



**Politechnika Krakowska
im. Tadeusza Kościuszki**

Wydział Informatyki i Telekomunikacji

Łukasz Rotko

Numer albumu: 139284

**Implementacja protokołu komunikacyjnego LIN w
urządzeniu zbudowanym w oparciu o
mikrokontroler**

**Implementation of the LIN communication
protocol in a device based on a microcontroller**

**Praca inżynierska
na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem:
Dr inż. Jerzy Białas

Kraków, ROK 2024

Spis treści

Wstęp	3
Rozdział 1. Mikrokontrolery i protokół LIN	5
1.1. Protokół LIN	5
1.1.1. Przedstawienie protokołu LIN	5
1.1.2. Kluczowe zasady działania protokołu LIN	7
1.2. Mikrokontrolery	10
1.2.1. Historia mikrokontrolerów	10
1.2.2. Przegląd mikrokontrolerów	12
1.3. Wybór mikrokontrolera	15
Rozdział 2. Oprogramowanie	19
2.1. Przygotowanie środowiska programistycznego	19
2.1.1. Wybór języka programowania	19
2.1.2. Wybór bibliotek	21
2.1. Implementacja protokołu LIN	22
2.1.1. Obsługa PIO	22
2.1.2. Proces deklaracji, przetwarzania i wysyłania danych	27
2.1.3. Wykorzystanie drugiego wątku do odczytu danych	32
2.2. Implementacja interfejsu użytkownika i automatyzacja flashowania	38
Rozdział 3. Prezentacja urządzenia i testy	41
3.1. Prezentacja działania urządzenia	41
3.1. Poprawność transmisji	44
3.2. Weryfikacja problemów i przedstawienie potencjalnych ulepszeń	47
Podsumowanie	49
BIBLIOGRAFIA	50
SPIS RYSUNKÓW	51

Wstęp

Wraz rozwojem mikrokontrolerów, stały się one nieodłączną częścią elektroniki, używanej w codziennym życiu każdego człowieka. Mikrokontrolery znajdują swoje zastosowanie w elektronice użytkowej takiej jak sprzęt AGD, a także mają swoje zastosowanie w przemyśle. Znaleźć je można zarówno w pralkach czy lodówkach jak i w sterownikach przemysłowych odpowiedzialnych za sterowanie różnymi maszynami.

W ostatnim czasie zyskały one również uznanie w gronie majsterkowiczów, którzy na własną rękę tworzą urządzenia np. do zdalnego sterowania oświetleniem domowym. W tym celu bardzo pomocne okazały się układy, które opierały się na mikrokontrolerach, jednak zawierały również dodatkowe peryferia. Oferują one różnorodne możliwości zastosowań w życiu codziennym jak i w środowisku profesjonalnym.

W niniejszej pracy przedstawiono rozwiązanie problemu pochodzącego ze środowiska zawodowego z wykorzystaniem takiej właśnie płytki. Problem polegał na automatyzacji testów urządzeń posługujących się wcześniej wspomnianym protokołem. Ze względu na niski koszt i możliwość oprogramowania tych urządzeń, zostały one wybrane na podstawę projektu, a wszelkie wymagania dotyczące funkcjonalności zostały zaczerpnięte z odpowiednich standardów ISO.

Zastosowania przedstawianego urządzenia ograniczają się głównie do dziedziny motoryzacyjnej, gdyż to w niej jest wykorzystywany protokół LIN. Jednak ze względu na szeroki zakres możliwości modyfikacji zaimplementowanego programu jest to nadal bardzo wszechstronne narzędzie. Jego mocną stroną jest zastosowanie w testach różnego rodzaju urządzeń posługujących się protokołem LIN.

Autor niniejszej pracy chciałby skierować szczególne podziękowania dla promotora Dr inż. Jerzego Białasa, który służył swoją pomocą i dobrą radą na każdym etapie jej tworzenia.

Celem pracy jest stworzenie oprogramowania na urządzenie oparte o mikrokontroler implementujące możliwość komunikacji protokołem LIN.

Rozdział pierwszy został poświęcony analizie zarówno mikrokontrolerów jak i protokołu LIN. Przedstawiono kluczowe wymagania, które definiowały zachowania urządzeń posługujących się protokołem LIN. Wykorzystano te informacje do wyboru mikrokontrolera, który został podstawą całego urządzenia

W drugim rozdziale szczegółowo opisano implementację oprogramowania. Opisano w nim język programowania, programy wspomagające programowanie oraz wykorzystane biblioteki. Następnie przedstawione zostały dodatkowe informacje dotyczące gotowych funkcjonalności wybranej płytki, kod źródłowy programu oraz jego działanie.

Ostatni rozdział przedstawia testy, które sprawdziły poprawność działania opracowanego urządzenia oraz jego ograniczenia. Rozpoczęto od prezentacji działania urządzenia w przykładowym scenariuszu. Testy sprawdziły zgodność implementacji z wcześniej założonymi wymaganiami. Końcowa część rozdziału została poświęcona analizie wszelkich napotkanych błędów oraz rozważaniom dotyczącym ich naprawienia.

Rozdział 1. Mikrokontrolery i protokół LIN

1.1. Protokół LIN

Protokół LIN towarzyszy w codziennym życiu wielu osobom. Wykorzystuje się go w dziedzinie motoryzacji, gdzie posługują się nim różne przełączniki w autach, motocyklach lub ciężarówkach. Mogą to być sterowniki wycieraczek, przyciski na kierownicy, bądź też przyciski do regulacji fotela. W tym rozdziale przedstawiono szczegółowo jego działanie, aby wyjaśnić wymagania dla oprogramowania zaimplementowanego w urządzeniu opartym na mikrokontrolerze komunikującym się protokołem LIN.

1.1.1. Przedstawienie protokołu LIN

Informacje dotyczące cech protokołu LIN są zawarte w standardzie ISO 17897 [1], do którego dostęp jest płatny. W związku z czym, w tym rozdziale odniesiono się również do dokumentu pt. „LIN Protocol and Physical Layer Requirements” [2] utworzonego przez firmę Texas Instruments, która udostępnia do grona publicznego streszczoną część informacji zawartych w normie.

Protokół LIN (Local Interconnect Network) jest tańszym zamiennikiem protokołu CAN (Controller Area Network), który natomiast jest bardziej zaawansowanym protokołem, również wykorzystywanym w dziedzinie motoryzacji. LIN jest prostszym, wolniejszym oraz bardziej awaryjnym protokołem, jednak przez mniejszy koszt produkcji stał się równie popularny. Przeważnie nie zastępuje się protokołu CAN protokołem LIN w całości, natomiast urządzenia w sieci LIN są połączone z magistralą CAN poprzez urządzenie typu master jako podsieć [2]. Jest on używany tylko do niektórych elementów pojazdu, gdyż krytyczne dla bezpieczeństwa podzespoły muszą spełniać bardziej zaawansowane normy.

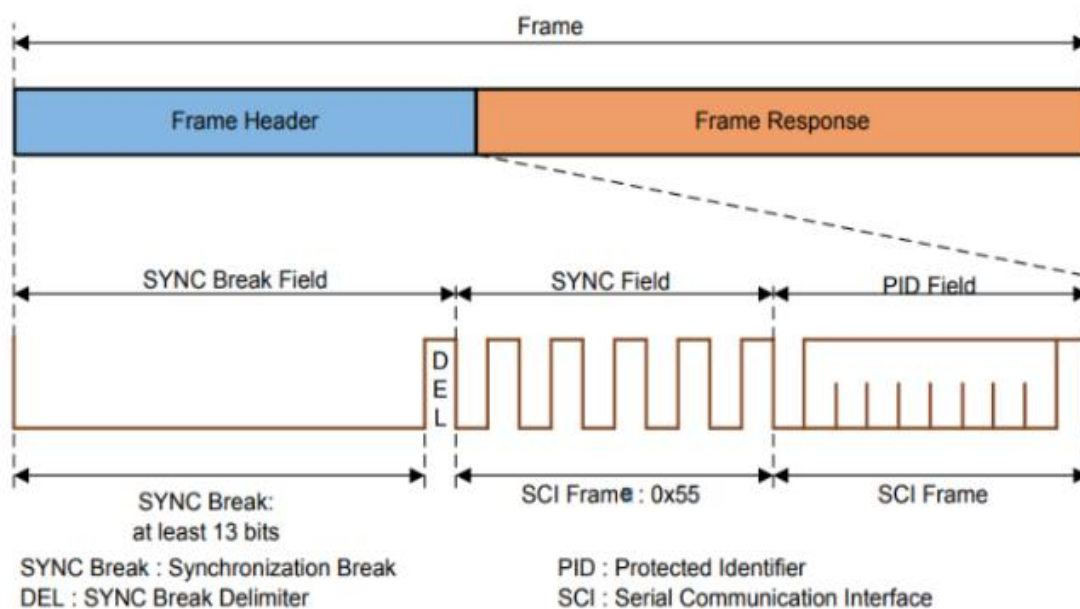
Zanim utworzono protokół LIN, próbowano wykorzystywać protokół UART jednak był on bardzo zawodny i nie miał żadnych zabezpieczeń przed utratą komunikacji, fałszowaniem danych, czy też synchronizacji. Dlatego utworzono protokół LIN, który nadal w tani sposób miał zaradzić wspomnianym problemom. Urządzenia w sieci LIN mają dwa typy, jest to „master” oraz „slave”. „Master” wysyła polecenia lub zapytania do urządzeń „slave”, które z kolei, mogą jedynie odpowiadać. Urządzenie

typu master może również służyć jako połączenie pomiędzy siecią LIN i siecią CAN [2].

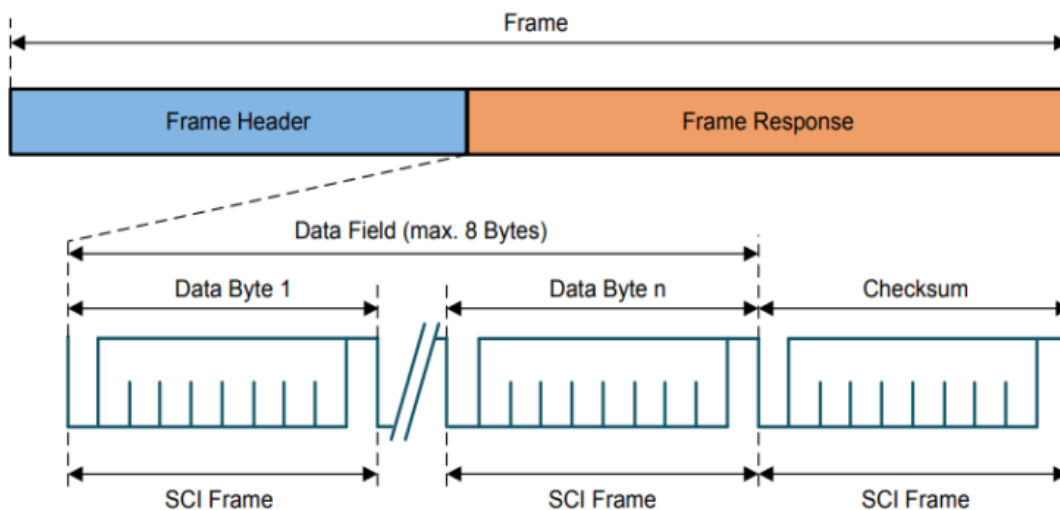
Protokół LIN operuje w zakresie prędkości 1-20kbit/s, jednak przeważnie dobierane są wartości 9600bit/s lub 19200bit/s oraz powinien utrzymywać komunikację na dystansie do 40 metrów. Wiadomości są wysyłane i odbierane na jednej linii, która składa się z dwóch stanów. Pierwszym jest stan dominujący – logiczne „0” oraz drugim, stan recesywny - logiczna „1”. Ich długość lub inaczej czas trwania określa stała czasowa o nazwie T-bit, która wynosi od 1μs do 5μs. Sama struktura wiadomości w protokole LIN (Rys. 1, Rys. 2) zawiera następujące segmenty [2]:

- Pole przerwy / wybudzenia (Break Field) – jest to początek wiadomości w postaci stanu dominującego, który trwa przynajmniej przez 13 T-bitów i kończy się stanem recesywnym,
- Pola synchronizacji (Sync Field) – jest to 8-bitowa liczba o wartości heksadecymalnej równej 0x55, przez co na linii naprzemiennie pojawiają się stany recesywne i dominujące w równych odstępach,
- Chroniony Identyfikator (PID) - jest to 8-bitowa liczba, która zawiera adres węzła typu „slave”, do którego było skierowane polecenie lub zapytanie, dodatkowo zawiera na końcu 2 bity parzystości P0 i P1 powstałe w wyniku operacji XOR z bitów od B1-B4 dla P1 oraz B1, B3, B4, B5 dla P2. Dodatkowo, identyfikatory dzielą się na różne rodzaje względem ich wartości:
 - Ramki o ID 0-59 są ramkami bezwarunkowymi, wykorzystywanymi do normalnej komunikacji,
 - Ramki o ID 60 (zapytanie od węzła master) oraz 61 (odpowiedź od węzła „slave”), które odpowiadają za diagnostykę lub konfigurację urządzenia, zawsze zawierają pełne 8 bitów danych,
- Pole danych (Data Field) – jeśli wiadomością jest polecenie, zawiera od 1 do 8 bajtów danych, jeśli jest to zapytanie to może być puste, jednak od razu po otrzymaniu ramki z identyfikatorem przez urządzenie „slave”, zostaje uzupełnione odpowiedzią od wskazanego węzła,

- Suma kontrolna (Checksum) – istnieją dwa rodzaje wykorzystywanej sumy, klasyczna, która odnosi się tylko do sekcji danych wykonując operacje bitową modulo-256, ulepszona wersja natomiast dodaje do operacji modulo-256 jeszcze wartość identyfikatora, obydwie operacje kończą się odwróceniem bitów.



Rys. 1. Graficzna ilustracja pola przerwy, pola synchronizacji oraz pola identyfikatora protokołu LIN [2].



Rys. 2. Graficzna ilustracja pola danych i pola sumy kontrolnej protokołu LIN [2].

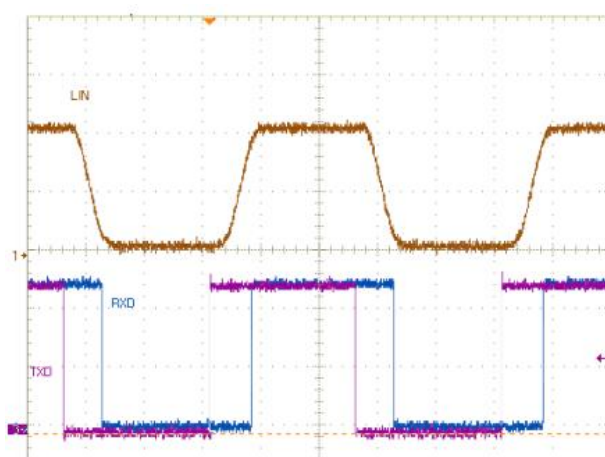
1.1.2. Kluczowe zasady działania protokołu LIN

W tym podrozdziale przedstawiono kryteria, które zostały wykorzystane, aby zaimplementować protokół LIN na urządzeniu opartym o mikrokontroler. Wymagania jakie przedstawia norma, są obowiązkowe dla urządzeń komercyjnych, jednak w

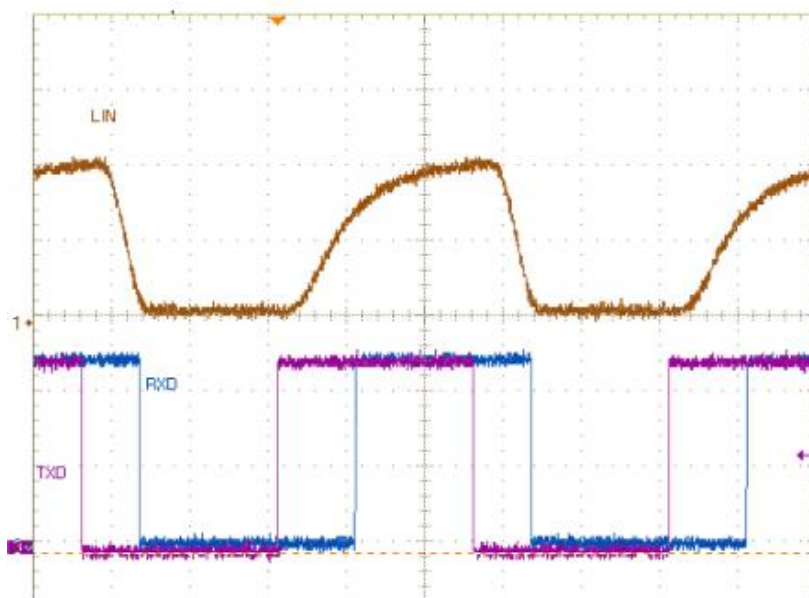
prezentowanym rozwiązaniu znajduje się jedynie podstawowa funkcjonalność. W tym jest zawarta możliwość komunikacji z innym urządzeniem, które posługuje się protokołem LIN oraz możliwość wyświetlania komunikacji użytkownikowi. Należy wspomnieć, iż według normy ISO 17897, protokół LIN operuje na zakresie 0-12V lub 0-24V jednak przedstawiono rozwiązanie operujące na 0-3.3V. Aby uzyskać wyższe przedziały napięcia, należałoby dodać układ zawierający transceiver, który transformuje sygnał z 3.3V na 12V lub 24V. Jego schemat możemy znaleźć w standardzie ISO 17897 [1].

Pierwszym krytycznym elementem będzie suma kontrolna. Urządzenie jest zobowiązane do sprawnego i bezbłędnego obliczania sumy kontrolnej, która albo będzie weryfikowana po otrzymaniu odpowiedzi lub obliczana przy jej wysyłaniu. Jeśli w którymś z tych dwóch przypadków nastąpi pomyłka, wiadomość może zostać zignorowana.

