

Układ został podzielony na pięć bloków funkcjonalnych:

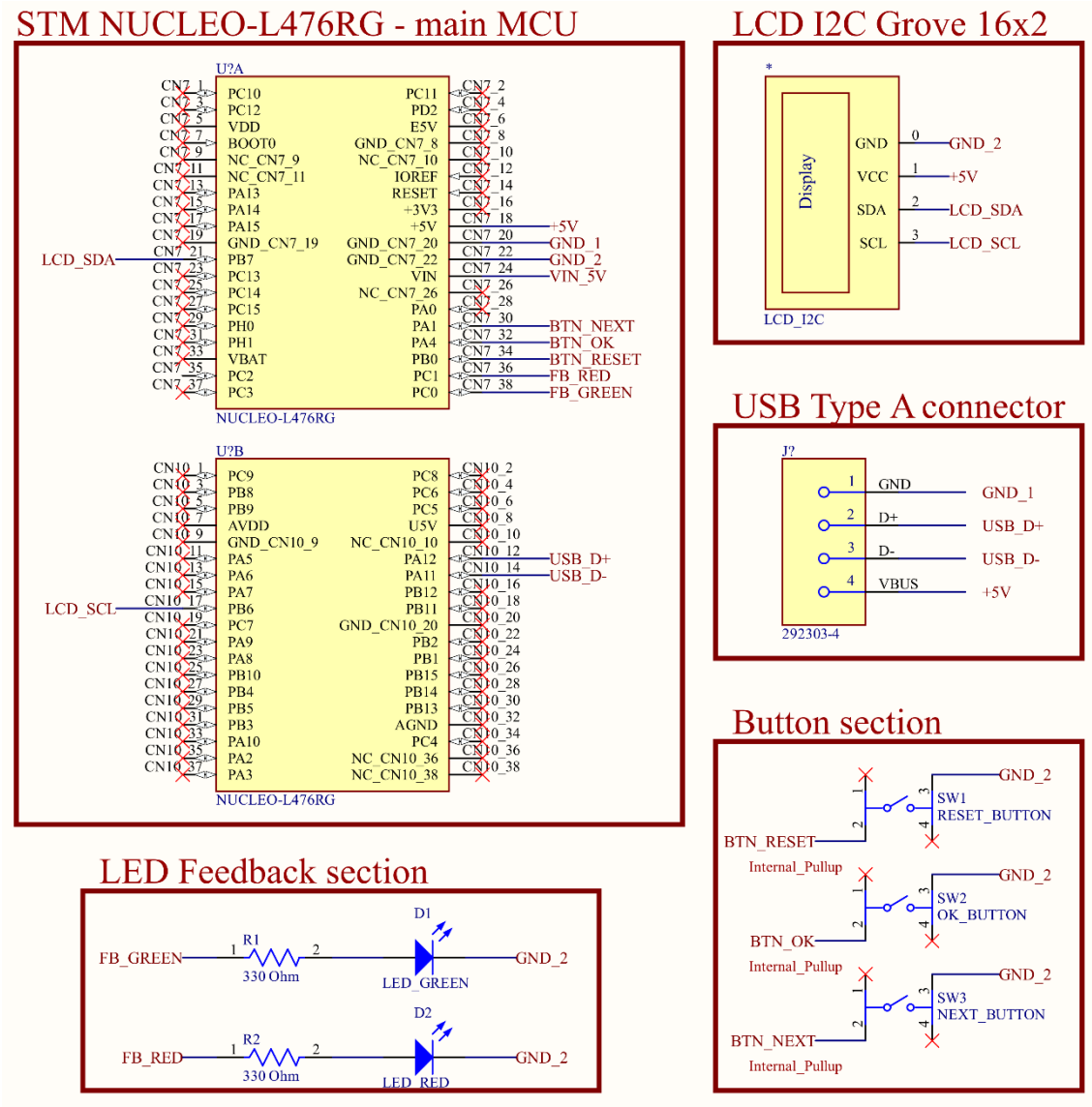
a) moduł mikrokontrolera (NUCLEO-L476RG),

b) interfejs USB Host (złącze USB Type-A),

c) wyświetlacz LCD 16×2 sterowany po I2C (Grove-16x2 LCD),

d) sekcję trzech przycisków sterujących (mikroprzyciski typu Tact Switch),

e) sekcję dwóch diod LED odpowiedzialnych za informację zwrotną (zielona i czerwona).



Rys. 15.: Schemat elektryczny układu, ***źródło: opracowanie własne***

a) Jednostka sterująca – STM32 NUCLEO-L476RG

Głównym elementem projektu jest płytka NUCLEO-L476RG z mikrokontrolerem STM32L476RG. Mikrokontroler realizuje logikę aplikacji (menu użytkownika oraz tryby nauki), obsługę USB w trybie Host, komunikację z wyświetlaczem po magistrali I2C oraz przetwarzanie wejść z przycisków i sterowanie diodami LED.

b) Interfejs USB Host dla elektronicznego instrumentu klawiszowego USB-MIDI

Instrument jest podłączany do złącza USB Type-A. Do mikrokontrolera doprowadzone są linie danych D+ (PA12) i D− (PA11) kontrolera USB OTG FS (Full-Speed) [19].

Linia VBUS doprowadza napięcie +5 V z linii zasilania płytki. Na podstawie odczytu deskryptora konfiguracji urządzenie deklaruje tryb self-powered (bmAttributes = 0xC0) oraz bMaxPower = 0 mA [źródło: Dodatek A]. VBUS jest jednak wykorzystywane do detekcji podłączenia hosta i inicjacji enumeracji [11].

Po stronie programowej urządzenie jest enumerowane przez middleware USB Host, a następnie przez część aplikacyjną (własna klasa/sterownik USB-MIDI) odbierane są zdarzenia MIDI.

c) Wyświetlacz tekstowy Grove LCD 16×2 (I2C)

W celu prezentacji menu użytkownika, list wyboru oraz informacji o aktualnym kroku lekcji zastosowano wyświetlacz Grove LCD 16×2 sterowany po magistrali I2C. Na schemacie widoczny jest moduł LCD z liniami SDA (PB7) i SCL (PB6) oraz zasilaniem VCC, które wynosi +5 V, ale układ jest też przystosowany do pracy przy zasilaniu +3.3 V [18]. Ostatnia linia to wspólna z mikrokontrolerem masa GND.

d) Sekcja przycisków sterujących (RESET / OK / NEXT)

Interfejs użytkownika ograniczono do trzech przycisków: RESET, OK oraz NEXT. Każdy przycisk jest podłączony w konfiguracji do masy (GND), a po stronie mikrokontrolera piny GPIO skonfigurowano z wewnętrznym rezystorem podciągającym do napięcia 3.3 V (Internal Pull-Up) [19]. Oznacza to, że stanem aktywnym jest stan niski – w stanie spoczynku wejście pinu ma stan wysoki. Na schemacie linie sygnałowe są oznaczone jako BTN_RESET, BTN_OK, BTN_NEXT i prowadzą bezpośrednio do pinów płytki.

e) Sekcja diod LED – informacja zwrotna (zielona i czerwona)

W projekcie zastosowano dwie diody w celu zapewnienia użytkownikowi prostej, natychmiastowej informacji zwrotnej. Przy każdym wciśnięciu klawisza instrumentu zaświeci się jedna z dwóch diod: zielona (poprawna dla danego kroku lekcji nuta) oraz czerwona (niepoprawna).

Diody sterowane są z wyjść mikrokontrolera FB_GREEN oraz FB_RED przez rezystory ograniczające prąd o wartości 330 Ω (R1 oraz R2). Stanem aktywnym jest stan wysoki na linii sterującej.

5.3 Część aplikacyjna – architektura i opis oprogramowania

5.3.1 Ogólny opis działania programu

Oprogramowanie projektu zrealizowano w architekturze modułowej, w której to wyróżniono:

- warstwę inicjalizacji i obsługi peryferiów (HAL + middleware),

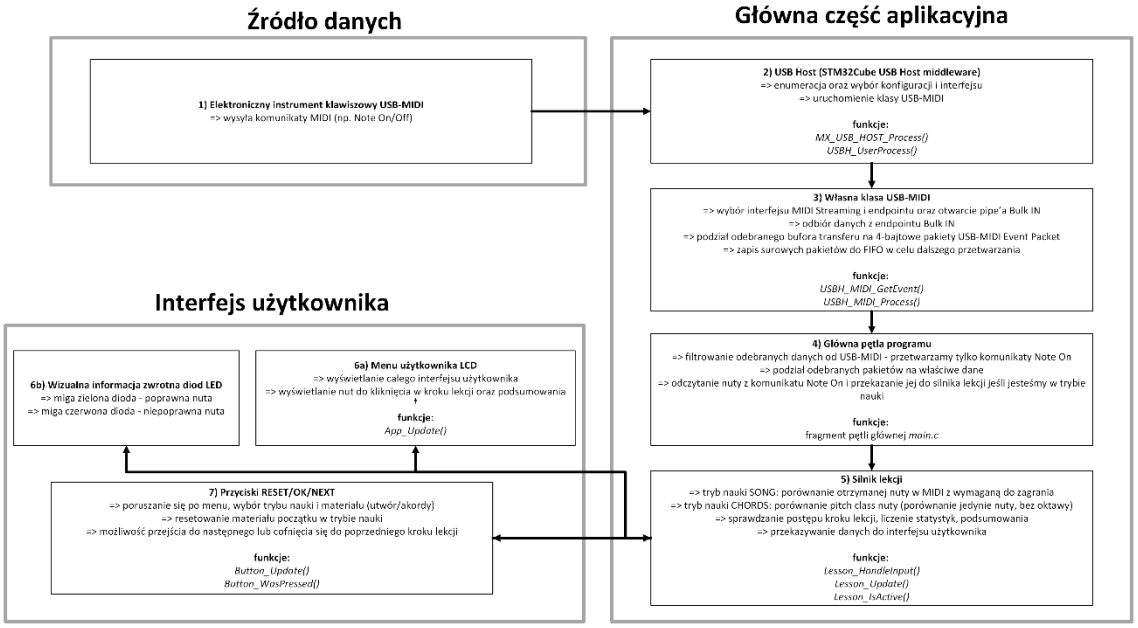
- warstwę sterowników (USB-MIDI, LCD, przyciski),

- warstwę logiki aplikacyjnej (menu użytkownika oraz silnik lekcji).

Wszystkie zadania są wykonywane cyklicznie w pętli głównej, a funkcje wymagające odmierzania czasu, takie jak sygnalizacja LED czy filtracja drgań styków przycisków, zrealizowano w sposób nieblokujący z wykorzystaniem licznika systemowego *HAL_GetTick()* (czas mierzony w ms).

Po uruchomieniu mikrokontrolera wykonywana jest standardowa inicjalizacja HAL – konfigurowane są zegary oraz peryferia, które zostały skonfigurowane wcześniej w STM32CubeMX: piny GPIO (przyciski oraz diody LED), magistrala I2C (wyświetlacz LCD) oraz USB w trybie Host. W kolejnym kroku inicjalizowany jest wyświetlacz Grove LCD 16x2, a do jego pamięci CGRAM ładowane są własne znaki – utworzone na potrzebę systemu lekcji symbole długości nut oraz znaki chromatyczne (bemol, krzyżyk). Po stronie aplikacyjnej uruchamiana jest maszyna stanów menu użytkownika, która odpowiada za wybór trybu nauki i uruchomienie lekcji.

Na rysunku poniżej pokazano przepływ danych w ujęciu funkcjonalnym projektu (źródło danych, część aplikacyjna, interfejs użytkownika). W obrębie głównej części aplikacyjnej można wyróżnić wypisane wcześniej warstwy: inicjalizacji (HAL/middleware), sterowników oraz logiki aplikacyjnej.



Rys. 16.: Schemat blokowy całego systemu, *źródło: opracowanie własne*

Schemat blokowy przedstawia ogólny przepływ informacji w systemie oraz podział odpowiedzialności pomiędzy konkretnymi modułami oprogramowania. Dane wejściowe pochodzą z elektronicznego instrumentu klawiszowego USB-MIDI i są przetwarzane przez warstwę USB Host (STM32Cube USB Host middleware), która realizuje enumerację urządzenia oraz uruchamia odpowiednią klasę.

Następnie sterownik klasy USB-MIDI odbiera dane z endpointu Bulk IN, dzieli odebrany bufor transferu na 4-bajtowe pakiety USB-MIDI Event Packet oraz buforuje surowe pakiety w kolejce FIFO w celu dalszego przetwarzania.

W kolejnych iteracjach pętli głównej programu zdarzenia MIDI są pobierane z kolejki FIFO, a następnie filtrowane (wykorzystywane są jedynie komunikaty Note On z niezerową prędkością velocity, które odpowiadają za wciśnięcie klawisza na instrumencie), po czym informacja o zagranej nucie przekazywana jest do silnika lekcji.

