Politechnika
Śląska

PRACA MAGISTERSKA

Uniwersalny blok wejść - wyjść analogowych i cyfrowych do komputera PC

Dawid NAJDA
Nr albumu: 281533

Kierunek: Automatyka i Robotyka
Specjalność: Robotyka

PROWADZĄCY PRACĘ
dr inż. Mirosław Magnuski
KATEDRA Elektroniki, Elektrotechniki i Mikroelektroniki
Wydział Automatyki, Elektroniki i Informatyki

Gliwice 2024

Tytuł pracy

Uniwersalny blok wejść - wyjść analogowych i cyfrowych do komputera PC

Streszczenie

Niniejsza praca magisterska przedstawia projekt i realizację uniwersalnego bloku wejść-wyjść analogowych i cyfrowych dla komputera PC. W ramach pracy stworzono urządzenie, które umożliwia zdalne pomiary i generowanie sygnałów oraz przesył rezultatów pomiarów do użytkownika za pomocą standardowych interfejsów komunikacyjnych. W pracy opisano zastosowane przetworniki, techniki komunikacji z peryferiami, najważniejsze elementy oprogramowania oraz sposób sterowania działaniem urządzenia. Przeprowadzono również kalibrację oraz eksperymenty, które potwierdziły funkcjonalność i poprawność działania zbudowanego prototypu. Praca kończy się omówieniem uzyskanych wyników oraz propozycjami dalszego rozwoju projektu.

Słowa kluczowe

zdalne połączenie, blok wejść-wyjść, przetworniki, analogowo-cyfrowe, cyfrowo-analogowe

Thesis title

Universal analog and digital input - output block for PC computer

Abstract

This master's thesis presents the design and implementation of a universal analog and digital input-output block for a PC. As part of the project, a device was created that enables remote measurements and signal generation, as well as the transmission of measurement results to the user via standard communication interfaces. The thesis describes the applied converters, techniques for communication with peripherals, key software components, and the method of controlling the device's operation. Calibration and experiments were also conducted, confirming the functionality and correctness of the constructed prototype. The thesis concludes with a discussion of the obtained results and proposals for further development of the project.

Key words

remote connection, input-output block, converters, analog-to-digital, digital-to-analog

Spis treści

Rozdział 1

Wstęp

Urządzenia pomiarowe znajdują szerokie zastosowanie w praktycznie każdej dziedzinie techniki i nauki, w przemyśle, w laboratoriach naukowych oraz po prostu przy zastosowaniach domowych. Oczywiście, w zaawansowanych procesach stosuje się dedykowane urządzenia zaprojektowane specjalnie do tego celu. Jednakże w wielu sytuacjach, szczególnie niestandardowych lub niespodziewanych, możliwe (a nawet konieczne) jest wykorzystanie urządzeń uniwersalnych, i dostosowanie ich do bieżących potrzeb, choćby ze względu na czas i koszt stworzenia rozwiązania dedykowanego. Zaletą w takich sytuacjach z pewnością jest wielofunkcyjność danego urządzenia, co pozwala ograniczyć miejsce, ilość przewodów czy czas. Dodatkowym pozytywem jest możliwość zdalnej komunikacji z takim urządzeniem, co pozwala na umieszczenie go na przemieszczającym się obiekcie lub w środowisku niesprzyjającym przebywaniu ludzi.

Najczęściej bezpośrednio mierzoną wielkością jest napięcie. Za jego pomocą zaś można mierzyć inne wielkości fizyczne, używając odpowiednich układów dokonujących analogowych konwersji. Są to na przykład: natężenie prądu, opór, pojemność, indukcyjność, natężenia pól elektrycznych i magnetycznych, natężenie światła i dźwięku, siłę nacisku, ciśnienie, wagę, masę, moment obrotowy, i wiele, wiele innych.

Najczęściej wyjściową wielkością jest również napięcie, zaś za jego pomocą można generować inne wielkości fizyczne, używając odpowiednich układów analogowych lub wzmacniaczy – np. natężenie prądu, pole elektryczne lub magnetyczne, światło, dźwięk, ruch liniowy lub obrotowy i inne.

W czasach przed epoką cyfrową wykorzystywało się różne urządzenia pomiarowe do przedstawienia danych lub przebiegów operatorowi, który na ich podstawie mógł podejmować jakieś decyzje. Współcześnie również są one wykorzystywane, jednak coraz częściej (szczególnie takie wymagające analizy) przetwarzane są komputerowo, dlatego dużą zaletą jest urządzenie pomiarowe, które działa również cyfrowo oraz potrafi w prosty sposób łączyć się z komputerem. Urządzenia cyfrowe ułatwiają też obserwację zdarzeń losowych, nawet pojedynczych.

1.1 Cel pracy

Celem pracy jest stworzenie uniwersalnego bloku wejść - wyjść analogowych i cyfrowych do komputera PC. Ma to być elektroniczne urządzenie pomiarowe, sterowane zdalnie za pomocą komputera osobistego. Powinno ono być możliwie uniwersalne – być w stanie przynajmniej w ograniczonym zakresie zastępować narzędzia takie jak multimetr [7], oscyloskop, generator funkcji/sygnałów [29] czy generator wzorców cyfrowych.

1.2 Zakres pracy

Budowany blok pomiarowy będzie wyposażony w wejścia i wyjścia analogowe, zarówno napięciowe jak i prądowe. Oprócz tego, dostępne będą również wejścia i wyjścia cyfrowe. Wszystkie porty analogowe powinny obsługiwać zarówno dodatnie jak i ujemne polaryzacje sygnałów. Wejścia analogowe powinny mieć wybieralne zakresy wartości. Urządzenie zostanie zbudowane na płytce uniwersalnej, która otrzyma obudowę stworzoną za pomocą drukarki `3D`. Niektóre komponenty (szczególnie mikroprocesor) będą użyte w postaci gotowych płytek w celu uniknięcia montażu elementów `SMD` oraz ze względu na generalnie mniejszy rozmiar, krótsze połączenia, a więc i lepsze działanie w tego typu rozwiązaniach. Całość zarządzana będzie przez mikrokontroler, oprogramowany w języku `C++`, z którym można będzie się komunikować za pomocą komputera osobistego.

Rozdział 2

Analiza tematu

2.1 Przetworniki

Przetworniki to układy elektroniczne mające na celu zamianę jakiejś wielkości fizycznej (najczęściej napięcia) na wartości cyfrowe (zdyskretyzowane i skwantowane) lub vice-versa.

2.1.1 Cel przetwarzania sygnałów

Przetwarzanie analogowo-cyfrowe

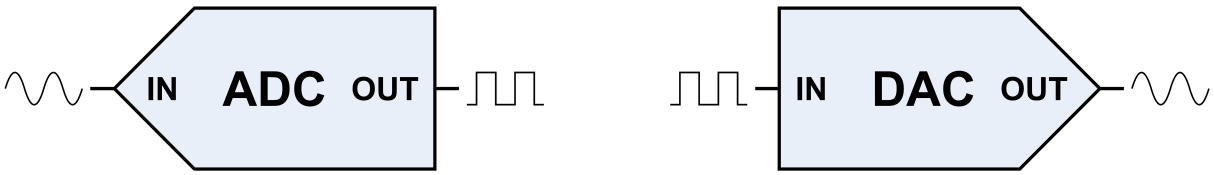
Przetwornik `A/C` (ang. *`A/D`* – *Analog to Digital*, *`ADC`* – *Analog-Digital Converter*) – jest to układ elektroniczny służący do zamiany sygnału analogowego na sygnał cyfrowy. Dzięki temu możliwe jest przetwarzanie ich w urządzeniach elektronicznych opartych o architekturę zero-jedynkową oraz przechowywanie na dostosowanych do tej architektury nośnikach danych.
Najczęściej bezpośrednio mierzoną wartością jest napięcie, zaś za jego pomocą można mierzyć inne wielkości fizyczne, używając odpowiednich układów analogowych – np. natężenie prądu i opór, natężenie pola elektrycznego i magnetycznego, pojemność i indukcyjność, natężenie światła lub dźwięku, siłę nacisku, wagę, masę, moment obrotowy, i wiele, wiele innych. Wyspecjalizowane przetworniki służą też do robienia zdjęć.

Przetwarzanie cyfrowo-analogowe

Przetwornik `C/A` (ang. *`D/A`* – *Digital to Analog*, *`DAC`* – *Digital-Analog Converter*) – jest to układ elektroniczny zamieniający sygnał cyfrowy na sygnał analogowy. Dzięki temu istnieje możliwość wpływu na systemy analogowe, mechaniczne, itp. za pomocą kontrolerów cyfrowych.
Najczęściej generowaną wartością jest napięcie, zaś za jego pomocą można tworzyć inne wielkości fizyczne, używając odpowiednich układów analogowych lub wzmacniaczy – np.

natężenie prądu, pole elektryczne lub magnetyczne, światło, dźwięk/muzykę, ruch liniowy lub obrotowy i inne.

2.1.2 Zasada działania przetworników



(a) Analogowo-cyfrowy (b) Cyfrowo-analogowy

Rys. 2.1: Symbole przetworników

Przetworniki analogowo-cyfrowe

Przetwarzanie `A/C` dzieli się ogólnie na 3 etapy: próbkowanie, kwantyzacja i kodowanie. Najpierw należy stworzyć sygnał dyskretny, reprezentujący sygnał ciągły w czasie za pomocą listy wartości nazywanych próbkami. Następnie przeprowadza się kwantyzację – czyli zastępuje próbki, które mogą przyjmować dowolne wartości, na jakąś skończoną ilość stanów, za pomocą wewnętrznych elementów przetwornika [15]. Finalnie, stan wewnętrzny zamieniany jest na odpowiednie kody binarne, które można potem wyprowadzić: bezpośrednio na zewnątrz – równolegle, lub przesłać szeregowo za pomocą dowolnego protokołu.

Przetworniki cyfrowo-analogowe

Przetwarzanie `C/A` dzieli się ogólnie na 3 etapy: dekodowanie, generacja, wystawianie. Najpierw, kod binarny jest odbierany, albo przez wejście równoległe, albo szeregowe. Potem jest zamieniany na odpowiedni stan wewnętrzny. Następnie ma miejsce generacja – za pomocą elementów ustawionych w poprzednim kroku, tworzona jest odpowiadająca wartość. Finalnie, jest ona przekazana na wyjście i podtrzymana za pomocą elementu buforującego [15].

Podsumowanie

Oba rodzaje przetworników są bardzo podobne pod względem idei działania. Oczywiście proces musi być w odwrotnej kolejności. Przetwarzanie `A/C` jest popularniejsze niż `C/A`, ze względu na to, że przeważnie zależy nam na zbieraniu informacji z wielu czujników lub elementów, na podstawie zbioru których opracowywane jest przez układ kontrolujący odpowiednie sterowanie jakimś elementem. Tak samo każde sprzężenie zwrotne, wykonane cyfrowo, musi mieć w sobie przetwornik `A/C`.

2.1.3 Parametry charakteryzujące przetworniki

Każdy przetwornik jest opisany przez kilka parametrów, z czego niektóre są wzajemnie zależne. Występują one w obu typach, jednak mogą nosić inną nazwę lub, zależnie od kierunku, ich funkcja może się zmieniać.

Zakres

Oznacza on zakres napięć wejściowych lub wyjściowych, który przetwornik może poprawnie odczytać lub wytworzyć; nazywany jest „prawidłowym zakresem konwersji" przetwornika. Zakres ma zawsze dolną i górną granicę: $U_{range} = U_{upper} - U_{lower}$. Najczęściej wyróżnia się dwa przypadki: $U_{lower} = 0\,\mathrm{V}$ – klasyfikowany jako jednobiegunowy (unipolarny), $U_{lower} = -U_{upper}$ – klasyfikowany jako dwubiegunowy (bipolarny).

Przetwornik `A/C` nie może dostarczyć wiarygodnych konwersji, jeśli napięcie wejściowe jest poza zakresem. Najczęściej zwrócony kod będzie odpowiadać napięciu granicznemu, które zostało przekroczone. Przetwornik `C/A` (pomijając celową konstrukcję) ze zrozumiałych względów nie przekroczy na wyjściu napięć zasilania.

Liczba bitów

Liczba bitów (oznaczana N) określa długość słowa wyjściowego lub wejściowego. Najczęściej stosowany jest kod w postaci liczby całkowitej, w przypadku przetworników bipolarnych przeważnie stosuje się kod z przesunięciem lub kod uzupełnień do dwóch, ze względu na powszechność tej reprezentacji w systemach komputerowych. Każdy bit oznacza podwojenie ilości poziomów kwantyzacji (gdyż jeden poziom kwantyzacji to jeden kod bitowy), zgodnie ze wzorem $\#_{codes} = 2^N$. Najniższy kod to 0, zaś najwyższy – $2^N - 1$.

Liczba bitów bezpośrednio określa również tzw. *dynamic Range*, jest to różnica między maksymalnym i minimalnym sygnałem, podana w decybelach.

$$\mathrm{DR} = 20\lg\left(\frac{2^N}{1}\right) = 20\lg(2)N \approx 6N\mathrm{dB}$$

Rozdzielczość

Pojęcie to oznacza najmniejszą różnicę w analogowym sygnale, jaką może wykryć lub wytworzyć przetwornik, odpowiadającą przełączeniu najniższego bitu. Wyznaczana jest jako stosunek zakresu napięcia do ilości kodów cyfrowych [16].

$$U_{LSB} = \frac{U_{range}}{2^N}$$

Częstotliwość

Częstotliwość określa ile próbek w danym okresie czasu może zostać pobrane. Podaje się ją w Hz lub sps (ang. *samples per second*, próbki na sekundę). Parametr ten może zostać również jako odwrotność, tzn. okres lub czas próbkowania. Określa on co ile może zostać pobrana lub wystawiona wartość analogowa. Jest to jeden z najważniejszych parametrów, gdyż określa maksymalną częstotliwość sygnału jaki przetwornik jest w stanie zmierzyć lub wytworzyć, zgodnie z twierdzeniem Nyquista–Shannona. Wpływa również na rozdzielczość w czasie, aliasing, zużycie energii oraz wymagania dotyczące przetwarzania danych cyfrowych.

2.2 Ocena jakości działania przetworników

Zrozumiałym jest, że przetworniki, jak każdy fizyczny obiekt, nie działają w sposób idealny. Nawet gdyby mogły, i tak sama idea działania narzuca pewne ograniczenia. Precyzja zależy od jakości wykonania oraz warunków otoczenia. Poniżej przedstawione są parametry opisujące dokładność i jakość działania przetworników.

2.2.1 Parametry termiczne

Zmiana temperatury pracy, czy to przez zmianę temperatury otoczenia, czy wydzielanie ciepła, ma wpływ na zachowanie fizycznych elementów przetwornika, a więc i na wynik jego pracy. Teoretycznie wszystkie rodzaje błędu mogą się zmieniać wraz z temperaturą, często jednak karty katalogowe uwzględniają już zalecany przedział temperatur pracy w wartościach błędów. Najczęściej jednak spotykany jest współczynnik zmian cieplnych napięcia przesunięcia zera, wyrażany w $\frac{\mu V}{^\circ C}$ lub $\frac{\%}{^\circ C}$.