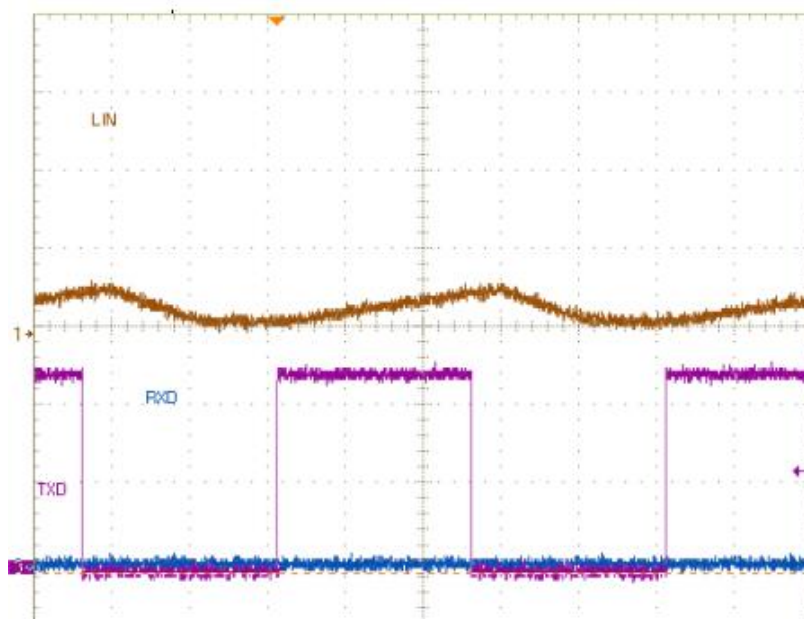
Sam sygnał, który zostanie wysyłany z urządzenia powinien spełniać warunki dotyczące struktury ramki oraz zadanymi przedziałami czasowymi określonymi w T-bitach. Niezmiernie ważnym jest, aby urządzenie było w stanie zmieniać stan recesywny na dominujący oraz odwrotnie w odpowiednim czasie. Jeśli zmiana będzie trwała za długo np. 0.5T-bit to może się okazać, iż urządzenie nie zrozumie przesyłanej wiadomości lub sama wiadomość zostanie zafałszowana poprzez przesunięcie bitów. Im zmiany sygnału stają się bardziej pionowe, tym większe szanse na pomyślną komunikację. Poniżej możemy zauważyć jak urządzenie obierające widzi wysyłany sygnał. Kanał TXD określa wiadomość nadawaną, natomiast RXD wiadomość odbieraną [2]:



Rys. 3. Sygnał z zbyt wolną zmianą stanu linii – nieczytelny przez odbiorcę [2].



Rys. 4. Sygnał z wolną zmianą stanu linii – czytelny przez odbiorcę [2].



Rys. 5. Sygnał z wolną zmianą stanu linii – czytelny przez odbiorcę [2].

Na powyższych rysunkach (Rys. 3-5) można zauważyć, iż istnieje przesunięcie w czasie nadawanego sygnału w Rys. 4. oraz pogorszenie tego zjawiska na Rys. 5. spowodowanym przez zbyt wolne podciąganie linii na poprawny poziom napięcia. Na Rys. 6 widać sytuację, gdzie nadajnik nie jest w stanie odpowiednio podciągnąć poziomu linii przez co wiadomość nie zostaje odczytana przez odbiorcę.

1.2. Mikrokontrolery

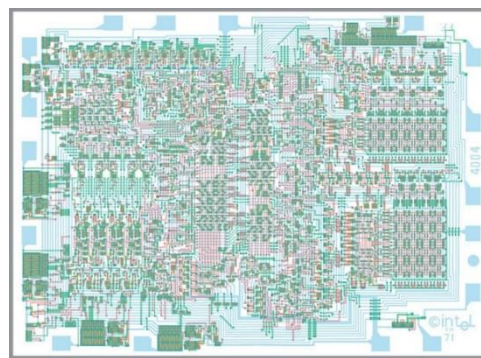
Mikrokontrolery to małe scalone systemy mikroprocesorowe składające się z wielu elementów. Poza jednostką centralną (CPU) zawierają również pamięć RAM, pamięć Flash oraz układy peryferyjne. Mikrokontrolery są stosowane w szerokiej gammie produktów. Są one używane np. w pralkach, lodówkach czy piekarnikach. Mają także zastosowania poza dziedziną elektroniki użytkowej. Można je spotkać w urządzeniach medycznych, robotyce oraz automatyce ogólnej a także w wielu innych dziedzinach. W tym rozdziale bliżej przedstawiona zostanie historia mikrokontrolerów, ich dostępne modele na rynku oraz sam mikrokontroler, który wykorzystano do budowy urządzenia obsługującego protokół LIN.

1.2.1. Historia mikrokontrolerów

Bazując na informacjach podanych w książce „Functional Reverse Engineering of Machine Tools” [3], urządzenie uznane za pierwszy mikroprocesor to Intel 4004 (Rys. 7) wyprodukowany przez firmę Intel w 1971 roku. Był on zaprojektowany architekturą harwardzką co oznaczało posiadanie osobnych pamięci dla programu oraz danych. Rodzina Intel 4004 (Rys. 6) wymagała 256 bajtów pamięci ROM. Same mikroprocesory odpowiadały za funkcję które posiadają procesory umieszczane w komputerach, co oznacza przyjmowanie instrukcji i ewentualne zwracanie wyniku operacji. Przed Intel 4004 firma, która go wyprodukowała, zajmowała się chipami pamięci. Kolejno Intel 4001/2 posiadały 256 bajtów pamięci ROM oraz 40 bajtów pamięci RAM, a Intel 4003 posiadał dodatkowo 10-bitowy rejestr przesuwany dla operacji wejścia/wyjścia. Wykorzystywano go do odczytu sygnałów z klawiatur, drukarek i innych urządzeń użytkowych.



Rys. 6. Zestaw chipów z rodziny Intel MCS-4: 4001, 4002, 4003 i 4004 [11].



Rys. 7. Zdjęcie matrycy procesora Intel 4004 [12].

Niestety mikroprocesor do pracy potrzebuje zewnętrznych elementów jak np. pamięci RAM odpowiedzialną za przechowanie danych i dodatkowych instrukcji oraz pamięci ROM, która przechowuje niezmiennie instrukcje np. do uruchomienia systemu. Elementy te były umieszczane osobno na płycie głównej, na której również montowany był mikroprocesor. Mikrokontroler zaś rozwiązuje ten problem integrując pamięci, peryferia wejścia i wyjścia, watchdogi, konwertery analogowe i kilka innych elementów w jednym układzie scalonym, których brakuje w mikroprocesorach. [4]

Inne różnice o których warto wspomnieć to min.:

- Mikroprocesor:
 - Architektura von Neumanna,
 - Bardziej skomplikowana metoda pobierania instrukcji z pamięci SRAM,
 - Przeważnie operują na 32-bitowej reprezentacji liczb (większy zakres liczb oraz lepsza dokładność),
- Mikrokontroler
 - Architektura harwardzka,
 - Mniejsza ilość cykli na instrukcję dzięki zintegrowanej pamięci RAM,
 - Przeważnie ograniczone do 8-bitowej lub 16-bitowej reprezentacji liczb.

Pierwszym mikrokontrolerem był wyprodukowany w 1976 roku, również przez firmę Intel, Intel 8048/49, który został wykorzystany w oryginalnej klawiaturze firmy IBM do komputerów personalnych. Następnie w roku 1980 początkiem rodziny MCS-51 był mikroprocesor nazwany 8051. Jego zaletami były: niski koszt, ulepszony zbiór instrukcji i dostępność narzędzi programistycznych. Był on wyposażony w 2 timery 16-bitowe, cztery 8-bitowe porty, 128 bajtów pamięci RAM oraz 4 kilobajty ROM [3].

W kolejnych latach strukturę MCS-51 rozwijały również takie firmy jak: AMD, ATMEL 89x, Philips i wiele innych. Dzięki temu rozwój technologii mikrokontrolerów przyspieszył i wyznaczono oddzielne rodzaje które odnosiły się do pamięci, rodzaju architektury czy zestawu instrukcji. Rozróżniane są takie typy jak PIC, AVR, ARM i kilka innych. PIC wykorzystywany jest w produktach firmy Microchip np. w PIC16. AVR możemy znaleźć w mikrokontrolerach serii ATmega lub Arduino. ARM to typ, z którego korzysta ARM Cortex-M0 lub ARM Cortex-M7 [3].

1.2.2. Przegląd mikrokontrolerów

Na rynku dostępne są bardzo różnorodne propozycje mikrokontrolerów. Rozpoczynając od małych płytek zawierających podstawowe funkcjonalności, aż do pełnoprawnych mini-komputerów z własnym systemem operacyjnym. Dzięki szerokiej gammie produktów, można wybrać dobrze dopasowany mikrokontroler do naszych potrzeb. Czasem to zaoszczędzi niepotrzebnego nadpłacania lub też pozwoli na dobór odpowiednio zaawansowanego sprzętu.

Pierwszym potencjalnym wyborem bazy urządzenia komunikującego się protokołem LIN jest mikrokontroler – Atmega328P. Jest on osadzony na popularnej płytce Arduino Uno Rev3 (Rys. 10) [\[5\]](#).

Ta wersja płytki Arduino zawiera:

- Jedno-rdzeniowy procesor AVR o częstotliwości maksymalnej 16 MHz,
- 32 KB pamięci flash, 2 KB SRAM, 1KB EEPROM,
- Dużo gotowych opcji wejścia/wyjścia jak np. I2C, SPI, PWM,
- 14 pinów ogólnych we/wy w tym 6 pinów z możliwościami modulacji szerokości impulsów (PWM).

Jest to płytka nazwana „Uno” ze względu na jej równoległą premierę z wersją 1.0 oprogramowania Arduino Software (IDE). Z tego względu jest to bardzo prosta i łatwa w obsłudze podstawa wielu projektów, przyjazna dla nowicjuszy. Oprogramowanie dla tej płytki opiera się na języku pochodnym od języka C. Język ten wprowadza niewielkie zmiany, gdzie najbardziej zauważalną jest początkowa struktura pliku wykonywalnego, która jest podzielona na sekcje deklaracji, jednorazowej funkcji startu oraz pętli wykonywanej przez cały czas działania płytki [\[5\]](#).

Środowisko programistyczne jest bardzo łatwe w obsłudze, zawiera gotowe przykłady podstawowych programów oraz dużo ułatwień takich jak prosty proces flashowania czy manager bibliotek wbudowany w samo środowisko. Całość została również opisana na stronie producenta w dokumentacji odnoszącej się do poszczególnych funkcji jak i całego środowiska Arduino Software.

Cena na okres 12.2023 wynosi ok. 105 zł, jednak istnieją zamienniki współpracujące z oprogramowaniem Arduino, których koszt to ok. 40 zł.

Kolejną propozycją jest płytka STM32 NUCLEO-L432KC (Rys. 8) z mikrokontrolerem STM32L432KCU6.

Ten produkt od firmy STMicroelectronics zawiera [6]:

- Jedno-rdzeniowy procesor ARM o częstotliwości maksymalnej 80 MHz,
- 256 KB pamięci flash, 64 KB SRAM,
- Interfejsy jak np. USB, SAI, I2C, SPI, USART, CAN, SWPMI, IRTIM,
- 11 timerów,
- 26 szybkich portów we/wy,
- Przetworniki A/D i D/A, wzmacniacz z wbudowanym PGA.

Producent STMicroelectronics wspiera swoje produkty oprogramowaniem STM32CubeIDE, które pozwala na wgrywanie programów na płytki. W porównaniu do przedstawianych propozycji, obsługa urządzenia w tym przypadku jest najbardziej skomplikowana, ale przez to również daje bardziej zaawansowane możliwości konfiguracji. Możliwość ręcznego przypisania funkcjonalności na danym pinie w innych propozycjach jest ustawiana za pomocą polecenia, w STM32CubeIDE możemy to zmienić za pomocą graficznego interfejsu obrazującego mikrokontroler. Dodatkowo, z wbudowanym debuggerem jesteśmy w stanie zaczytywać wartości rejestrów, czy timerów, co znacznie upraszcza pracę z kodem. Język, który wykorzystuje STM32 NUCLEO to C.

Dokumentacja i poradniki znajdują się na stronie producenta w formie plików pdf bądź w postaci wpisu na stronie internetowej podzielonego na rozdziały. Przykładowe programy również są dostępne na stronie producenta. Dodatkowej pomocy nie brakuje ze względu na popularność tej platformy. Na platformie youtube lub forach o danej tematyce znajduje się wiele poradników dotyczących płytek STM32 NUCLEO.

Cena na okres 12.2023 wynosi ok. 60 zł.

Ostatnią propozycją jest mikrokontroler osadzony na Raspberry Pi Pico (Rys. 9) - RP2040.

Zaprojektowana przez firmę Raspberry Pi płytka oferuje min. [7]:

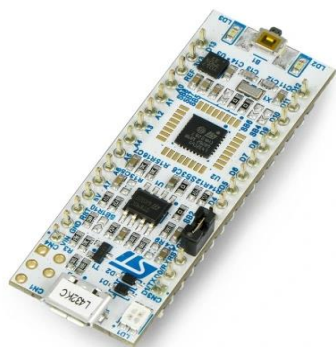
- Dwurdzeniowy procesor ARM o taktowaniu maksymalnym 133 MHz,
- 264kB wewnętrznej pamięci RAM i 16 MB dodatkowej pamięci flash,
- Dużo gotowych opcji wejścia/wyjścia jak np. I2C, SPI,
- bootloader UF2 umieszczony w pamięci ROM,
- 26 pinów ogólnych we/wy 8 Programowalnych pinów we/wy (PIO).

Dodatkowo można dokupić jeszcze moduł do bezprzewodowej komunikacji lub zestaw do debugowania. Są to dobrze przemyślane dodatki. Z jednej strony użytkownik nie nadpłaca za funkcjonalności, które przeważnie nie są wykorzystywane w wielu projektach. Z drugiej strony, jeśli użytkownik nie posiada innych urządzeń zdolnych pełnić pracę debuggera jak np. oscyloskop to również skorzysta na takiej ofercie.

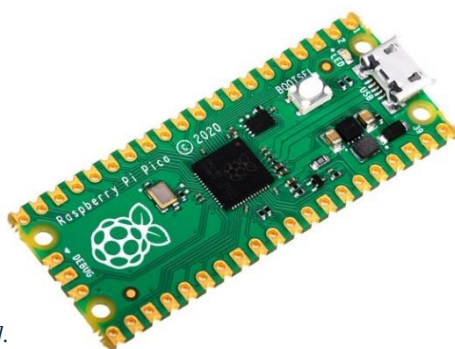
Języki, w których możemy tworzyć oprogramowanie to C/C++ lub Python z pomocą bibliotek MicroPython. Pod obydwa języki zostały przygotowane zestawy narzędzi (SDK), aby w łatwy sposób zacząć pracę z urządzeniem.

Dokumentacja jest rozwinięta bardzo dobrze na obydwu platformach i opisuje wszystkie funkcjonalności oraz dostępne instrukcje. Dodatkowo na repozytorium producenta znajdują się przykładowe programy wymagające jedynie wgrania na płytkę, co znacznie przyspiesza zrozumienie działania niektórych funkcji.

Cena za wersję podstawową (bez dodatków) na okres 12.2023 wynosi ok. 22 PLN.



Rys. 10. STM32 NUCLEO-L432KC [7].



Rys. 9. Raspberry Pi Pico [6].



Rys. 8. Arduino UNO [5].

1.3. Wybór mikrokontrolera

Kluczowym elementem całego urządzenia jest mikrokontroler. Od niego zależy jak bardzo można rozbudować urządzenie posługujące się protokołem LIN. Aby dokonać odpowiedniego wyboru mikrokontrolera, należy bliżej przyjrzeć się wymaganiom jakie to urządzenie powinno spełniać.

Znajomość zasad jakimi kieruje się protokół LIN pozwala na wybranie urządzenia z takimi podzespołami, które będą go obsługiwać w sposób sprawny i bezawaryjny. Należy jednak założyć odpowiedni zapas możliwości w dobieranej płytce ze względu na potencjalne rozszerzenie wymagań podczas pracy nad projektem. Biorąc to pod uwagę można stwierdzić, że urządzenie musi być elastyczne pod względem swojego zastosowania. Może posłużyć jako np. urządzenie pracujące bez potrzeby interakcji użytkownika lub wręcz przeciwnie, z ciągłym nadzorem pod postacią wyświetlania komunikacji w czasie rzeczywistym do przeanalizowania przez użytkownika.

Funkcjonalność takiego urządzenia powinna zawierać komunikację przynajmniej w trybie simplex. Należy jednak pamiętać, że taka komunikacja pozwoli jedynie na pracę w trybie odczytu wiadomości lub wydawania poleceń osobno. Niestety w realnym zastosowaniu jest to przeważnie niepraktyczne, gdyż komunikacja za pomocą protokołu LIN odbywa się w trybie half-duplex, co oznacza dwustronny naprzemienny kierunek wysyłania wiadomości. Zaimplementowanie tego trybu wymagałoby możliwości odczytu i wysyłania danych w czasie rzeczywistym. Jest to możliwe do implementacji na jednorzeniowym procesorze jednak wymagany byłby bardzo szybki procesor lub kolejkovanie wiadomości w dużych odstępach czasowych, aby jednostka centralna zdążyła ze wszystkimi operacjami. Natomiast wykorzystanie dwóch rdzeni, jednego do odczytu, drugiego do wysyłania wiadomości, powinno zwiększyć znacznie prędkość i możliwości urządzenia. W każdym przypadku coś poświęcamy, jednak rozsądniejszym jest wykorzystanie drugiego podejścia, gdyż na tę chwilę nie wiadomo, ile w praktyce zajmą kalkulacje i konwersje danych z sygnałów analogowych na cyfrowe.

Wcześniej wspomniane zastosowanie jest uznane jako ogólne. Przez to kolejną funkcjonalnością o jaką należy zadbać jest wyświetlanie danych komunikacji. Najbardziej praktycznym rozwiązaniem jest wyświetlanie komunikacji w czasie

rzeczywistym w konsoli oraz zapis historii w postaci pliku np. tekstowego. Takie funkcjonalności pozwoliłyby na analizę działania urządzenia, z którym byłaby prowadzona komunikacja. Pozwoliłoby to na odpluskwanie błędów samego urządzenia, sprawdzenie jego zachowania podczas testów, czy też zwykłą kontrolę i odczyt komunikacji, jeżeli zajdzie taka potrzeba bez konieczności podpięcia zaawansowanych analizatorów takich jak oscyloskop czy analizator stanów.