Moduł lekcji definiuje dwa tryby nauki: tryb utworu (SONG) lub tryb akordów (CHORDS), które lekko różnią się sposobem przetwarzania odczytanych z instrumentu

danych. Tryb nauki składa się z „kroków lekcji”, gdzie jeden krok składa się z jednej do trzech nut. Moduł lekcji weryfikuje i aktualizuje postępy realizacji kolejnych kroków oraz przekazuje użytkownikowi informację zwrotną. Wyniki działania silnika lekcji prezentowane są użytkownikowi w postaci komunikatów na wyświetlaczu LCD oraz sygnalizacji diodami LED. Równolegle obsługiwane są trzy przyciski sterujące, które umożliwiają nawigację po menu oraz sterowanie przebiegiem lekcji, zależnie od aktualnego stanu aplikacji.

5.3.2 Podział modułów oprogramowania

W celu uporządkowania struktury programu dokonano podziału oprogramowania na moduły odpowiadające wybranym funkcjom systemu.

Tabela poniżej przedstawia zestawienie plików źródłowych i nagłówkowych wraz z ich rolą w systemie. Dodatkowo wskazano odniesienie do bloków schematu blokowego systemu (Rys. 16.), co ułatwia powiązanie struktury kodu z architekturą funkcjonalną aplikacji.

Tabela 1.: Zestawienie modułów oprogramowania, plików projektu oraz ich odniesienia do schematu blokowego (Rys. 16.)

Moduł	Pliki	Rola w systemie	Odniesienie w schemacie blokowym (Rys. 16.)
Inicjalizacja systemu i pętla główna programu	*main.c/.h*	Inicjalizacja peryferiów (GPIO/I2C/USB Host), start LCD i menu użytkownika, cykliczna obsługa całego systemu w pętli oraz filtracja odebranych komunikatów MIDI	*4 oraz etap inicjalizacji*
Obsługa USB w trybie Host (STM32Cube middleware)	*usb_host.c/.h*	Inicjalizacja stosu, enumeracja urządzenia, rejestracja klasy USB-MIDI, cykliczne przetwarzanie zdarzeń hosta. Udostępnia stan aplikacji USB (połączony, odłączony)	*2*

Sterownik klasy USB-MIDI po stronie Hosta (klasa własna)	*usbh_midi.c/.h*	Wybór interfejsu MIDI Streaming oraz endpointów, otwarcie pipe Bulk IN, odbiór danych, podział bufora transferu na 4-bajtowe pakiety USB-MIDI Event Packet i buforowanie w FIFO	*3*
Silnik lekcji i weryfikacja poprawności (feedback LED/LCD)	*lesson.c/.h*	Tryb SONG i CHORDS, obsługa kroków lekcji, weryfikacja nut (dokładny MIDI lub pitch class), statystyki i podsumowanie, nieblokujące sterowanie LED (feedback), generowanie treści na LCD w czasie lekcji	*5 + 6b*
Menu interfejsu użytkownika oraz system nawigacji menu	*app.c/.h*	Maszyna stanów interfejsu: ekran powitalny, menu główne, listy (utwory/paczki akordów), start lekcji, przekazywanie przycisków do silnika lekcji w trybie nauki	*6a*
Filtracja drgań zestyków oraz obsługa przycisków	*button.c/.h*	Odczyt przycisków i detekcja naciśnięć z filtrowaniem drgań styków	*7*
Sterownik LCD Grove 16×2 po I2C	*grove_lcd16x2_i2c.c/.h*	Inicjalizacja, komendy, wyświetlanie tekstu, pozycjonowanie kursora i znaki własne (CGRAM)	*6a*
Baza danych utworów (kroki lekcji trybu nauki SONG)	*songs.c/.h*	Dane utworów dla trybu SONG: tytuły, kroki lekcji, nuty do zagrania (do 3 w jednym kroku), symbole długości nut	*Dane dla 5 i 6a*
Baza danych akordów (kroki lekcji trybu nauki CHORDS)	*chords.c/.h*	Dane paczek akordów dla trybu CHORDS: zestawy akordów, nazwy akordów i ich składowe (do 3 nut w jednym kroku)	*Dane dla 5 i 6a*
Mapowania nut i pomocnicze operacje na MIDI	*notes.c/.h*	Funkcje pomocnicze do parsowania nazw nut i konwersji na numer MIDI / z MIDI na nazwę (użyteczne w testach i rozwoju danych)	*Brak – narzędzie pomocnicze*

5.3.3 Inicjalizacja systemu i pętla główna *(main.c/.h)*

a) Rola plików

Pliki *main.c* oraz *main.h* stanowią punkt wejścia programu oraz miejsce inicjalizacji peryferiów skonfigurowanych w STM32CubeMX.

Plik źródłowy *main.c* zawiera wywołania funkcji inicjalizacyjnych wygenerowanych przez CubeMX (HAL, zegary, GPIO, USB Host, I2C), a następnie uruchamia część aplikacyjną projektu (LCD, menu). W pętli głównej realizowane jest cykliczne utrzymanie stosu USB Host, odczyt zdarzeń MIDI, obsługa przycisków oraz aktualizacja interfejsu użytkownika.

Plik nagłówkowy *main.h* pełni rolę wspólnego nagłówka projektu: dołącza bibliotekę HAL (*stm32l4xx_hal.h*) oraz zawiera definicje makr sprzętowych (m.in. piny LED oraz przycisków *RESET*, *OK*, *NEXT*). W projekcie zastosowano także dyrektywy preprocesora *#error*, które pomagają wykryć brakujące mapowania pinów na etapie budowania programu.

b) Sekwencja startowa programu - inicjalizacja

W funkcji *main()* wykonywana jest standardowa sekwencja uruchomieniowa: *HAL_Init()* – reset peryferiów oraz SysTick, konfiguracja zegarów poprzez *SystemClock_Config()*, a następnie inicjalizacja peryferiów GPIO, USB Host oraz I2C.

Po stronie aplikacyjnej wykonywana jest inicjalizacja wyświetlacza Grove LCD 16×2 funkcją *GroveLCD_Init()*, a następnie do pamięci CGRAM przesyłane są własne znaki – symbole długości nut oraz znaki chromatyczne. Dzięki temu interfejs użytkownika może prezentować elementy notacji muzycznej mimo ograniczeń tekstowego wyświetlacza 16×2.

```
152   HAL_Init();
153
154   /* USER CODE BEGIN Init */
155   /* USER CODE END Init */
156
157   /* Configure the system clock. */
158   SystemClock_Config();
159
160   /* USER CODE BEGIN SysInit */
161   /* USER CODE END SysInit */
162
163   /* Initialize all configured peripherals. */
164   MX_GPIO_Init();
165   MX_USB_HOST_Init();
166   MX_I2C1_Init();
167
168   /* USER CODE BEGIN 2 */
169   /* SWV/ITM banner (visible in CubeIDE ITM Data Console). */
170   printf("\r\n==== SN_Keyboard_assistant started (SWV printf active) ====\r\n");
171
172   /* LCD initialization (Grove 16x2 over I2C). */
173   GroveLCD_Init(&lcd, &hi2c1, GROVE_LCD_I2C_ADDR_7BIT_DEFAULT);
174
175   /* Upload custom chars into CGRAM slots 0..6. */
176   GroveLCD_CreateChar(&lcd, 0, CH_WHOLE);
177   GroveLCD_CreateChar(&lcd, 1, CH_HALF);
178   GroveLCD_CreateChar(&lcd, 2, CH_QUARTER);
179   GroveLCD_CreateChar(&lcd, 3, CH_EIGHTH);
180   GroveLCD_CreateChar(&lcd, 4, CH_SIXTEENTH);
181   GroveLCD_CreateChar(&lcd, 5, CH_SHARP);
182   GroveLCD_CreateChar(&lcd, 6, CH_FLAT);
```

Rys. 17.: Inicjalizacja startowa w projekcie, *źródło: opracowanie własne*

Na końcu inicjalizacji uruchamiana jest maszyna stanów menu użytkownika funkcją *App_Init()*, która odpowiada za ekran powitalny, przejście do menu głównego oraz nawigację po listach i uruchamianie lekcji.

c) Pętla główna – koordynacja pracy modułów

Następnie program przechodzi do wykonywania pętli głównej, gdzie cyklicznie są wykonywane 4 główne kroki:

1. Obsługa USB w trybie Host – wywołanie *MX_USB_HOST_Process()* obsługuje zdarzenia stosu USB Host, czyli enumerację, transfery i pracę klas.
2. Diagnostyka stanu USB – stan *Appli_state* (ustawiany w *usb_host.c*) jest porównywany z poprzednim stanem, aby wypisywać komunikaty debugujące tylko przy zmianie stanu.
3. Odbiór i filtracja MIDI – aplikacja pobiera zdarzenia 4-bajtowe przez funkcję *USBH_MIDI_GetEvent()*, filtruje komunikaty tylko do Note On z niezerową

wartością velocity (reszta jest odrzucana) i jeśli użytkownik jest w trybie nauki to przekazuje numer nuty do silnika lekcji funkcją *Lesson_HandleInput().*

4. Obsługa menu interfejsu użytkownika – na samym końcu następuje aktualizacja modułu przycisków *Button_Update()* oraz maszyny stanów UI (interfejsu użytkownika) funkcją *App_Update()*.

```c
  while (1)
  {
    /* Maintain USB Host stack (enumeration, transfers, class handling). */
    MX_USB_HOST_Process();

    /* Print USB app state transitions only once (no spam every loop). */
    if (Appli_state != prevState)
    {
      switch (Appli_state)
      {
        case APPLICATION_START:
          printf("State: APPLICATION_START (device connected)\r\n");
          break;
        case APPLICATION_READY:
          printf("State: APPLICATION_READY (MIDI class active)\r\n");
          break;
        case APPLICATION_DISCONNECT:
          printf("State: APPLICATION_DISCONNECT (device disconnected)\r\n");
          break;
        default:
          printf("State: %d\r\n", Appli_state);
          break;
      }
      prevState = Appli_state;
    }

    /* Read MIDI events and forward NOTE ON only when the lesson is active. */
    if (Appli_state == APPLICATION_READY || Appli_state == APPLICATION_START)
    {
      uint8_t midi_event[4];
      if (USBH_MIDI_GetEvent(&hUsbHostFS, midi_event) == USBH_OK)
      {
        uint8_t status  = midi_event[1] & 0xF0;
        uint8_t note    = midi_event[2];
        uint8_t vel     = midi_event[3];

        if (status == 0x90 && vel != 0)  /* NOTE ON */
        {
          if (Lesson_IsActive())
          {
            /* Lesson_HandleInput treats 0..127 as MIDI notes. */
            Lesson_HandleInput(note);
          }
        }
        else
        {
          /* NOTE OFF / other messages are ignored in this file. */
        }
      }
    }

    /* Poll buttons (debounce/edge) and run UI/menu logic. */
    Button_Update();
    App_Update();

    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
  }
```

Rys. 18.: Pętla główna pliku *main.c*, *źródło: opracowanie własne*

Taki podział sprawia, że *main.c* nie implementuje szczegółowej logiki lekcji ani obsługi USB-MIDI, a jedynie koordynuje pracę modułów i przekazuje dane między

warstwą sterowników, a logiką aplikacyjną. Rozwiązanie to zwiększa przejrzystość działania programu, oraz ułatwia modyfikację oraz rozszerzanie jego struktury funkcjonalnej.

5.3.4 Warstwa USB Host i zarządzanie stanem połączenia *(usb_host.c/.h)*

a) Rola plików

Pliki *usb_host.c* oraz *usb_host.h* stanowią warstwę integracyjną wygenerowaną przez STM32CubeMX, której zadaniem jest uruchomienie mikrokontrolera jako urządzenie USB w trybie Host oraz powiązanie biblioteki STM32Cube USB Host z częścią aplikacyjną projektu. Moduł inicjalizuje Hosta USB, rejestruje obsługiwane klasy USB oraz zapewnia cykliczne przetwarzanie zdarzeń stosu, czyli enumerację urządzenia, obsługę transferów i wywoływanie funkcji klas [13].

Dodatkowo moduł udostępnia uproszczony mechanizm informowania aplikacji o stanie połączenia w postaci zmiennej globalnej *Appli_state* typu *ApplicationTypeDef*. Jest ona aktualizowana w funkcji zwrotnej (tzw. callback) *USBH_UserProcess()*, która jest wywoływana automatycznie przez stos Hosta w reakcji na zdarzenia takie jak podłączenie, odłączenie oraz aktywacja klasy urządzenia [13].

W ramach projektu modyfikacje pliku *usb_host.c* ograniczono do rejestracji własnej klasy USB-MIDI oraz utrzymania prostego mechanizmu stanów połączenia *Appli_state*. Pozostała część modułu wynika bezpośrednio z kodu generowanego przez STM32CubeMX i udostępnianej przez STMicroelectronics biblioteki USB Host.

b) Inicjalizacja stosu USB Host

Funkcja *MX_USB_HOST_Init()* realizuje standardową sekwencję uruchomienia hosta USB i składa się z następujących kroków:

1. Inicjalizacja biblioteki Hosta *USBH_Init()*, w którym przekazywany jest uchwyt (czyli referencję do jego struktury) oraz funkcja callback *USBH_UserProcess()*, która służy do informowania aplikacji o zdarzeniach połączenia, takich jak podłączenie, odłączenie oraz aktywacja klasy urządzenia [13].