2.2.2 Monotoniczność

W przetwornikach `C/A`, jest to zdolność układu do zmiany napięcia wyłącznie w tym samym kierunku co zmiana wejściowego słowa cyfrowego. Oznacza to, że jeśli wartość zadana rośnie, to napięcie na wyjściu nie może zmaleć i vice versa. Jest to cecha pożądana w źródłach sygnałów niskiej częstotliwości oraz w przetwornikach wykorzystywanych jako programowalny trymer (nastawny komponent elektryczny służący do kalibracji działania układu).

2.2.3 Nieliniowość przetwarzania

Wszystkie przetworniki cierpią na nieliniowość ze względu na niedoskonałości w procesie produkcji. Powoduje to, że napięcie nie jest idealnie (liniowo lub inaczej, w przypadku specjalnego nieliniowego przetwornika) powiązane ze słowem. Nieliniowości podaje się jako

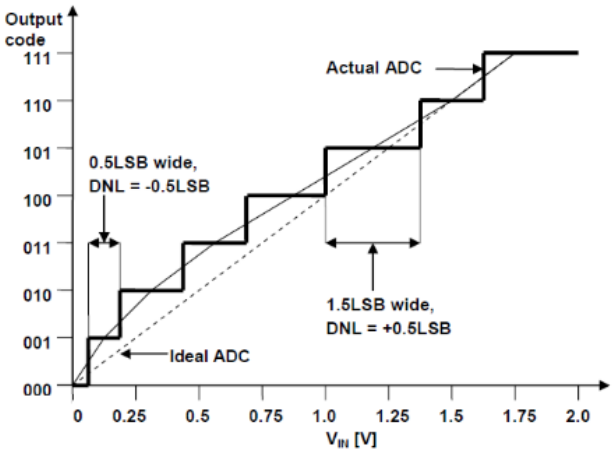
procent wartości maksymalnej lub jako krotność napięcia nominalnego odpowiadającemu najniższemu bitowi U_n.

Nieliniowość różniczkowa

Nieliniowość różniczkowa (ang. *DNL – differential nonlinearity*) określa się przez wyznaczenie różnic między sąsiednimi wartościami napięcia, wywołujących (dla `A/C`) lub wywoływanych przez (dla `C/A`) zmianę słowa o wartość najniższego bitu [16]. Określa ona więc błąd jednorodności szerokości poziomów kwantyzacji. Nieliniowość różniczkowa jest wyznaczana jako maksymalna różnica (zarówno dodatnia jak i ujemna) pomiędzy rzeczywistą U_r a nominalną U_n szerokością danego poziomu kwantyzacji:

$$\epsilon_d = \frac{|U_r - U_n|_{max}}{U_n}\texttt{LSB}$$

Dla przykładu: Na poniższym rysunku, w maksymalnym przypadku, szerokość stopnia wynosi 1.5 `LSB` (o wartości wyjściowej 101). Przez to `DNL` wyniesie +0.5 `LSB`. Natomiast w minimalnym przypadku szerokość kroku wynosi tylko 0.5 `LSB` (dla wartości wyjściowej 001), czyli o 0.5 `LSB` mniej niż oczekiwana. Finalnie `DNL` będzie wynosić ±0.5 `LSB`.



Rys. 2.2: Przykład wyznaczania nieliniowości różniczkowej [2]

W skrajnym przypadku, niektóre kody wyjściowe mogą nigdy nie wystąpić i utracona zostaje monotoniczność. Wtedy dolna granica `DNL` wychodzi poza -1 `LSB`.

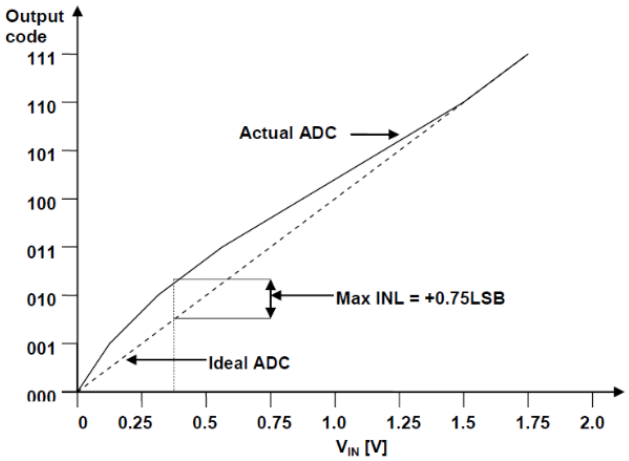
Nieliniowość całkowa

Nieliniowość całkowa (ang. ***INL*** *– integral nonlinearity*) jest określana jako $(\Delta U_I)_{max}$ – maksymalna różnica napięcia pomiędzy rzeczywistą charakterystyką przetwarzania $Q = f(U_r)$ lub $U_r = f(Q)$ a charakterystyką idealną U_i; odniesiona do nominalnej wartości napięcia U_n odpowiadającej najmniej znaczącemu bitowi [16]. Charakterystykę idealną wyznacza się jako prostą łączącą skrajne punkty zakresu przetwarzania, charakterystykę

rzeczywistą natomiast jako linię łączącą środki przedziałów napięcia odpowiadających kolejnym wartościom cyfrowym przetwornika.

$$\epsilon_r = \pm \frac{|U_r - U_i|_{max}}{U_n} \texttt{LSB}$$

INL można interpretować jako pewnego rodzaju sumę nieliniowości różniczkowych (DNL). Na przykład kilka kolejnych ujemnych wartości DNL przesuwa rzeczywistą krzywą w lewo, a więc podnosi ją powyżej krzywej idealnej, jak pokazano na rysunku poniżej. INL w tym przypadku jest dodatni. Ujemne INL wskazują, że rzeczywista krzywa znajduje się poniżej krzywej idealnej. Podsumowując, *rozkład* nieliniowości różniczkowej określa nieliniowość całkową przetwornika.



Rys. 2.3: Przykład wyznaczania nieliniowości całkowej [2]

2.2.4 Błąd niezrównoważenia

Znany też jako błąd przesunięcia zera; jest określany (dla A/C) przez wartość napięcia wejściowego potrzebną do przejścia od zerowej wartości słowa wyjściowego do następnej większej wartości, lub (dla C/A) wartość napięcia wyjściowego odpowiadającą zerowej wartości słowa wejściowego. Mierzony jako przesunięcie w stosunku do charakterystyki idealnej, tj. $\frac{1}{2}$LSB. W większości nowoczesnych przetworników jest jednak możliwa całkowita kompensacja tego błędu.

2.2.5 Błąd kwantyzacji

W przetwornikach A/C, błąd kwantyzacji jest wprowadzony przez proces kwantyzacji, obecny nawet w idealnym urządzeniu. Ze względu na ograniczoną długość słowa, wartości ciągłe są zamieniane na zbiór kodów o ograniczonej liczebności. Błąd ten jest zależny od przebiegu sygnału, do tego nie liniowo od wartości napięcia. W idealnym przetworniku A/C, błąd jest równomiernie rozłożony w przedziale $\pm\frac{1}{2}$ LSB (wartość bezwzględna błędu

jest mniejsza niż pół rozdzielczości). Dla sygnału o wartościach rozłożonych równomiernie w zakresie przetwarzania, stosunek mocy sygnału do zakłócenia kwantyzacji (ang. *signal-to-quantization-noise ratio*) jest równy *Dynamic Range*:

$$\mathrm{SQNR} = \mathrm{DR}$$

2.2.6 Współczynnik zawartości harmonicznych

Współczynnik zawartości harmonicznych i szum (ang. *THD+N – Total harmonic distortion and noise*) określa zawartość zniekształceń i zakłóceń wprowadzonych do sygnału przez przetwornik C/A. Jest wyrażony jako stosunek sumy mocy niechcianych składowych harmonicznych oraz szumu do mocy sygnału pożądanego.

2.3 Techniki pomagające poprawić jakość

2.3.1 Dither

W przetwornikach `A/C`, jakość może być zazwyczaj poprawiona przez tzw. *dithering*. Jest to proces dodania bardzo słabego szumu (np. białego) do mierzonego sygnału. Pozwala to rozłożyć błąd kwantyzacji z przebiegu „piłokształtnego" na dużo bardziej stały. Poprawia też precyzję pomiaru ciągłego, gdyż zamiast zaokrąglania w jedną stronę, przez odchyły może powodować przeskok bitów, a więc przy uśrednianiu w czasie – lepszą precyzję. Niestety w zamian delikatnie wzmacnia zakłócenia. Proces ten jest używany np. przy redukcji długości słowa (m.in. przy zmianie jakości obrazów) lub w układach całkujących, gdzie drobny błąd kwantyzacji mógłby się nawarstwiać w czasie.

2.3.2 Budowa przetworników

Niektóre przetworniki mają zewnętrzne napięcie odniesienia, zaś inne – wewnętrzne. Jest ono tak nazywane, ponieważ to w stosunku do jego wartości są porównywane próbki. W przypadku przetworników `A/C`, względem napięcia odniesienia mierzone są wartości napięcia na wejściu. W przypadku przetworników `C/A`, względem napięcia odniesienia produkowane są wartości wyjściowe. W obu przypadkach oznacza to, że napięcie odniesienia jest maksymalnym napięciem jakie przetwornik może zmierzyć lub wygenerować i jednocześnie odpowiada najwyższemu kodowi bitowemu.

Bardzo często napięcie odniesienia jest dobrane jako iloczyn o postaci

$$U_{ref} = 2^n \cdot 10^{-m}\ \mathrm{V}$$

gdzie $n, m \in \mathbb{N}$. Ma to na celu uproszczenie przeliczania z systemu dwójkowego na dziesiętny. Przykład: $n = 12$ i $m = -3$, więc $U_{ref} = 4.096\,\mathrm{V}$. Dla przetwornika 12-bitowego

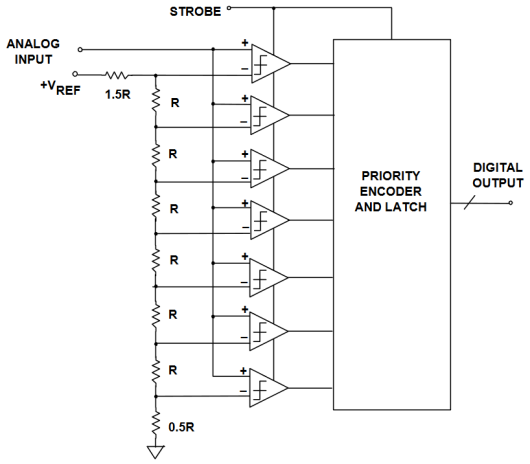
istnieje 4096 możliwych stanów, dlatego w idealnym urządzeniu osiągamy rozdzielczość 1 mV.

2.3.3 Budowa przetworników analogowo-cyfrowych

Ze względu na sposób działania wyróżnia się trzy metody pracy: bezpośrednia, pośrednia, kompensacyjna. Istnieje kilka podstawowych typów, które różnią się zasadą działania oraz osiągami. Stosują one jedną z powyższych metod. Dla każdego z nich mogą istnieć różne modyfikacje, które poprawiają w jakiś sposób osiągi.

Przetwornik błyskawiczny

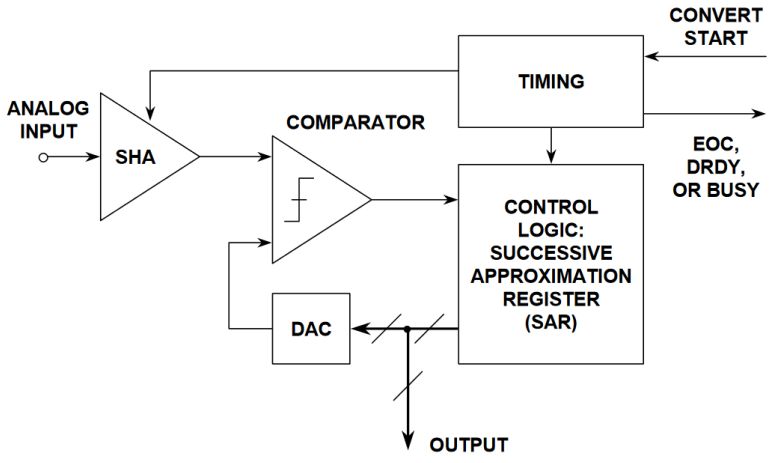
Przetwornik z rodzaju bezpośrednich, nazywany także błyskawicznym (ang. *Flash*). Działa na zasadzie bezpośredniego i (najczęściej) jednoczesnego porównania wartości napięcia wejściowego z szeregiem napięć odniesienia reprezentujących poszczególne poziomy kwantowania za pomocą szeregu komparatorów analogowych. Rezultatem tego porównania jest kod jedynkowy (ang. *unary*) zwany też termometrycznym (ang. *thermometer code*) ze względu na działanie analogiczne do termometru cieczowego – interesuje nas najwyższa osiągnięta pozycja, czyli ilość pozytywnych wyników porównań. Kod taki wprowadzany jest na specjalny koder, który wyprowadza wartość cyfrową (dwójkową) odpowiadającą sygnałowi wejściowemu. Podstawową zaletą takich przetworników jest szybkość działania, na którą składają się wyłącznie dwa czynniki: opóźnienie na komparatorze analogowym oraz opóźnienie na koderze cyfrowym. Uzyskiwane częstotliwości są nawet do kilku rzędów wielkości większe od pozostałych typów przetworników `A/C`. Niestety, ze względu na potrzebę zastosowania równoległego pomiaru dla każdej możliwej wartości, liczba elementów podwaja się z każdym bitem. Zwiększenie rozdzielczości wymaga też zwiększenia precyzyjności napięć odniesienia, uzyskiwanych zazwyczaj z dzielnika oporowego napięcia odniesienia. Zwiększenie ilości komparatorów skutkuje dodatkowo zwiększeniem pojemności wejściowej, a więc ograniczeniem pasma wejściowego sygnału, co jest sprzeczne z główną zaletą. Podsumowując, najczęściej mają one nie więcej niż 8 bitów rozdzielczości.



Rys. 2.4: Schemat przetwornika *Flash* [4]

Przetwornik sukcesywnej aproksymacji

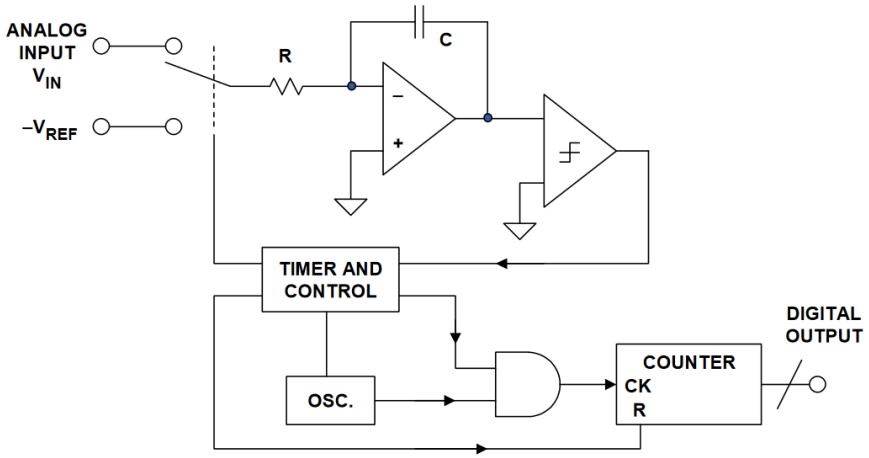
Przetwornik z sukcesywną aproksymacją (ang. *SAR – Successive Approximation Register*) należy do rodziny kompensacyjnych. Porównuje on wartości napięcia wejściowego z napięciem porównania wytworzonym za pomocą wewnętrznego przetwornika `C/A` w iteracyjnym procesie zarządzanym przez układ sterujący. Algorytm działa na zasadzie wyszukiwania binarnego, tzn. polega na włączaniu kolejnych bitów słowa, poczynając od najwyższego bitu (`MSB`), aż do osiągnięcia ostatniego bitu (`LSB`). W przypadku kiedy napięcie wejściowe będzie mniejsze od napięcia porównania, to dany bit słowa jest resetowany; w przeciwnym razie pozostawiana jest wartość „1”. Następnie wykonywana jest kolejna iteracja algorytmu. Tak wytworzone słowo jest reprezentacją cyfrową napięcia wejściowego. Ze względu na iteracyjny charakter pracy przetwornika, jego częstotliwość próbkowania jest zależna od długości słowa, szybkości pracy przetwornika `C/A`, komparatora i układu sterującego. Mimo tego, są one uważane za najbardziej uniwersalne, będąc średnimi zarówno w prędkości jak i dokładności.