Następnie pozostaje rozważenie sposobu zapisu danych. Zakładając, że urządzenie powinno mieć zastosowanie ogólne, należy obsłużyć przynajmniej możliwość zapisu do pliku jak i wyświetlanie komunikacji w czasie rzeczywistym. Wyświetlanie danych np. w konsoli nie wyczerpuje miejsca w pamięci poza tymczasowym miejscem np. na jedną wiadomość, która później jest zastępowana następną. Dzięki temu należy zwrócić uwagę na metodykę odzyskiwania danych po zakończonej części komunikacji. Przedstawione wcześniej mikrokontrolery posiadają od 32-255 KB pamięci, którą w tym celu można wykorzystać. Dokładną ilość informacji jaką możemy zapisać na takim nośniku (w minutach lub ilości wiadomości) podano w rozdziale 3. Poprzez zapisywanie danych w pamięci wewnętrznej spełniamy wymóg zapisu historii jednak istnieje jeszcze jedna możliwość, która znacznie zwiększa zakres zapisu. Możliwym jest zapis wiadomości za pomocą interfejsów I2C, które połączone do komputera personalnego np. poprzez USB, są w stanie wysyłać wiadomości na znacznie większy dysk. W tej pracy opisano jedynie dwa pierwsze podejścia, ponieważ uznano, iż są one wystarczające do praktycznego zastosowania urządzenia opartego na mikrokontrolerze, komunikującego się protokołem LIN.

Prędkość z jaką mikrokontroler może wysyłać wiadomości oraz zmieniać stan linii komunikacyjnej jest równie istotnym aspektem. Najczęściej protokoły określają pewne przedziały czasowe dla wiadomości oraz ich pojedynczych bitów. Przy zbyt wolnej reakcji procesora lub jego przeciążeniu podczas wymiany danych, mogłoby dojść do zafałszowania lub całkowitego braku wiadomości. Przez ten aspekt, należy wybrać odpowiednio sprawny model, który nie będzie miał problemu z nadążeniem za ciasnymi ograniczeniami czasowymi.

Ostatnim wymaganiem jakie ma spełniać takie urządzenie to wygoda użytkownika. Nie jest to wymaganie krytyczne do funkcjonowania, jednak poprawia praktyczność takiego urządzenia. Pod tym hasłem można wyznaczyć takie aspekty jak

łatwość w uruchomieniu programu na urządzeniu lub nieskomplikowany proces modyfikacji programu. Samo środowisko wymagane do obsługi urządzenia poza dokumentacją i przykładami powinno zawierać przejrzystą strukturę oraz prosty proces instalacji. Dodatkowym ułatwieniem jest wykorzystanie popularnych technologii, ponieważ istnieje wtedy większa szansa na szybsze przyswojenie nowych funkcjonalności, jeśli np. język programowania nie będzie obcym elementem. Ostatnim dodatkiem jest możliwość zautomatyzowania procesu aktualizacji programu oraz jego współpracy z innymi elementami jak zasilacze, które mogą być sterowane zdalnie na bazie skryptu. Przykładowo urządzenie powinno mieć możliwość rozpoczęcia działania za pomocą polecenia, które zostanie umieszczone w innym skrypcie, aby zsynchronizować się z innymi komponentami.

Biorąc pod uwagę powyższe wymagania możemy wykluczyć dwie płytki oparte na mikrokontrolerach:

- Arduino Uno, ponieważ
 - 32 KB pamięci nie są wystarczającą ilością pamięci na rejestrację historii komunikacji w postaci pliku o praktycznej zawartości,
 - Ogólne osiągi są znacznie gorsze, za wyższą cenę od pozostałych propozycji,
 - Duży rozmiar płytki, może przeszkadzać w niektórych zastosowaniach,
- STM32 NUCLEO, ponieważ
 - Pomimo dobrych specyfikacji, jest to platforma z cięższą do opanowania obsługą,
 - Język C oraz osobne oprogramowanie STM32CubeIDE może utrudnić synchronizację z innymi urządzeniami używanymi po równolegle z płytką (np. zasilacze zaprogramowane w języku Python),
 - Skąpa ilość informacji na temat pisania własnego protokołu komunikacyjnego.

Płytką na jaką opisano oprogramowanie została Raspberry Pi Pico z mikrokontrolerem RP2040. Główne przychylne argumenty to:

- Programowalne piny (PIO) pozwalające na dokładne określenie zmian na linii, a co za tym idzie, możliwość napisania dowolnego protokołu od podstaw,
- Cena, która jest znacznie niższa od pozostałych propozycji,
- Gotowe przykłady innych protokołów jak np. UART, dostępnych na platformie GitHub, podatnych na modyfikację,
- Prostota obsługi, wiele możliwych sposobów na metodykę pracy z płytką,
- Wysokopoziomowy język Python, dobrze udokumentowane biblioteki oraz wiele wpisów na forach wspomagające pracę z urządzeniem.

Rozdział 2. Oprogramowanie

2.1. Przygotowanie środowiska programistycznego

W każdym urządzeniu równie ważnym elementem poza warstwą elektryczną jest oprogramowanie. Dzięki niemu można określić bardzo precyzyjne instrukcje, które zostaną wykonane przez zaprogramowaną płytkę. Jednak, aby można było wgrać program dla mikrokontrolera, należy przygotować środowisko programistyczne oraz przeanalizować potencjalne pomocnicze narzędzia. W poniższym rozdziale opisano cały proces implementacji protokołu LIN na Raspberry Pi Pico.

2.1.1. Wybór języka programowania

Przyjmując Raspberry Pi Pico jako podstawę narzędzia do komunikacji protokołem LIN, istnieją dwa możliwe języki programowania do wyboru. Pierwszym z nich jest język C, który:

- jest kompilowanym językiem przez co zyskuje na prędkości wykonywanych instrukcji w trakcie egzekucji programu,
- proces konfiguracji środowiska oraz proces flashowania urządzenia jest skomplikowany i zajmuje znacznie dłużej od środowiska na języku Python,
- zasób dokumentacji, wpisów na forach internetowych oraz przykładowych programów jest porównywalny do tych dla języka Python,
- przez niską warstwę abstrakcji i brak automatycznego zarządzania pamięcią jest to język trudniejszy do opanowania oraz prototypowanie oprogramowania również zajmuje więcej czasu.

Tworzenie oprogramowania w języku C spełnia wszystkie wymagania do zaprogramowania Raspberry Pi Pico jako urządzenia posługującego się protokołem LIN. Jednak ze względu na trudności związane z procesem pisania kodu, które znaczne wydłużyłyby czas produkcji, wybrano język Python. Dodatkowymi zaletami pracy w tym języku są:

- szybszy proces konfiguracji Raspberry Pi Pico oraz prosty proces flashowania (za pomocą narzędzi polecanych przez autorów - ThonnyIDE),

- pomimo bycia językiem interpretowanym, co sprawia, że jest wolniejszy w wykonywaniu instrukcji, proces prototypowania jest znacznie ułatwiony,
- wysoko-poziomowość języka również przyspiesza proces pisania oprogramowania,
- posiada równie bogate zasoby dokumentacji i wsparcia na forach co język C
- język Python ułatwia synchronizację urządzenia z innymi peryferiami takimi jak np. zasilacze, które również korzystają z tego języka w skryptach np. do ich zdalnego uruchamiania,
- istnieją gotowe rozwiązania służące do szerszej automatyzacji flashowania takie jak „rshell”, które pozwala na wywoływanie dowolnego programu w dowolnym skrypcie Python za pomocą poleceń batch.

Językiem, w którym ostatecznie zostało napisane oprogramowanie został Python. Na czas pisania programu łatwość implementacji i skrócony czas realizacji programu były przeważającymi atrybutami nad językiem C. Dodatkowym atutem była praktyczność, gdyż urządzenie wykorzystano w projekcie narzędzi do testowania urządzeń i cały system również został zaprojektowany w Pythonie, przez co integracja nie stanowiła większego problemu.

Do flashowania próbki wykorzystano ThonnyIDE oraz wyprodukowanym przez firmę IDEA, PyCharm Community Edition ze wsparciem Rshell. Narzędzie ThonnyIDE pozwala na bardzo uproszczony proces wgrywania kodu na płytkę. Jednak jego prostota wiąże się z ograniczeniami, a zostają one zauważalne, gdy zaistnieje potrzeba importowania dodatkowych bibliotek, podział głównego programu na więcej niż 1 plik oraz automatyzację lub synchronizację z innymi skryptami.

Dlatego też, został on wykorzystany w fazie prototypowania, a końcowa wersja opiera się na oprogramowaniu od firmy IDEA. PyCharm nie jest narzędziem przystosowanym do pracy z Raspberry Pi Pico i nie zna bibliotek przewidzianych dla RP2040, jednak uzupełnia ten brak wieloma udogodnieniami takimi jak: wbudowana kontrola wersji, menager bibliotek języka Python, narzędzia wspomagające pracę z kodem pod kątem przeszukiwania czy refaktoryzacji kodu. Pozwala on również na pracę ze skryptami napisanymi w języku Batch, co jest bardzo przydatne do automatyzacji oraz synchronizacji ze skryptami zewnętrznymi.

2.1.2. Wybór bibliotek

Kolejnym krytycznym elementem całego urządzenia są dobrze dobrane biblioteki, które wzbogacają jego funkcjonalność. Poza podstawowymi modułami zawartymi w Micropython wykorzystano kilka pomocniczych bibliotek do automatyzacji lub poszerzenia możliwości opisywanego urządzenia.

Sama podstawa, którą jest Micropython zawiera min. moduły:

- Machine - odpowiada za obsługę pinów, przygotowanych protokołów, czy też Timery,
- rp2 – służy do obsługi programowalnych wejść/wyjść, jest to najważniejsza biblioteka za pomocą, której można zadeklarować własny protokół szeregowy,
- _thread – wspomaga wykorzystanie drugiego wątku procesora zaimplementowanego w RP2040,
- time, utime – biblioteki służące do wywoływania przerw na określony czas,
- gc – służy do czyszczenia pamięci, pomimo automatycznego pozbywania się zbędnych danych w języku Python, w niektórych momentach, wykorzystanie tego modułu usprawnia działanie programu,
- sys – zawiera funkcje systemowe RP2040, został wykorzystany do przerywania pracy programu, gdy zostanie wykryty krytyczny błąd.

Dodatkowe biblioteki, które wspomagają pracę z urządzeniem z poziomu komputera to:

- os – zarządza ścieżkami do plików wykonywalnych lub konfiguracyjnych,
- subprocess – moduł pozwalający na uruchomienie CMD z poziomu skryptu Python, umożliwia automatyzację flashowania płytki.

Ostatnim zewnętrznym elementem z którego korzysta oprogramowanie implementujące protokół LIN na urządzeniu opartym o mikrokontroler jest „rshell”. Jest to narzędzie służące do zdalnego połączenia się z powłoką systemową RP2040, na który wgrano Micropython. Dodatkowo daje możliwość przenoszenia plików z i na pamięć flash Raspberry Pi Pico co zostało wykorzystane do automatyzacji flashowania programów oraz odzyskiwania historii komunikacji.

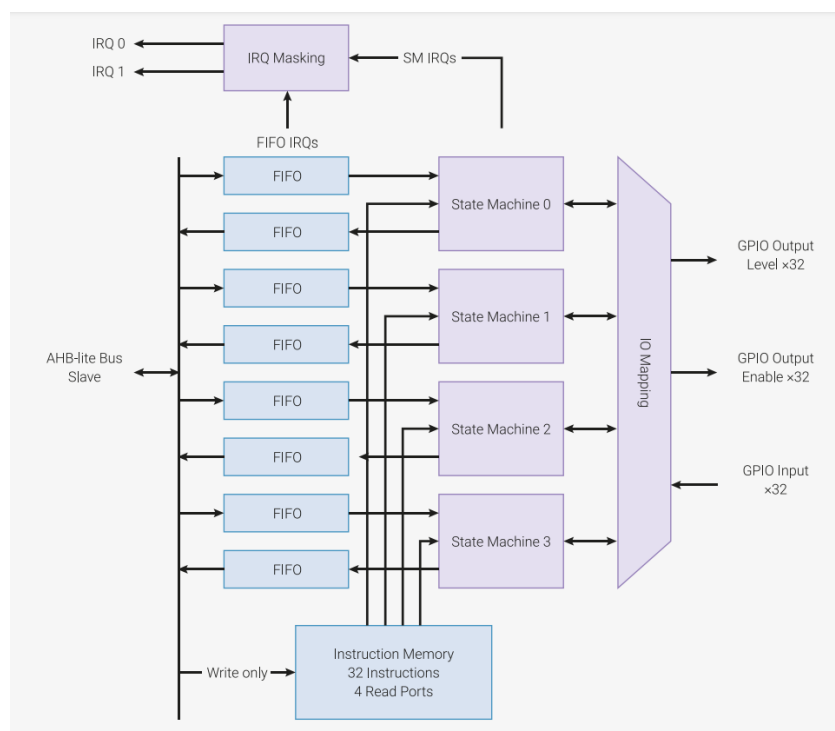
2.1. Implementacja protokołu LIN

Najważniejszym etapem całego projektu, a zarazem najtrudniejszym jest tworzenie oprogramowania. W poniższym rozdziale przedstawiono funkcjonujący kod w najnowszej wersji oprogramowania implementującego protokół LIN na Raspberry Pi Pico oraz proces jego tworzenia.

2.1.1. Obsługa PIO

Najważniejszą częścią całej funkcjonalności są programowalne wejścia/wyjścia. Odpowiadają za instrukcje dotyczące aktywności na linii LIN. Dzięki nim można bardzo dokładnie zadeklarować przedziały czasowe dla zmian stanu linii lub też za zacytywanie wszystkich zdarzeń.

Aby rozpocząć pracę nad programowaniem PIO, należy zrozumieć ich działanie. W RP2040 są 2 identyczne bloki PIO składające się z 4 maszyn stanów na każdy blok.

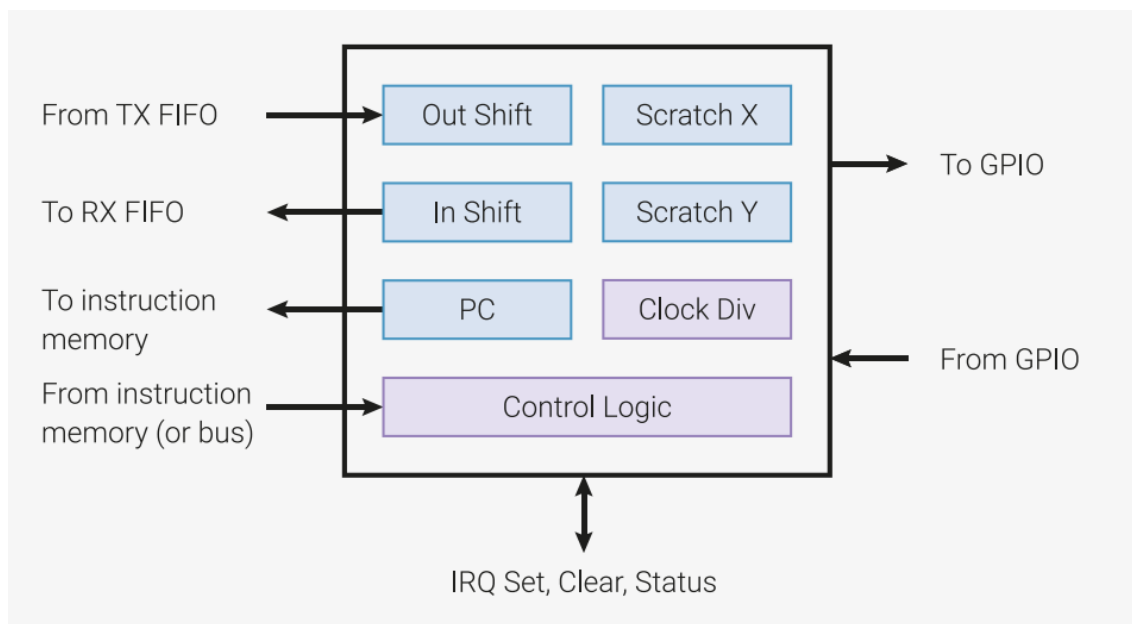


Rys. 11. Schemat blokowy PIO [8].

Na powyższym rysunku przedstawiono schemat blokowy PIO. Maszyny stanów wykonują instrukcje znajdujące się w pamięci dzielonej, która przechowuje do 30 instrukcji [8]. Mogą obsługiwać aż do 30 pinów we/wy. Każda z nich jest wyposażona w:

- 32-bitowe rejestry przesuwne ISR i OSR,

- 32-bitowe rejestry tymczasowego X i Y,
- 4x32-bitowe busy FIFIO w każdym kierunku (TX/RX) z możliwością przeprogramowania na 8x32 w jednym kierunku,
- Elastyczne mapowanie GPIO,
- Dynamiczny dostęp do pamięci,
- Flagi IRQ.



Rys. 12. Schemat blokowy maszyny stanów [8].

Na powyższej ilustracji możemy zauważyć kierunek przepływu danych w maszynie stanów. Pozwala ona na dokładniejsze zrozumienie zastosowania każdego elementu. „Out Shift” (OSR) to rejestr przesuwany z którego dane są wysyłane na wyjście GPIO, „In Shift” (ISR) odpowiada za transfer danych z powrotem do programu. „Scratch X” i „Scratch Y” to tymczasowe rejestry przechowujące zmienne, które są np. iteratorem pętli. „Clock Div” odpowiada za kontrolowanie długości cykli, które mogą się wykonywać z częstotliwością od do 133Mhz. „Control Logic” to pamięć, w której przechowywane są instrukcje dla danej maszyny. Dodatkowo Flagi IRQ to element służący do tworzenia przerw, które wspomagają synchronizację pracy maszyn stanów. Z kolei elastyczne mapowanie GPIO pozwala na zadeklarowanie pina jako wejście, wyjście lub sygnał stały (1 lub 0) oraz jako „Sideset”, co pozwala zmianę stanu pina podczas wykonywania innej operacji.