2. Rejestracja sterownika klasy USB poprzez funkcję *USBH_RegisterClass()*. W standardowej konfiguracji użytej w projekcie STM32Cube USB Host nie udostępnia gotowej klasy USB-MIDI, dlatego zaimplementowano własny sterownik klasy MIDI po stronie Hosta (opisany w podnagłówku „5.3.5 Własna implementacja obsługi klasy USB-MIDI (*usbh_midi.c/.h*)") na podstawie specyfikacji USB-MIDI [14].
3. Uruchamiana jest funkcja *USBH_Start()*, która aktywuje stos Hosta, tym samym aktywując obsługę urządzeń USB typu Device podłączonych do mikrokontrolera.

Opisany powyżej proces oraz funkcję są typowym interfejsem biblioteki STM32Cube USB Host [13]. Funkcje biblioteki zwracają kod statusu (np. *USBH_OK*), w przypadku błędu wykonywana jest funkcja *Error_Handler()*, która wyłącza przerwania i zatrzymuje program w pętli nieskończonej.

```
69  void MX_USB_HOST_Init(void)
70  {
71    /* USER CODE BEGIN USB_HOST_Init_PreTreatment */
72
73  /** *** VERY IMPORTANT ***
74    * remember to change: if (USBH_RegisterClass(&hUsbHostFS, USBH_HID_CLASS) != USBH_OK)
75    * to
76    * (USBH_RegisterClass(&hUsbHostFS, USBH_MIDI_CLASS) != USBH_OK)
77    * it change every time during generating code via STM32CubeMX
78    */
79    /* USER CODE END USB_HOST_Init_PreTreatment */
80
81    /* Init host Library, add supported class and start the library. */
82    if (USBH_Init(&hUsbHostFS, USBH_UserProcess, HOST_FS) != USBH_OK)
83    {
84      Error_Handler();
85    }
86    if (USBH_RegisterClass(&hUsbHostFS, USBH_MIDI_CLASS) != USBH_OK)
87    {
88      Error_Handler();
89    }
90    if (USBH_Start(&hUsbHostFS) != USBH_OK)
91    {
92      Error_Handler();
93    }
94    /* USER CODE BEGIN USB_HOST_Init_PostTreatment */
95    /* USER CODE END USB_HOST_Init_PostTreatment */
96  }
```

Rys. 19.: Inicjalizacja mikrokontrolera w pliku *usb_host.c* jako urządzenia USB w trybie Host, *źródło: opracowanie własne*

Kluczowa uwaga dla działania programu: rejestracja klasy odbywa się w miejscu, które STM32CubeMX modyfikuje do stanu domyślnego (rejestracji klasy HID) przy każdej ponownej generacji kodu – oznacza to, że przy modyfikacji ustawień mikrokontrolera i regeneracji kodu należy zmienić typ rejestrowanej klasy na

USBH_MIDI_CLASS, dzięki czemu program zarejestruje sterownik klasy obsługujący USB-MIDI.

c) Przetwarzanie zdarzeń i stany aplikacji (*Appli_state*)

Właściwe przetwarzanie zdarzeń Hosta, takich jak podłączenie czy odłączanie urządzeń, jest realizowane w funkcji *MX_USB_HOST_Process()*, która jest wywoływana cyklicznie z pętli głównej programu.

Funkcja ta przekazuje sterowanie do *USBH_Process()*, która utrzymuje pracę stosu Hosta USB, czyli obsługuje wykrywanie urządzenia, enumerację oraz wywołuje procedury zarejestrowanej klasy [13], w tym przypadku USB-MIDI.

Zdarzenia takie jak podłączenie urządzenia, aktywacja klasy lub rozłączenie sygnalizowane są aplikacji przez *USBH_UserProcess()*, w której aktualizowana jest zmienna *Appli_state*. Zmienna ta sygnalizuje warstwie aplikacyjnej stan połączenia:

- *APPLICATION_START* (urządzenie podłączone),
- *APPLICATION_READY* (klasa aktywna),
- *APPLICATION_DISCONNECT* (urządzenie odłączone)

```
 * user callback definition
 */
static void USBH_UserProcess  (USBH_HandleTypeDef *phost, uint8_t id)
{
  /* USER CODE BEGIN CALL_BACK_1 */
  switch(id)
  {
  case HOST_USER_SELECT_CONFIGURATION:
  break;

  case HOST_USER_DISCONNECTION:
  Appli_state = APPLICATION_DISCONNECT;
  break;

  case HOST_USER_CLASS_ACTIVE:
  Appli_state = APPLICATION_READY;
  break;

  case HOST_USER_CONNECTION:
  Appli_state = APPLICATION_START;
  break;

  default:
  break;
  }
  /* USER CODE END CALL_BACK_1 */
}
```

Rys. 20.: Obsługa zdarzeń Hosta oraz aktualizacja *Appli_state* w funkcji *USBH_UserProcess()*, *źródło: opracowanie własne*

Warto podkreślić, że opisany tutaj moduł nie otwiera pipe'ów dla endpointów interfejsu MIDI Streaming (Bulk IN). Otwarcie pipe'a Bulk IN oraz uruchomienie odbioru danych realizowane jest w sterowniku klasy USB-MIDI (*usbh_midi.c*) podczas inicjalizacji klasy.

5.3.5 Własna implementacja obsługi klasy USB-MIDI *(usbh_midi.c/.h)*

a) Rola plików

Pliki *usbh_midi.c* oraz *usbh_midi.h* implementują autorski sterownik klasy USB Host dla urządzeń USB-MIDI (interfejs MIDI Streaming). Moduł działa jako klasa rejestrowana w bibliotece STM32Cube USB Host i jest uruchamiany automatycznie po aktywacji klasy, czyli gdy w deskryptorach urządzenia zostanie znaleziony interfejs MIDI Streaming (Audio 0x01, MIDI Streaming 0x03) [14]. W odróżnieniu od pliku *usb_host.c*, który jedynie inicjalizuje stos Hosta i rejestruje klasę, właściwa obsługa interfejsu MIDI, taka jak identyfikacja interfejsu MIDI Streaming w deskryptorach

konfiguracji, wybór endpointów oraz otwarcie pipe Bulk IN, jest realizowana właśnie w tym module.

Sterownik został zaprojektowany jako minimalna implementacja trybu MIDI IN (odbiór danych z instrumentu). Odbierane dane są interpretowane jako sekwencja 4-bajtowych pakietów USB-MIDI Event Packet zgodnie ze specyfikacją USB-MIDI [14], a następnie buforowane w kolejce FIFO, aby tempo przetwarzania danych przez logikę aplikacyjną nie ograniczało obsługi transferów USB.

b) Struktury danych i interfejs modułu (*usbh_midi.h*)

Opis całej struktury sterownika klasy znajduje się w pliku nagłówkowym, który definiuje:

- stałe opisujące klasę i podklasę USB (Audio / MIDI Streaming),

- rozmiary buforów (*USBH_MIDI_MAX_PACKET_SIZE = 64* dla transmisji Full Speed oraz *USBH_MIDI_EVENT_FIFO_SIZE* jako rozmiar kolejki FIFO),

- typ stanu wewnętrznego w jakim znajduje się sterownik klasy *MIDI_StateTypeDef* (*MIDI_IDLE*, *MIDI_TRANSFER*, *MIDI_ERROR*),

- strukturę *MIDI_HandleTypeDef*, która przechowuje: identyfikatory pipe'ów, adresy endpointów, bufor odbioru jednego pakietu USB oraz kolejkę FIFO dla zdarzeń MIDI.

Warstwa aplikacyjna ma dostęp tylko do funkcji *USBH_MIDI_GetEvent()*, która pobiera pojedyncze zdarzenie (4 bajty) z kolejki FIFO.

c) Inicjalizacja klasy – wybór interfejsu i endpointów (*USBH_MIDI_Init*)

Funkcja *USBH_MIDI_Init()* jest wywoływana przez bibliotekę hosta w momencie aktywacji klasy. Do jej zadań należą:

1. Utworzenie uchwytu klasy (*MIDI_HandleTypeDef*) i podpięcie go pod *phost->pActiveClass->pData*, czyli pola przeznaczonego na dane prywatne aktualnie aktywnej klasy. Dzięki temu kolejne wywołania funkcji klasy, np. *USBH_MIDI_Process()*, mają dostęp do tego samego kontekstu: endpointów, pipe'ów, buforów oraz wskaźników FIFO.

2. Wybór interfejsu MIDI Streaming – sterownik przeszukuje deskryptory konfiguracji urządzenia i wybiera interfejs, który spełnia warunki zgodne ze standardem USB-MIDI: klasa Audio (0x01) oraz podklasa MIDI Streaming (0x03) [14].
3. Wybór endpointów oraz otwarcie pipe'ów – sterownik analizuje endpointy jakie posiada interfejs i wyszukuje endpoint Bulk IN (kierunek IN + typ Bulk). Dla znalezionego endpointu otwierany jest pipe funkcjami biblioteki STM32Cube USB Host: *USBH_AllocPipe()* i *USBH_OpenPipe()*. Endpoint Bulk OUT zostaje wykryty i otwarty pomocniczo, ale w tej pracy nie jest używany.
4. Na końcu inicjalizowana jest kolejka FIFO (wskaźniki *head*/*tail*) oraz stan klasy ustawiany jest na *MIDI_IDLE*.

```
  /* Allocate memory for MIDI class handle */
  MIDI_Handle = (MIDI_HandleTypeDef *)USBH_malloc(sizeof(MIDI_HandleTypeDef));
  if (MIDI_Handle == NULL)
  {
    printf("USBH_MIDI_Init: Failed to allocate MIDI class handle\r\n");
    return USBH_FAIL;
  }
  memset(MIDI_Handle, 0, sizeof(MIDI_HandleTypeDef));
  phost->pActiveClass->pData = (void *)MIDI_Handle;

  /* Find the MIDI Streaming interface (Audio class 0x01, subclass 0x03) */
  uint8_t interface = 0xFF;
  for (uint8_t idx = 0; idx < phost->device.CfgDesc.bNumInterfaces; idx++)
  {
    if ((phost->device.CfgDesc.Itf_Desc[idx].bInterfaceClass == USB_MIDI_CLASS_CODE) &&
        (phost->device.CfgDesc.Itf_Desc[idx].bInterfaceSubClass == USB_MIDI_SUBCLASS_STREAMING))
    {
      interface = idx;
      break;
    }
  }
  if (interface == 0xFF)
  {
    printf("USBH_MIDI_Init: No MIDI Streaming interface found\r\n");
    return USBH_FAIL;
  }
  printf("USBH_MIDI_Init: MIDI Streaming interface found at index %d\r\n", interface);

  /* Get the interface descriptor and parse its endpoints */
  USBH_InterfaceDescTypeDef *itf_desc = &phost->device.CfgDesc.Itf_Desc[interface];
  for (uint8_t ep_idx = 0; ep_idx < itf_desc->bNumEndpoints; ep_idx++)
  {
    USBH_EpDescTypeDef *ep_desc = &itf_desc->Ep_Desc[ep_idx];
    uint8_t ep_addr = ep_desc->bEndpointAddress;
    uint8_t ep_type = ep_desc->bmAttributes & 0x03U;  /* lower 2 bits indicate transfer type */

    /*
     * Bulk IN endpoint:
     * - direction bit set (0x80)
     * - transfer type bulk (0x02)
     */
    if ((ep_addr & 0x80U) && (ep_type == 0x02U))
    {
      /* MIDI IN endpoint (Bulk IN) */
      MIDI_Handle->InEp = ep_addr;
      MIDI_Handle->InEpSize = ep_desc->wMaxPacketSize;

      /*
       * IMPORTANT:
       * RxBuffer is sized to USBH_MIDI_MAX_PACKET_SIZE (64).
       * The code passes InEpSize to USBH_BulkReceiveData() later.
       * This assumes InEpSize <= USBH_MIDI_MAX_PACKET_SIZE (typical for FS bulk).
       */
      MIDI_Handle->InPipe = USBH_AllocPipe(phost, MIDI_Handle->InEp);
      USBH_OpenPipe(phost, MIDI_Handle->InPipe, MIDI_Handle->InEp,
                    phost->device.address, phost->device.speed,
                    USBH_EP_BULK, MIDI_Handle->InEpSize);
      USBH_LL_SetToggle(phost, MIDI_Handle->InPipe, 0);

      printf("USBH_MIDI_Init: Bulk IN endpoint 0x%02X (pipe %d) opened, max packet %d bytes\r\n",
             MIDI_Handle->InEp, MIDI_Handle->InPipe, MIDI_Handle->InEpSize);
    }
  /* Initialize FIFO and state */
  MIDI_Handle->EventFIFOHead = 0;
  MIDI_Handle->EventFIFOTail = 0;
  MIDI_Handle->state = MIDI_IDLE;
  printf("USBH_MIDI_Init: MIDI FIFO initialized (head=%u, tail=%u)\r\n",
         MIDI_Handle->EventFIFOHead, MIDI_Handle->EventFIFOTail);

  /* Indicate successful initialization */
  printf("USBH_MIDI_Init: MIDI class driver initialized successfully\r\n");
  status = USBH_OK;
  return status;
```

Rys. 21.: Najważniejsze fragmenty funkcji *USBH_MIDI_Init()* – inicjalizacja klasy, otwarcie pipe'a Bulk IN oraz inicjalizacja bufora FIFO, *źródło: opracowanie własne*

d) Odbiór danych – maszyna stanów oraz buforowanie FIFO (*USBH_MIDI_Process*)

Funkcja *USBH_MIDI_Process()* jest cyklicznie wywoływana przez stos USB Host, w ramach funkcji *USBH_Process()*, która jest wywoływana w głównej pętli programu z poziomu *MX_USB_HOST_Process()*.