Rys. 2.5: Schemat przetwornika `SAR` [4]

Przetwornik podwójnie całkujący

Metoda podwójnego całkowania (ang. *Dual Slope*) jest jednym z najdokładniejszych sposobów na pomiar wartości analogowej. Należy ona do metod pośrednich. W pierwszej części, do elementu całkującego doprowadzone jest napięcie mierzone. Całkowanie tego napięcia trwa ustalony czas (najczęściej 20 ms). W drugiej fazie do wejścia elementu całkującego dołączone jest napięcie odniesienia o biegunowości przeciwnej do wejścia. W czasie rozładowywania, układ zlicza impulsy zegara. Kiedy napięcie wyjściowe z całkowacza osiągnie wartość zero, układ kończy zliczanie impulsów. Znając czas ładowania, czas rozładowania, oraz napięcie rozładowujące, z prostej proporcji wyliczane jest napięcie wejściowe. Bardziej zaawansowane konstrukcje posiadają napięcia odniesienia o obu polaryzacjach lub nawet różnych wartościach (np. 10x) dla danej polaryzacji. Pozwala to przyspieszyć rozładowywanie układu całkującego. Ze względu na użycie kładu całkującego, zegarów i innych elementów bazujących na upływie czasu, dokładność przetwornika jest uzyskana kosztem częstotliwości przetwarzania.

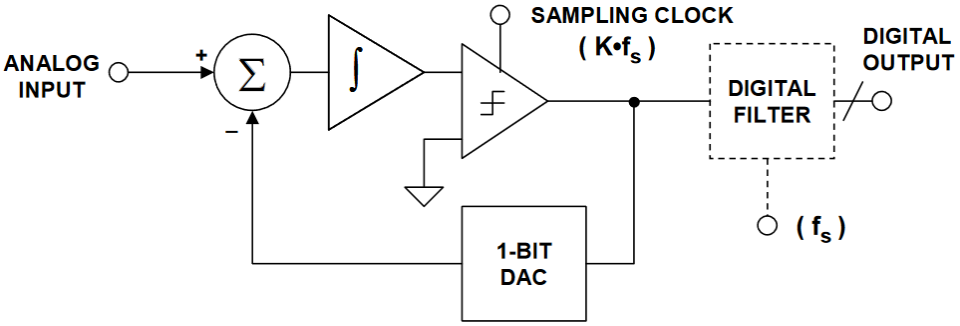


Rys. 2.6: Schemat przetwornika *Dual Slope* [4]

Przetwornik Sigma-Delta

Topologia ta jest jedną z najbardziej skomplikowanych [10]. Napięcie wejściowe jest podane na element całkujący, którego wyjście jest porównywane do jakiegoś niezerowego napięcia granicznego. Po przekroczeniu tego progu, załącza się bramka, która powoduje pomniejszenie napięcia wejściowego o napięcie odniesienia. Zakładając, że napięcie wejściowe zawsze będzie mniejsze od odniesienia, powoduje to zmianę polaryzacji oraz początek rozładowywania całkowacza. Wyjście bramki stanowi sygnał cyfrowy – ciąg bitów będący modulacją gęstości impulsów (ang. *PDM – Pulse-Density Modulation*). Jest ona koncepcyjnie zbliżona do modulacji szerokością impulsów (ang. *PWM – Pulse-Width Modulation*), jednak nie ma stałego okresu. Następnie zlicza się w ustalonym okresie czasu ilość bitów w stanie wysokim. Ta ilość w stosunku do ilości wszystkich bitów jest propor-

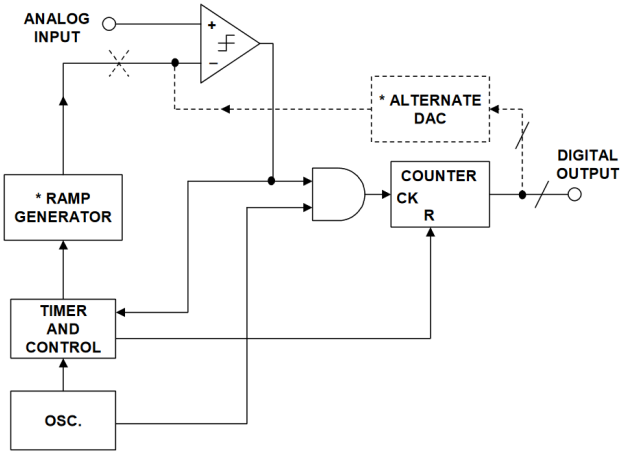
cjonalna do stosunku napięcia wejściowego do napięcia odniesienia. Z prostej proporcji można wyznaczyć wartość napięcia na wejściu. Przetworniki tego typu, ze względu na element całkujący uśredniający wpływ zakłóceń, oferują nieosiągalną dla innych precyzję, przy zachowaniu względnie wysokich częstotliwości. Niesie to jednak ze sobą koszty produkcji.



Rys. 2.7: Schemat przetwornika ΣΔ [4]. Symulacja

Przetwornik napięcia na czas

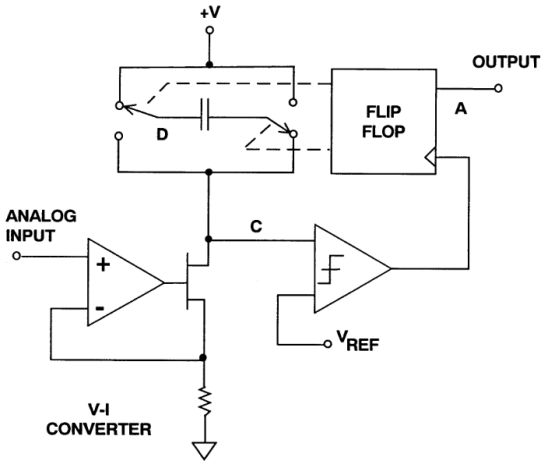
Metoda „napięcie na czas” (ang. *Ramp-Compare*) polega na generowaniu napięcia piłokształtnego o przebiegu $0 \div U_{ref}$ lub $\pm U_{ref}$ a następnie wprowadzeniu go razem z napięciem mierzonym na wejście komparatora. W ten sposób otrzymujemy ciąg bitów reprezentujący modulację szerokością impulsów (ang. `PWM` – *Pulse-Width Modulation*). Wystarczy z pomocą szybkiego zegara zmierzyć jaka część okresu składa się z bitów w stanie wysokim lub niskim, a następnie z prostej proporcji wyznaczyć napięcie wejściowe. Do zalet tego typu przetwornika należy bardzo nieskomplikowana konstrukcja; do wad zaś duża zależność od precyzji przebiegu piłokształtnego, który w najprostszym przypadku jest wytwarzany przez prosty element całkujący, na który bardzo duży wpływ może mieć np. temperatura. Z tego powodu zamiast analogowego generatora, można zastosować przetwornik `C/A`, którego napięcie wyjściowe wzrasta wraz z przebiegiem zegara.



Rys. 2.8: Schemat przetwornika *Ramp-Compare* [4]

Przetwornik napięcie-częstotliwość

Metoda „napięcie na częstotliwość" (ang. *Voltage-to-Frequency*) wykorzystuje przetwornik napięcia na częstotliwość (`VFC`). Jest to oscylator, którego częstotliwość jest liniowo proporcjonalna do napięcia sterującego. Impulsy są następnie zliczane w jakimś okresie czasu i przeliczane na napięcie na podstawie charakterystyki `F(U)`. Ten typ przetwornika jest monotoniczny, wolny od brakujących kodów, całkuje szum i może zużywać bardzo mało energii. Jest również bardzo przydatny w zastosowaniach telemetrycznych, ponieważ `VFC`, który jest mały, tani i zużywa niewiele energii, można zamontować na obiekcie doświadczalnym i komunikować się z licznikiem za pomocą łącza telemetrycznego.



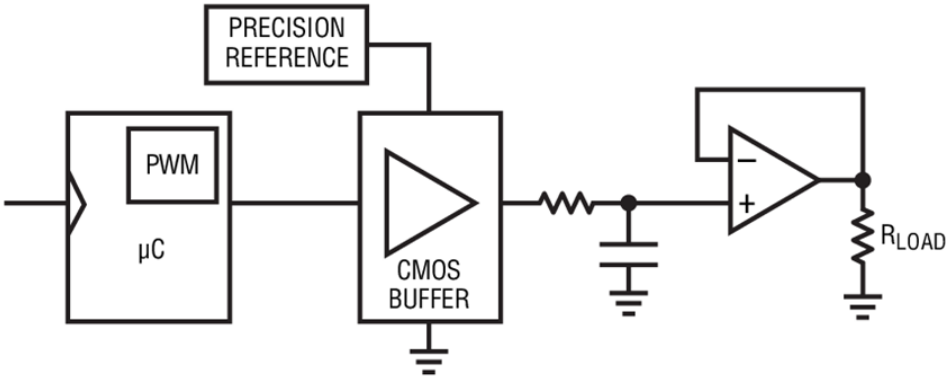
Rys. 2.9: Schemat przetwornika *Voltage-to-Frequency* [4]

2.3.4 Budowa przetworników cyfrowo-analogowych

Istnieje kilka podstawowych typów, które różnią się zasadą działania oraz osiągami. Często wykorzystują one metody analogiczne do przetworników A/C [4].

Przetwornik z modulatorem szerokości impulsów

Przetwornik działa dzięki szeroko znanej metodzie, polegającej na zmianie wypełnienia sygnału – jest nią wspomniane wcześniej PWM. Modulacja szerokości impulsów jest najczęściej wykonywana poprzez przełączanie tranzystorów pomiędzy stanem przewodzenia a stanem zaporowym. Kontrola takiego procesu jest też bardzo prosta w implementacji, sterownik musi tylko w odpowiednim momencie włączać lub wyłączać elementy kluczujące. W stanie zaporowym natężenie prądu jest zerowe, zaś w stanie przewodzenia spadek napięcia jest bardzo niewielki, dlatego straty praktycznie nie występują. Przez prostotę realizacji oraz bardzo wysoką wydajność, jest to metoda często stosowana do regulacji nie tyle napięcia, co mocy. Ze względu na pracę przerywaną, najlepiej działa w połączeniu z odbiornikami które z natury są inercyjne: silniki, wentylatory, pompy (moment bezwładności), żarówki, grzałki (bezwładność cieplna), zasilacze impulsowe (prąd w dławiku). Przy wystarczająco wysokiej częstotliwości przełączania, może jednak obsługiwać sterowanie mocą w układach uznawanych za statyczne, np. oświetlenie LEDowe. W zastosowaniach bardziej „typowych" dla C/A – czyli wytwarzanie precyzyjnego napięcia – oprócz wysokiej częstotliwości przełączania wymagane są analogowe filtry dolnoprzepustowe w celu wygładzenia przebiegu.



Rys. 2.10: Schemat przetwornika PWM

Przetwornik z modulatorem gęstości impulsów

Przetwornik tego typu bazuje na metodzie PDM (ang. *Pulse-Density Modulation*). Jest on odpowiednikiem przetwornika A/C $\Sigma\Delta$. Działanie jest podobne do PWM, oferuje jednak mniejsze zakłócenia na wyjściu w zamian za zwiększenie częstotliwości przełączania, a więc ewentualnych strat mocy. Wynika to z tego, że PWM ma stały okres i wszystkie bity

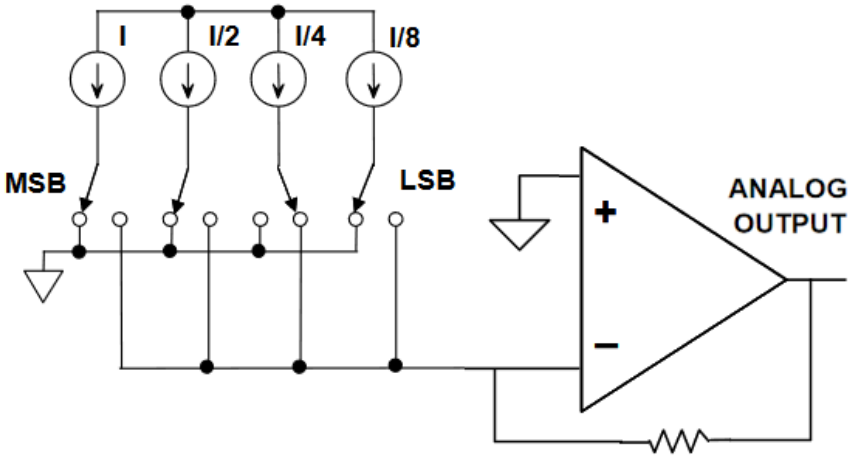
w stanie wysokim lub niskim są w takim okresie zgrupowane razem.

Przetwornik ważony dwójkowo

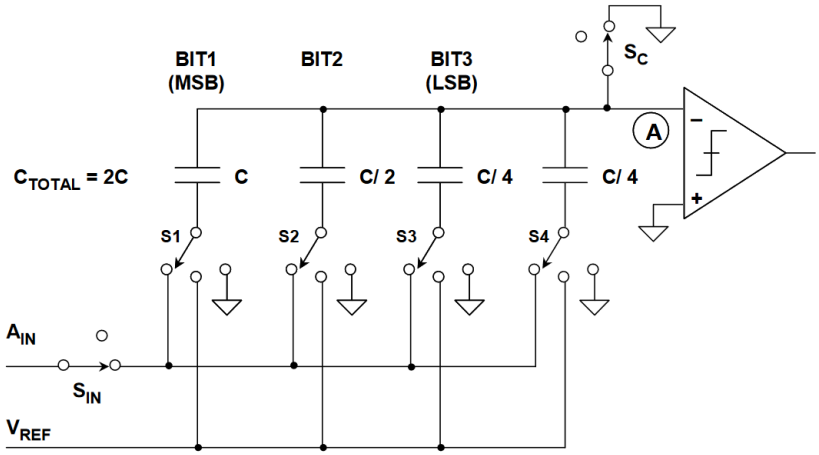
Przetwornik ważony binarnie (ang. *binary-weighted*) może być zbudowany na kilka sposobów. Mają one jednak wspólną cechę – dla każdego bitu słowa zawiera pewien zestaw elementów, które są dobrane tak, że każdy ma w węźle sumacyjnym dwa razy większą wagę od poprzedniego, zaczynając od najniższego bitu. Istnieje kilka wariantów, różniących się implementacją:

1. Napięcie odniesienia podane jest na zestaw rezystorów o podwajających się wartościach (tak więc prąd maleje dwukrotnie). Każdy z nich może być dołączony lub nie do wspólnego punktu. Sumaryczny prąd jest następnie zamieniany na napięcie. Opcjonalnie zamiast rezystorów można zastosować odpowiednio wyskalowane źródła prądowe.
2. Zamiast źródeł prądowych można wykorzystać kondensatory. Po naładowaniu napięciem odniesienia, wybrane z nich są przełączane do masy, co skutkuje pojemnościowym dzielnikiem napięcia.
3. Jedno źródło prądowe zasila zespół oporników połączony szeregowo, przy czym każdy może być indywidualnie zwarty – suma spadków napięć daje bezpośrednio wyjście.
4. Zamiast rezystorów o wartościach podwajających się, wykorzystano tzw. drabinkę `R-2R`. Ze względu na szczególną strukturę, prąd płynący do każdej kolejnej odnogi staje się dwukrotnie mniejszy. Zależnie od ułożenia, otrzymujemy wyjście napięciowe na zasadzie dzielnika oporowego, lub po zsumowaniu odpowiednich odnóg, wyjście prądowe, zamieniane na napięcie.