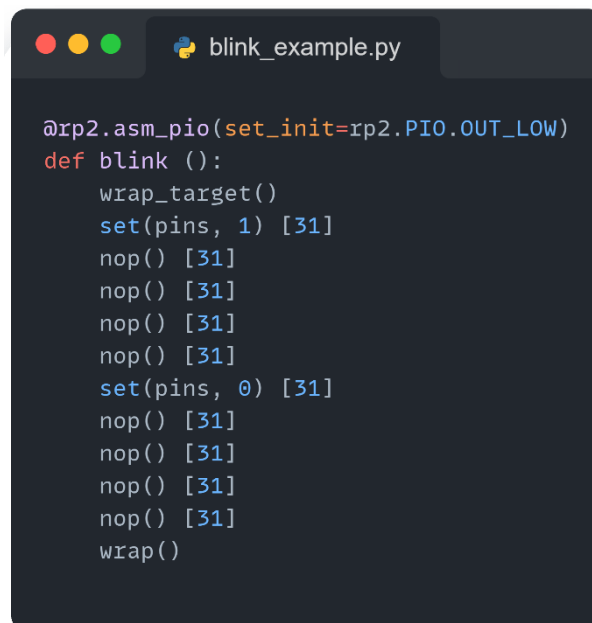
Ze względu na mały zasób pamięci na instrukcję, maszyny stanów mają dedykowany język, który przypomina język Assembler i zawiera 9 instrukcji:

- JMP – przeskakuje do wskazanego adresu w instrukcjach, jeśli podane stwierdzenie jest prawdziwe,
- WAIT – oczekuje na zadeklarowany stan na wskazanym pinie lub fladze,
- IN – przesuwa wskazaną ilość bitów ze źródła (np. z tymczasowego rejestru) do ISR,
- OUT – przesuwa wskazaną ilość bitów z OSR do odbiorcy (np. do tymczasowego rejestru),
- PUSH – wypycha całą zawartość ISR do RX FIFO, po wykonaniu wszystkie 32-bity ISR zostają zapełnione zerami,
- PULL – ładuje OSR danymi z TX FIFO w postaci 32-bitowego słowa,
- MOV – przenosi dane ze źródła do odbiorcy (np. z rejestru tymczasowego X do OSR),
- IRQ – ustawia lub czyści stan wskazanej flagi,
- SET – ustawia wskazaną wartość na odbiorcy (np. Zmienia wartość rejestru tymczasowego Y na wartość 5).

Wszystkie instrukcje mają możliwość opóźnienia przed przejściem do kolejnej linijki o 31 cykli lub w przypadku, gdy korzystamy z funkcjonalności „Sideset” czas opóźnienia jednej instrukcji skraca się do 7 cykli.

Dodatkowo istnieje możliwość „zawijania” linijek instrukcji. Jest to uproszczona funkcjonalność polecenia JMP. Gdy program napotka instrukcję „WRAP”, to przeskakuje do miejsca w pamięci, gdzie zostało zadeklarowane „WRAP_TARGET” i kontynuuje wykonywanie kolejnych instrukcji rozpoczynając z tamtego adresu.

Implementacja kliku z tych funkcjonalności w Micropython została zaprezentowana na przykładzie mrugania diodą led na poniższym rysunku:



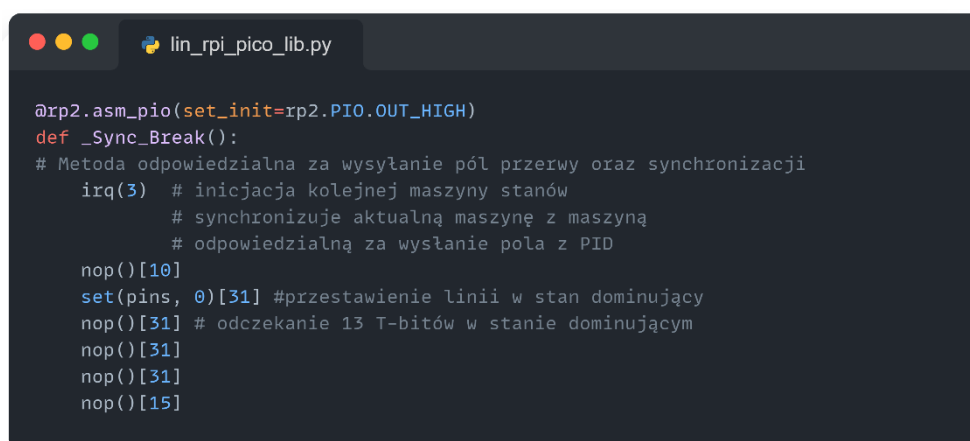
```
blink_example.py

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink():
    wrap_target()
    set(pins, 1) [31]
    nop() [31]
    nop() [31]
    nop() [31]
    nop() [31]
    set(pins, 0) [31]
    nop() [31]
    nop() [31]
    nop() [31]
    nop() [31]
    wrap()
```

Rys. 13. Ilustracja przykładowego programu dla maszyny stanów.

Pierwsza linijka kodu pozwala zrozumieć interpreterowi, aby traktować tę metodę jako polecenia kierowane wyłącznie dla maszyn stanów oraz deklaruje początkowy stan wskazanego pinu równy 0. Można również zauważyć wykorzystanie poleceń „wrap”, „set” oraz opóźnień, które są deklarowane przy poleceniu za pomocą kwadratowych nawiasów. Operacja „nop ()” to operacja przeniesienia bitu między tymczasowymi rejestrami X i Y, która zwyczajnie służy jako opóźnienie.

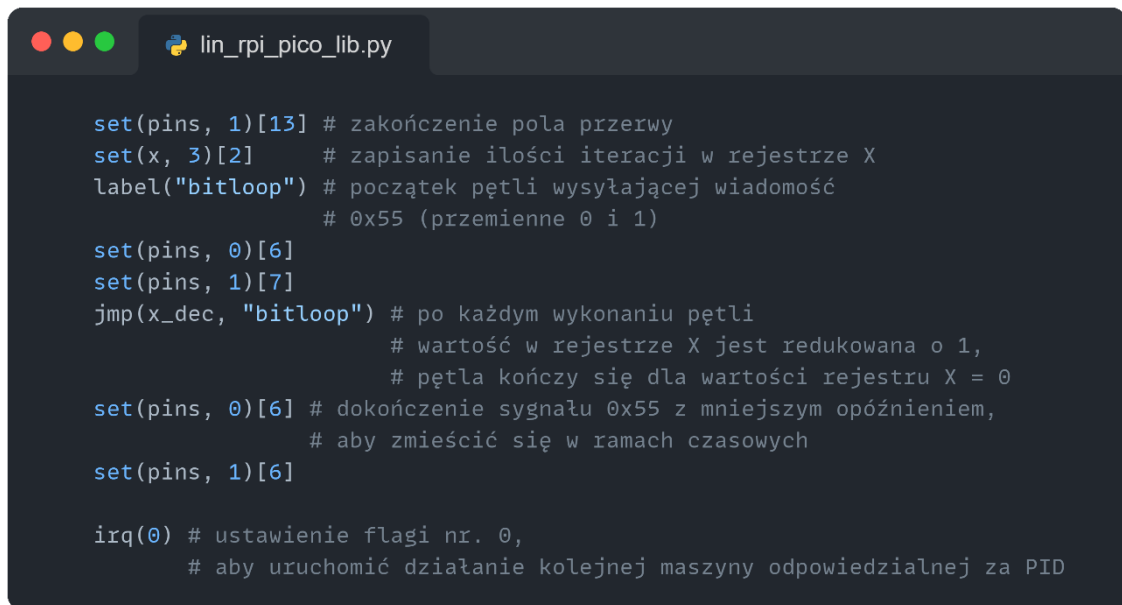
Praktyczne wykorzystanie PIO do symulacji protokołu LIN wymaga bardzo precyzyjnych instrukcji dla maszyn stanów, aby zachować kluczowe zasady protokołu. Jeśli czas zmian na linii odchodzi od standardowych wartości na więcej niż 20%, to zwiększa się możliwość destabilizacji lub całkowitej utraty komunikacji.



```
lin_rpi_pico_lib.py

@rp2.asm_pio(set_init=rp2.PIO.OUT_HIGH)
def _Sync_Break():
    # Metoda odpowiedzialna za wysyłanie pól przerwy oraz synchronizacji
    irq(3) # inicjacja kolejnej maszyny stanów
           # synchronizuje aktualną maszynę z maszyną
           # odpowiedzialną za wysłanie pola z PID
    nop()[10]
    set(pins, 0)[31] #przestawienie linii w stan dominujący
    nop()[31] # odczekanie 13 T-bitów w stanie dominującym
    nop()[31]
    nop()[31]
    nop()[15]
```

Rys. 14. Metoda opisująca instrukcje pola przerwy oraz synchronizacji dla PIO cz.1



```
set(pins, 1)[13] # zakończenie pola przerwy
set(x, 3)[2]      # zapisanie ilości iteracji w rejestrze X
label("bitloop") # początek pętli wysyłającej wiadomość
                  # 0x55 (przemienne 0 i 1)

set(pins, 0)[6]
set(pins, 1)[7]
jmp(x_dec, "bitloop") # po każdym wykonaniu pętli
                      # wartość w rejestrze X jest redukowana o 1,
                      # pętla kończy się dla wartości rejestru X = 0
set(pins, 0)[6] # dokończenie sygnału 0x55 z mniejszym opóźnieniem,
                # aby zmieścić się w ramach czasowych
set(pins, 1)[6]

irq(0) # ustawienie flagi nr. 0,
        # aby uruchomić działanie kolejnej maszyny odpowiedzialnej za PID
```

Rys. 15. Metoda opisująca instrukcje pola przerwy oraz synchronizacji dla PIO cz.2

Powyżej (Rys. 15) podano instrukcje, które zostają umieszczone w pamięci dzielonej maszyn. „Sync_Break” oraz „Data_Pid” są metodami wgranymi do pamięci, ze względu na ograniczenie do 30 instrukcji. Proces przejścia z wysyłania pola synchronizacji do danych, jest na tyle sprawny, iż nie przeszkadza w poprawnej komunikacji, jednak nie opiera się on tylko na zwykłej zmianie maszyny wykonującej polecenia. Jedną z sztuczek zapobiegającym opóźnieniom w tym przypadku jest przedwczesne uruchomienie kolejnych maszyn oraz niepełne wysłanie ostatniego bitu. Linia komunikacji pozostaje w stanie recesywnym po zakończeniu pola synchronizacji, przez co można przerwać pracę pierwszej maszyny wcześniej, aby zostawić więcej czasu maszynie odpowiedzialnej za kolejne dane.

Na poniższym rysunku (Rys. 16) przedstawiono działanie maszyny odpowiedzialnej za wysyłanie pól PID, danych oraz sumy kontrolnej. Do rejestrów, które przechowują wartości tych danych, dostarczany jest ciąg liczb przekonwertowanych na pojedyncze bity. Wszystkie pola zostają ze sobą scalone, ponieważ usprawnia to transfer i zapobiega opóźnieniom.

```
lin_rpi_pico_lib.py

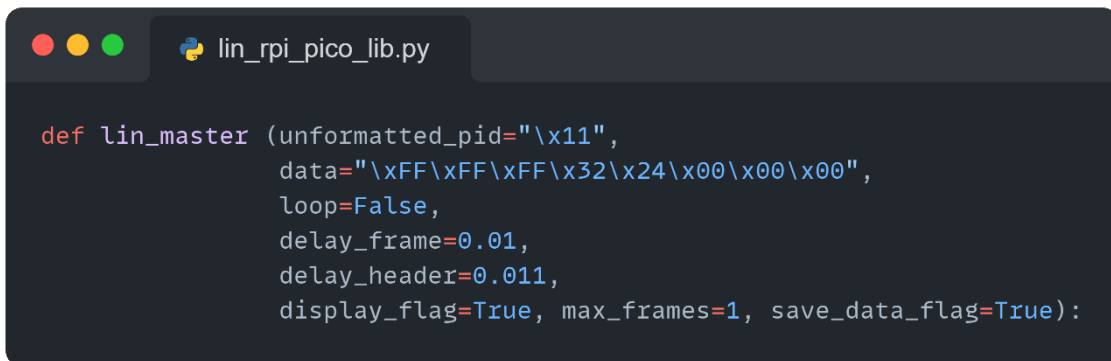
@rp2.asm_pio(sideset_init=PIO.OUT_HIGH,
             out_init=PIO.OUT_HIGH,
             out_shift_dir=PIO.SHIFT_RIGHT)
def _Data_Pid():
    # Metoda odpowiedzialna za wysyłanie pól PID, danych oraz sumy kontrolnej
    pull() # zaciągnięcie danych do wysłania do OSR
    set(x, 7).side(0)[7] # przygotowanie pętli do wysyłania kolejnych słów
    label("bitloop")
    out(pins, 1)[6] # wypchnięcie danych z OSR na pin
    jmp(x_dec, "bitloop")
    nop().side(1)[6]
    irq(2) # poinformowanie o zakończeniu wysyłania wskazanej wiadomości
```

Rys. 16. Metoda opisująca instrukcje pola identyfikatora oraz danych dla PIO

2.1.2. Proces deklaracji, przetwarzania i wysyłania danych

Aby skorzystać z powyżej wymienionych funkcji PIO, należy w odpowiedni sposób przekazać im wymagane dane. Główną metodą, która odpowiada za podstawową komunikację jest „lin_master”. Dzięki niej można odczytywać aktywność na linii oraz wysyłać wiadomości z Raspberry Pi Pico protokołem LIN jako urządzenie typu master. Jest to metoda przystosowana na modyfikację i można ją dostosować za pomocą następujących argumentów:

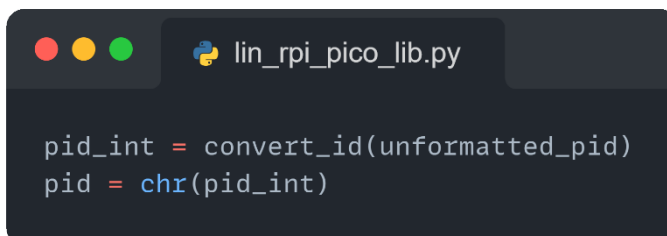
- `unformatted_pid` (string) – wartość bajtu identyfikatora wiadomości w postaci heksadecymalnej,
- `data` (string) – ciąg bajtów danych wysyłanych w wiadomości w postaci heksadecymalnej,
- `loop` – określa, czy wskazana ramka powinna być wysyłana w nieskończonej pętli,
- `delay_frame` – odstęp czasowy pomiędzy ramkami (wymagany `loop = True`),
- `delay_header` – odstęp czasowy po wysłaniu zapytania (aplikowany podczas wysyłania samego identyfikatora bez danych),
- `display_flag`, `max_fames`, `save_data_flag` – argumenty odnoszące się do zapisu danych, decydują kolejno o wyświetlaniu danych, ilości pobieranych ramek przed ich wyświetleniem, zapisem do pliku.



```
def lin_master (unformatted_pid="\x11",
                data="\xFF\xFF\xFF\x32\x24\x00\x00\x00",
                loop=False,
                delay_frame=0.01,
                delay_header=0.011,
                display_flag=True, max_frames=1, save_data_flag=True):
```

Rys. 18. Główna metoda programu obsługująca proces komunikacji protokołem LIN oraz przyjmowane przez nią argumenty.

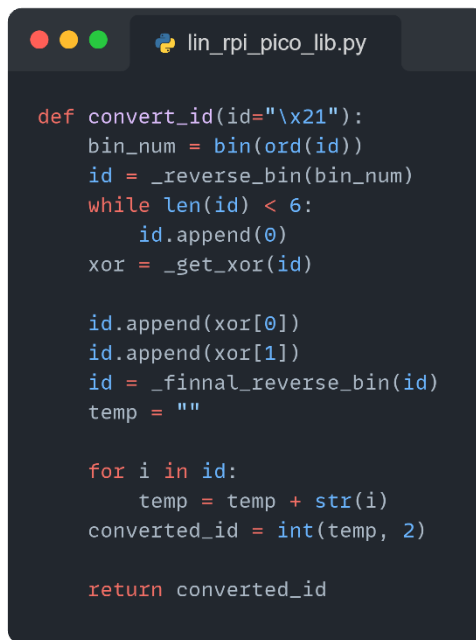
W funkcji „lin_master” (Rys. 18), pierwszym etapem jest przygotowanie danych. Należy pamiętać, iż wartość PID jest wzbogacana o bity parzystości, a ramka kończy się sumą kontrolną. Obliczenia te są wykonywane automatycznie przez program, aby aplikacja była bardziej praktyczna. Obsługują to: metoda „convert_id” oraz wbudowana funkcja „chr”. Druga z nich jedynie przetwarza liczbę z typu całkowitego na typ char. Jest to wymagana operacja, gdyż maszyny stanów przyjmują znaki i tak też są sformułowane argumenty głównej metody. Jednak przez modyfikacje jakich należy dokonać, początkowo znak PID zamieniany jest na liczbę binarną, a następnie całkowitą dla ułatwienia obliczeń.



```
pid_int = convert_id(unformatted_pid)
pid = chr(pid_int)
```

Rys. 17. Fragment kodu przedstawiający funkcje odpowiadające za konwersję danych.