Moduł realizuje prostą maszynę stanów:

- *MIDI_IDLE* – zgłaszany jest nowy transfer IN przez *USBH_BulkReceiveData()*, a stan przechodzi do *MIDI_TRANSFER*,

- *MIDI_TRANSFER* – sterownik sprawdza stan ostatniego transferu USB (URB), wykorzystując z biblioteki funkcję *USBH_LL_GetURBState()* i w zależności od odczytanego stanu wykonuje konkretne kroki:

  - *USBH_URB_DONE* – odczytywana jest liczba odebranych bajtów, a bufor jest dzielony na kolejne 4-bajtowe pakiety USB-MIDI Event Packet. Każde zdarzenie jest dopisywane do FIFO (krok indeksów wynosi 4 bajty). Następnie natychmiast zgłaszany jest kolejny transfer IN, aby utrzymać ciągłe odpytywanie endpointu,

  - *USBH_URB_STALL* – wykonywane jest skasowanie warunku STALL z wykorzystaniem funkcji *USBH_ClrFeature()* i ponowienie transferu,

  - *USBH_URB_ERROR* – sterownik przechodzi do stanu *MIDI_ERROR*.

```
231   switch (MIDI_Handle->state)
232   {
233     case MIDI_IDLE:
234       /* Start a new IN transfer */
235       printf("USBH_MIDI_Process: State=MIDI_IDLE, initiating IN transfer\r\n");
236       USBH_BulkReceiveData(phost, MIDI_Handle->RxBuffer,
237                            MIDI_Handle->InEpSize, MIDI_Handle->InPipe);
238       MIDI_Handle->state = MIDI_TRANSFER;
239       printf("USBH_MIDI_Process: State -> MIDI_TRANSFER (waiting for data)\r\n");
240       status = USBH_BUSY;
241       break;
242
243     case MIDI_TRANSFER:
244       /* Check the state of the last USB transfer */
245       urb_state = USBH_LL_GetURBState(phost, MIDI_Handle->InPipe);
246       if (urb_state == USBH_URB_DONE)
247       {
248         /* One USB packet received */
249         length = USBH_LL_GetLastXferSize(phost, MIDI_Handle->InPipe);
250         printf("USBH_MIDI_Process: URB done, received %lu bytes\r\n", (unsigned long)length);
251
252         if (length > 0 && length <= USBH_MIDI_MAX_PACKET_SIZE)
253         {
254           /* Split data into 4-byte USB-MIDI event packets and push to FIFO */
255           uint32_t i = 0;
256           while (i < length && (length - i) >= 4)
257           {
258             uint16_t nextHead = (MIDI_Handle->EventFIFOHead + 4) % USBH_MIDI_EVENT_FIFO_SIZE;
259             if (nextHead == MIDI_Handle->EventFIFOTail)
260             {
261               /* FIFO full: drop remaining events in this packet */
262               break;
263             }
264
265             memcpy(&MIDI_Handle->EventFIFO[MIDI_Handle->EventFIFOHead],
266                    &MIDI_Handle->RxBuffer[i], 4);
267             MIDI_Handle->EventFIFOHead = nextHead;
268             i += 4;
269           }
270
271           uint32_t eventsAdded = i / 4;
272           if (eventsAdded > 0)
273           {
274             printf("USBH_MIDI_Process: Added %lu MIDI events to FIFO\r\n", (unsigned long)eventsAdded);
275           }
276           if (i < length)
277           {
278             printf("USBH_MIDI_Process: MIDI FIFO overflow, dropped %lu bytes\r\n",
279                    (unsigned long)(length - i));
280           }
281         }
282         else if (length > USBH_MIDI_MAX_PACKET_SIZE)
283         {
284           /* Safety log: should not happen for FS bulk with 64-byte packets */
285           printf("USBH_MIDI_Process: Packet size %lu exceeds max %d, ignoring\r\n",
286                  (unsigned long)length, USBH_MIDI_MAX_PACKET_SIZE);
287         }
288
289         /* Immediately submit the next read transfer to keep continuous polling */
290         USBH_BulkReceiveData(phost, MIDI_Handle->RxBuffer,
291                              MIDI_Handle->InEpSize, MIDI_Handle->InPipe);
292
293         /* Stay in MIDI_TRANSFER */
294         status = USBH_OK;
295       }
```

Rys. 22.: Fragment funkcji *USBH_MIDI_Process()* – przejście stanów oraz dzielenie bufora na 4-bajtowe zdarzenia, *źródło: opracowanie własne*

Zastosowanie bufora FIFO powoduje, że logika aplikacyjna nie musi przetwarzać danych natychmiastowo po otrzymaniu pakietu USB, tym samym nie blokując przesyłania danych po magistrali USB.

e) Pobieranie zdarzeń przez logikę aplikacyjną (*USBH_MIDI_GetEvent*)

Funkcja *USBH_MIDI_GetEvent()* udostępnia aplikacji kolejne zdarzenia MIDI w postaci 4-bajtowych pakietów. Jeśli FIFO jest puste, funkcja natychmiast zwraca kod *USBH_FAIL*, co oznacza brak nowych zdarzeń do odczytu. Dzięki temu odczyt jest nieblokujący – pętla główna nie oczekuje na dane USB, ponieważ ich odbiór realizowany jest niezależnie w *USBH_MIDI_Process()*, która dopisuje odebrane pakiety do FIFO. Po pobraniu zdarzenia wskaźnik ogona bufora FIFO (*tail*) jest przesuwany o 4 bajty.

```
USBH_StatusTypeDef USBH_MIDI_GetEvent(USBH_HandleTypeDef *phost, uint8_t *event_buf)
{
  MIDI_HandleTypeDef *MIDI_Handle = (MIDI_HandleTypeDef *)phost->pActiveClass->pData;
  if (MIDI_Handle == NULL)
  {
    return USBH_FAIL;  /* Class not initialized / device not ready */
  }

  /* FIFO empty */
  if (MIDI_Handle->EventFIFOHead == MIDI_Handle->EventFIFOTail)
  {
    return USBH_FAIL;
  }

  /* Copy one 4-byte event from FIFO to user buffer */
  for (uint8_t j = 0; j < 4; j++)
  {
    event_buf[j] = MIDI_Handle->EventFIFO[MIDI_Handle->EventFIFOTail + j];
  }

  /* Advance tail by 4 (one event) */
  MIDI_Handle->EventFIFOTail = (MIDI_Handle->EventFIFOTail + 4) % USBH_MIDI_EVENT_FIFO_SIZE;
  return USBH_OK;
}
```

Rys. 23.: Funkcja *USBH_MIDI_GetEvent()*, odczyt pojedynczego zdarzenia z bufora FIFO, *źródło: opracowanie własne*

5.3.6 Silnik lekcji: weryfikacja dźwięków i generowanie informacji zwrotnej *(lesson.c/.h)*

a) Rola plików

Pliki *lesson.c* oraz *lesson.h* implementują silnik lekcji, czyli główną logikę trybu nauki w projekcie. Moduł ten odpowiada za:

- obsługę dwóch trybów nauki: SONG (kroki z konkretnymi nutami MIDI do zagrania) oraz CHORDS (kroki jako akordy dopasowywane po klasie wysokości, czyli można zagrać dźwięk z dowolnej oktawy),

- weryfikację poprawności zagranych dźwięków,

- prowadzenie statystyk (trafienia + błędy / liczba prób),

- generowanie informacji zwrotnej, po naciśnięciu klawisza przez użytkownika za pomocą diod LED,

- prezentowanie bieżącego kroku lekcji oraz ekranu podsumowania, po zakończeniu wszystkich kroków, na wyświetlaczu LCD.

Co ważne moduł nie komunikuje się bezpośrednio z USB ani MIDI. Otrzymuje wyłącznie zdarzenia wejściowe przekazywane z warstwy aplikacyjnej przez funkcję *Lesson_HandleInput()* (jako nuty MIDI oraz przyciski), dzięki czemu pozostaje niezależny od sposobu dostarczenia danych.

b) Interfejs modułu i stan wewnętrzny (*lesson.h* oraz *lesson.c*)

W pliku nagłówkowym *lesson.*h znajdują się następujące funkcje, które są udostępniane jako API silnika:

- *Lesson_StartSong(Song *song)* – rozpoczęcie lekcji w trybie utworu,

- *Lesson_StartChordExercise(ChordPack *pack)* – rozpoczęcie ćwiczeń akordów,

- *Lesson_HandleInput(uint8_t input)* – obsługa pojedynczego zdarzenia wejściowego,

- *Lesson_IsActive()* – informacja, czy lekcja jest aktywna,

- *Lesson_Update()* – funkcja okresowa do realizacji zadań czasowych (m.in. LED).

Dodatkowo zdefiniowano również specjalne kody wejściowe dla przycisków, nazwane *LESSON_INPUT_BTN_OK*, *LESSON_INPUT_BTN_NEXT* oraz *LESSON_INPUT_BTN_RESET* jako wartości 0xF1–0xF3, czyli większe od 0x7F, co pozwala rozróżnić je od nut MIDI (które zajmują wartości od 0x0 do 0x7F) w jednym wspólnym interfejsie *Lesson_HandleInput()*.

W *lesson.c* stan lekcji jest przechowywany w zmiennych statycznych:

- wskaźnik wybranych do trybu nauki danych *currentSong* lub *currentChordPack*,

- flaga aktywności trybu nauki *lessonActive,*

- indeks kroku lekcji *currentStepIndex* oraz liczba kroków *totalSteps*,

- *stepHit[3]* – tablica oznaczająca, które elementy kroku (maks. 3 nuty) zostały już poprawnie trafione,

- *correctPlayed*, *wrongPlayed*, *totalPlayed* – liczniki naciśnięć klawiszy wykorzystywane do statystyk

- *lessonState* – prosty automat, mający dwa stany: RUNNING oraz SUMMARY,

- znaczniki do nieblokującego sterowania LED (flagi + HAL_GetTick()).

c) Uruchamianie lekcji i inicjalizacja kroku

Uruchamianie trybu nauki wywołujemy za pomocą funkcji *Lesson_StartSong()* lub *Lesson_StartChordExercise()*, które realizują analogiczny schemat startu:

1. Ustawienie źródła danych (song/pack), po których silnik weryfikuje poprawność granych nut oraz ustala parametry sesji *currentStepIndex = 0*, *totalSteps* oraz *lessonActive*,
2. Wyzerowanie statystyk liczników naciśnięć *correctPlayed*, *wrongPlayed* oraz *totalPlayed*,
3. Reset tablicy *stepHit[]* oraz ustawienie stanu na *LESSON_STATE_RUNNING*,
4. Wyłączenie diod LED i wyzerowanie ich znaczników czasowych,
5. Wyświetlenie pierwszego kroku lekcji na LCD poprzez funkcję *DisplaySongStep()* lub *DisplayChordStep()*.

Dzięki temu rozpoczęcie lekcji jest deterministyczne (zawsze przebiega według tej samej sekwencji), a dalsza praca działa już wyłącznie zdarzeniowo – przetwarzając informacje wejściowe podawane z pętli głównej *main.c* – lub okresowo poprzez cykliczne wywoływanie *Lesson_Update()*.

d) Mechanizm weryfikacji nut i sterowanie przebiegiem lekcji (*Lesson_HandleInput*)

Centralnym punktem logiki jest funkcja *Lesson_HandleInput()*, która obsługuje trzy główne przypadki:

1. Ekran podsumowania (*LESSON_STATE_SUMMARY*)

- odebrane nuty MIDI (wejście z instrumentu) są ignorowane,

- dowolny przycisk kończy lekcję (*lessonActive* = false) i zwraca sterowanie do interfejsu użytkownika (*app.c/h*)

2. Nuta MIDI (wartości od 0 do 127)

Po odebraniu nuty zwiększany jest licznik prób *totalPlayed*++, a następnie wykonywana jest weryfikacja zależna od trybu nauki:

Tryb SONG: dopasowanie po dokładnym numerze MIDI (czyli z uwzględnieniem oktawy).

Silnik przeszukuje wymagane nuty bieżącego kroku lekcji i jeśli znajdzie niezaliczoną pozycję równą odebranej z pętli głównej nucie, to oznacza ją jako trafioną *stepHit[i] = true*. Mechanizm ten powoduje także, że ponowne zagranie tej samej nuty w ramach już zaliczonego slotu nie jest ponownie liczone jako trafienie i zostanie zaklasyfikowane jako błąd dla danego kroku.