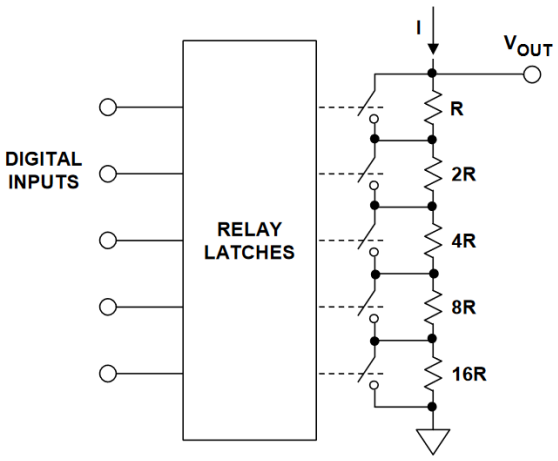
Jest to jedna z najszybszych metod konwersji, jednak wadą jest niska dokładność ze względu na trudność w wyprodukowaniu komponentów o precyzyjnych wartościach. Wariant drabinkowy częściowo pozwala zapobiec temu problemowi. Oporniki o tym samym rzędzie wielkości są dużo łatwiejsze do dokładnego wyprodukowania, a potrzeba ich jedynie dwa razy więcej niż w podstawowej topologii. Wersja z kondensatorami jest używana w przetwornikach `A/C SAR`, gdzie szybkie rozładowanie nie jest problemem, oraz ze względu na brak potrzeby dodatkowych układów *sample and hold*. Dodatkowo, na jednym układzie wykonuje się i konwersję `A/C` i `C/A`, co znacznie upraszcza strukturę.



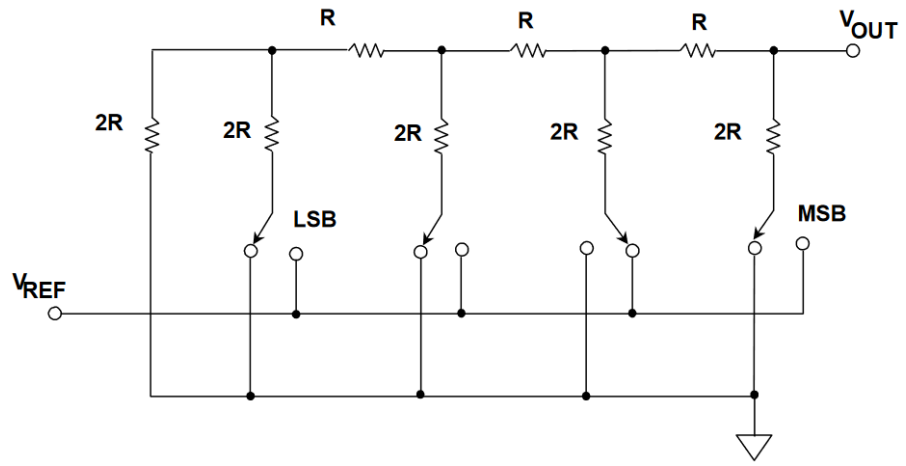
Rys. 2.11: Schemat przetwornika ważonego dwójkowo, rezystorowy [4]



Rys. 2.12: Schemat przetwornika ważonego dwójkowo, kondensatorowy [4]



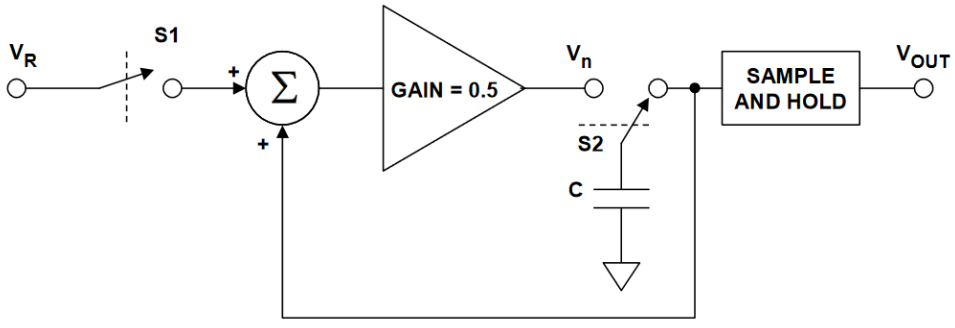
Rys. 2.13: Schemat przetwornika ważonego dwójkowo, napięciowy [4]



Rys. 2.14: Schemat przetwornika ważonego dwójkowo, drabinka R-2R [4]

Przetwornik cykliczny

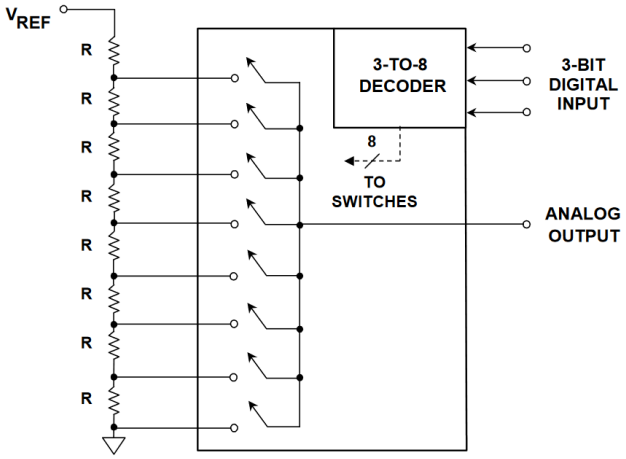
Przetwornik cykliczny, zwany też przetwornikiem sukcesywnej aproksymacji, jest odpowiednikiem przetwornika `A/C SAR`. Nazwa bierze się ze względu na to, że bity podawane są pojedynczo, a więc posiada on jeden element kluczujący. Zależnie czy bity podawane są od najniższego czy najwyższego, wymagane są lekkie zmiany, natomiast zasada działania jest ta sama. Przetwornik działa na zasadzie pętli dodatniego sprzężenia zwrotnego. Zależnie od wartości bitu, jedno z wejść sumatora jest podłączane do napięcia odniesienia lub masy, przepuszczane przez odpowiednie wzmocnienie, wraca do drugiego wejścia sumatora i proces powtarza się, dodając napięcie ustawione przez kolejny bit. W przypadku kolejności `LSB`→ `MSB`, wzmocnienie powinno wynosić 0.5; w przeciwnym przypadku, jest ono równe 2 [4]. Jak łatwo się domyślić, w praktyce rozpoczęcie od najniższego bitu będzie dawać bardziej dokładne rezultaty, ze względu na to, że najbardziej znaczący bit nie będzie wielokrotnie przechodził przez pętlę, przy okazji akumulując błąd. Mnożenie napięcia odniesienia przez 0.5 i dodawanie gwarantuje też, że wynik zawsze znajdzie się w przedziale $0 \div U_{ref}$. Mnożenie przez 2 wymaga albo późniejszego przeskalowania, albo odpowiedniego pomniejszenia napięcia odniesienia (co znów powoduje niedokładności).



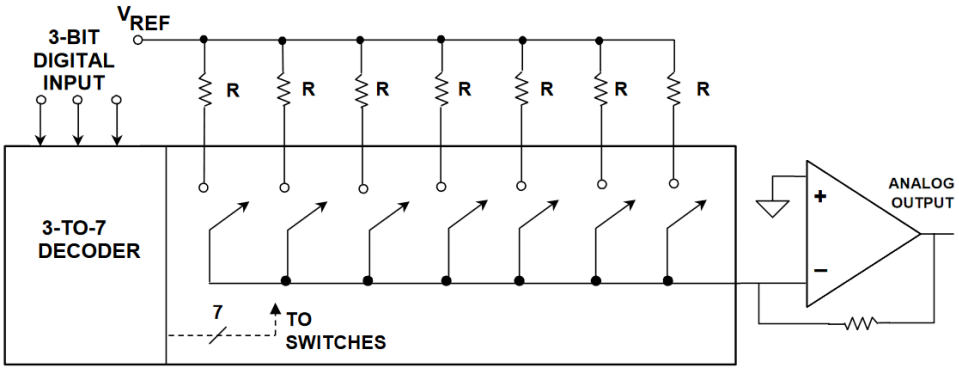
Rys. 2.15: Schemat przetwornika cyklicznego [4]

Przetwornik termometryczny

Przetwornik termometryczny jest odpowiednikiem przetwornika `A/C` Flash. Zawiera on tyle zestawów elementów ile możliwych wartości. Mogą to być rezystory, źródła napięciowe lub źródła prądowe o tej samej wartości. Wartość binarna jest przeliczona na kod unarny. W przypadku źródeł napięciowych (np. oporowy dzielnik napięcia), wybierane jest najwyższe i odpowiada ono bezpośrednio wyjściu. W przypadku źródeł prądowych (np. zbiór równoległych identycznych rezystorów) powoduje włączenie dokładnie takiej ilości elementów jaka jest podana w słowie, zaś później prąd zamieniany jest na napięcie. Architektura tego typu jest szybka i najdokładniejsza ze wszystkich. Niestety kosztem są duże wymogi pod względem ilości komponentów, co ogranicza rozdzielczość lub wymaga procesu produkcji układów scalonych wysokiej skali integracji (ang. *high-density IC*).



Rys. 2.16: Schemat przetwornika termometrycznego, napięciowy [4]



Rys. 2.17: Schemat przetwornika termometrycznego, prądowy [4]

Przetwornik hybrydowy

Przetworniki hybrydowe, jak nazwa wskazuje, łączą wcześniejsze metody w jednym urządzeniu. Większość obecnie produkowanych układów jest właśnie tego typu, ze względu

na łączenie pożądanych cech (niski koszt, wysoka częstotliwość, dobra precyzja). Jednym z przykładów jest przetwornik „segmentowy", który łączy metodę termometryczną na najwyższych bitach, dla zwiększonej precyzji, z metodą ważoną dwójkowo dla niższych bitów, w celu zmniejszenia ilości potrzebnych komponentów.

2.4 Komunikacja z peryferiami

Częścią praktycznie każdego bardziej skomplikowanego układu cyfrowego jest procesor, który steruje pracą całości bądź części układu. Procesor musi mieć możliwość komunikacji z każdą istotniejszą częścią takiego układu.

Istnieją dwa podejścia do podłączenia urządzeń do procesora:

- ciasna integracja: urządzenie ma swój dedykowany interfejs i staje się w pewnym sensie częścią specyfikacji procesora (np. dedykowane instrukcje czy specjalne rejestry procesora do obsługi danego urządzenia)
- luźna integracja: urządzenie jest podłączone do procesora przez standardową szynę (zazwyczaj współdzieloną z innymi urządzeniami) i jest widoczne dla procesora za pomocą zwykłych instrukcji dostępu do pamięci — pewna część fizycznej przestrzeni adresowej jest wydzielona dla urządzenia i odpowiada ono na instrukcje procesora, które piszą bądź czytają ten obszar; podejście takie nazywa się MMIO (memory-mapped I/O)

Ciasna integracja jest używana rzadko (najczęściej gdy projektujemy jednocześnie procesor i jego urządzenia peryferyjne) i w przypadku procesorów ogólnego przeznaczenia używa się wyłącznie luźnej integracji.

Do podłączania zewnętrznych modułów najczęściej wykorzystywane są ustandaryzowane protokoły komunikacyjne, których obsługa jest często zaimplementowana w procesorach za pomocą wyżej wymienionych integracji. Kilka takich najpopularniejszych protokołów jest wymienionych poniżej.

2.4.1 Układ UART

Uniwersalny asynchroniczny nadajnik-odbiornik (ang. *Universal Asynchronous Receiver-Transmitter*) to interfejs do asynchronicznej komunikacji szeregowej, w którym można konfigurować format danych i prędkość transmisji. `UART` pobiera bajty danych i przesyła poszczególne bity w sposób sekwencyjny, od najmniej do najbardziej znaczącego, otoczone bitami startu i stopu, dzięki czemu kanał komunikacyjny zapewnia precyzyjne taktowanie. W miejscu docelowym drugi `UART` ponownie składa bity w kompletne bajty. Każdy `UART` zawiera rejestr przesuwny, który jest podstawową metodą konwersji pomiędzy formami szeregowymi i równoległymi [30]. Transmisję informacji złączem szeregowym inicjuje bit

startowy (logiczne zero). W dalszej kolejności przesłana zostaje informacja w postaci 7, 8 lub 9 kolejnych bitów (w zależności od ustalonej konfiguracji urządzenia). Za zakończenie transmisji odpowiedzialny jest bit stopu (logiczna jedynka). Całość tworzy tzw. ramkę `UART`, która zawiera w sobie kompletną przesyłaną informację.

2.4.2 Magistrala I2C

`I2C` lub `IIC` (ang. *Inter-Integrated Circuit*) to synchroniczna magistrala komunikacyjna, opracowana w 1982 roku przez firmę Philips Semiconductors. Jest szeroko stosowana do podłączania układów peryferyjnych o niższej prędkości do procesorów i mikrokontrolerów w komunikacji wewnątrzpłytowej na małe odległości. Jest to protokół typu *master-slave*, co oznacza, że jedno urządzenie kontrolujące (*master*) komunikuje się z jednym lub wieloma urządzeniami podrzędnymi (*slaves*).

`I2C` do transmisji wykorzystuje dwie dwukierunkowe linie w topologii magistrali: `SDA` – linia danych (ang. *Serial DAta*) i `SCL` – linia zegara (ang. *Serial CLock*). Obie linie są na stałe podłączone do źródła zasilania poprzez rezystory podciągające (ang. *pull-up*). Wszystkie nadajniki są typu otwarty kolektor lub otwarty dren, co powoduje, że logiczne zero jest dominujące. I2C używa logiki dodatniej, a więc stan niski na magistrali odpowiada „0” logicznemu, natomiast stan wysoki „1” logicznej [35].