Funkcja „convert_id” (Rys. 17 i Rys. 19) odwzorowuje wskazane przez normę ISO obliczenia. W tym celu rozpoczyna od konwersji zwykłego identyfikatora na typ binarny, następnie przeprowadza operację XOR. Na koniec zwraca wartość PID w postaci liczby całkowitej. Konwersja na znak char celowo nie została umieszczona w ciele poniższej funkcji, ponieważ jest często wykorzystywana do ręcznego sprawdzania wartości PID, a wynik w postaci liczby całkowitej jest znacznie czytelniejszy od znaku.



```
def convert_id(id="\x21"):
    bin_num = bin(ord(id))
    id = _reverse_bin(bin_num)
    while len(id) < 6:
        id.append(0)
    xor = _get_xor(id)

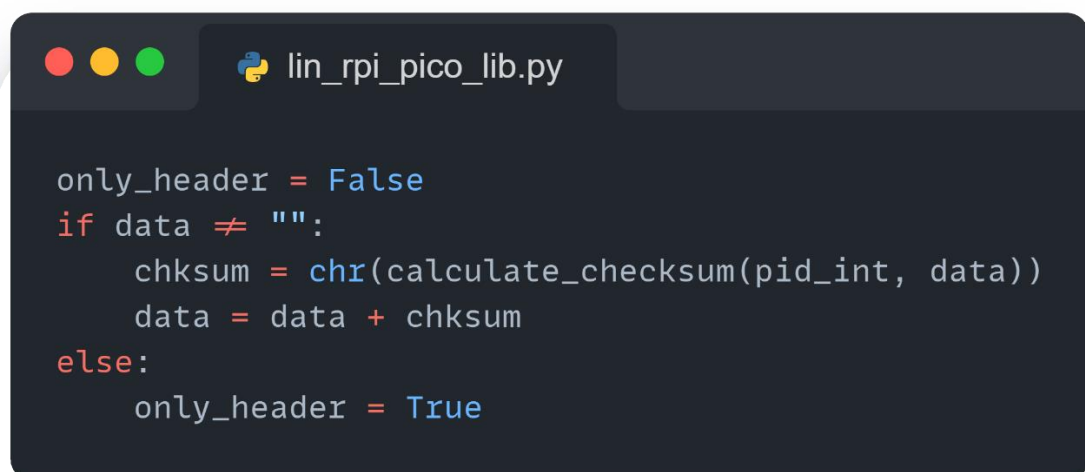
    id.append(xor[0])
    id.append(xor[1])
    id = _final_reverse_bin(id)
    temp = ""

    for i in id:
        temp = temp + str(i)
    converted_id = int(temp, 2)

    return converted_id
```

Rys. 19. Ciało metody "convert_id".

Kolejnym etapem metody „lin_master”, jest sprawdzenie, czy wiadomością jest zapytanie czy polecenie (Rys. 20). Zapytanie można łatwo rozróżnić poprzez sprawdzenie czy wiadomość zawiera jakiejkolwiek wartości w polu danych pole danych. Jeśli zawiera – jest to polecenie, jeśli nie – zapytanie. Na tej podstawie jest ustawiana zmienna typu boolean. Dzięki niej możemy przewidzieć, jakie będą maszyny stanów należy aktywować.



```
only_header = False
if data != "":
    checksum = chr(calculate_checksum(pid_int, data))
    data = data + checksum
else:
    only_header = True
```

Rys. 20. Fragment kodu sprawdzający typ nadawanej wiadomości.

Następnie następuje inicjalizacja maszyn stanów odpowiedzialnych za wysyłanie wiadomości. Korzystając z biblioteki rp2 można zainicjalizować maszyny w sposób zaprezentowany na rysunku umieszczonym poniżej.

```
lin_rpi_pico_lib.py

while True:

    sm = rp2.StateMachine(
        0, _Sync_Break, freq=8 * BAUD_RATE,
        set_base=Pin(PIO_TX_PIN),
        sideset_base=Pin(PIO_TX_PIN),
        out_base=Pin(10))

    sm1 = rp2.StateMachine(
        1, _Data_Pid, freq=8 * BAUD_RATE,
        set_base=Pin(PIO_TX_PIN),
        sideset_base=Pin(PIO_TX_PIN),
        out_base=Pin(PIO_TX_PIN),
        jmp_pin=Pin(PIO_TX_PIN))

    sm2 = rp2.StateMachine(
        2, _Data_Pid, freq=8 * BAUD_RATE,
        set_base=Pin(PIO_TX_PIN),
        sideset_base=Pin(PIO_TX_PIN),
        out_base=Pin(PIO_TX_PIN),
        jmp_pin=Pin(PIO_TX_PIN))


    sm3 = rp2.StateMachine(
        3, _Data_Pid, freq=8 * BAUD_RATE,
        set_base=Pin(PIO_TX_PIN),
        sideset_base=Pin(PIO_TX_PIN),
        out_base=Pin(PIO_TX_PIN),
        jmp_pin=Pin(PIO_TX_PIN))
```

Rys. 21. Fragment kodu przedstawiający inicjalizację maszyny stanów w ciele głównej metody.

Argumenty klasy „StateMachine” wykorzystane do inicjalizacji to:

- id – wartość od 0 do 7 określająca jedną z 4 dostępnych maszyn na każde PIO,
- program – zestaw instrukcji jakimi ma się kierować dana maszyna,
- freq – odnosi się do częstotliwości cykli zegara z jaką maszyna ma pracować,
- set_base, sideset_base, out_base – zmienna odnosząca się do GPIO, na których przeprowadzane będą operacje wyjścia,
- jmp_pin – zmienna odnosząca się do GPIO, na których będzie sprawdzany warunek instrukcji JMP ustawiony na weryfikowanie stanu danego pina.

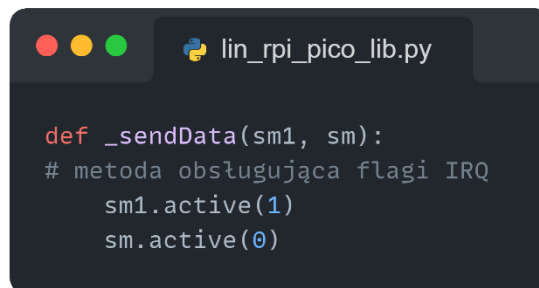
Maszyny stanów są przypisane do zmiennych, ponieważ później należy je aktywować w odpowiednim momencie, a takie rozwiązanie jest zgodne z poprawnymi praktykami programowania. Inicjalizowane są 4 maszyny, po jednej na każde pole wiadomości LIN. Pierwsza odpowiada za pole przerwy, druga za PID, trzecia za pole danych i ostatnia wysyła sumę kontrolną. Są one aktywowane zależnie od typu wiadomości za pomocą flag:

A screenshot of a code editor window titled 'lin_rpi_pico_lib.py'. The code defines an interrupt request (IRQ) handler for a state machine (sm). It uses lambda functions to call _sendData, _sm2active, and _sm3active. The code is as follows:

```
sm.irq(lambda p: _sendData(sm1, sm))
if not only_header:
    sm2.irq(lambda s: _sm2active(sm2))
else:
    sm2.irq(lambda s: _handler())
sm3.irq(lambda x: _sm3active(sm3))
```

Rys. 23. Konfiguracja flag IRQ.

Flagi IRQ przedstawione na powyższym rysunku podlegają danej maszynie stanów przez co deklarujemy ją metodą zawartą w klasie StateMachine. Wymaga ona podania serii działań np. w postaci funkcji lambda, które mają nastąpić po aktywowaniu flagi.

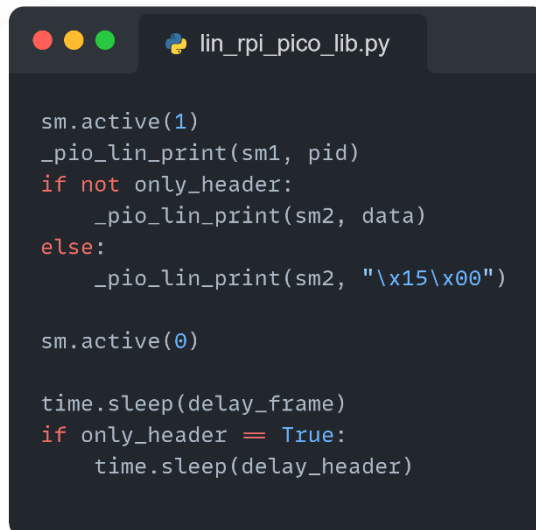
A screenshot of a code editor window titled 'lin_rpi_pico_lib.py'. The code defines a method _sendData that takes sm1 and sm as arguments. It calls sm1.active(1) and sm.active(0). The code is as follows:

```
def _sendData(sm1, sm):
    # metoda obsługująca flagi IRQ
    sm1.active(1)
    sm.active(0)
```

Rys. 22. Przykładowa metoda zarządzająca działaniem flag IRQ.

Metoda lambda dla maszyny 0 to „_sendData” (Rys. 22), która zatrzymuje jej operacje oraz aktywuje działanie kolejnej maszyny. Kolejne flagi są umieszczone w ciele instrukcji wgranych do pamięci dzielonej maszyn.

Wszystkie maszyny pracują niezależnie od głównego programu. Na poniższym rysunku (Rys. 24) przedstawiono aktywację maszyny o ID = 0, odpowiedzialną za pole przerwy oraz metodę „pio_lin_print”, która przesyła dane z drugiego przyjmowanego argumentu do TX FIFO maszyny wskazanej w pierwszym argumencie funkcji. W przypadku maszyny przypisanej do zmiennej „sm2”, której zadaniem jest wysyłanie pola danych, gdy pole danych jest puste i przesyłane jest zapytanie to podmienia się wartość tego pola losowymi danymi. Jest to działanie wymuszone ze względu na blokadę pracy maszyny w przypadku pustego rejestru OSR. Maszyna nie może być zablokowana, ponieważ odpowiada również za poprawną synchronizację pomiędzy innymi maszynami.



```
sm.active(1)
_pio_lin_print(sm1, pid)
if not only_header:
    _pio_lin_print(sm2, data)
else:
    _pio_lin_print(sm2, "\x15\x00")

sm.active(0)

time.sleep(delay_frame)
if only_header == True:
    time.sleep(delay_header)
```

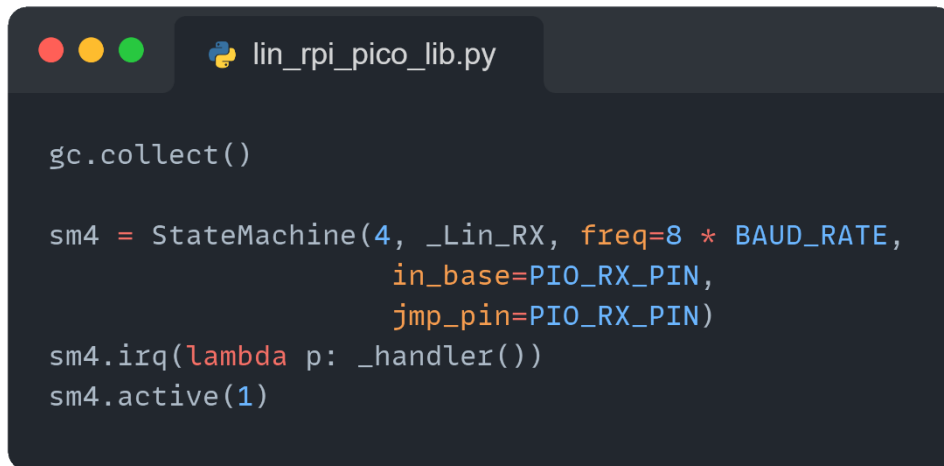
Rys. 24. Wywołanie funkcji odpowiedzialnych za odczyt danych.

Na koniec wprowadzone zostało opóźnienie po zakończeniu wysyłania ramki. W przypadku, gdy wiadomością jest zapytanie, dodawany jest większy przedział czasowy, aby odczytać całą odpowiedź.

2.1.3. Wykorzystanie drugiego wątku do odczytu danych

Przedstawiona w poprzednich rozdziałach funkcjonalność spełnia wszystkie wymagania dotyczące wysyłania wiadomości w formie protokołu LIN. Następną częścią jest omówienie fragmentów kodu, które odpowiadają za odbiór wiadomości oraz ich wyświetlanie bądź też zapis. Dane rozwiązanie prezentuje również technikę programowania współbieżnego, gdzie wykorzystywany jest drugi wątek procesora zawartego w RP2040, aby umożliwić odczyt i wysyłanie danych jednocześnie, w czasie rzeczywistym.

Niestety implementacja drugiego wątku w Micropython nie została w pełni dokończona, dlatego dodatkowo należy zadbać o sprzątanie pamięci, gdy pracują obydwa wątki. Z pomocą przychodzi biblioteka „gc” odpowiedzialna za sprzątanie niepotrzebnych danych z pamięci. W programie wykorzystano tylko funkcję „gc.collect” w miejscach, gdzie przy dużym natłoku danych, ich nadmiar powodował nieprawidłowe zachowania programu.



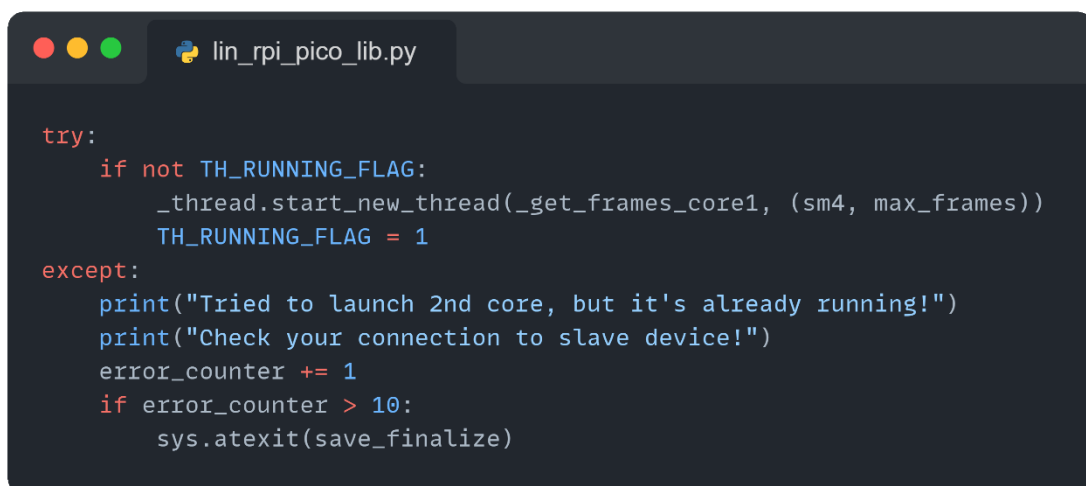
```
lin_rpi_pico_lib.py

gc.collect()

sm4 = StateMachine(4, _Lin_RX, freq=8 * BAUD_RATE,
                    in_base=PIO_RX_PIN,
                    jmp_pin=PIO_RX_PIN)
sm4.irq(lambda p: _handler())
sm4.active(1)
```

Rys. 25. Inicjalizacja maszyny stanów odpowiedzialnej za odczyt danych.

Na powyższym rysunku, poza sprzątaniem zbędnych danych, można zauważyć deklarację maszyny o ID = 4. Należy ona do drugiego PIO i jej funkcjonalność została przypisana drugiemu wątkowi. Wykorzystuje ona zbiór instrukcji „Lin_RX”, które są odpowiedzialne za odbiór danych z wskazanego pinu i przekazywania ich przez ISR do programu do dalszej manipulacji. Deklaracja flagi dla tej maszyny zawiera funkcję „handler”, która nie zawiera instrukcji jednak sama zmiana stanu flagi jest widoczna dla innych maszyn co również wspomaga synchronizację pomiędzy nimi.



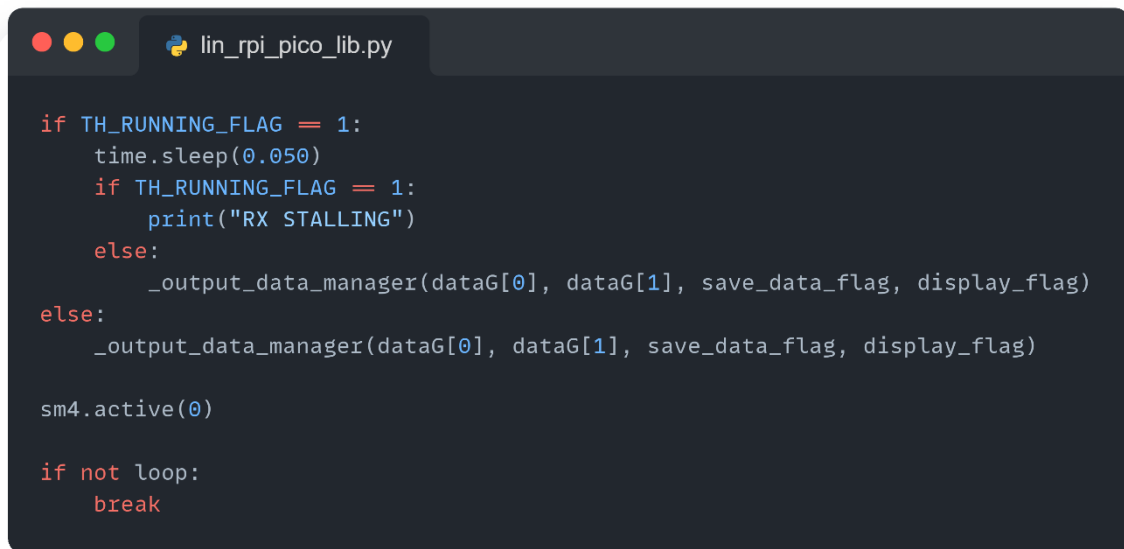
```
lin_rpi_pico_lib.py

try:
    if not TH_RUNNING_FLAG:
        _thread.start_new_thread(_get_frames_core1, (sm4, max_frames))
        TH_RUNNING_FLAG = 1
except:
    print("Tried to launch 2nd core, but it's already running!")
    print("Check your connection to slave device!")
    error_counter += 1
    if error_counter > 10:
        sys.atexit(save_finalize)
```

Rys. 26. Fragment kodu przedstawiający obsługę drugiego wątku procesora.

Na Rys. 26 pokazano wykorzystanie biblioteki „thread”, która obsługuje dodatkowy wątek. Jako argumenty przyjmuje funkcję, którą wykonuje drugi wątek oraz argumenty danej funkcji. W tym przypadku uruchamia się metoda „get_frames_core1” z argumentami wskazującymi maszynę stanów oraz maksymalną ilość zaczytywanych danych przed ich przesłaniem z ISR do programu.