W przypadku trafienia zwiększany jest licznik *correctPlayed*, uruchamiana jest zielona dioda i silnik automatycznie przechodzi do kolejnego kroku lekcji lub podsumowania (jeśli był to ostatni krok), jeśli wszystkie wymagane nuty zostały zagrane.

W przypadku błędu zwiększany jest licznik *wrongPlayed* i uruchamiana dioda czerwona.

Tryb CHORDS: dopasowanie po klasie wysokości *note % 12*, co pozwala zagrać składniki akordu w dowolnej oktawie.

Odebrana nuta jest redukowana do pitch class (zmienna *playedPC*), a następnie porównywana z oczekiwanymi klasami wysokości akordu, wyznaczanymi funkcją *NoteToPitchClass()* na podstawie litery nuty oraz znaku chromatycznego.

Dalej mechanizm działania jest analogiczny jak w trybie SONG: trafienie zapala zieloną diodę i powoduje przejście dalej, jeśli wszystkie nuty dla tego kroku lekcji zostały poprawnie zagrane, a błąd zapala diodę czerwoną.

```
    /* --- MIDI note (0..127) --- */
    if (input <= 0x7F)
    {
        totalPlayed++;

        if (currentSong != NULL)
        {
            SongStep *step = &currentSong->steps[currentStepIndex];
            uint8_t slots = step->noteCount;
            if (slots > 3) slots = 3;

            bool matched = false;
            for (uint8_t i = 0; i < slots; i++)
            {
                if (!stepHit[i] && (uint8_t)step->notes[i].midiNote == input)
                {
                    stepHit[i] = true;
                    matched = true;
                    break;
                }
            }

            if (matched) {
                correctPlayed++;
                LedBlinkGreen();

                /* Auto-advance when all required notes are hit */
                if (IsStepComplete()) {
                    AdvanceOrSummary();
                }
            } else {
                wrongPlayed++;
                LedBlinkRed();
            }

            return;
        }

        if (currentChordPack != NULL)
        {
            Chord *chord = &currentChordPack->chords[currentStepIndex];
            uint8_t slots = chord->noteCount;
            if (slots > 3) slots = 3;

            uint8_t playedPC = (uint8_t)(input % 12U);
            bool matched = false;

            for (uint8_t i = 0; i < slots; i++)
            {
                if (stepHit[i]) continue;
                int8_t expectedPC = NoteToPitchClass(chord->notes[i].letter, chord->notes[i].accidental);
                if (expectedPC >= 0 && (uint8_t)expectedPC == playedPC)
                {
                    stepHit[i] = true;
                    matched = true;
                    break;
                }
            }

            if (matched) {
                correctPlayed++;
                LedBlinkGreen();

                /* Auto-advance when chord tones are all hit */
                if (IsStepComplete()) {
                    AdvanceOrSummary();
                }
            } else {
                wrongPlayed++;
                LedBlinkRed();
            }

            return;
        }

        return;
    }
```

Rys. 24.: Fragment funkcji *Lesson_HandleInput()* – logika trybu nauki SONG oraz CHORDS, *źródło: opracowanie własne*

3. Przyciski *OK*, *NEXT* oraz *RESET*

- Przycisk *OK* powoduje pominięcie bieżącego kroku lekcji – brakujące do zaliczenia kroku nuty są doliczane jako poprawne, a następnie następuje przejście do kolejnego kroku lekcji.

- Przycisk *NEXT* powoduje cofnięcie do poprzedniego kroku lekcji (nazwa wynika z ogólnej konwencji przycisku w UI), a jeśli użytkownik jest już na kroku 0 (startowym), to przycisk kończy lekcję.

- Przycisk *RESET* powoduje powrót do kroku 0, a jeśli użytkownik jest już na kroku 0, to przycisk kończy lekcję.

```
446     /* --- Buttons --- */
447     if (input == LESSON_INPUT_BTN_OK)
448     {
449         /* Skip current step: count remaining slots as correct and move forward */
450         AddMissingSlotsAsCorrect();
451         AdvanceOrSummary();
452     }
453     else if (input == LESSON_INPUT_BTN_NEXT)
454     {
455         /* Go to previous step; if already at step 0 -> exit lesson */
456         if (currentStepIndex > 0)
457         {
458             currentStepIndex--;
459             ResetStepHit();
460             if (currentSong) DisplaySongStep(&currentSong->steps[currentStepIndex]);
461             else DisplayChordStep(&currentChordPack->chords[currentStepIndex]);
462         }
463         else
464         {
465             lessonActive = false;
466             ResetStepHit();
467             HAL_GPIO_WritePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin, GPIO_PIN_RESET);
468             HAL_GPIO_WritePin(RED_LED_GPIO_Port, RED_LED_Pin, GPIO_PIN_RESET);
469             greenLedOn = false;
470             redLedOn = false;
471         }
472     }
473     else if (input == LESSON_INPUT_BTN_RESET)
474     {
475         /* Reset to step 0; if already at step 0 -> exit lesson */
476         if (currentStepIndex != 0)
477         {
478             currentStepIndex = 0;
479             ResetStepHit();
480             if (currentSong) DisplaySongStep(&currentSong->steps[0]);
481             else DisplayChordStep(&currentChordPack->chords[0]);
482         }
483         else
484         {
485             lessonActive = false;
486             ResetStepHit();
487             HAL_GPIO_WritePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin, GPIO_PIN_RESET);
488             HAL_GPIO_WritePin(RED_LED_GPIO_Port, RED_LED_Pin, GPIO_PIN_RESET);
489             greenLedOn = false;
490             redLedOn = false;
491         }
492     }
493 }
```

Rys. 25.: Fragment funkcji *Lesson_HandleInput()* – logika obsługi przycisków, *źródło: opracowanie własne*

Dzięki takiemu rozwiązaniu użytkownik jest w stanie poruszać się po danej lekcji – może priorytetyzować konkretne fragmenty utworu lub konkretne akordy, na których nauce chce się skupić.

e) Informacja zwrotna i praca nieblokująca (diody LED oraz *Lesson_Update*)

Przy każdym naciśnięciu klawisza generowana jest informacja zwrotna LED, która została zrealizowana w sposób nieblokujący:

- funkcje *LedBlinkGreen()* oraz *LedBlinkRed()* włączają diodę i zapisują czas startu z wykorzystaniem licznika systemowego *HAL_GetTick()*,

- funkcja *Lesson_Update()* jest wywoływana cyklicznie w pętli głównej programu i wyłącza diody po upływie czasu zdefiniowanego w stałej *LED_BLINK_MS*, domyślnie ustawionego jako 120 ms.

Takie rozwiązanie powoduje, że logika lekcji nie zatrzymuje programu opóźnieniami *HAL_Delay()*, a pętla główna może nadal obsługiwać USB, UI oraz LCD.

5.3.7 Interfejs użytkownika: menu i nawigacja po trybach nauki *(app.c/.h)*

a) Rola plików

Pliki *app.c* oraz *app.h* implementują logikę interfejsu użytkownika w postaci prostej maszyny stanów. Moduł odpowiada za nawigację po ekranach LCD, wybór trybu nauki oraz uruchamianie lekcji na podstawie danych aplikacyjnych, czyli utworów i paczek akordów. Interfejs użytkownika jest sterowany trzema przyciskami: OK, NEXT oraz RESET.

W przeciwieństwie do modułu *lesson.c*, który zajmuje się weryfikacją poprawności gry, moduł *app.c* pełni funkcję warstwy UI wyświetlając odpowiednie ekrany, interpretując naciśnięcia przycisków i w odpowiednim momencie przekazując je do silnika lekcji.

b) Model stanu i dane pomocnicze (*app.h* / *app.c*)

W pliku nagłówkowym *app.h* zdefiniowano typ wyliczeniowy *AppState*, opisujący stany w jakich może znajdować się interfejs użytkownika:

- ekran powitalny *APP_STATE_WELCOME*,

- menu główne *APP_STATE_MENU_MAIN*,

- lista utworów *APP_STATE_MENU_SONGS*,

- lista paczek akordów *APP_STATE_MENU_CHORDPACKS*,

- ekran legendy symboli *APP_STATE_VIEW_LEGEND*,

- uruchomiona lekcja utworu *APP_STATE_LESSON_SONG* lub akordów *APP_STATE_LESSON_CHORD*.

W *app.c* stan aplikacji jest przechowywany w zmiennej statycznej *appState*, a nawigacja po listach w danych stanach realizowana jest przez indeksy: *mainMenuIndex*, *songListIndex* oraz *chordPackIndex*. Moduł korzysta ze wspólnej instancji LCD *extern GroveLCD_t lcd*, inicjalizowanej wcześniej w *main.c*.

c) Ekrany LCD i sposób prezentacji informacji

Plik źródłowy *app.c* zawiera zestaw funkcji renderujących poszczególne ekrany: *DisplayWelcomeScreen()*, *DisplayMainMenu()*, *DisplaySongsList()*, *DisplayChordPacksList()* oraz *DisplayNotesLegend()*. Każda z nich czyści wyświetlacz i zapisuje nową zawartość dostosowaną do ograniczeń LCD 16×2.

Menu główne posiada trzy opcje do wyboru przez użytkownika: legendę znaków, listę utworów, listę akordów.

W dolnym wierszu LCD wyświetlana jest krótka podpowiedź sterowania (np. „NEXT=Down OK=Sel”). Dla ekranów list zastosowano prostą konwencję wizualną („> ” przed nazwą), aby zaznaczyć aktualnie wybraną pozycję.

Dodatkowo ekran legendy prezentuje symbole nut i znaki chromatyczne, wykorzystując znaki własne w pamięci CGRAM (załadowane wcześniej w sekwencji startowej programu).

d) Obsługa przycisków i przejścia pomiędzy stanami (*App_Update*)

Centralną funkcją modułu jest funkcja *App_Update()*, wywoływana cyklicznie w pętli głównej programu.

Funkcja sprawdza, czy w bieżącej iteracji wykryto zdarzenie naciśnięcia przycisku *Button_WasPressed()*, a następnie – zależnie od stanu *AppState* – wykonuje odpowiednie przejścia w maszynie stanów UI:

- Przycisk *OK* zatwierdza wybór, używany do przejścia z ekranu powitalnego do menu, wejścia do podmenu czy uruchomienia lekcji dla wybranej pozycji.

- Przycisk *NEXT* przewija pozycje w menu lub po elementach list, indeks jest zawijany modulo liczby pozycji, czyli po ostatnim elemencie wracamy na samą górę listy.

- Przycisk *RESET* pełni rolę powrotu, np. z list/legendy do menu głównego.

W stanie lekcji *APP_STATE_LESSON_SONG* lub *APP_STATE_LESSON_CHORD* menu przestaje reagować na przyciski jako nawigację, a zdarzenia są mapowane na kody *LESSON_INPUT_BTN_** i przekazywane do silnika lekcji z wykorzystaniem *Lesson_HandleInput()*.

Powrót do listy następuje po zakończeniu lekcji, co jest wykrywane przez sprawdzenie *Lesson_IsActive()*, np. po wciśnięciu dowolnego przycisku na ekranie podsumowania.

```
184     if (Button_WasPressed(BUTTON_RESET))
185     {
186         switch (appState)
187         {
188             case APP_STATE_MENU_MAIN:
189                 /* RESET is ignored in the main menu */
190                 break;
191 
192             case APP_STATE_MENU_SONGS:
193             case APP_STATE_MENU_CHORDPACKS:
194             case APP_STATE_VIEW_LEGEND:
195                 /* Back to main menu */
196                 appState = APP_STATE_MENU_MAIN;
197                 DisplayMainMenu();
198                 break;
199 
200             case APP_STATE_LESSON_SONG:
201             case APP_STATE_LESSON_CHORD:
202                 /* Forward reset/cancel to the lesson engine */
203                 Lesson_HandleInput(LESSON_INPUT_BTN_RESET);
204 
205                 /* If the lesson ended, return to the list */
206                 if (!Lesson_IsActive())
207                 {
208                     if (appState == APP_STATE_LESSON_SONG) {
209                         appState = APP_STATE_MENU_SONGS;
210                         DisplaySongsList();
211                     } else {
212                         appState = APP_STATE_MENU_CHORDPACKS;
213                         DisplayChordPacksList();
214                     }
215                 }
216                 break;
217 
218             default:
219                 break;
220         }
221     }
```

Rys. 26.: Fragment *App_Update()* – obsługa przycisku RESET, *źródło: opracowanie własne*

e) Integracja z silnikiem lekcji

Uruchomienie trybu nauki następuje w momencie naciśnięcia przycisku *OK* będąc w liście wyboru piosenek lub paczek akordów, wywoływana jest odpowiednia funkcja:

- *Lesson_StartSong(&songs[songListIndex])* dla utworów,

- *Lesson_StartChordExercise(&chordPacks[chordPackIndex])* dla akordów.