`I2C` jest magistralą zorientowaną bajtowo, a więc bity grupowane są po 8. Dane są wysyłane w kolejności od najbardziej znaczącego bitu do najmniej znaczącego. Po przesłaniu 8 bitów w jednym kierunku, przesyłany jest dodatkowy bit potwierdzenia odebrania danych `ACK` (lub `NACK` w przypadku braku potwierdzenia) w kierunku przeciwnym.

Pierwszym bajtem jest zawsze nadawany przez urządzenie master adres urządzenia slave, który oprócz 7 bitów właściwego adresu zawiera bit kierunku transmisji na najniższej pozycji. Wartość „0” tego bitu oznacza transmisję od mastera do slave'a (zapis), podczas gdy wartość „1” kierunek przeciwny (odczyt). Po pierwszym bajcie przesyłane zostają dane.

2.4.3 Interfejs SPI

`SPI` (ang. *Serial Peripheral Interface*) – szeregowy interfejs urządzeń peryferyjnych. Jeden z najczęściej używanych interfejsów komunikacyjnych pomiędzy systemami mikroprocesorowymi a układami peryferyjnymi.

Komunikacja poprzez SPI odbywa się synchronicznie za pomocą 3 linii:

- `MOSI` (ang. *master output slave input*) – dane dla układu peryferyjnego,
- `MISO` (ang. *master input slave output*) – dane z układu peryferyjnego,
- `SCLK` (ang. *serial clock*) – sygnał zegarowy (taktujący).

Do aktywacji wybranego układu peryferyjnego służy dodatkowo linia CS (ang. *Chip Select* — wybór układu podrzędnego) lub adresacja układów. W drugim przypadku, w przesyłanej wiadomości zawarty musi być adres urządzenia, które po jego rozpoznaniu przyjmuje pozostałe bajty. Adresowanie układów wykorzystywane jest szczególnie podczas pracy z rozbudowanymi systemami, których poszczególne części można programować niezależnie, także po zamontowaniu na płytce [6].

Aby rozpocząć komunikację, master najpierw wybiera urządzenie podrzędne. Podczas każdego cyklu zegara SPI następuje transmisja pojedynczego bitu w trybie pełnego dupleksu. Master wysyła bit na linii MOSI, podczas gdy slave wysyła bit na linii MISO, a następnie każdy odczytuje odpowiadający im bit przychodzący. Ta kolejność jest zachowywana nawet wtedy, gdy zamierzony jest tylko jednokierunkowy transfer danych.

2.5 Połączenie między urządzeniami

Do komunikacji pomiędzy samodzielnymi urządzeniami, głównie ze względu na typowo dużo większy dystans niż między peryferiami, stosuje się inne metody. Wymagają one adekwatnej mocy, szczególnie przy wyższych prędkościach przesyłu, oraz odpowiednich zabezpieczeń przed błędami związanymi z możliwymi zakłóceniami. Kilka możliwych rozwiązań zostało omówionych poniżej, zarówno przewodowych jak i zdalnych.

2.5.1 Łącze RS-232

RS-232 to standard szeregowej transmisji danych między urządzeniami elektronicznymi. Opisuje sposób połączenia urządzeń końcowych danych (np. komputer) oraz urządzeń komunikacji danych (np. modem). Określa nazwy styków złącza oraz przypisane im sygnały, a także specyfikację elektryczną obwodów wewnętrznych. Standard ten definiuje normy wtyczek i przewodów portów szeregowych typu COM.

RS-232 jest magistralą komunikacyjną przeznaczoną do szeregowej transmisji danych. Najbardziej popularna wersja tego standardu, RS-232C pozwala na transfer na odległość nieprzekraczającą 15 m z szybkością maksymalną 20 $\frac{\text{kbit}}{\text{s}}$.

2.5.2 Łącze RS-485

RS-485 składa się z różnicowego (symetrycznego) nadajnika, dwuprzewodowego toru transmisyjnego i różnicowego odbiornika. Umożliwia podłączenie wielu nadajników i odbiorników (maksymalnie do 32). Ograniczenie wynika z ograniczeń energetycznych nadajnika. Najczęściej stosowaną topologią dla takich standardów jest topologia magistrali. Zasięg to około 1200 m. Prędkości transmisji jakie można uzyskać to 35 $\frac{\text{Mbit}}{\text{s}}$ (do 10 m), i 100 $\frac{\text{kbit}}{\text{s}}$ (do 1200 m). RS-485 jest najczęściej stosowanym interfejsem przewodowym w

sieciach przemysłowych – z prostego powodu, ponieważ przesył różnicowy zapobiega wpływowi zakłóceń zewnętrznych na transmisję danych. Na bazie tego interfejsu opracowano wiele protokołów komunikacyjnych.

2.5.3 Magistrala USB

USB, uniwersalna magistrala szeregowa (ang. *Universal Serial Bus*) -– komputerowe złącze komunikacyjne (tak zwany port lub interfejs) zastępujące stare porty szeregowe i porty równoległe.

Jedną z ważniejszych cech portu USB jest zgodność ze standardem *Plug and Play*. Architektura USB składa się z serwera (hosta), wielu portów USB oraz urządzeń do nich podłączonych. Host USB może zarządzać wieloma kontrolerami, a każdy kontroler może udostępniać jeden lub więcej portów USB. Urządzenia można z sobą łączyć, tworząc sieć o topologii drzewa wykorzystując do tego koncentratory USB [33].

Współcześnie podstawowy wariant oferuje przepustowość rzędu 480 $\frac{\text{Mbit}}{\text{s}}$. Transmisja danych przez port odbywa się w trybie *half duplex* na jednej parze przewodów. Istnieją nowsze, szybsze warianty, jednak nie wszystkie porty i nie wszystkie urządzenia je wspierają.

2.5.4 Standard Ethernet

Ethernet to rodzina technologii przewodowych sieci komputerowych, powszechnie stosowanych do połączeń internetowych. Został on wprowadzony na rynek w 1980 roku; od tego czasu został udoskonalony, aby obsługiwać wyższe przepływności, większą liczbę węzłów i większe odległości łączy, ale zachowuje znaczną kompatybilność wsteczną.

Klasyczne sieci Ethernet mają cztery cechy wspólne. Są to: format ramki, parametry czasowe, podstawowe reguły obowiązujące przy ich projektowaniu, proces transmisji. Standardem jest izolacja o wytrzymałości minimum 250 V AC między kablem a komputerem. Kabel jest z założenia odizolowany galwanicznie; to znaczy moduły służące do obsługi Ethernetu posiadają zintegrowane swego rodzaju mini transformatory na każdej parze żył. Przez to można łączyć układy o różnicy potencjałów nawet rzędu kilku kV.

Stacje Ethernet komunikują się, wysyłając sobie nawzajem pakiety danych: bloki danych indywidualnie wysyłane i dostarczane. Adaptery są dostarczane z zaprogramowanym globalnie unikalnym 48-bitowym adresem MAC, dzięki czemu każda stacja Ethernet ma unikalny adres. Adresy te służą do określenia zarówno miejsca docelowego, jak i źródła każdego pakietu danych. Ethernet ustanawia połączenia (*link-layer*), które można zdefiniować przy użyciu adresu docelowego oraz źródłowego. Po odebraniu transmisji odbiornik na podstawie adresu docelowego określa, czy transmisja ma znaczenie dla stacji, czy też powinna zostać zignorowana. Interfejs sieciowy zwykle nie akceptuje pakietów adresowanych do innych stacji Ethernet.

Warstwa fizyczna Ethernet ewoluowała przez znaczny okres czasu i obejmuje fizyczne interfejsy koncentryczne, skrętkowe i światłowodowe o prędkościach od 1 $\frac{\text{Mbit}}{\text{s}}$ do 400 $\frac{\text{Gbit}}{\text{s}}$.

Najczęściej stosowane formy to `10BASE-T`, `100BASE-TX` i `1000BASE-T`. Wszystkie trzy wykorzystują skrętkę komputerową i złącza modułowe `8P8C`. Działają odpowiednio z szybkością 10 $\frac{\text{Mbit}}{\text{s}}$, 100 $\frac{\text{Mbit}}{\text{s}}$ i 1 $\frac{\text{Gbit}}{\text{s}}$.

Datagram nazywany jest pakietem lub ramką. Pakiet służy do opisu całej jednostki transmisyjnej i zawiera nagłówek, ogranicznik początkowy ramki (ang. ***SFD - start frame delimiter***) i przedłużenie nośnika (ang. *carrier extension*) -– dopełnienie za ramką do 512 B. Ramka rozpoczyna się nagłówkiem ramki zawierającym źródłowy i docelowy adres `MAC`. Środkowa część ramki składa się z danych użytkowych, w tym wszelkich nagłówków innych protokołów (na przykład protokołu internetowego) znajdujących się w ramce. Ramka kończy się 32-bitowym cyklicznym kodem nadmiarowym, który służy do wykrywania uszkodzeń przesyłanych danych.

2.5.5 Standard Wi-Fi

`Wi-Fi` to rodzina protokołów sieci bezprzewodowych, które są powszechnie używane w sieciach lokalnych urządzeń i dostępie do Internetu, umożliwiając wymianę danych pobliskich urządzeń cyfrowych za pomocą fal radiowych. `Wi-Fi` jest zaprojektowane tak, aby bezproblemowo współpracować ze swoim przewodowym odpowiednikiem, Ethernetem.

Pasma radiowe `Wi-Fi` najlepiej sprawdzają się w przypadku korzystania „w zasięgu wzroku". Wiele typowych przeszkód, takich jak ściany, filary, urządzenia gospodarstwa domowego itp., może znacznie zmniejszyć zasięg, ale pomaga to również zminimalizować zakłócenia między różnymi sieciami w zatłoczonym otoczeniu. Zasięg wynosi od około 20 m do nawet 150 m. Oferuje prędkość przesyłu do 100 $\frac{\text{Mbit}}{\text{s}}$, lecz przy większych dystansach, prędkość przesyłu zazwyczaj spada.

Standard zapewnia kilka różnych zakresów częstotliwości radiowych do użytku w komunikacji `Wi-Fi`: pasma 900 MHz, 2.4 GHz, 3.6 GHz, 4.9 GHz, 5 GHz, 5.9 GHz i 60 GHz. Najczęściej używane są 2.4 GHz i 5 GHz.

Dane są zorganizowane w ramki, które są bardzo podobne do ramek Ethernet w warstwie łącza danych, ale z dodatkowymi polami adresowymi. Adresy `MAC` są używane jako adresy sieciowe do kierowania w sieci `LAN`.

2.5.6 Standard Bluetooth

Bluetooth to standard technologii bezprzewodowej krótkiego zasięgu, używany do wymiany danych między urządzeniami stacjonarnymi i mobilnymi na małe odległości oraz do budowania sieci osobistych (`PAN`). Korzysta z fal radiowych w paśmie częstotliwości 2.4 GHz i oferuje zasięg przeważnie do 10 m oraz prędkość przesyłu do około 50 $\frac{\text{Mbit}}{\text{s}}$.

2.6 Wybór sposobu komunikacji

Każdy sposób połączenia urządzenia z komputerem narzuca też jakiś specyficzny format komunikacji czy wiadomości. Jeśli zostanie użyty protokół internetowy (czy w postaci Ethernetu czy Wi-Fi), należy rozważyć sposób przesyłania wiadomości i danych między budowanym urządzeniem a urządzeniem zarządzającym. Kilka możliwych rozwiązań jest przedstawionych poniżej.

2.6.1 Gniazda (sockets)

Gniazdo sieciowe to struktura oprogramowania, która służy jako punkt końcowy do wysyłania i odbierania danych w sieci. Ze względu na standaryzację protokołów `TCP/IP` w rozwoju Internetu, termin gniazdo sieciowe jest najczęściej używany w kontekście zestawu protokołów internetowych i dlatego często nazywany jest także gniazdem internetowym. W tym kontekście gniazdo jest zewnętrznie identyfikowane przez inne hosty na podstawie adresu gniazda, który jest triadą protokołu transportowego, adresu IP i numeru portu.

Interfejs programowania aplikacji dla stosu protokołów sieciowych tworzy tzw. „uchwyt" (ang. *handle*) dla każdego gniazda utworzonego przez aplikację, powszechnie nazywany deskryptorem gniazda. W systemach operacyjnych typu `Unix` ten deskryptor jest rodzajem deskryptora pliku. Jest on przechowywany przez proces aplikacji i używany przy każdej operacji odczytu i zapisu w kanale komunikacyjnym.

W momencie tworzenia za pomocą `API` gniazdo sieciowe jest powiązane z kombinacją typu protokołu sieciowego, który będzie używany do transmisji, adresu sieciowego hosta i numeru portu. Porty służą jako adresowalny zewnętrznie (z sieci) komponent lokalizacji, dzięki czemu inne hosty mogą nawiązywać połączenia.

Aplikacja może komunikować się ze zdalnym procesem poprzez wymianę danych protokołem `TCP/IP`, znając kombinację typu protokołu, adresu `IP` i numeru portu. Ta kombinacja jest często nazywana adresem gniazda. Jest to skierowany w stronę sieci uchwyt dostępu do gniazda sieciowego. Zdalny proces ustanawia gniazdo sieciowe we własnej instancji stosu protokołów i wykorzystuje interfejs `API` sieci do łączenia się z aplikacją, przedstawiając własny adres gniazda do wykorzystania przez aplikację.

Interfejs, którego programy używają do komunikacji ze stosem protokołów przy użyciu gniazd sieciowych, nazywany jest *socket `API`*. Tworzenie programów użytkowych korzystających z tego interfejsu nazywa się programowaniem gniazdowym lub programowaniem sieciowym. Interfejsy `API` gniazd internetowych są zwykle oparte na standardzie gniazd Berkeley. W standardzie gniazd *Berkeley* gniazda są formą deskryptora pliku, ze względu na filozofię Uniksa, że "wszystko jest plikiem".

Gniazdo strumieniowe zapewnia niezawodny, sekwencyjny i unikalny przepływ bezbłędnych danych bez granic rekordów, z dobrze zdefiniowanymi mechanizmami tworzenia i niszczenia połączeń oraz raportowania błędów. W Internecie gniazda strumieniowe są

zwykle implementowane przy użyciu protokołu TCP, dzięki czemu aplikacje mogą działać w dowolnej sieci przy użyciu protokołu TCP/IP.

Programowanie bezpośrednio na gniazdach oferuje praktycznie nieograniczone możliwości, natomiast jednocześnie niskopoziomowość jest wadą, ze względu na konieczność samodzielnej obsługi bardzo wielu problemów. Po stronie klienta: najpierw trzeba otworzyć gniazdo (i pamiętać o zamknięciu przed wyjściem z programu). Następnie połączyć się z serwerem, używając odpowiedniej struktury przechowującej adres. Po tym możemy wysyłać i odbierać dane. Po stronie serwera: najpierw trzeba zarejestrować usługę w systemie oraz przejść w tryb nasłuchiwania. Następnie trwa oczekiwanie na połączenie, które finalnie zostaje odebrane. Tworzy to nowe gniazdo, przez które można wysyłać i odbierać dane (oczywiście tu również należy je później zamknąć). Obsługa działania, sprawdzanie poprawności, błędów (errno), szykowanie struktur danych, wszystko wymaga od programisty użycia sporej liczby funkcji [32]. Niektóre kody oznaczają błąd nieodwracalny, niektóre można obejść, zaś jeszcze inne są wręcz oczekiwane w pewnych sytuacjach. Różne ustawienia powodują pojawianie się nowych kodów błędów lub zmianę zachowań już istniejących. Dodatkowo, całkowicie po stronie użytkownika leży wybór i implementacja metody sprawdzenia, czy dane wysłane i otrzymane zostały całe, czy jeszcze nie, czy może nigdy już nie dotrą; dobór limitów czasowych i wiele, wiele innych.