Dodatkowo zaimplementowano obsługę błędów dotyczących pracy drugiego wątku. Blokują one dalsze wykonywanie kodu przez 2 wątek np. w przypadku niepołączonych linii komunikacji między Raspberry Pi Pico, a urządzeniem typu slave. Poniżej (Rys. 27) przedstawiono sposób zapobiegania utraci komunikacji podczas trwającej sesji wymiany wiadomości. Gdy następuje przerwa, pojawia się komunikat o braku aktywności na RX FIFO. Po przywróceniu połączenia, program może kontynuować swoje zadanie. Błędy są również notowane w przypadku wyświetlania informacji w konsoli lub w zapisanym pliku.



```
lin_rpi_pico_lib.py

if TH_RUNNING_FLAG == 1:
    time.sleep(0.050)
    if TH_RUNNING_FLAG == 1:
        print("RX STALLING")
    else:
        _output_data_manager(dataG[0], dataG[1], save_data_flag, display_flag)
else:
    _output_data_manager(dataG[0], dataG[1], save_data_flag, display_flag)

sm4.active(0)

if not loop:
    break
```

Rys. 27. Obsługa błędów związanych z pracą drugiego wątku procesora.

Ostatnim elementem głównego programu jest czytanie wymiany danych na wskazanej linii. Proces pobierania danych rozpoczyna metoda „get_frames_core1” (Rys. 28), która zajmuje się pobieraniem danych z ISR do globalnej zmiennej w programie. Zaciągnięcie danych wykonuje polecenie „get” z klasy StateMachine. Pobiera ona ciąg bitów z RX FIFO wskazanej maszyny. W przypadku braku danych do zaciągnięcia, blokuje dalsze wykonywanie kodu, oczekując na kolejne bity [9]. Z tego powodu ograniczamy maksymalny rozmiar pojedynczego zaciągnięcia do przesunięcia tylko 24 bitów.

```
lin_rpi_pico_lib.py

global dataG
global start
global TH_RUNNING_FLAG
current_frame = []
all_frames = []
all_frames_time = []
sleep_time = int(1200 * 9600 / BAUD_RATE)

for frame in range(max_frames):
    wait_for_end = False
    while True:
        current_frame.append(hex(sm4.get() >> 24))
        wait_for_end = True
        time.sleep_us(sleep_time)
        if (wait_for_end and sm4.rx_fifo() == 0) or len(current_frame) > 9:
            if current_frame != []:
                all_frames.append(current_frame)
                curr_time = time.ticks_diff(utime.ticks_ms(), start)
                all_frames_time.append(
                    [int(curr_time / (60000 * 60)) % 60,
                     int(curr_time / 60000) % 60,
                     int((curr_time / 1000) % 60),
                     curr_time % 1000])
                current_frame = []
            break

dataG = [all_frames, all_frames_time]
TH_RUNNING_FLAG = 0
```

Rys. 28. Instrukcje wykonywane przez drugi wątek procesora.

Kolejne instrukcje warunkowe przedstawione powyżej służą do usprawnienia procesu pobierania, zabezpieczają niepoprawnym działaniem oraz decydują o zakończeniu zaciągania danych. Gdy odebrane zostały wszystkie wymagane informacje, metoda przechodzi do etapu zapisu wartości do globalnej listy zmiennych „dataG” wraz z dodaniem znacznika czasu dla każdej ramki. Na koniec funkcji drugi wątek kończy swoje działanie co też jest komunikowane za pomocą globalnej zmiennej „TH_RUNNING_FLAG”.

Wszystkie ramki zebrane przez 2 wątek są przekazywane do głównego programu, który je wyświetla za pomocą metody „output_data_manager” (Rys.29). Zarządza on sposobem prezentowania danych oraz zabezpiecza program na wypadek przyjęcia błędnych danych.

```
lin_rpi_pico_lib.py

def _output_data_manager(data, data_time, save_data_flag=True, display_flag=True):
    try:
        if save_data_flag:
            save_data(data, data_time)
        if display_flag:
            display_data(data, data_time)
    except:
        print("Badly formatted Frame")
```

Rys. 29. Metoda zarządzająca wyświetlaniem oraz zapisywaniem historii komunikacji urządzenia.

Jako argumenty przyjmuje listę danych, osobną listę z pieczętkami czasowymi wiadomości oraz flagi określające, czy powinny zostać zapisane do pliku lub czy też powinny być wyświetlane w czasie rzeczywistym.

Metoda odpowiadająca za zapis danych do pliku przedstawiona została poniżej.

```
lin_rpi_pico_lib.py

def save_data(data, data_time):
    global frame_counter_save
    global csv_file
    comma = ""
    simple_data = ""
    for i in range(0, len(data)):
        shift_data = 10 - len(data[i])
        if len(data[i]) == 1:
            shift_data -= 1

        for x in range(shift_data):
            comma = comma + ","

        curr_data_byte = data[i][1:len(data[i]) - 1]
        for j in curr_data_byte:
            simple_data += j + ","

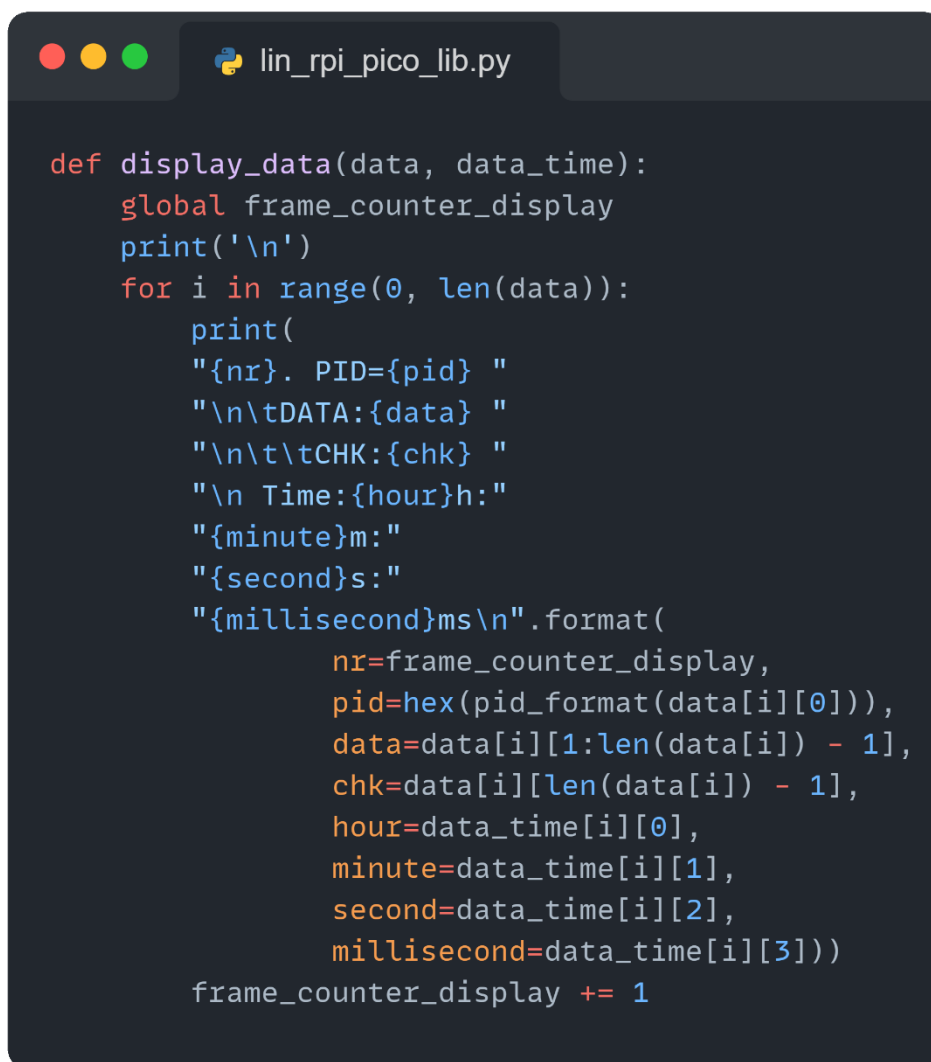
        csv_frame = f"{frame_counter_save}. ,\n" \
                    f"{{hex(pid_format(data[i][0]))}},\n" \
                    f"{{simple_data}} {{comma}}\n" \
                    f"{{data[i][len(data[i]) - 1]}},\n" \
                    f"{{data_time[i][0]}h}\n" \
                    f"{{data_time[i][1]}m}\n" \
                    f"{{data_time[i][2]}s}\n" \
                    f"{{data_time[i][3]}ms}\n"

        frame_counter_save += 1
        csv_file.write('%s' % (csv_frame))
```

Rys. 30. Metoda odpowiedzialna za zapis danych w pliku csv.

Jej funkcjonalność polega na odpowiedniej obróbce danych. Ważne informacje dotyczące komunikacji są wyciągane z globalnych list, aby przedstawić je w sposób czytelny. Dodatkowo formatuje ona dane w taki sposób, aby można było je zapisać w pliku o rozszerzeniu .csv. Końcowe polecenie tej metody to „csv_file.write” dopisuje przetworzoną ramkę do przygotowanego pliku typu csv. We wcześniej przygotowanym pliku znajdują się dodatkowe informacje dotyczące danej sesji komunikacji, takie jak czas rozpoczęcia komunikacji oraz jej zakończenia. Wszystkie operacje na plikach dotyczących historii komunikacji są zautomatyzowane i nie wymagają ingerencji użytkownika, bądź przygotowania szablonów plików.

Wyświetlaniem danych w czasie rzeczywistym zajmuje się natomiast metoda „display_data” (Rys. 31), która za pomocą zwykłego polecenia „print” wypisuje kolejno dane z listy zawierającej ramki w przystępnym układzie.



```
lin_rpi_pico_lib.py

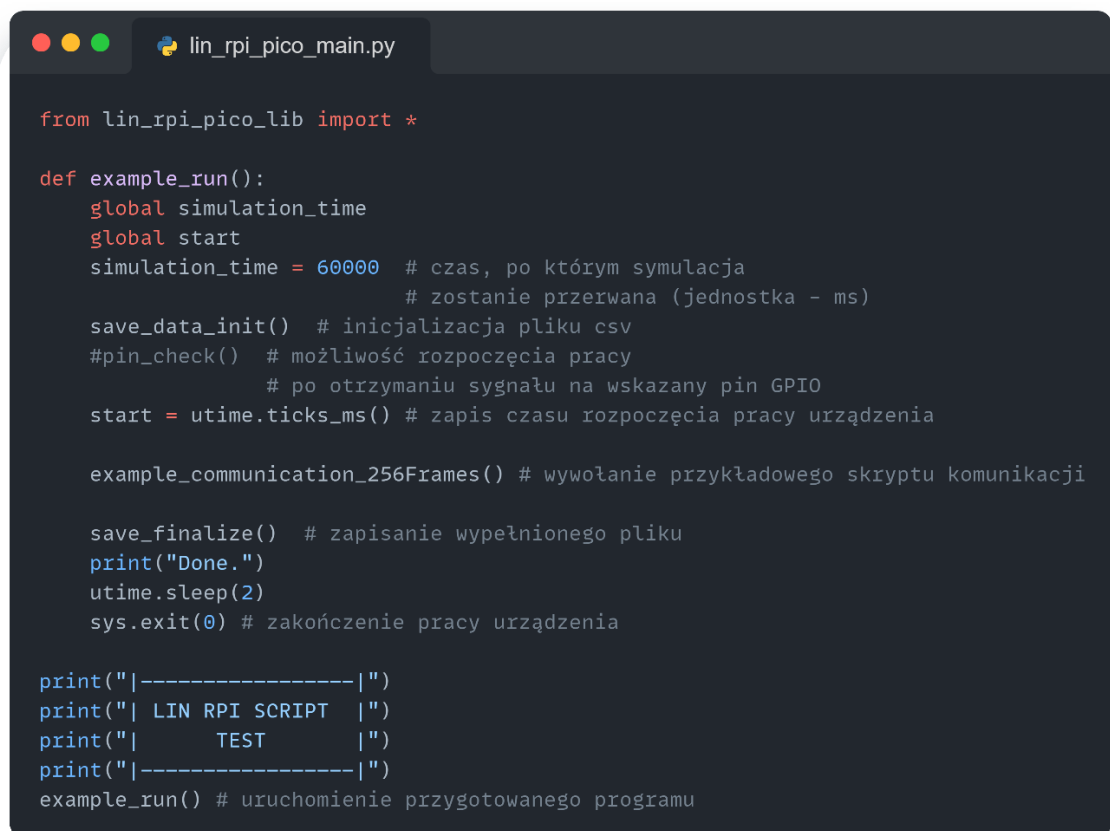
def display_data(data, data_time):
    global frame_counter_display
    print('\n')
    for i in range(0, len(data)):
        print(
            "{nr}. PID={pid} "
            "\n\tDATA:{data} "
            "\n\t\tCHK:{chk} "
            "\n Time:{hour}h:"
            "{minute}m:"
            "{second}s:"
            "{millisecond}ms\n".format(
                nr=frame_counter_display,
                pid=hex(pid_format(data[i][0])),
                data=data[i][1:len(data[i]) - 1],
                chk=data[i][len(data[i]) - 1],
                hour=data_time[i][0],
                minute=data_time[i][1],
                second=data_time[i][2],
                millisecond=data_time[i][3])
        frame_counter_display += 1
```

Rys. 31. Metoda odpowiedzialna za wyświetlanie komunikacji urządzenia w czasie rzeczywistym.

2.2. Implementacja interfejsu użytkownika i automatyzacja flashowania

Do przedstawionej w poprzednim podrozdziale biblioteki utworzono również przykładowy skrypt konfiguracyjny do zawartych w niej funkcji oraz skrypt Batch wspomagający wgrywanie całego oprogramowania na Raspberry Pi Pico. Biblioteka pomimo możliwości działania samodzielnie, została podzielona na plik z funkcjami oraz na plik konfiguracyjny/startowy (Rys. 32) ze względu na łatwość i praktyczność jej wykorzystania. Podczas użytkowania narzędzia symulującego protokół LIN często przychodzi potrzeba zmiany informacji wysyłanych na linii. Dostępne możliwości konfiguracji programu zawierają:

- Zmianę czasu trwania programu
- Zmianę sposobu rozpoczęcia pracy – z opóźnieniem czasowym lub poprzez wykrycie aktywności na wskazanym pinie GPIO (przydatne, gdy płytką jest zsynchronizowana z innymi urządzeniami za pomocą przekaźników)
- Deklarację i inicjalizację zapisu historii komunikacji do plików
- Wybór przygotowanego skryptu z biblioteki

A screenshot of a code editor window titled 'lin_rpi_pico_main.py'. The code is written in Python and defines a function 'example_run()' which sets up a simulation, initializes data, checks for a signal on a GPIO pin, and runs a communication script. It also includes a main block that prints a header and calls 'example_run()'.

```
from lin_rpi_pico_lib import *

def example_run():
    global simulation_time
    global start
    simulation_time = 60000 # czas, po którym symulacja
                           # zostanie przerwana (jednostka - ms)
    save_data_init() # inicjalizacja pliku csv
    #pin_check() # możliwość rozpoczęcia pracy
                  # po otrzymaniu sygnału na wskazany pin GPIO
    start = utime.ticks_ms() # zapis czasu rozpoczęcia pracy urządzenia

    example_communication_256Frames() # wywołanie przykładowego skryptu komunikacji

    save_finalize() # zapisanie wypełnionego pliku
    print("Done.")
    utime.sleep(2)
    sys.exit(0) # zakończenie pracy urządzenia

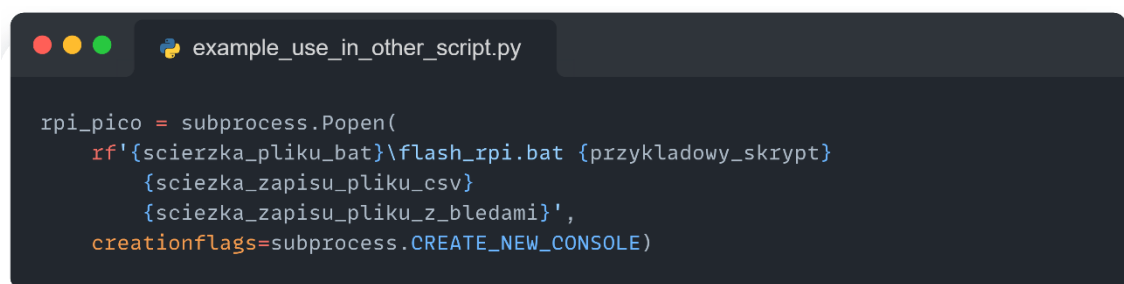
print("|-----|")
print("| LIN RPI SCRIPT |")
print("| TEST |")
print("|-----|")
example_run() # uruchomienie przygotowanego programu
```

Rys. 32. Przykładowy program wykorzystujący bibliotekę implementującą protokół LIN dla Raspberry Pi Pico.

Umieszczenie programu w pamięci Raspberry Pi Pico oraz odzyskiwanie plików obsługuje skrypt napisany w języku Batch utylizujący narzędzie Rshell. Przyjmuje on następujące argumenty:

- skrypt konfiguracyjny
- ścieżkę do miejsca zapisu pliku historii komunikacji
- ścieżkę do miejsca zapisu pliku z błędami

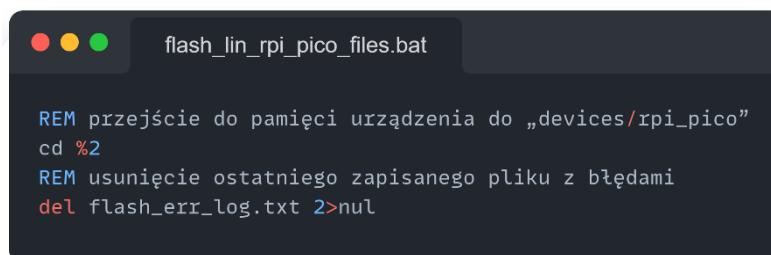
Można go wywołać ręcznie za pomocą konsoli CMD i PowerShell lub poprzez implementację biblioteki „subprocess” (Rys. 33) w skrypcie Python:

A screenshot of a code editor window titled 'example_use_in_other_script.py'. The code defines a variable 'rpi_pico' as a subprocess.Popen object. It passes a list of arguments to the 'args' parameter: a path to a batch file with a placeholder for a script name, a path to a CSV file, and a path to a log file. The 'creationflags' parameter is set to 'subprocess.CREATE_NEW_CONSOLE'.