Od tego momentu UI działa w stanie lekcji i przekazuje przyciski do silnika, a sam silnik odpowiada za weryfikację nut oraz sterowanie prezentacją przebiegu lekcji. Na końcu *App_Update()* wykonywane jest także cykliczne wywołanie *Lesson_Update()*, które pozwala utrzymać okresowe funkcje silnika lekcji (np. zadania czasowe) bez blokowania pętli głównej.

5.3.8 Obsługa wejść użytkownika *(button.c/.h)*

a) Rola plików

Pliki *button.c* oraz *button.h* implementują obsługę trzech przycisków sterujących *RESET*, *NEXT*, *OK* w sposób odporny na drgania styków (tzw. debouncing) oraz z detekcją zbocza stanu stabilnego. Dzięki takiemu rozwiązaniu uzyskano zdarzenie typu „press”, czyli jednorazowy impuls logiczny, który jest generowany jedynie w chwili naciśnięcia – niezależnie od tego jak długo trzymany jest przycisk.

Moduł ten stanowi więc warstwę pośrednią pomiędzy surowym odczytem GPIO a logiką aplikacyjną (*app.c* oraz *lesson.c*). W związku z tym moduły wyższego poziomu nie muszą samodzielnie wykrywać przejścia wciśnięty/puszczony ani pilnować, aby jedno przytrzymanie nie było liczone wielokrotnie.

W projekcie założono, że przyciski są podłączone jako aktywne stanem niskim (w spoczynku wejście ma stan wysoki dzięki wewnętrznemu rezystorowi podciągającemu (internal pull-up), a naciśnięcie ściąga sygnał do masy. W kodzie oznacza to: 1 = puszczony, 0 = wciśnięty.

b) Interfejs modułu (*button.h)*

Plik nagłówkowy *button.h* udostępnia trzy funkcje:

- *Button_Init()* – inicjalizacja modułu oraz opcjonalna konfiguracja GPIO jako tryb input + pull-up,

- *Button_Update()* – funkcja cykliczna realizująca filtrację drgań i wykrywanie zbocza,

- *Button_WasPressed(ButtonType button)* – zwraca *true* tylko raz na fizyczne naciśnięcie (tzw. one-shot), po czym automatycznie kasuje zdarzenie.

Identyfikatory przycisków są wspólne dla całej aplikacji dzięki typowi enumerycznemu *ButtonType*. Definiujemy *BUTTON_RESET*, *BUTTON_NEXT*, *BUTTON_OK* oraz *BUTTON_COUNT*. Ten ostatni nie jest fizycznym przyciskiem, ale jest wykorzystywany do definiowania rozmiarów tablicy i umożliwia wygodne przetwarzanie wszystkich przycisków w pętli.

c) Debouncing i wykrywanie zdarzenia „press” (*button.c*)

W *button.c* zastosowano algorytm czasowy z progiem *DEBOUNCE_MS = 30 ms*. Dla każdego przycisku przechowywany jest zestaw następujących zmiennych:

- *last_raw_level* – ostatnia pobrana próbka surowego poziomu GPIO,

- *last_change_ms* – czas ostatniej zmiany próbki, mierzony z wykorzystaniem *HAL_GetTick()*,

- *stable_level* – zaakceptowany, czyli ustabilizowany, poziom wejścia,

- *pressed_event* – zatrzask zdarzenia naciśnięcia (jednorazowy).

Cały algorytm działa w następujący sposób:

1. Funkcja *Button_Update()* iteruje po wszystkich przyciskach i dla każdego wywołuje *read_raw()*, który odczytuje GPIO i sprawdza stan logiczny na wejściu.
2. Każda zmiana stanu surowego zapisuje znacznik czasu ostatniej zmiany *last_change_ms = now.*

3. Jeżeli od ostatniej zmiany minęło co najmniej *DEBOUNCE_MS* i surowy poziom różni się od aktualnego *stable_level*, to zostaje zaakceptowany jako nowy stan stabilny.

4. Zdarzenie naciśnięcia powstaje tylko przy stabilnym przejściu ze stanu puszczonego na wciśnięty oraz jest zapisywane w zmiennej *pressed_event*.

```
99  void Button_Update(void)
100 {
101     uint32_t now = HAL_GetTick();
102 
103     for (uint8_t i = 0; i < BUTTON_COUNT; i++)
104     {
105         ButtonType b = (ButtonType)i;
106         uint8_t raw = read_raw(b);
107 
108         /* Track raw-level changes and reset debounce timer on any transition. */
109         if (raw != g_btn[i].last_raw_level) {
110             g_btn[i].last_raw_level = raw;
111             g_btn[i].last_change_ms = now;
112         }
113 
114         /* If the raw level has been stable long enough, accept it as stable. */
115         if ((now - g_btn[i].last_change_ms) >= DEBOUNCE_MS)
116         {
117             if (raw != g_btn[i].stable_level)
118             {
119                 /* Stable edge detected */
120                 uint8_t prev = g_btn[i].stable_level;
121                 g_btn[i].stable_level = raw;
122 
123                 /* Press event = released->pressed transition (1->0) */
124                 if (prev == 1U && raw == 0U) {
125                     g_btn[i].pressed_event = 1U;
126                 }
127             }
128         }
129     }
130 }
```

Rys. 27.: Funkcja *Button_Update()* – implementacja generacji zdarzenia typu press, *źródło: opracowanie własne*

Dzięki takiemu rozwiązaniu logika wyższego poziomu nie musi czekać bezczynnie na stabilizację i nie reaguje wielokrotnie na jedno fizyczne kliknięcie.

d) Odczyt zdarzeń naciśnięcia przez aplikację (*Button_WasPressed*)

Button_WasPressed() realizuje model zdarzeniowy, zatem jeśli *pressed_event* jest ustawione, to funkcja zwraca *true* i natychmiast zeruje flagę. W praktyce oznacza to, że w *main.c* konieczne jest wywoływanie funkcji *Button_Update()* cyklicznie, natomiast

app.c oraz *lesson.c* mogą sprawdzać zdarzenia bez obawy o wielokrotne zliczenie pojedynczego przytrzymania przycisku.

5.3.9 Sterownik wyświetlacza LCD *(grove_lcd16x2_i2c.c/.h)*

a) Rola plików

Pliki *grove_lcd16x2_i2c.c* oraz *grove_lcd16x2_i2c.h* implementują sterownik dla wyświetlacza Grove LCD 16×2 komunikującego się z mikrokontrolerem po magistrali I2C. Moduł udostępnia proste API do inicjalizacji, czyszczenia ekranu, ustawiania kursora oraz wypisywania tekstu.

b) Interfejs modułu (*grove_lcd16x2_i2c.h*)

Najważniejsze funkcje udostępniane przez interfejs to:

- *GroveLCD_Init()* – inicjalizacja LCD,

- *GroveLCD_Clear()*, *GroveLCD_SetCursor()* – podstawowa obsługa ekranu (czyszczenie i ustawianie kursora),

- *GroveLCD_WriteChar()*, *GroveLCD_Print()* – wypisywanie znaków i tekstu,

- *GroveLCD_CreateChar()* – definicja znaków własnych w CGRAM, wyświetlacz udostępnia osiem slotów.

c) Transmisja I2C – komendy i dane (*grove_lcd16x2_i2c.c*)

W implementacji rozdzielono zapis komend i danych poprzez użycie prefiksu sterującego, co powoduje że komendy są wysyłane z prefiksem 0x80, a dane z prefiksem 0x40 [18]. Komunikacja jest realizowana z wykorzystaniem funkcji *HAL_I2C_Mem_Write()* z biblioteki STM32 HAL.

```
54  * @brief Low-level helper: write one command byte to the LCD command register.
55  */
56 static HAL_StatusTypeDef lcd_write_cmd(GroveLCD_t *lcd, uint8_t cmd)
57 {
58     return HAL_I2C_Mem_Write(
59         lcd->hi2c,
60         lcd_hal_addr(lcd),
61         GROVE_LCD_REG_CMD,
62         I2C_MEMADD_SIZE_8BIT,
63         &cmd,
64         1,
65         lcd->timeout_ms
66     );
67 }
68
69 /**
70  * @brief Low-level helper: write one data byte to the LCD data register.
71  */
72 static HAL_StatusTypeDef lcd_write_data(GroveLCD_t *lcd, uint8_t data)
73 {
74     return HAL_I2C_Mem_Write(
75         lcd->hi2c,
76         lcd_hal_addr(lcd),
77         GROVE_LCD_REG_DATA,
78         I2C_MEMADD_SIZE_8BIT,
79         &data,
80         1,
81         lcd->timeout_ms
82     );
83 }
```

Rys. 28.: Fragment pliku *grove_lcd16x2_i2c.c*, funkcje wysyłające komendy oraz dane, *źródło: opracowanie własne*

d) Procedura inicjalizacji wyświetlacza

Funkcja *GroveLCD_Init()* realizuje inicjalizację wyświetlacza. W pierwszym kroku konfigurowany jest tryb pracy kontrolera LCD jako wyświetlacza 16×2, a następnie wysyłany jest zestaw komend konfigurujących sposób prezentacji treści: włączenie wyświetlania, ustawienia kursora oraz tryb automatycznego przesuwania kursora podczas dopisywania kolejnych znaków.

W sterowniku nie jest odczytywana flaga zajętości kontrolera LCD (busy flag), dlatego w miejscach wymagających dłuższego czasu wykonania po stronie wyświetlacza – w szczególności dla operacji *GroveLCD_Clear()* i *GroveLCD_Home()* – zastosowano krótkie opóźnienia czasowe *HAL_Delay()*, które zapewniają poprawne zakończenie komendy przed wysłaniem kolejnych danych.

e) Znaki własne (CGRAM)

Funkcja *GroveLCD_CreateChar()* zapisuje do pamięci CGRAM 8-bajtowy wzorzec znaku (dostępne sloty 0–7). Pozwala to prezentować symbole niedostępne w standardowym zestawie znaków LCD 16×2. W projekcie wykorzystano to do wyświetlania symboli notacji muzycznej w trybie nauki, takich jak długości nut oraz znaki chromatyczne.

5.3.10 Zasoby danych aplikacji (utwory i akordy) oraz mapowanie nut *(songs.c/.h, chords.c/.h, notes.c/.h)*

a) Rola modułów

Pliki *songs.c/.h*, *chords.c/.h* oraz *notes.c/.h* tworzą warstwę danych aplikacyjnych oraz zestaw funkcji pomocniczych, które są wykorzystywane przez logikę interfejsu użytkownika *app.c* oraz silnik lekcji *lesson.c*. Moduły te nie sterują sprzętem ani nie utrzymują stanu aplikacji – dostarczają one jedynie statyczne definicje utworów/akordów oraz proste operacje związane z mapowaniem nut.

b) Definicje utworów (*songs.c/.h*)

Moduł ten zawiera zestaw utworów dostępnych w trybie SONG. Każdy utwór opisany jest przez:

- tytuł (nazwa wyświetlana w menu),

- liczbę kroków lekcji,

- tablicę kroków, w których każdy krok lekcji zawiera 1–3 wymagane nuty (w tym wypadku numery MIDI z uwzględnieniem oktawy) jak i zapis do prezentacji na LCD (litera, znak chromatyczny oraz ikona długości).

Dzięki wyświetlaniu ikon długości nut użytkownik wie jak długo powinien grać dany dźwięk – w tym prototypie jednak nie jest to weryfikowane.

Dane te są wykorzystywane przez *app.c* do listy wyboru oraz przez *lesson.c* do weryfikacji poprawności gry w trybie SONG.

c) Definicje paczek akordów (*chords.c/.h*)

Ten moduł przechowuje paczki akordów dla trybu CHORDS. Struktury danych zawierają:

- nazwy paczek akordów (przedstawiane jako pozycje menu),

- listy akordów w paczce,

- składowe akordów (1–3 dźwięki) wraz z informacją pomocniczą – nazwą akordu, w celu ułatwienia jego zapamiętania.

W trybie CHORDS dopasowanie odbywa się po klasie wysokości (pitch class), dlatego dane akordów są interpretowane przez *lesson.c* w sposób niezależny od oktawy. Oznacza to, że jeśli musimy zagrać dźwięk np. C, to możemy zagrać dowolne C na naszym instrumencie klawiszowym, ponieważ akordy nie są przypisane do konkretnych oktaw.

d) Mapowanie nut i funkcje pomocnicze (*notes.c/.h*)

Ostatni moduł *notes.c/.h* dostarcza funkcje pomocnicze do operacji na nutach, takie jak konwersje pomiędzy nazwą nuty a numerem MIDI oraz operacje używane przy przygotowaniu i testowaniu danych (np. wyznaczanie klasy wysokości). Funkcje te ułatwiają rozbudowę bazy danych utworów i akordów oraz zwiększają czytelność kodu w pozostałych modułach.

Moduł *notes.c/.h* nie jest wymagany do działania bieżącej wersji aplikacji, jednak stanowi przygotowaną warstwę pomocniczą na potrzeby przyszłej rozbudowy projektu, np. o wczytywanie własnych utworów (konwersja nazw nut do numerów MIDI i odwrotnie).

Rozdział 6

Wynik działania układu

Wynikiem działania układu jest funkcjonowanie kompletnego systemu od strony użytkownika, który obejmuje nawigację po interfejsie, realizację trybów nauki oraz prezentację informacji zwrotnej i wyników lekcji.