Z wymienionych przyczyn, bardzo pożądany jest wyższy poziom abstrakcji, który pozwoli na zachowanie funkcjonalności przy jednoczesnym ukryciu wszystkich tych niepożądanych, zarówno przez programistę jak i użytkownika, cech i wymogów.

2.6.2 WebSocket

WebSocket jest protokołem komunikacyjnym, zapewniającym dwukierunkowy kanał wymiany danych poprzez jedno połączenie TCP. Zarówno WebSocket jak i HTTP są zlokalizowane na 7 warstwie w modelu OSI i zależą od TCP na warstwie 4. Pomimo faktu, że są one różne, WebSocket został zaprojektowany do działania na portach przypisanych do HTTP. Aby osiągnąć kompatybilność z HTTP, *handshake* WebSocket'u wykorzystuje nagłówek HTTP Upgrade, aby przełączyć komunikację.

Protokół WebSocket umożliwia interakcję między przeglądarką internetową (lub innym klientem), a serwerem sieciowym przy niższym obciążeniu niż alternatywne rozwiązania pół-dupleksowe, takie jak np. odpytywanie HTTP (ang. *polling*), ułatwiając przy tym znacznie przesyłanie danych w czasie rzeczywistym do i z serwera. Jest to możliwe dzięki zapewnieniu znormalizowanego sposobu wysyłania przez serwer treści do klienta bez uprzedniego żądania klienta i umożliwienia przesyłania komunikatów tam i z powrotem przy zachowaniu aktywnego połączenia. W ten sposób między klientem, a serwerem może odbywać się dwukierunkowa wymiana danych [36].

Niestety wymaga jednocześnie zwykłego serwera HTTP [11], dodatkowo pozostaje pro-

blem obsługi wielu równoległych połączeń do ograniczonego sprzętu.

2.6.3 Serwer HTTP

Po dokładniejszym przeanalizowaniu problemu, stało się jasne, iż połączenie w pełni dwukierunkowe, oferowane przez `WebSocket`, jest zbędne. Wystarczający jest prosty serwer `HTTP` [37].

Serwer `WWW` to oprogramowanie komputerowe (i sprzęt na którym ono pracuje), który przetwarza żądania za pośrednictwem protokołu `HTTP` (protokół sieciowy stworzony do dystrybucji treści internetowych) lub jego bezpiecznego wariantu `HTTPS`. Klient użytkownika – zwykle przeglądarka internetowa – inicjuje komunikację, wysyłając żądanie dotyczące strony internetowej lub innego zasobu za pomocą protokołu `HTTP`, a serwer odpowiada treścią tego zasobu lub komunikatem o błędzie. Serwer `WWW` może również akceptować i przechowywać zasoby wysyłane od klienta użytkownika, jeśli jest do tego skonfigurowany.

Program serwera `WWW` zwykle wykonuje kilka zadań: odczytuje i stosuje ustawienia znajdujące się w konfiguracji; rozpoczyna nasłuchiwanie połączeń/żądań klientów; zarządza połączeniami klientów; odbiera żądania: czyta i weryfikuje żądania `HTTP`, wykonuje tłumaczenie ścieżki `URL` wraz z różnymi kontrolami bezpieczeństwa, identyfikuje znane nagłówki i odczytuje ich wartości; wykonuje lub odrzuca żądaną metodę `HTTP`: zarządza komunikacją z programami zewnętrznymi/modułami wewnętrznymi służącymi do generowania treści dynamicznych, sprawdzając dostępność, rozpoczęcie i ostatecznie zatrzymanie wykonywania; odpowiada na żądania klientów wysyłając odpowiednie wiadomości `HTTP` (np. żądane zasoby lub komunikaty o błędach); opcjonalnie zapisuje komunikaty procesu o wykrytych anomaliach lub innych znaczących zdarzeniach (np. w żądaniach klientów lub w jego wewnętrznym funkcjonowaniu) [3].

Serwery obsługują kilka różnych tzw. metod: `GET`, `POST`, `PUT`, `HEAD`, `DELETE`, `PATCH`, `OPTIONS`, `CONNECT`, `TRACE`. Najpopularniejsze są jednak `GET` i `POST`, szczególnie w przeglądarkach internetowych.

Metoda żądania `GET` pobiera informacje z serwera. W ramach żądania do adresu `URL` można dodać *query string*, czyli jakieś dane w postaci par {nazwa, wartość}. Ich długość jest jednak ograniczona (maksymalna `URL` długość 2048 znaków) i muszą być w specjalny sposób (`application/x-www-form-urlencoded`) kodowane i dekodowane.

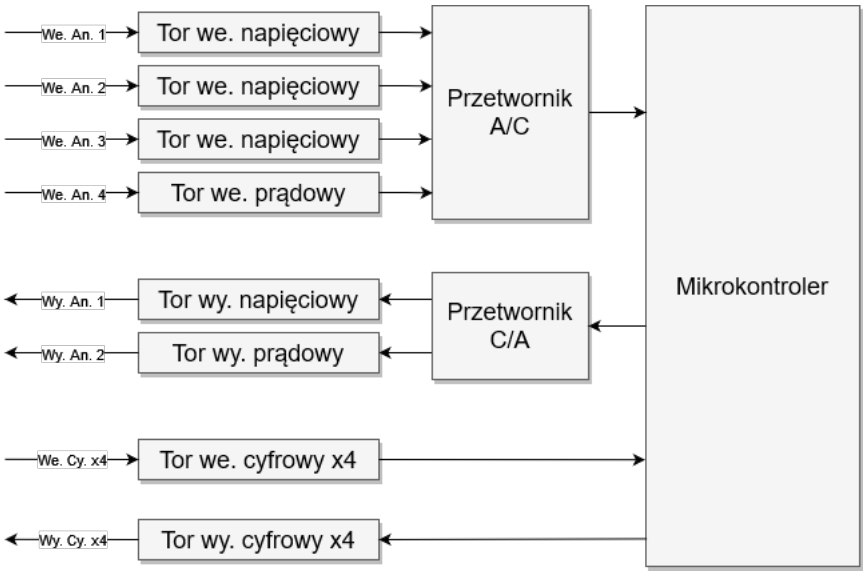
Z założenia metoda `POST` żąda, aby serwer `WWW` zaakceptował dane zawarte w treści żądania, w celu ich przetworzenia lub przechowania. Jest często używany podczas przesyłania pliku lub wypełnionego formularza. W ramach żądania `POST` w treści żądania można wysłać do serwera względnie dowolną ilość danych, o typie albo zgodnym z danymi metody powyżej, lub dowolnymi danymi binarnymi, za pomocą kodowania `multipart/form-data`. Pole nagłówka w żądaniu zwykle wskazuje typ kodowania tre-

ści wiadomości.

2.7 Koncepcja pracy

Koncepcja urządzenia przedstawiona jest na schemacie 2.18 – ogólny diagram pokazujący komponenty i połączenia między nimi.

Budowane urządzenie jest wyposażone w cztery wejścia analogowe: trzy napięciowe i jedno prądowe, oraz dwa wyjścia analogowe: jedno napięciowe i jedno prądowe. Oprócz tego, dostępne są również cztery wejścia i cztery wyjścia cyfrowe. Wszystkie porty analogowe obsługują zarówno dodatnie jak i ujemne wartości sygnałów. Wejścia cyfrowe działają z napięciem dodatnim 3.3 V - 24 V oraz są zabezpieczone przeciwko napięciom ujemnym. Wyjścia cyfrowe działają z napięciem do 50 V oraz są zabezpieczone przeciwko napięciom ujemnym i zbyt wysokim prądom. Wszystkie porty powinny móc działać (mierzyć, generować) z częstotliwością przynajmniej 50 kHz. Całość zarządzana jest poprzez mikrokontroler `ESP32`, z którym można się komunikować za pomocą komputera PC. Do operacji analogowych zostaną wykorzystane zewnętrzne chipy, które są podłączone do mikrokontrolera. Porty analogowe wyposażone są w złącza `BNC`, zaś porty cyfrowe w rozłączalne listwy zaciskowe. Wejścia analogowe mają do wyboru trzy możliwe przedziały. Do pomiarów wykorzystany jest przetwornik typu `SAR` ze względu na ich uniwersalność, tzn. przeciętne zarówno częstotliwość i rozdzielczość, oraz względną prostotę konstrukcji. Do wytwarzania napięcia wykorzystano przetwornik typu *resistive string*, czyli termometryczny, ze względu na bardzo szybkie działanie, niską nieliniowość różniczkową oraz gwarantowaną monotoniczność.



Rys. 2.18: Ogólny diagram urządzenia

2.7.1 Zadawanie zadań urządzeniu

Ponieważ jedyne bezpośrednie sterowanie urządzeniem jest możliwe poprzez tworzenie i wgrywanie kodu w języku `C++` – co wymaga jego dobrej znajomości oraz znajomości samego sprzętu; odpowiedniego komputera, oprogramowania, kompilatora, środowiska, bibliotek i połączenia fizycznego; jest czasochłonne i podatne na błędy – musiała zostać stworzona inna metoda sterowania urządzeniem, w celu wykonywania pożądanych pomiarów, ustawiania wyjść, itd.

Rozwiązaniem może być stworzenie własnego, prostego języka, zawierającego polecenia służące do sterowania podzespołami urządzenia, oraz opcjonalnie różne instrukcje pomocnicze, służące do zmiany przepływu sterowania, takie jak: skok, warunek, pętla. Oczywiście wymaga to również stworzenia interpretera, który będzie odczytywał instrukcje i wykonywał odpowiedni kod maszynowy obsługujący urządzenie [28].

Format binarny

Można założyć, że każde wyrażenie zajmuje pewną stałą liczbę bajtów, tzn. jakiś rozmiar operacji (np. 1 bajt), jakiś rozmiar argumentu (np. 4 bajty) itd. Zadanie takie może potem być wykonane bezpośrednio w czasie odczytu, ponieważ instrukcje są już w formacie łatwo zrozumiałym dla procesora. Napisanie go jest jednak niewygodne bez dedykowanego kompilatora, szczególnie w językach wysokopoziomowych, np. ze względu na typowanie słabe, wymóg różnych operacji bitowych czy odpowiedniej końcówkowości – kolejności bajtów w słowach. Ten format jest jednakże użyteczny jako reprezentacja w pamięci.

Format tekstowy

Drugim sposobem jest zapis instrukcji i argumentów w postaci zwykłych ciągów znaków. Instrukcje najlepiej kilkuliterowe (*opcode*, mnemoniki, akronimy) oraz argumenty (głównie liczby) przekazywane w zapisie dziesiętnym (ewentualnie szesnastkowym lub ósemkowym). Format ten jest bardzo prosty do pisania zarówno ręcznego jak i bardziej zautomatyzowanego. Wymaga jednak parsera, który zamieni postać tekstową na reprezentację w pamięci (drzewo składniowe, *abstract syntax tree,* ***AST***) [28].

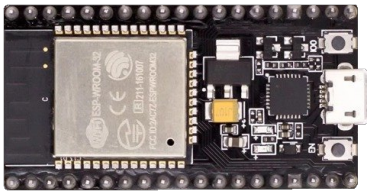
Rozdział 3

Konstrukcja budowanego urządzenia

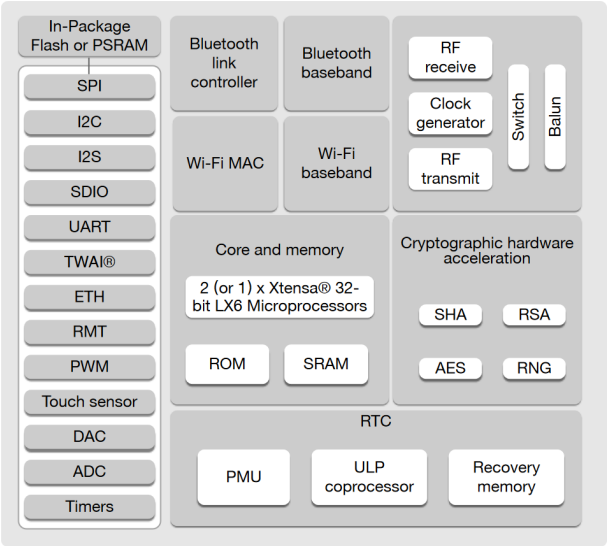
3.1 Użyte podzespoły

3.1.1 Mikrokontroler

Układ jest budowany w oparciu o mikrokontroler `ESP32`, ze względu na jego stosunkowo niski koszt, szeroki wachlarz możliwości (32-bitowy 2-rdzeniowy procesor; 4 MiB pamięci `Flash` i 520 KiB pamięci `RAM`; liczne `GPIO`; protokoły oraz peryferia: `UART`, `I2C`, `SPI`, `I2S`; wsparcie `Wi-Fi` i `Bluetooth`), oraz niskie zużycie energii (ok. 0.5 W). Na rysunku 3.1 przedstawiono jedną z wielu dostępnych płytek ewaluacyjnych. Oprócz samego modułu `ESP32`, posiada ona wbudowany regulator napięcia, port `USB` oraz konwerter `USB-UART`. Pod metalową pokrywą modułu kryją się m.in. właściwy procesor, pamięć Flash, oscylator kwarcowy oraz wyprowadzona jest antena. Na rysunku 3.2 pokazany jest uproszczony schemat blokowy mikrokontrolera: zawarte moduły, peryferia, obsługiwane protokoły i inne.