Rys. 33. Przykładowe wykorzystanie programu w innych skryptach napisanych w języku Python.

Dodatkowy argument funkcji „Popen” o nazwie „creationflags” pozwala na otwarcie konsoli w oddzielnym oknie zamiast w tej wbudowanej w dane IDE. Obiekt „rpi_pico” zawiera również metodę „wait”, która pozwala na zatrzymanie programu na czas trwania komunikacji pomiędzy urządzeniami.

Pierwszym kluczowym elementem skryptu jest obsługa błędów (Rys. 34). W przypadku, gdy wystąpi nieprzewidziany błąd, zostanie on zapisany do pliku tekstowego, aby wspomóc użytkownika w przypadku problemów. Jako że wszystkie komunikaty o błędach są domyślnie wypisywane w konsoli CMD jako stderr, można przekierować ten strumień do pliku. Każde przekierowanie z strumienia „2” jest dopisane po każdym poleceniu związanym z rshell

A screenshot of a batch script window titled 'flash_lin_rpi_pico_files.bat'. The script contains three lines: a comment in Polish about navigating to the device memory, a 'cd' command with a placeholder '%2', another comment about deleting the previous log file, and a 'del' command for 'flash_err_log.txt' with '2>nul' to redirect stderr to the null device.

Rys. 34. Skrypt batch odpowiedzialny za wgrywanie programu na Raspberry Pi Pico. cz.1

Każde przekierowanie z strumienia „2” jest dopisane po każdym poleceniu związanym z rshell. Następnie skrypt przechodzi do etapu wgrywania wymaganych plików na Raspberry Pi Pico (Rys. 35). Wczytywane są 3 pliki: main.py, lib.py oraz reset.py (Rys. 36) oraz następuje weryfikacja całej operacji oraz wyświetlenie plików znajdujących się w pamięci płytki.

```
flash_lin_rpi_pico_files.bat

echo "Copying [%1] data generation script to rpi pico main ..."
rshell --buffer-size=512 --quiet "cp scripts\\%1.py /pyboard/main.py" 2>> flash_err_log.txt

echo "Copying library for data generation script to rpi pico main ..."
rshell --buffer-size=512 --quiet "cp rpi_lib\\lin_rpi_pico_lib.py /pyboard/lib.py" 2>> flash_err_log.txt

echo "Copying reset to rpi pico ..."
rshell --buffer-size=512 --quiet "cp rpi_lib\\reset.py /pyboard/reset.py" 2>> flash_err_log.txt

echo "Library check"
rshell --buffer-size=512 --quiet "ls /pyboard" 2>> flash_err_log.txt
```

Rys. 35. Skrypt batch odpowiedzialny za wgrywanie programu na Raspberry Pi Pico. cz.2

Plik main.py zawiera instrukcje z pliku konfiguracyjnego, lib.py wszystkie funkcje z biblioteki, a reset.py zawiera tylko jedną instrukcję, która resetuje urządzenie. Jest uruchamiana jako pierwsza, ponieważ przy normalnym wgrywaniu pliku, użytkownik musi ręcznie importować plik main.py, aby go uruchomić. Poprzez reset systemu Raspberry Pi Pico, uruchamiany jest automatycznie plik main.py, co usprawnia cały proces uruchamiania programów na płytce.

```
flash_lin_rpi_pico_files.bat

rshell --buffer-size=512 "repl ~ import reset ~ import main ~" 2>> flash_err_log.txt
```

Rys. 36. Skrypt batch odpowiedzialny za wgrywanie programu na Raspberry Pi Pico. cz.3

Po zakończeniu działania programu wgranego do pamięci, skrypt batch odzyskuje pliki z pamięci urządzenia i zapisuje je we wskazanych lokalizacjach (Rys. 37). Kończy pracę wysyłając „0” jeśli cały proces przebiegł pomyślnie.

```
flash_lin_rpi_pico_files.bat

move flash_err_log.txt %3

REM change directory to logs directory
cd %3
rshell --buffer-size=512 --quiet "cp /pyboard/com_data* ." 2>> flash_err_log.txt
REM cleanup Rpi Pico memory
rshell --buffer-size=512 --quiet "rm /pyboard/com_data* ." 2>> flash_err_log.txt

echo "Finished."
timeout 2
exit 0
```

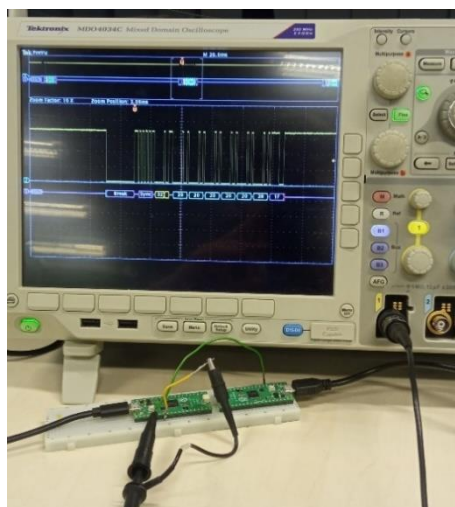
Rys. 37. Skrypt batch odpowiedzialny za wgrywanie programu na Raspberry Pi Pico. cz.4

Rozdział 3. Prezentacja urządzenia i testy

Ostatnim etapem projektu jest prezentacja pełnej implementacji protokołu LIN na urządzeniu opartym o mikrokontroler. W tym celu poza prezentacją pracy urządzenia za pomocą konsoli wykorzystano również oscyloskop. Poniższy rozdział zawiera opis i ilustracje przedstawiające pełen zakres funkcjonalności narzędzia oraz testy sprawdzające poprawność jego działania.

3.1. Prezentacja działania urządzenia

Ze względu na ograniczenia spowodowane poufnością informacji dotyczących urządzeń komercyjnych, testy odbyły się poprzez wykorzystanie drugiej płytki Raspberry Pi Pico zaprogramowanej w tryb nadawania wiadomości. Do dodatkowej analizy komunikacji pomiędzy bliźniaczymi płytkami wykorzystano oscyloskop MDO4034C wyprodukowany przez firmę Tektronix. Jest to profesjonalne narzędzie, które pozwoli uwiarygodnić uzyskane wyniki. Poniżej przedstawiono obydwie płytki Raspberry Pi Pico połączone ze sobą na kanale LIN, do którego podpięto oscyloskop.



Rys. 38. Zdjęcie oscyloskopu wpiętego na linii komunikacyjnej pomiędzy płytkami Raspberry Pi Pico.

Głównym analizatorem symulacji protokołu LIN jest konsola, wyświetlana po uruchomieniu skryptu rozpoczynającego pracę płytki (Rys.39). Na poniższym zrzucie ekranu znajdują się pierwsze linijki przedstawiane użytkownikowi po uruchomieniu skryptu batch. Przekazują one instrukcje postępowania w przypadku niepoprawnej konfiguracji lub wadliwego połączenia między komputerem, a płytką. Przeważnie port USB przez, który urządzenie jest połączone, jest znajdowane automatycznie przez „rshell”. Jednak w przypadku, gdy wpiętych płytek jest 2 lub więcej, należy wskazać odpowiednie porty w skrypcie batch za pomocą sugerowanej komendy.

Następnie wyświetlone zostają operacje flashowania oprogramowania na Raspberry Pi Pico, zawartość pamięci płytki oraz sam proces inicjalizujący REPL.

```
WybierzRaspberry Pi Pico flash script
Instructions"
Shell will connect to rpi com port and update main file automatically."
If connection attempt with Raspberry Pi Pico failed, "
try to connect with specific port "rshell -p com8 --buffer-size=512"
Copying [CT25_rpi] data generation script to rpi pico main ...
Copying 'C:\Users\0HZ2F9\Documents\Archive\sample-project\python_env\project_specific\devices\rpi_pico\
Copying library for data generation script to rpi pico main ...
Copying 'C:\Users\0HZ2F9\Documents\Archive\sample-project\python_env\project_specific\devices\rpi_pico\
Library check"
in_rpi_transceiver_lib.py      com_data_1_1_2021_0-0-2.csv      com_data_24_10_2023_16-55-49.csv
in_rpi_transceiver_lib_beta.py com_data_24_10_2023_16-47-19.csv com_data_24_10_2023_16-57-13.csv
main.py                       com_data_24_10_2023_16-47-57.csv
reset.py                      com_data_24_10_2023_16-52-33.csv
Connecting to COM10 (buffer-size 512)...
Trying to connect to REPL connected
Retrieving sysname ... rp2
Testing if sys.stdin.buffer exists ... Y
Retrieving root directories ... /com_data_1_1_2021_0-0-2.csv/ /com_data_24_10_2023_16-47-19.csv/ /com_d
lin_rpi_transceiver_lib.py/ /lin_rpi_transceiver_lib_beta.py/ /main.py/ /reset.py/
Setting time ... Oct 24, 2023 17:01:09
Evaluating board_name ... pyboard
Retrieving time epoch ... Jan 01, 1970
Entering REPL. Use Control-X to exit.
```

Rys. 39. Zawartość konsoli po uruchomieniu skryptu batch cz.1

Gdy inicjalizacja przebiegnie pomyślnie, uruchamiany jest skrypt „reset.py”, który resetując płytkę startuje działanie opisane w programie głównym „main.py”. Poniżej (Rys. 40) przedstawiono działanie programu, który w czasie rzeczywistym śledzi symulowaną komunikację protokołem LIN. Wyświetla on oddzielnie pola identyfikatora (PID), danych (DATA) oraz sumy kontrolnej (CHK) w formacie heksadecymalnym. Każda ramka jest numerowana oraz zapisywany jest czas jej otrzymania przez urządzenie.

```
MicroPython v1.19.1 on 2022-06-18; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>>
>>> import reset ; import main
MPY: soft reboot
|-----|
| LIN RPI SCRIPT |
| CT25 |
|-----|
-----seq_256_frames_sws()-----

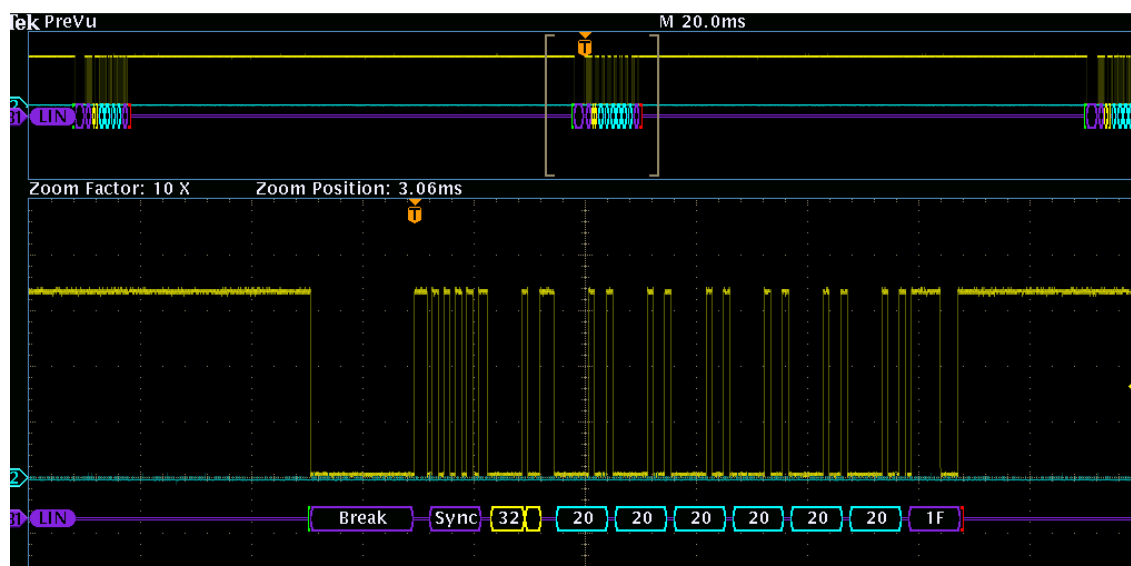
0. PID=0x20
   DATA: ['0x20', '0x20', '0x20', '0x20', '0x20', '0x20']
   CHK: 0x1f
Time: 0h:0m:0s:269ms

1. PID=0x30
   DATA: ['0x31', '0x31', '0x31', '0x31']
   CHK: 0x4a
Time: 0h:0m:0s:358ms

2. PID=0x20
   DATA: ['0x20', '0x20', '0x20', '0x20', '0x20', '0x20']
   CHK: 0x1f
Time: 0h:0m:0s:451ms
```

Rys. 40. Zawartość konsoli po uruchomieniu skryptu batch cz.2

Rzetelność komunikacji potwierdza również oscyloskop, który został doczepiony na linii komunikacyjnej LIN. Do ułatwienia odczytu uruchomiono dekodowanie sygnału na protokół LIN. Poniżej zamieszczono zdjęcie przechwyconej ramki z PID = 0x32, która min. jest wysyłana pomiędzy płytkami Raspberry Pi Pico.



Rys. 41. Sygnał przechwycony przez oscyloskop.

Zdjęcie zbliżone jest na fragment o długości 20ms, aby dane zawarte w ramce były czytelne. Jednorazowo pobierane są próbki przez 200ms i w mniejszym panelu górnym widać 2 ramki na początku i końcu pomiaru, co wskazuje, że wiadomości są wysyłane w odstępach około 100ms. Przybliżenie ukazujące ramkę o PID = 0x32 przedstawia również bajty danych = 0x20 oraz sumę kontrolną = 0x1F. Dzięki tym informacją można wywnioskować, że dane ukazywane w konsoli pokrywają się ze zdjęciami oscyloskopu. Wiadomość, więc jest zrozumiała dla wszystkich uczestników wymiany wiadomości, co dowodzi, iż wiadomości są wysyłane oraz odbierane pomyślnie.

Po zakończeniu sesji komunikacyjnej, program kończy działanie zapisując historię wymiany do pliku. Nazwa pliku zostaje wyświetlona, a następnie plik zostaje przeniesiony we wskazane wcześniej miejsce na dysk. Po odczekaniu kilku sekund cały proces kończy się zamknięciem konsoli (Rys. 42).

```

236. PID=0x30
      DATA: ['0x31', '0x31', '0x31', '0x31']
      CHK: 0x4a
Time: 0h:0m:24s:142ms

Saved to "com_out_11_1_2024_16-49-6.csv".

Press CTRL+X to export.

Done.
MicroPython v1.19.1 on 2022-06-18; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>>      1 file(s) moved.
Copying '/pyboard/com_data_11_1_2024_16-49-6.csv' to 'C:\Users\0
"Finished."

Waiting for 1_seconds, press a key to continue ...

```

Rys. 42. . Zawartość konsoli po uruchomieniu skryptu batch cz.3

Dane z pliku csv mogą zostać zaimportowane do programu Excel, gdzie po wybraniu domyślnego stylu formatowania danych prezentują tak jak na rysunku poniżej.

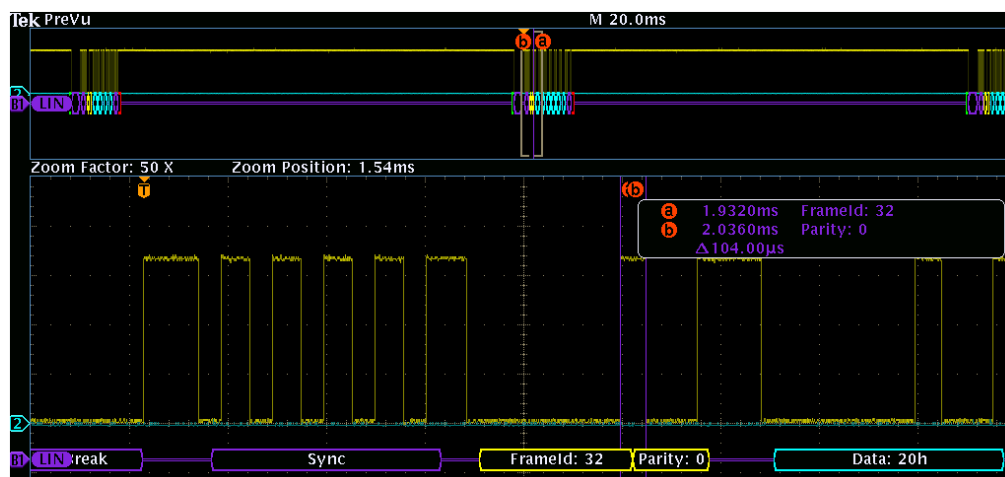
Number	Pid	Data0	Data02	Data2	Data3	Data4	Data5	Data6	Data7	Checksum	Frame Time
69 166.	0x20	0x20	0x20	0x20	0x20	0x20	0x20			0x1f	0h 0m 21s 664ms
70 167.	0x30	0x31	0x31	0x31	0x31					0x4a	0h 0m 21s 753ms
71 168.	0x20	0x20	0x20	0x20	0x20	0x20	0x20			0x1f	0h 0m 21s 846ms
72 169.	0x30	0x31	0x31	0x31	0x31					0x4a	0h 0m 21s 935ms
73 170.	0x20	0x20	0x20	0x20	0x20	0x20	0x20			0x1f	0h 0m 22s 27ms
74 171.	0x30	0x31	0x31	0x31	0x31					0x4a	0h 0m 22s 117ms
75 172.	0x20	0x20	0x20	0x20	0x20	0x20	0x20			0x1f	0h 0m 22s 209ms
76 173.	0x30	0x31	0x31	0x31	0x31					0x4a	0h 0m 22s 299ms
77 174.	0x20	0x20	0x20	0x20	0x20	0x20	0x20			0x1f	0h 0m 22s 391ms
78 175.	0x30	0x31	0x31	0x31	0x31					0x4a	0h 0m 22s 481ms
79 Start_Date: 11/1/2024 17:0:27											
80 End_Date: 11/1/2024 17:0:49											

Rys. 43. Zawartość wygenerowanego pliku csv z historią wiadomości.