6.1 Uruchomienie urządzenia

Po podaniu zasilania urządzenie inicjalizuje peryferia oraz uruchamia wyświetlacz LCD. Następnie prezentowany jest ekran startowy informujący o gotowości systemu.

Rys. 29.: Ekran startowy urządzenia, ***źródło: opracowanie własne***

Przejście do dalszej obsługi następuje po naciśnięciu przycisku *OK*.

6.2 Menu główne i nawigacja

Menu główne umożliwia użytkownikowi wybór jednej z trzech opcji:

- Legenda symboli (wyświetlenie znaków używanych w trybach nauki),

- Tryb nauki SONG (nauka utworów),

- Tryb nauki CHORDS (nauka akordów).

Nawigacja po menu użytkownika odbywa się trzema przyciskami:

- *NEXT* – przejście do kolejnej pozycji listy (przewijanie),

- *OK* – zatwierdzenie wyboru / wejście do wybranego trybu,

- *RESET* – powrót do poprzedniego ekranu.

Rys. 30.: Menu główne interfejsu użytkownika, *źródło: opracowanie własne*

6.3 Legenda symboli

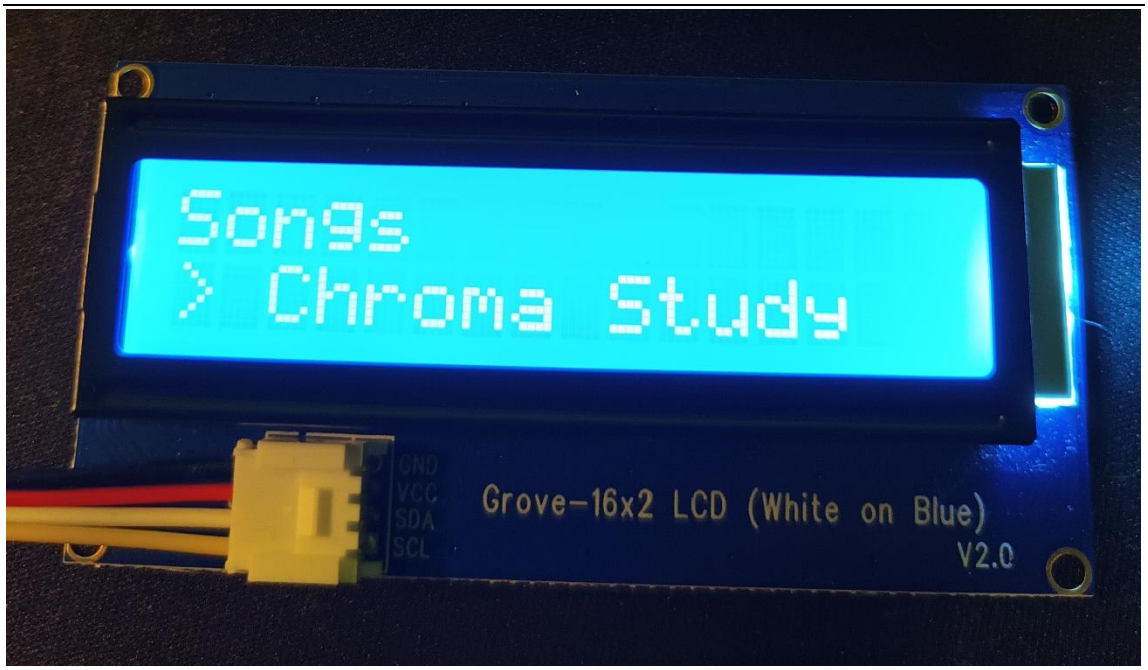
Po wybraniu pozycji „Icons” użytkownikowi wyświetla się ekran z symbolami używanymi w krokach lekcji. Wykorzystywane są znaki własne LCD do symboli długości nut oraz znaków chromatycznych, aby mimo ograniczeń wyświetlacza 16×2 możliwe było przedstawienie podstawowych elementów notacji.

Rys. 31.: Ekran legendy symboli, *źródło: opracowanie własne*

Powrót do menu głównego realizowany jest za pomocą przycisku *RESET*.

6.4 Tryb nauki SONG

Po wybraniu pozycji „Songs” użytkownik przenosi się do listy utworów dostępnych do zagrania w trybie nauki SONG. Użytkownik przewija listę przyciskiem *NEXT*, a wybrany utwór zatwierdza przyciskiem *OK*.



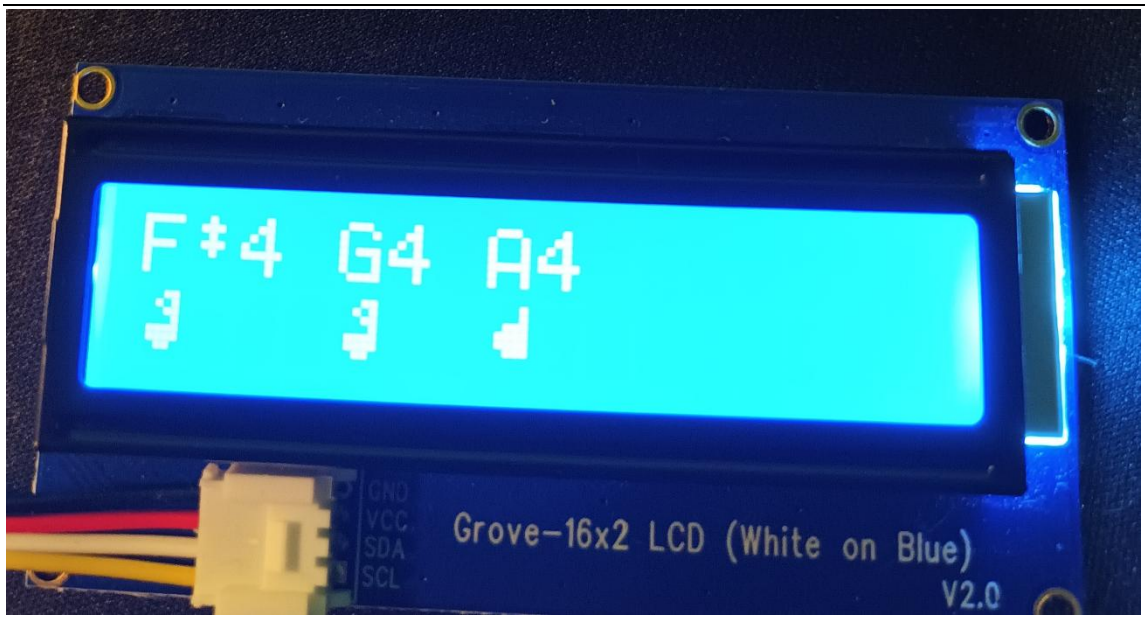
Rys. 32.: Lista utworów w trybie SONG, ***źródło: opracowanie własne***

Po wybraniu utworu następuje uruchomienie lekcji, a urządzenie przechodzi do prezentacji kolejnych kroków. Każdy krok zawiera 1–3 wymagane dźwięki do zagrania, wraz z ikonką o długości danego dźwięku. Użytkownik gra na instrumencie klawiszowym podłączonym przez USB, a mikrokontroler ocenia poprawność w czasie rzeczywistym:

- zielona dioda LED sygnalizuje poprawne trafienie wymaganej nuty,

- czerwona dioda LED sygnalizuje błąd.

Wszystkie trafienia i błędy są zapisywane w statystykach, w celu ich późniejszego wyświetlania na podsumowaniu. Po poprawnym zagraniu wszystkich wymaganych dźwięków w danym kroku program przechodzi do kroku następnego.



Rys. 33.: Ekran lekcji w trybie SONG (przykładowy krok), *źródło: opracowanie własne*

W trakcie lekcji użytkownik ma możliwość sterowania przebiegiem ćwiczenia z użyciem przycisków:

- *OK* – pominięcie bieżącego kroku (pozostałe w danym kroku lekcji nuty są zaliczane jako trafienia),

- *RESET* – powrót do kroku startowego (krok 0) lub powrót do listy wyboru utworów, jeśli użytkownik jest już na kroku startowym,

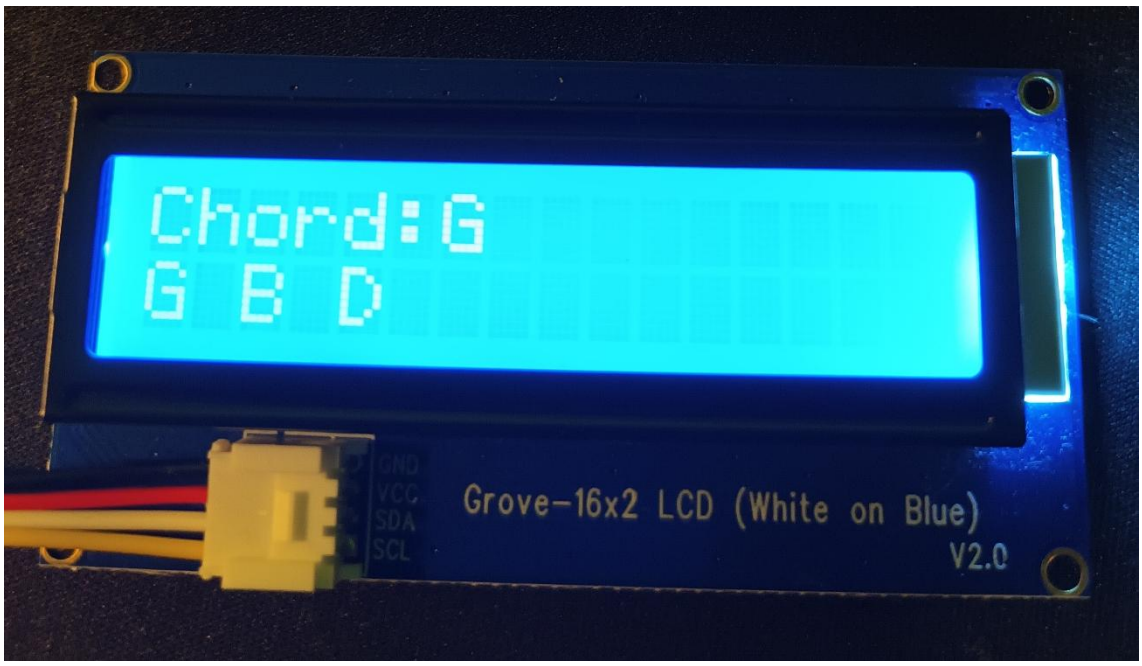
- *NEXT* – cofnięcie się o jeden krok lekcji do tyłu (lub zakończenie, jeśli użytkownik jest na kroku startowym).

6.5 Tryb nauki CHORDS

Po wybraniu pozycji „Chords” w menu głównym, użytkownik zostaje przeniesiony do listy dostępnych paczek akordów do nauki w trybie CHORDS. Prezentacja listy dostępnych materiałów oraz sterowanie odbywa się analogicznie jak w trybie SONG.

W trybie nauki CHORDS użytkownik wybiera paczki akordów, co oznacza, że jedna lekcja składa się z przynajmniej kilku różnych trójdźwięków. Każdy krok lekcji odpowiada jednemu akordowi, a przejście do kolejnego kroku następuje po poprawnym zagraniu wszystkich jego składowych dźwiękowych. Weryfikacja odbywa się po klasie

wysokości, dzięki czemu dźwięki można zagrać w dowolnej oktawie. Informacja zwrotna jest identyczna jak w trybie SONG (zielona/czerwona dioda).

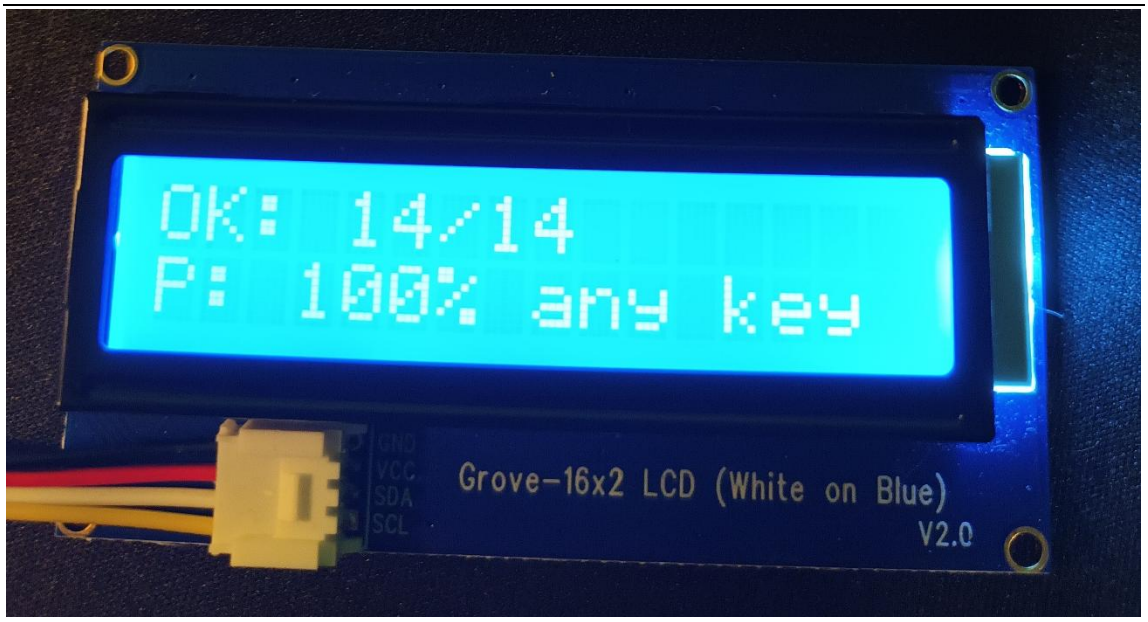


Rys. 34.: Ekran lekcji w trybie CHORDS (przykładowy krok), ***źródło: opracowanie własne***

Przyciski w trakcie ćwiczeń działają analogicznie jak w trybie SONG (pomijanie kroku, cofanie, reset/zakończenie).