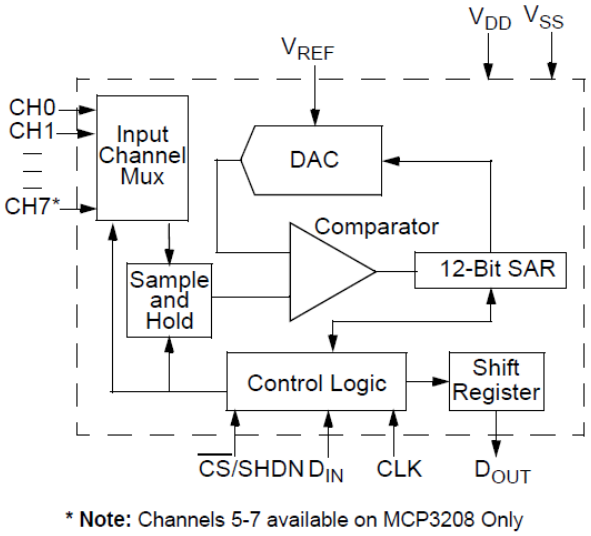
Rys. 3.1: Płytka ewaluacyjna z chipem `ESP-WROOM-32`



Rys. 3.2: Funkcyjny diagram blokowy procesora

3.1.2 Przetworniki A/C i C/A

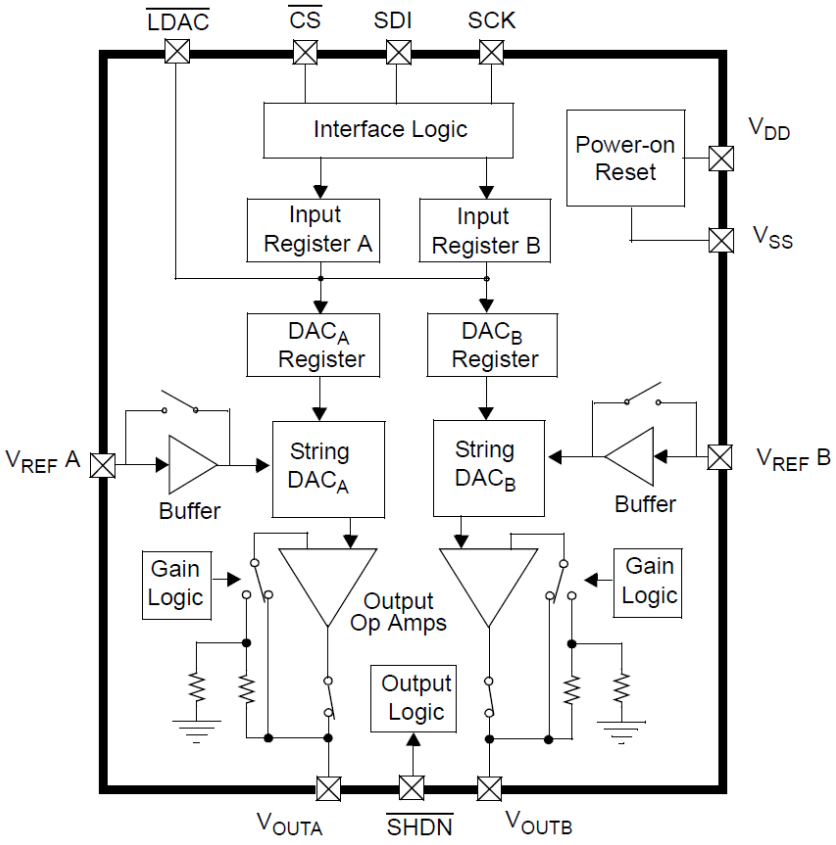
W pracy został wykorzystany przetwornik analogowo-cyfrowy `MCP3204` firmy Microchip [21]. Posiada on 4 wejścia analogowe, które mogą być używane niezależnie, lub jako wejścia różnicowe parami. Umożliwia próbkowanie z częstotliwością do 100 ksps oraz rozdzielczością 12 bitów. Zużywa bardzo mało energii, tj. w czasie pracy przy napięciu 5 V pobiera mniej niż 1 mA. Komunikuje się z mikroprocesorem za pomocą szybkiego protokołu szeregowego – `SPI`. Układ jest wyposażony w wejście napięcia odniesienia, które można prawie dowolnie wybrać.



Rys. 3.3: Schemat blokowy przetwornika `MCP3204` [21]

Do konwersji w drugą stronę, użyty został przetwornik cyfrowo-analogowy `MCP4922`, również firmy Microchip. Posiada on 2 wyjścia analogowe. Umożliwia generację z często-

tliwością do 1 Msps oraz oferuje również 12 bitów precyzji. Układ jest wyposażony w dwa wejścia napięcia odniesienia, osobne dla każdego z wyjść [22].



Rys. 3.4: Schemat blokowy przetwornika `MCP4922` [22]

3.1.3 Wzmacniacze

Przetwornik `A/C` posiada wbudowane kondensatory przechowujące próbki. Wymaga przez to wzmacniacza, który dostarczy potrzebny prąd ładowania. W tym celu wykorzystano układ `MCP6024` firmy Microchip, ze względu na identyczne zasilanie jak sam przetwornik oraz wysokie wzmocnienie $A_{Vol} \approx 120\,\mathrm{dB} \equiv 1\,\frac{\mathrm{MV}}{\mathrm{V}}$ oraz niskie napięcie niezrównoważenia (ang. *input offset voltage*) $V_{OS} < 250\,\mu\mathrm{V}$ [23].

Jako wzmacniacz wejść napięciowych umieszczony za dzielnikami, wykorzystany został układ `LT1014DIN` firmy Texas Instruments. Tak jak poprzednio wymieniony, ma on wysokie wzmocnienie $A_{Vol} > 1\,\frac{\mathrm{MV}}{\mathrm{V}}$ oraz niskie napięcie niezrównoważenia $V_{OS} < 250\,\mu\mathrm{V}$ [19].

Do pomiaru prądu wymagany jest wzmacniacz pomiarowy (instrumentalny), najlepiej z zewnętrznie regulowanym wzmocnieniem, w celu uniknięcia przełączania boczników. Został do tego wykorzystany układ `INA122P` firmy Texas Instruments. Ma on niskie napięcie niezrównoważenia wejścia $V_{OS} < 250\,\mu\mathrm{V}$ oraz niski prąd polaryzacji wejścia $I_b < 25\,\mathrm{nA}$. Wzmocnienie ustawia się podłączanym zewnętrznie rezystorem, zgodnie ze

wzorem $G = 5 + \frac{200\,\mathrm{k\Omega}}{R_g}$ [12].

W budowie wyjść wykorzystano wzmacniacze `LM2902N` firmy Texas Instruments [18], ze względu na ich dobry stosunek jakości do ceny. W celu zwiększenia mocy wyjść, zastosowano pary komplementarnych tranzystorów bipolarnych `BD139` i `BD140`.

Napięcia odniesienia: 2.048 V, 4.096 V, −2.048 V i −4.096 V zostały wytworzone przy pomocy układów generujących precyzyjne napięcie: `LM4040A20` i `LM4040A41` firmy Texas Instruments [17]. Ich wyjścia są buforowane za pomocą dwóch wzmacniaczy `MCP607` firmy Microchip, wybranych ze względu na wysokie wzmocnienie $A_{Vol} \approx 120\,\mathrm{dB} \equiv 1\,\frac{\mathrm{MV}}{\mathrm{V}}$ oraz niskie napięcie niezrównoważenia wejścia $V_{OS} < 250\,\mu\mathrm{V}$ [24]. Jeden wzmacniacz jest zasilany napięciem 0–5 V, zaś drugi – przeciwną polaryzacją.

3.1.4 Inne układy

Jako bufor wejść cyfrowych został wykorzystany układ `SN74HC125` firmy Texas Instruments. Potrafi on przyjąć aż 20 mA na każde wejście w celu utrzymania właściwego przedziału napięcia wejściowego, co w połączeniu z odpowiednimi rezystorami pozwala na znaczny przedział wartości napięcia równoważnego za stanem wysokim [31]. Do sterowania wyjściami cyfrowymi użyto generycznej macierzy Darlingtonów `XD2003` firmy Xinluda, w połączeniu z rezystorami limitującymi prąd.

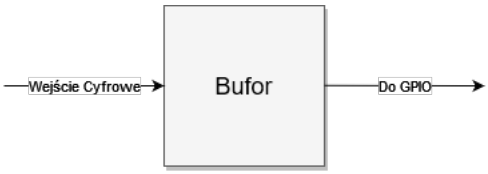
Do selekcji dzielników wejść napięciowych oraz wzmocnienia wejścia prądowego wykorzystano przekaźniki `V23100 V4005-A010` firmy TE Connectivity. Do sterowania przekaźnikami wykorzystano macierze Darlingtonów `ULN2803A` firmy STMicroelectronics [34]. Aby podłączyć ww. macierze do mikrokontrolera, użyto ekspanderów `MCP23008` firmy Microchip, ze względu na komunikację protokołem `I2C` oraz kompatybilne zasilanie [20].

3.1.5 Schematy ideowe torów

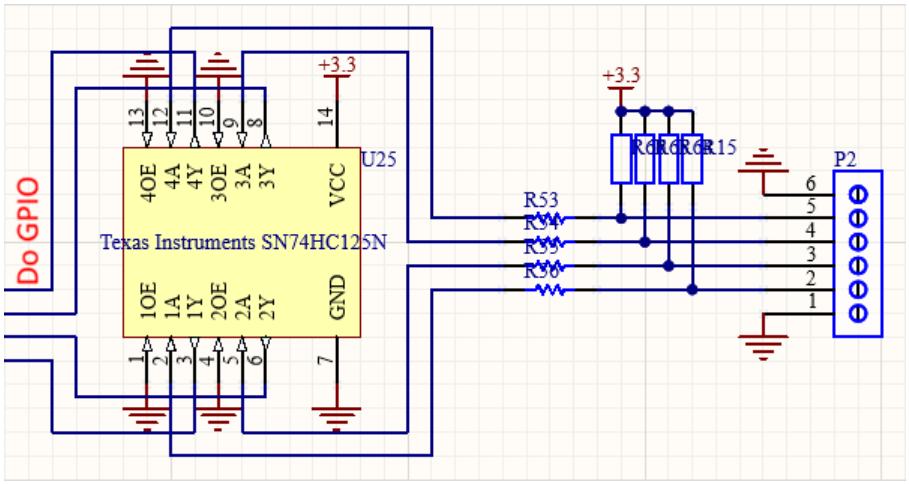
Poniżej przedstawione są diagramy blokowe i schematy ideowe wszystkich torów wejściowych i wyjściowych. Przy projektowaniu zostały uwzględnione zalecenia producentów poszczególnych układów oraz ogólnych „dobrych praktyk projektowania”, np. opisanych w dokumentach firmy Analog Devices [13][14]. Pełny schemat ideowy urządzenia umieszczony jest w dodatku B.

Tory cyfrowe wejściowe

Tory cyfrowe są bardzo proste w budowie oraz nie wymagają żadnego pośrednika, tzn. są bezpośrednio połączone do `GPIO` mikrokontrolera. Tor wejściowy wykorzystuje układ buforujący. Posiada on wbudowane zabezpieczenie przeciwko napięciu wejściowemu o wartościach przekraczających napięcie zasilania.



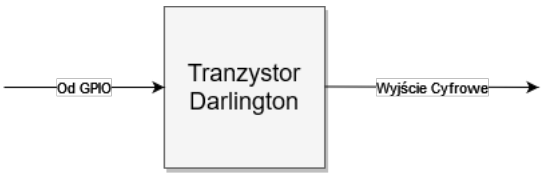
Rys. 3.5: Diagram blokowy toru cyfrowego wejściowego



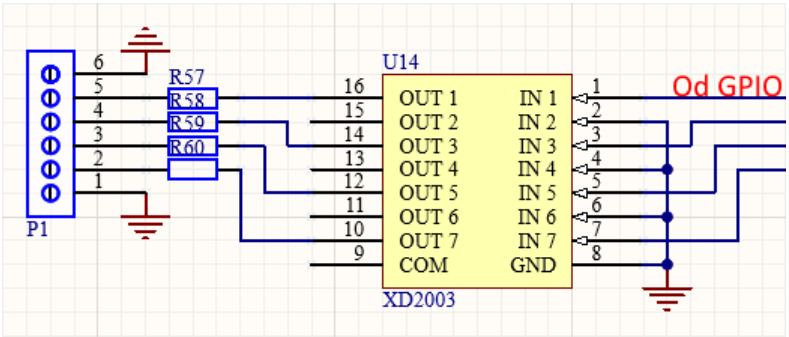
Rys. 3.6: Schemat ideowy toru cyfrowego wejściowego

Tory cyfrowe wyjściowe

Tor wyjściowy to po prostu wzmocnienie sygnału z mikrokontrolera za pomocą tranzystora bipolarnego w układzie Darlington.



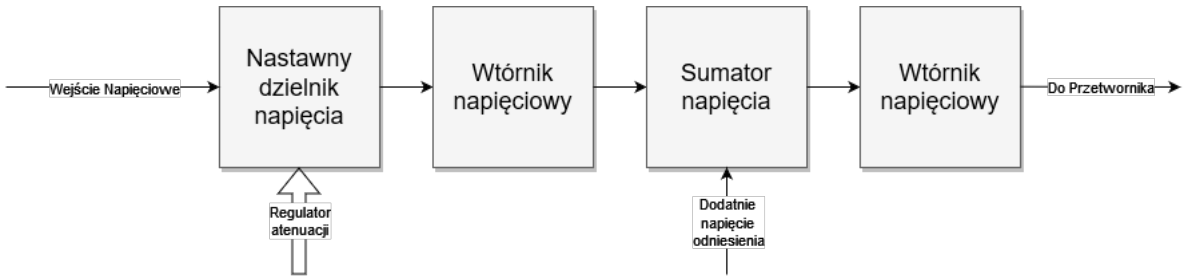
Rys. 3.7: Diagram blokowy toru cyfrowego wyjściowego



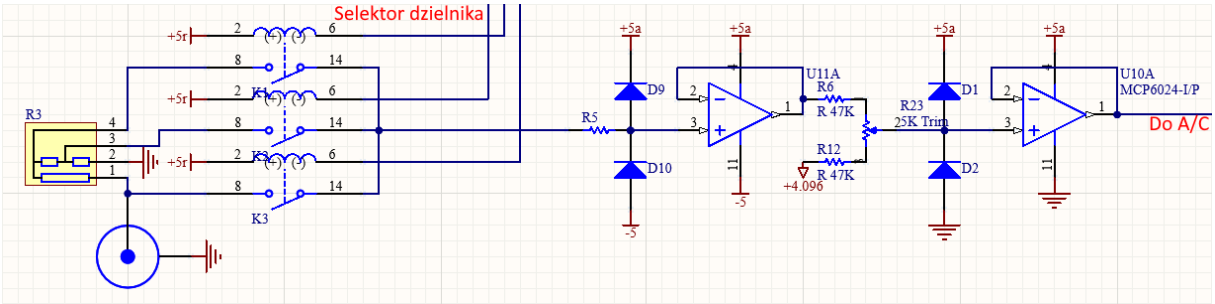
Rys. 3.8: Schemat ideowy toru cyfrowego wyjściowego

Tory analogowe wejściowe

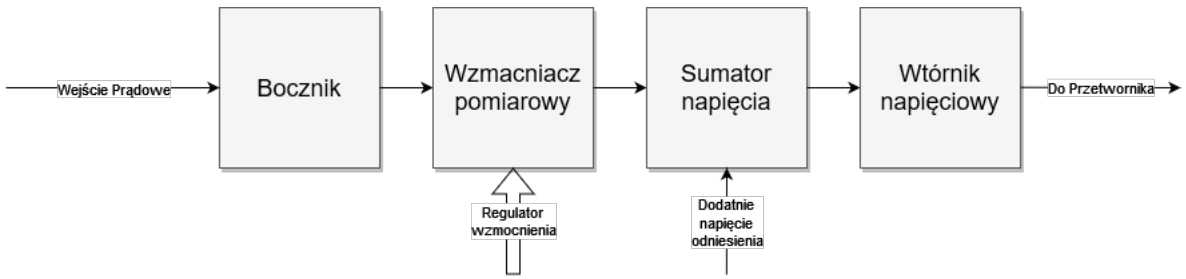
Tory wejściowe, zarówno napięciowe (rysunek 3.9) jak i prądowy (rysunek 3.11), są zbliżone w działaniu. Jedyna różnica jest taka, że tory napięciowe wykorzystują na wejściu nastawny dzielnik napięcia oraz wtórnik napięciowy, zaś tor prądowy jest wyposażony w bocznik i wzmacniacz pomiarowy z regulowanym wzmocnieniem. Następnie w obu typach sygnał zostaje przesunięty z przedziału $0 \pm 4.096\,\mathrm{V}$ na $2.048 \pm 2.048\,\mathrm{V}$. Kolejny wtórnik napięciowy przenosi go na wejścia przetwornika.