3.1. Poprawność transmisji

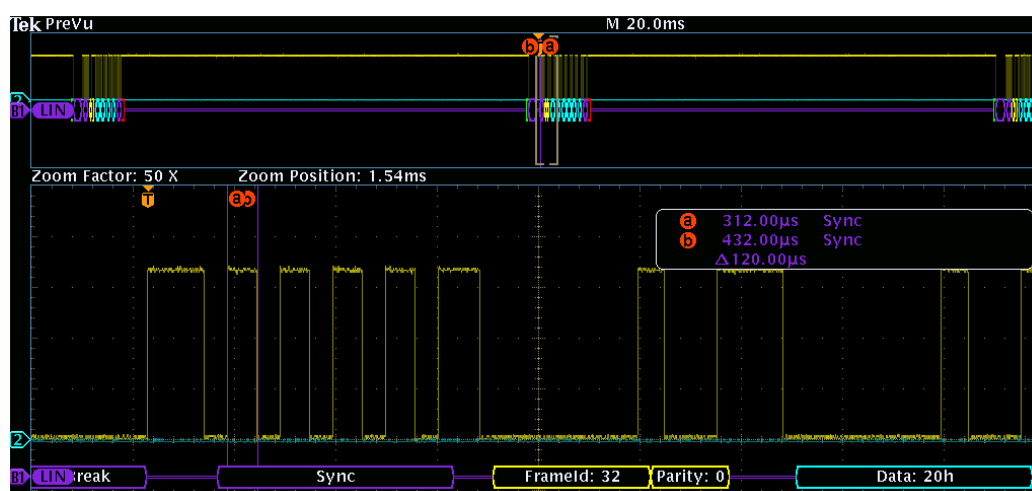
Aby przedstawić możliwości opracowanego urządzenia, przeprowadzono również podstawowe testy. Dotyczyły one podstawowej funkcjonalności urządzenia pod względem praktycznego zastosowania. Nie korzystano z norm ISO 17897 w pełni ze względu na brak implementacji transceivera wymaganego do komunikacji na poziomie 12V. Sprawdzono natomiast poprawność komunikacji pod względem przedziałów czasowych, odnawiania pracy po nastąpieniu awarii sytemu oraz dystansu jaki jest możliwy do utrzymania wymiany danych.

Najpierw sprawdzona została częstotliwość zmian stanu linii. Według oprogramowania powinna ona wynosić standardowe 9600 kbit/s. Aby to sprawdzić wystarczy zmierzyć długość trwania jednego bitu lub bajtu. Jako że szukana wartość częstotliwości to 9600 kbit/s, można stwierdzić, że 1 bit powinien trwać dokładnie $\frac{1}{9600} s = 104\mu s$.



Rys. 44. Pomiar długości jednego bitu w polu identyfikatora wysyłanego przez Raspberry Pi Pico

Na powyższym rysunku przedstawiony został pomiar czasu trwania bitu recesywnego w polu identyfikatora ramki. W prawym górnym rogu widoczny jest czas pomiędzy znacznikami „a” i „b”, który wynosi wspomniane 104 μs . Udało się uzyskać idealny wynik jednak, należy pamiętać o tym, że urządzenia często przewidują 10% - 20% marginesu błędu. Czyli czas trwania bitu mógł wynosić od 83 μs do 124 μs . Można to zauważyć mierząc bit pola synchronizacji, przedstawiony na rysunku poniżej, który nie powoduje błędów w komunikacji pomimo innej długości bitu recesywnego.



Rys. 45. Pomiar długości jednego bitu w polu synchronizacji wysyłanego przez Raspberry Pi Pico

[illegible]

tek PreVu

M 1.00 s

B1 Vertical
a -740mdiv

B1 LIN

Zoom Factor: 5 X

Zoom Position: 414ms

B1 LIN

46

Na powyższych zdjęciach, można zauważyć wspomnianą przerwę w komunikacji. Linia została zaciągnięta do stanu dominującego jednak jej nierówny kształt wskazuje na brak połączenia z jakimkolwiek urządzeniem. Sama konsola wyświetla komunikat o braku informacji na RX FIFO, które jest odpowiedzialny z ich odbiór. Przerwanie trwało niecałe 2 sekundy i po ponownym podłączeniu sygnału pomiędzy urządzeniami widać, że wiadomości są nadal odbierane w poprawny sposób. Jest to widoczne zarówno w konsoli jak i na ekranie oscyloskopu.

3.2. Weryfikacja problemów i przedstawienie potencjalnych ulepszeń

Pomimo poprawnie działających podstawowych funkcjonalności płytki i oprogramowania, nie jest ona idealnym narzędziem. Podczas testów zauważono również pewne błędy w przedstawianej implementacji protokołu LIN na urządzeniu opartym o mikrokontroler. W poniższym podrozdziale opisano wady opracowanego rozwiązania oraz potencjalne ulepszenia całego narzędzia.

Przedstawiane problemy odnoszą się wyłącznie do zaimplementowanego oprogramowania oraz jego przewidzianych funkcjonalności. Nie zostały zatem poruszone kwestie warstwy elektrycznej ze względu na brak transceivera, który jest jej podstawą do poprawnego funkcjonowania w zakresie napięć 12V lub 24V.

Pierwszym błędem z jakim boryka się urządzenie jest niestabilność pracy. Przejawia się ona przez losowe błędy podczas jego działania spowodowane czynnikami zewnętrznymi. Często są to przedwczesne przerwania wykonywanych poprzez odpięcie jej zasilania lub przerwanie procesu w konsoli. W chwili, gdy płytka zostaje wyłączona podczas pracy, zdarzało się, że zapętląła ostatnie zadane jej instrukcje. Podczas gdy płytka kontynuowała np. wysyłanie wiadomości, uniemożliwiała tym wgranie nowego programu. Również odzyskanie historii komunikacji było niemożliwe, gdyż plik jest zapisywany dopiero po ukończeniu pracy, co nigdy nie następuje w przypadku zapętlenia instrukcji. Wymagany jest wtedy twardy reset urządzenia poprzez wgranie na nowo lub odcięcie zasilania na dłuższy okres i ponowne wgranie programu na płytkę. W trakcie pracy nad płytką zaimplementowano kilka metod zapobiegania tym błędom takich jak zatrzymanie pracy, jeśli błąd będzie wykrywany przez określoną ilość czasu. Jednak były one w pełni skuteczne przez co nie zostały zaprezentowane w pracy.

Kolejnym problemem jaki napotkano podczas próby implementacji protokołu LIN jest zbyt wolne działanie drugiego wątku procesora zawartego w RP2040. Co prawda pozwala on na jednoczesny odczyt komunikacji podczas nadawania wiadomości w trybie pracy jako urządzenie typu „master”. Jednak Gdy podjęto próbę implementacji trybu „slave” okazało się to być większym wyzwaniem. W trybie „master” nadawaniem wiadomości zajmuje się główny wątek natomiast odbiór wiadomości jest odizolowany na drugim wątku. Współdzielenie danych zostało zminimalizowane pomiędzy wątkami, ponieważ większa ilość wspólnych danych powodowała niechciane opóźnienia w pracy urządzenia. W trybie „slave” praca urządzenia polegałaby na odebraniu początku wiadomości z identyfikatorem i wysłania odpowiedzi w czasie nie dłuższym niż 1 T-bit. Jako że wymiana danych pomiędzy wątkiem odbierającym wiadomość a wątkiem nadającym jest wolna, powodowało to problemy z wysłaniem odpowiedzi na zadany czas. Prawdopodobnym rozwiązaniem tego problemu jest dalsze rozwijanie kodu programu lub jego przeniesienie na język C, aby usprawnić cały proces przetwarzania danych. Dodatkowo zauważono wzrost prędkości przekazywania danych, gdy komunikacja nie była wyświetlana w czasie rzeczywistym w konsoli.

Następną wadą urządzenia jest jego mała pamięć. Pozwala ona jedynie na zapisanie historii 15-20 minut komunikacji. Rozważano implementację rozwiązania, jednak na dany etap rozwoju, 10 minut było wystarczającym wynikiem. Potencjalnymi usprawnieniami może być np. dodanie płytki z kartą SD taką jak „Adafruit Industries LLC 4682” [\[10\]](#). Należałoby wtedy zaimplementować obsługę przesyłania danych na dodatkową płytkę za pomocą protokołu SPI .

Ostatnim ulepszeniem przewidzianym dla urządzenia jest implementacja graficznego interfejsu użytkownika. Przygotowanie przykładowych metod w prezentowanych plikach źródłowych nie jest możliwe najprzyjemniejsze w obsłudze przez końcowego użytkownika. Jest to rozwiązanie praktyczne jednak zamknięcie całego programu w jednej wykonywalnej aplikacji ułatwiłoby jego obsługę. W podstawowym zastosowaniu, można przygotować panel, gdzie użytkownik będzie miał możliwość wyboru trybu pracy, możliwość deklaracji wysyłanych ramek oraz sposób zapisu lub wyświetlania danych.

Podsumowanie

Cel pracy jakim była implementacja protokołu LIN na urządzeniu opartym o mikrokontroler został pomyślnie ukończony. Przygotowano odpowiednią analizę zarówno wymagań związanych z protokołem LIN, jak i zbadano dostępne urządzenia zawierające wystarczające osiągi dla przewidzianego rozwiązania. Wyszczególniono również fragmenty oprogramowania odpowiedzialnego za komunikację na magistrali LIN.

Testy, które zostały przeprowadzone na urządzeniu potwierdziły poprawne działanie urządzenia. Wiarygodność testów została poparta poprzez profesjonalne narzędzia do analizy sygnałów. Za pomocą testów wyszczególniono poprawnie działające zabezpieczenia przed awariami, jak i również ograniczenia samego urządzenia.

Dalszy rozwój oraz usprawnienia projektu pozwoliłyby na zwiększenie niezawodności oraz zakresu możliwości przedstawianego narzędzia. Wymaga ono min. opracowania dokładniejszego systemu zapobiegania awariom, usprawnienia metod odpowiedzialnych za wymianę danych pomiędzy wątkami oraz funkcjonalności zapisywania plików na nośniki zewnętrzne. Można kontynuować pracę w środowisku Micropython jednak należałoby również zbadać działanie urządzenia opisanego w języku C.

Prowadzone są prace nad aspektem warstwy elektrycznej urządzenia, aby było ono możliwe do zweryfikowania przez normę ISO jako pełnoprawny węzeł magistrali LIN. Implementacja transceivera oraz dodatkowych układów przygotowanych pod testy innych urządzeń, znacznie zwiększą możliwości wykorzystania urządzenia.

Praktycznym zastosowaniem takiego urządzenia, które również jest już częściowo wykorzystywane, są testy przełączników produkowanych w przemyśle motoryzacyjnym. Ze względu na niskie koszty, dużą elastyczność oprogramowania oraz płytki jak i możliwości synchronizacji z zewnętrznymi programami jest to bardzo przystępne narzędzie. Praktycznym przykładem jego zastosowania jest wsparcie automatyzacji testów z normy ISO 17897.

BIBLIOGRAFIA

- [1] Standardization International Organization, *ISO 17897 1-8*, 2016.
- [2] E. Hackett, *LIN Protocol and Physical Layer Requirements*, 2018.
- [3] W. Ahmed Khan, G. Abbas, K. Rahman, G. Hussain i C. Aimal Edwin (Redaktorzy), *Functional Reverse Engineering of Machine Tools*, 2019, str. 22-24.
- [4] J. T. Wunderlich, *Focusing on the Blurry Distinction between Microprocessors and Microcontrollers*, Proc. of ASEE Nat'l Conf. 1999.: Elizabethtown College and Purdue University, 1999.
- [5] S.r.l. Arduino, „Arduino Uno Rev3,” [Data uzyskania dostępu: 20.12.2023] <https://store.arduino.cc/products/arduino-uno-rev3>.
- [6] BOTLAND, „STM32 NUCLEO-L432KC - STM32L432KCU6 ARM Cortex M4,” [Data uzyskania dostępu: 20.12.2023]: <https://botland.store/stm32-nucleo/7607-stm32-nucleo-l432kc-stm32l432kcu6-arm-cortex-m4-5904422336028.html>.
- [7] Raspberry Pi. Raspberry Pi Pico. [Data uzyskania dostępu: 20.12.2023] <https://www.raspberrypi.com/products/raspberry-pi-pico/>.
- [8] Raspberry Pi, „RP2040 Datasheet”, Str 309-310, [Data uzyskania dostępu: 20.12.2023] <https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf>.
- [9] Raspberry Pi, „rp2 — functionality specific to the RP2040,” [Data uzyskania dostępu: 20.12.2023] <https://docs.micropython.org/en/latest/library/rp2.StateMachine.html>.
- [10] DigiKey, „DigiKey,” [Data uzyskania dostępu: 23.12.2023] <https://www.digikey.com/en/products/detail/adafruit-industries-llc/4682/12822319>.
- [11] Adafruit, „Schematic, Mask, and Die Shot of Intel’s 4004 CPU from 1971” [Data uzyskania dostępu: 23.12.2023] <https://en.wikichip.org/wiki/File:MCS-4.jpg#file>
- [12] WikiChip LLC, „File:MCS-4.jpg” [Data uzyskania dostępu: 23.12.2023] <https://blog.adafruit.com/2016/06/13/schematic-mask-and-die-shot-of-intels-4004-cpu-from-1971/>

SPIS RYSUNKÓW

Rys. 1. Graficzna ilustracja pola przerwy, pola synchronizacji oraz pola identyfikatora protokołu LIN [2].....	7
Rys. 2. Graficzna ilustracja pola danych i pola sumy kontrolnej protokołu LIN [2].	7
Rys. 3. Sygnał z zbyt wolną zmianą stanu linii – nieczytelny przez odbiorcę [2].	8
Rys. 4. Sygnał z wolną zmianą stanu linii – czytelny przez odbiorcę [2].	9
Rys. 5. Sygnał z wolną zmianą stanu linii – czytelny przez odbiorcę [2].	9
Rys. 6. Zestaw chipów z rodziny Intel MCS-4: 4001, 4002, 4003 i 4004 [11].....	10
Rys. 7. Zdjęcie matrycy procesora Intel 4004 [12].....	10
Rys. 10. Arduino UNO [5].....	14
Rys. 9. Raspberry Pi Pico [6].....	14
Rys. 8. STM32 NUCLEO-L432KC [7].....	14
Rys. 11. Schemat blokowy PIO [8].	22
Rys. 12. Schemat blokowy maszyny stanów [8].	23
Rys. 13. Ilustracja przykładowego programu dla maszyny stanów.....	25
Rys. 14. Metoda opisująca instrukcje pola przerwy oraz synchronizacji dla PIO cz.1 ..	25
Rys. 15. Metoda opisująca instrukcje pola przerwy oraz synchronizacji dla PIO cz.2 ..	26
Rys. 16. Metoda opisująca instrukcje pola identyfikatora oraz danych dla PIO	27
Rys. 17. Fragment kodu przedstawiający funkcje odpowiadające za konwersję danych.	28
Rys. 18. Główna metoda programu obsługująca proces komunikacji protokołem LIN oraz przyjmowane przez nią argumenty.	28
Rys. 19. Ciało metody "convert_id".	29
Rys. 20. Fragment kodu sprawdzający typ nadawanej wiadomości.....	29
Rys. 21. Fragment kodu przedstawiający inicjalizację maszyny stanów w ciele głównej metody.	30
Rys. 22. Przykładowa metoda zarządzająca działaniem flag IRQ.	31
Rys. 23. Konfiguracja flag IRQ.	31
Rys. 24. Wywołanie funkcji odpowiedzialnych za odczyt danych.	32
Rys. 25. Inicjalizacja maszyny stanów odpowiedzialnej za odczyt danych.	33
Rys. 26. Fragment kodu przedstawiający obsługę drugiego wątku procesora.	33
Rys. 27. Obsługa błędów związanych z pracą drugiego wątku procesora.	34
Rys. 28. Instrukcje wykonywane przez drugi wątek procesora.....	35

Rys. 29. Metoda zarządzająca wyświetlaniem oraz zapisywaniem historii komunikacji urządzenia.	36
Rys. 30. Metoda odpowiedzialna za zapis danych w pliku csv.	36
Rys. 31. Metoda odpowiedzialna za wyświetlanie komunikacji urządzenia w czasie rzeczywistym.	37
Rys. 32. Przykładowy program wykorzystujący bibliotekę implementującą protokół LIN dla Raspberry Pi Pico.....	38
Rys. 33. Przykładowe wykorzystanie programu w innych skryptach napisanych w języku Python.	39
Rys. 34. Skrypt batch odpowiedzialny za wgrywanie programu na Raspberry Pi Pico. cz.1	39
Rys. 35. Skrypt batch odpowiedzialny za wgrywanie programu na Raspberry Pi Pico. cz.2	40
Rys. 36. Skrypt batch odpowiedzialny za wgrywanie programu na Raspberry Pi Pico. cz.3	40
Rys. 37. Skrypt batch odpowiedzialny za wgrywanie programu na Raspberry Pi Pico. cz.4	40
Rys. 38. Zdjęcie oscyloskopu wpiętego na linii komunikacyjnej pomiędzy płytkami Raspberry Pi Pico.....	41
Rys. 39. Zawartość konsoli po uruchomieniu skryptu batch cz.1	42
Rys. 40. Zawartość konsoli po uruchomieniu skryptu batch cz.2	42
Rys. 41. Sygnał przechwycony przez oscyloskop.	43
Rys. 42. . Zawartość konsoli po uruchomieniu skryptu batch cz.3	44
Rys. 43. Zawartość wygenerowanego pliku csv z historią wiadomości.....	44
Rys. 44. Pomiar długości jednego bitu w polu identyfikatora wysyłanego przez Raspberry Pi Pico.....	45
Rys. 45. Pomiar długości jednego bitu w polu synchronizacji wysyłanego przez Raspberry Pi Pico.....	45
Rys. 46. Zdjęcie ekranu oscyloskopu ukazujące przerwę w połączeniu pomiędzy urządzeniami.....	46
Rys. 47. Informacja o awarii wyświetlana przez konsolę.....	46