6.6 Ekran podsumowania

Po ukończeniu wszystkich kroków lekcji (w trybie SONG lub CHORDS) urządzenie wyświetla ekran podsumowania zawierający wynik ćwiczenia (suma poprawnych naciśnięć na wszystkie naciśnięcia oraz procent skuteczności). Zakończenie podsumowania i powrót do listy następuje po naciśnięciu dowolnego przycisku.



Rys. 35.: Ekran podsumowania wyświetlany po zakończeniu lekcji, *źródło: opracowanie własne*

Wnioski

Celem niniejszej pracy było zaprojektowanie i implementacja prototypu układu wspomagającego naukę gry na instrumencie klawiszowym, opartego na mikrokontrolerze STM32 i komunikacji USB w trybie Host. Zrealizowany system umożliwia bezpośrednie podłączenie instrumentu MIDI przez interfejs USB, analizę odebranych zdarzeń muzycznych oraz prowadzenie użytkownika przez interaktywne tryby nauki z wykorzystaniem prostego interfejsu użytkownika i informacji zwrotnej.

W ramach pracy zaprojektowano rozwiązanie obejmujące część sprzętową w postaci doboru i integracji gotowych modułów (płytka STM32, wyświetlacz LCD, przyciski, diody LED) oraz autorską część programową. Opracowano architekturę oprogramowania z wyraźnym podziałem na warstwy: obsługę USB Host, autorską implementację klasy USB-MIDI, silnik lekcji odpowiedzialny za weryfikację poprawności gry, interfejs użytkownika oparty na maszynie stanów oraz moduły pomocnicze i sterowniki peryferiów. Szczególną uwagę poświęcono poprawnej obsłudze komunikacji USB, w tym odbiorowi danych MIDI niezależnemu od głównej logiki aplikacji i buforowaniu ich w kolejce FIFO.

Opracowany prototyp umożliwia pracę w dwóch trybach nauki: SONG oraz CHORDS. Zastosowanie wyświetlacza LCD 16×2 oraz trzech przycisków pozwoliło na stworzenie prostego, lecz czytelnego interfejsu użytkownika, umożliwiającego nawigację po menu, wybór materiału do nauki oraz obserwację postępów. Informacja zwrotna w postaci diod LED zapewnia natychmiastową ocenę poprawności zagranego dźwięku, co zwiększa interaktywność i użyteczność urządzenia.

Zrealizowany system stanowi funkcjonalny prototyp, który może być podstawą do dalszego rozwoju. Potencjalne kierunki rozbudowy obejmują m.in.:

- realizacja sprzętowej nakładki, umieszczanej na instrumencie klawiszowym, która mapowałaby klawisze z diodami LED, zapalając je przy odpowiednich nutach,

- dodanie możliwości wgrywania własnych materiałów edukacyjnych w formacie MIDI,

- rozbudowę interfejsu użytkownika, np. poprzez zastosowanie większego wyświetlacza graficznego,

- dodanie możliwości zapętlania konkretnych fragmentów w trybie nauki,

- obsługę komunikacji MIDI w obu kierunkach, co pozwoliłoby np. nagrywać własne utwory na mikrokontroler,

- rozszerzenie ilości nut na krok lekcji (aktualnie trzy),

- rozszerzenie bazy danych utworów oraz akordów,

- zastosowanie systemu operacyjnego czasu rzeczywistego (RTOS) w celu dalszej separacji zadań i zwiększenia skalowalności projektu.

- realizację układu w postaci dedykowanego obwodu drukowanego.

Bibliografia

[1] MIDI Association, MIDI History Chapter 6-MIDI Begins 1981-1983, https://midi.org/midi-history-chapter-6-midi-begins-1981-1983, stan na dzień 21.12.2025

[2] MIDI Association, About MIDI-Part 4:MIDI Files, https://midi.org/about-midi-part-4midi-files, stan na dzień 21.12.2025

[3] MIDI Association, MIDI 1.0 Detailed Specification, https://midi.org/midi-1-0-detailed-specification, stan na dzień 21.12.2025

[4] MIDI Association, Standard MIDI Files Specification, https://midi.org/standard-midi-files-specification, stan na dzień 21.12.2025

[5] CHD Elektroservis, MIDI Communications Tips, https://www.chd-el.cz/support/application/app001-midi/, stan na dzień 21.12.2025

[6] Strona aplikacji Flowkey, https://www.flowkey.com/en, stan na dzień 20.05.2025

[7] Strona aplikacji Synthesia, https://www.synthesiagame.com/, stan na dzień 20.05.2025

[8] Oficjalna strona CASIO, https://www.casio.com/us/search/?query=CASIO%20LK&type=product, stan na dzień 21.12.2025

[9] CASIO, Instrukcja użytkownika CASIO LK-S250, https://www.casio.com/content/dam/casio/global/support/manuals/electronic-musical-instruments/pdf/008-en/l/LKS250_usersguide_B_EN.pdf, stan na dzień 21.12.2025

[10] Strona aplikacji Chordana Play, https://web.casio.com/app/en/play/top.html, stan na dzień 21.12.2025

[11] USB Implementers Forum, The Original USB 2.0 specification released on April 27, 2000, https://www.usb.org/document-library/usb-20-specification,

stan na dzień 28.12.2025

[12] Infineon, USB 101: An Introduction to Universal Serial Bus 2.0 https://www.infineon.com/assets/row/public/documents/cross-divisions/42/infineon-an57294-usb-101-an-introduction-to-universal-serial-bus-2.0-applicationnotes-en.pdf?fileId=8ac78c8c7cdc391c017d072d8e8e5256&, stan na dzień 28.12.2025

[13] STMicroelectronics, UM1720 – STM32Cube USB Host library (ST), https://www.st.com/resource/en/user_manual/um1720-stm32cube-usb-host-library-stmicroelectronics.pdf, stan na dzień 28.12.2025

[14] USB Implementers Forum, Universal Serial Bus Device Class Definition for MIDI Devices, https://www.usb.org/sites/default/files/midi10.pdf, stan na dzień 29.12.2025

[15] STMicroelectronics, STM32 configuration and initialization C code generation, https://www.st.com/resource/en/data_brief/stm32cubemx.pdf, stan na dzień 30.12.2025

[16] STMicroelectronics, STM32CubeIDE user guide - User manual, https://www.st.com/resource/en/user_manual/um2609-stm32cubeide-user-guide-stmicroelectronics.pdf, stan na dzień 30.12.2025

[17] Altium, Altium Designer Documentation, https://www.altium.com/documentation/altium-designer, stan na dzień 30.12.2025

[18] Seeed Studio, Oficjalna wiki elementu oraz dokumentacja tam zawarta, https://wiki.seeedstudio.com/Grove-16x2_LCD_Series/#tech-support--product-discussion, stan na dzień 30.12.2025

[19] STMicroelectronics, Datasheet - STM32L476xx, https://www.st.com/resource/en/datasheet/stm32l476je.pdf, stan na dzień 30.12.2025

Dodatek A

Deskryptory elektronicznego instrumentu klawiszowego CASIO USB-MIDI

Do ich odczytu skorzystano z aplikacji *Thesycon USB Descriptor Dumper*, link do strony producenta.

Information for device CASIO USB-MIDI (VID=0x07CF PID=0x6803):

Connection Information:

Device current bus speed: FullSpeed

Device supports USB 1.1 specification

Device supports USB 2.0 specification

Device address: 0x0002

Current configuration value: 0x01

Number of open pipes: 2

Device Descriptor:

0x12 bLength

0x01 bDescriptorType

0x0200 bcdUSB

0x00 bDeviceClass

0x00 bDeviceSubClass

0x00 bDeviceProtocol

0x40 bMaxPacketSize0 (64 bytes)

0x07CF idVendor

0x6803 idProduct

0x0100 bcdDevice

0x01 iManufacturer "CASIO"

0x02 iProduct "CASIO USB-MIDI"

0x00 iSerialNumber

0x01 bNumConfigurations

Configuration Descriptor:

0x09 bLength

0x02 bDescriptorType

0x0065 wTotalLength (101 bytes)

0x02 bNumInterfaces

0x01 bConfigurationValue

0x00 iConfiguration

0xC0 bmAttributes (Self-powered Device)

0x00 bMaxPower (0 mA)

Interface Descriptor:

0x09 bLength

0x04 bDescriptorType

0x00 bInterfaceNumber

0x00 bAlternateSetting

0x00 bNumEndPoints

0x01 bInterfaceClass (Audio Device Class)

0x01 bInterfaceSubClass (Audio Control Interface)

0x00 bInterfaceProtocol (Audio Protocol undefined)

0x00 iInterface

AC Interface Header Descriptor:

0x09 bLength

0x24 bDescriptorType

0x01 bDescriptorSubtype

0x0100 bcdADC

0x0009 wTotalLength (9 bytes)

0x01 bInCollection

0x01 baInterfaceNr(1)

Interface Descriptor:

0x09 bLength

0x04 bDescriptorType

0x01 bInterfaceNumber

0x00 bAlternateSetting

0x02 bNumEndPoints

0x01 bInterfaceClass (Audio Device Class)

0x03 bInterfaceSubClass (MIDI Streaming Interface)

0x00 bInterfaceProtocol (Audio Protocol undefined)

0x00 iInterface

MS Interface Header Descriptor:

0x07 bLength

0x24 bDescriptorType

0x01 bDescriptorSubtype

0x0100 bcdMSC

0x0041 wTotalLength (65 bytes)

MS MIDI IN Jack Descriptor:

0x06 bLength

0x24 bDescriptorType

0x02 bDescriptorSubtype

0x01 bJackType

0x01 bJackID

0x00 iJack

MS MIDI IN Jack Descriptor:

0x06 bLength

0x24 bDescriptorType

0x02 bDescriptorSubtype

0x02 bJackType

0x02 bJackID

0x00 iJack

MS MIDI OUT Jack Descriptor:

0x09 bLength

0x24 bDescriptorType

0x03 bDescriptorSubtype

0x01 bJackType

0x03 bJackID

0x01 bNrInputPins

0x02 baSourceID(1)

0x01 baSourcePin(1)

0x00 iJack

MS MIDI OUT Jack Descriptor:

0x09 bLength

0x24 bDescriptorType

0x03 bDescriptorSubtype

0x02 bJackType

0x04 bJackID

0x01 bNrInputPins

0x01 baSourceID(1)

0x01 baSourcePin(1)

0x00 iJack

Endpoint Descriptor (Audio/MIDI 1.0):

0x09 bLength

0x05 bDescriptorType

0x01 bEndpointAddress (OUT endpoint 1)

0x02 bmAttributes (Transfer: Bulk / Synch: None / Usage: Data)

0x0040 wMaxPacketSize (64 bytes)

0x00 bInterval

0x00 bRefresh

0x00 bSynchAddress

MS Bulk Data Endpoint Descriptor:

0x05 bLength

0x25 bDescriptorType

0x01 bDescriptorSubtype

0x01 bNumEmbMIDIJack

0x01 baAssocJackID(1)

Endpoint Descriptor (Audio/MIDI 1.0):

0x09 bLength

0x05 bDescriptorType

0x82 bEndpointAddress (IN endpoint 2)

0x02 bmAttributes (Transfer: Bulk / Synch: None / Usage: Data)

0x0040 wMaxPacketSize (64 bytes)

0x00 bInterval

0x00 bRefresh

0x00 bSynchAddress

MS Bulk Data Endpoint Descriptor:

0x05 bLength

0x25 bDescriptorType

0x01 bDescriptorSubtype

0x01 bNumEmbMIDIJack

0x03 baAssocJackID(1)

Microsoft OS Descriptor is not available. Error code: 0x0000001F

String Descriptor Table

Index LANGID String

0x00   0x0000 0x0409

0x01   0x0409 "CASIO"

0x02   0x0409 "CASIO USB-MIDI"

Connection path for device:

USB xHCI Compliant Host Controller

Root Hub

CASIO USB-MIDI (VID=0x07CF PID=0x6803) Port: 1

Running on: Windows 10 or greater (Build Version 26200)

Brought to you by TDD v2.19.0

Spis ilustracji