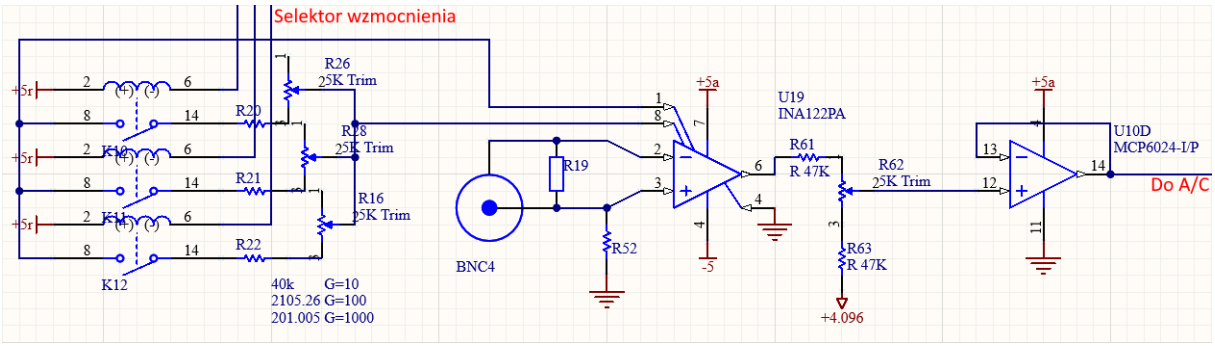
Rys. 3.9: Diagram blokowy toru napięciowego wejściowego



Rys. 3.10: Schemat ideowy toru napięciowego wejściowego



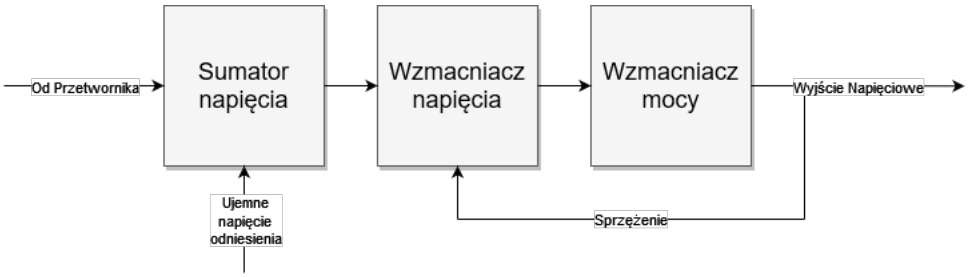
Rys. 3.11: Diagram blokowy toru prądowego wejściowego



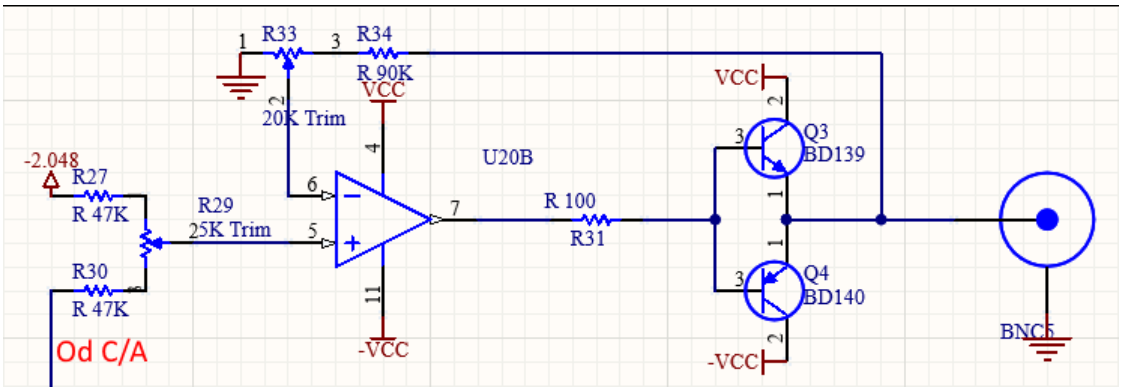
Rys. 3.12: Schemat ideowy toru prądowego wejściowego

Tory analogowe wyjściowe

Oba tory wyjściowe są na początku identyczne – napięcie z przetwornika ($0 \div 4.096\,\mathrm{V}$) jest zsumowane z ujemnym napięciem odniesienia, w celu osiągnięcia symetrycznego przedziału $\pm 1.024\,\mathrm{V}$. Dla toru napięciowego trafia ono bezpośrednio na główny wzmacniacz, który jednocześnie odpowiada za przeskalowanie go oraz za sterowanie elementami mocy, które są połączone z gniazdem `BNC`.



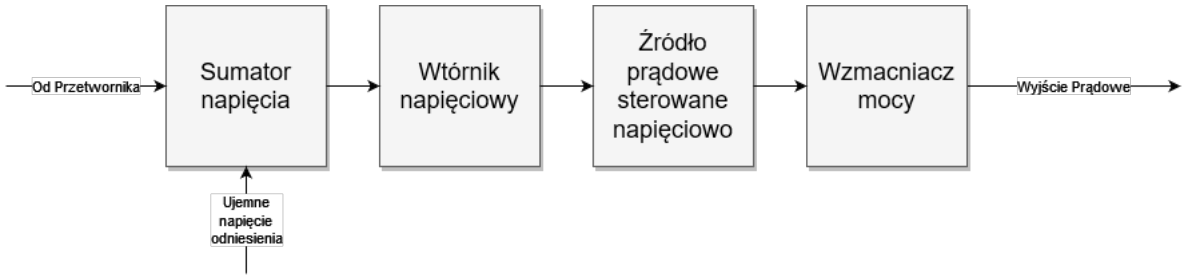
Rys. 3.13: Diagram blokowy toru napięciowego wyjściowego



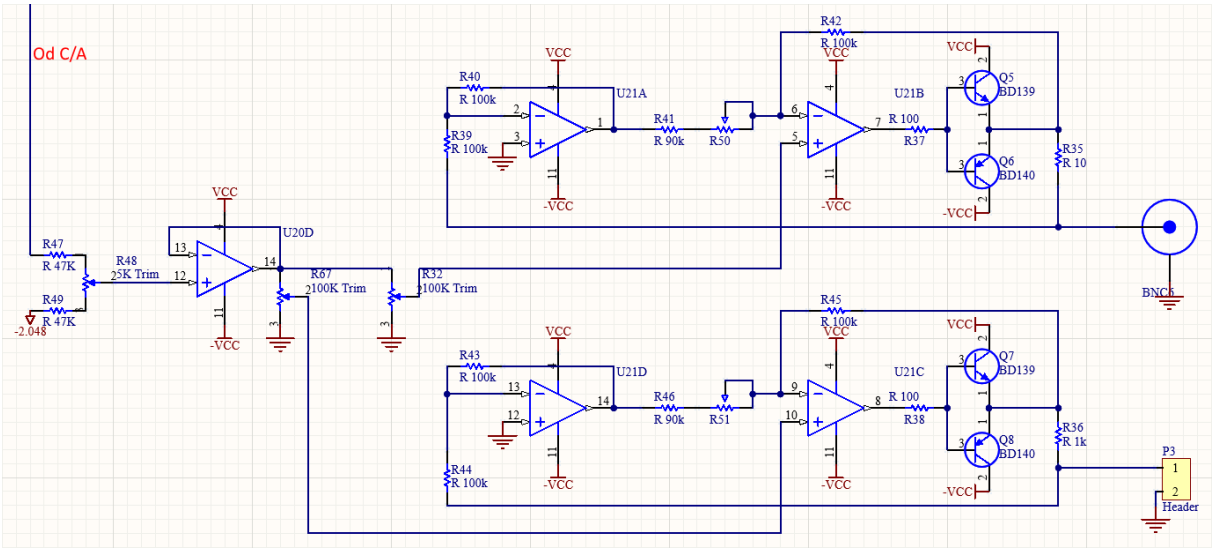
Rys. 3.14: Diagram blokowy toru napięciowego wyjściowego

Wyjście prądowe jest zbudowane jako źródło prądowe sterowane napięciowo. Składa się ono z dwóch wzmacniaczy. Jeden odpowiada za sterowanie elementami mocy, zaś drugi mierzy spadek na boczniku (co „oblicza” prąd), a następnie odpowiednio koryguje

główny wzmacniacz, w celu uzyskania charakterystyki prądowej zależnej tylko od napięcia wejściowego, nie zaś od rezystancji podłączonej do portu. Zbudowane zostało też drugie wyjście prądowe, z bocznikiem o większym oporze. Pozwala ono wytworzyć mniejszy, lecz dużo precyzyjniejszy prąd. Jest ono jednak sterowane jednocześnie z głównym źródłem, tym samym wyjściem przetwornika.



Rys. 3.15: Schemat ideowy toru prądowego wyjściowego



Rys. 3.16: Schemat ideowy toru analogowy prądowego wyjściowego

3.2 Połączenie z komputerem

Komunikacja między urządzeniem a komputerem mogła być zaimplementowana na wiele sposobów. Rozwiązania klasyczne - `RS232/RS485` – niestety są przestarzałe i niewygodne w użyciu (grubość kabla, dystans, szybkość przesyłu).

Początkowo rozważane było połączenie przez `USB`, jednak ma ono wady: wymaga konwertera `USB-UART` po stronie kontrolera oraz emulatora portu `COM` po stronie komputera. Do tego wymaga pośrednika odpowiadającego za odizolowanie elektryczne układu od komputera, co jest jednym z założeń projektu. Możliwe są dwie topologie, gdzie układ odpowiedzialny za izolację jest umieszczony przed lub za konwerterem. Izolator dla `UART` jest dużo prostszy ze względu na fakt, że dana linia przesyła informacje tylko w jedną

stronę, do czego wystarczy prosty optoizolator. USB wymaga dwukierunkowego przesyłu na obu żyłach, co komplikuje budowę. Z drugiej strony, izolator umieszczony za konwerterem powoduje konieczność zasilania konwertera z komputera, a więc naraża port lub komputer na uszkodzenie w przypadku uszkodzenia nawet samego przewodu.

Lepszym rozwiązaniem jest komunikacja przez port Ethernet. Jego porty posiadają układy izolujące po obu stronach, więc nawet w przypadku uszkodzenia przewodu, komputer nie jest narażony. Przewód może być podpięty bezpośrednio między urządzeniami (*crossover*), gdzie umożliwia transmisję na sporą odległość, lub też poprzez sieć - modem/router itd. - co pozwala na całkowicie zdalną pracę z urządzeniem, z dowolnej odległości. Jest fabrycznie wspierany przez prawie każdy komputer oraz łatwy do podłączenia do mikrokontrolera - wymaga tylko gotowego modułu z portem Ethernet. Dodatkowo pozwala też w standardach 10BASE-T i 100BASE-T na bardzo proste przesłanie zasilania po dwóch nieużywanych parach w przewodzie. Umożliwiają to splittery, odłączające te pary od wtyczki RJ45 dla złącza zasilającego.

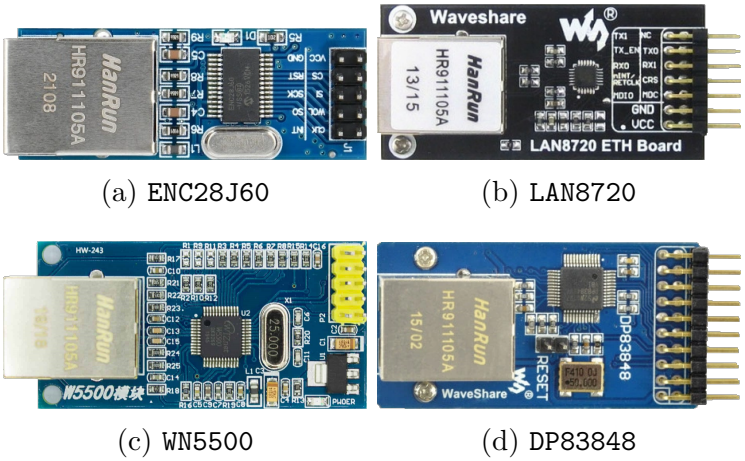
Na rysunku 3.17 przedstawiono kilka dostępnych komercyjnie układów. Mimo iż wyglądają podobnie, różnią się działaniem. Wszystkie nazwy odnoszą się zarówno do samych układów, jak i gotowych płytek, na których się znajdują. ENC28J60 i WN5500 komunikują się poprzez protokół SPI. Oznacza to jednocześnie, że muszą same implementować stos TCP/IP oraz warstwę MAC. Moduł LAN8720 i DP83848 łączą się za pomocą interfejsu RMII (modyfikacja *Media-Independent Interface*, oznaczona *Reduced*, wykorzystuje 10 przewodów, w porównianiu do 18 dla MII). Jest on dedykowany do łączenia warstwy PHY i MAC. Oznacza to, że moduły te implementują tylko warstwę fizyczną, zaś zadaniem mikrokontrolera jest zarządzanie warstwą MAC.

PHY to warstwa fizyczna – najniższa w stosie OSI. Implementuje ją każde urządzenie łączące się w jakikolwiek fizyczny sposób, zarówno przewodowy jak i bezprzewodowy. Warstwa MAC (ang. *Medium Access Control*) to druga w stosie OSI, m.in. odpowiada za fragmentację wiadomości i pomaga w korekcji błędów.

Zostało przetestowane działanie dwóch modułów. ENC28J60 nie jest bezpośrednio wspierany przez Espressif, tylko wymaga dodatkowych kodów, do tego posiada długą listę errata. Udało się go uruchomić, jednak jego praca jest bardzo niestabilna i przeważnie przestawał działać po upływie maksymalnie kilku minut.

Moduł LAN8720 co prawda zużywa więcej wejść mikrokontrolera, jednak to on okazał się rozwiązaniem działającym stabilnie, o łatwym połączeniu i implementacji w oprogramowaniu, bez problemu komunikując się z innymi urządzeniami, np. routerem. Wystarczyło skonfigurować sygnał zegara jako idący od procesora do modułu. Mimo długich połączeń prototypu, nie było problemu ani z zasilaniem, ani z zakłóceniami sygnałów. Wadą okazała się jednak prędkość przesyłu, jak wynikło z testów, osiągająca szczytowo 3.5 kB/s gdzie najwolniejszy standard Ethernet to 10 MB/s, zaś moduł według specyfikacji powinien obsługiwać nawet 100 MB/s. Być może takie parametry są osiągalne na

dedykowanej płytce drukowanej.



(a) ENC28J60 (b) LAN8720

(c) WN5500 (d) DP83848

Rys. 3.17: Przykładowe moduły Ethernet

Ostatecznie wybranym rozwiązaniem okazało się połączenie za pomocą wbudowanego w ESP32 komponentu Wi-Fi, które nie dość, że zaoferowało dużo większą przepustowość i kompletne pozbycie mechanicznego połączenia z komputerem, ale dodatkowo pozwoliło na zwolnienie jednego z bardzo pożądanych interfejsów SPI oraz innych GPIO. Niestety wadą połączenia bezprzewodowego jest możliwość wystąpienia np. niespodziewanych opóźnień czy chwilowych wstrzymań transferu, co może spowodować nieaktualność danych lub w skrajnym przypadku nawet przepełnienie bufora próbek. Dodatkowo jakość przesyłu będzie dużo bardziej zależna od dystansu niż przy połączeniu przewodowym.

3.3 Oprogramowanie

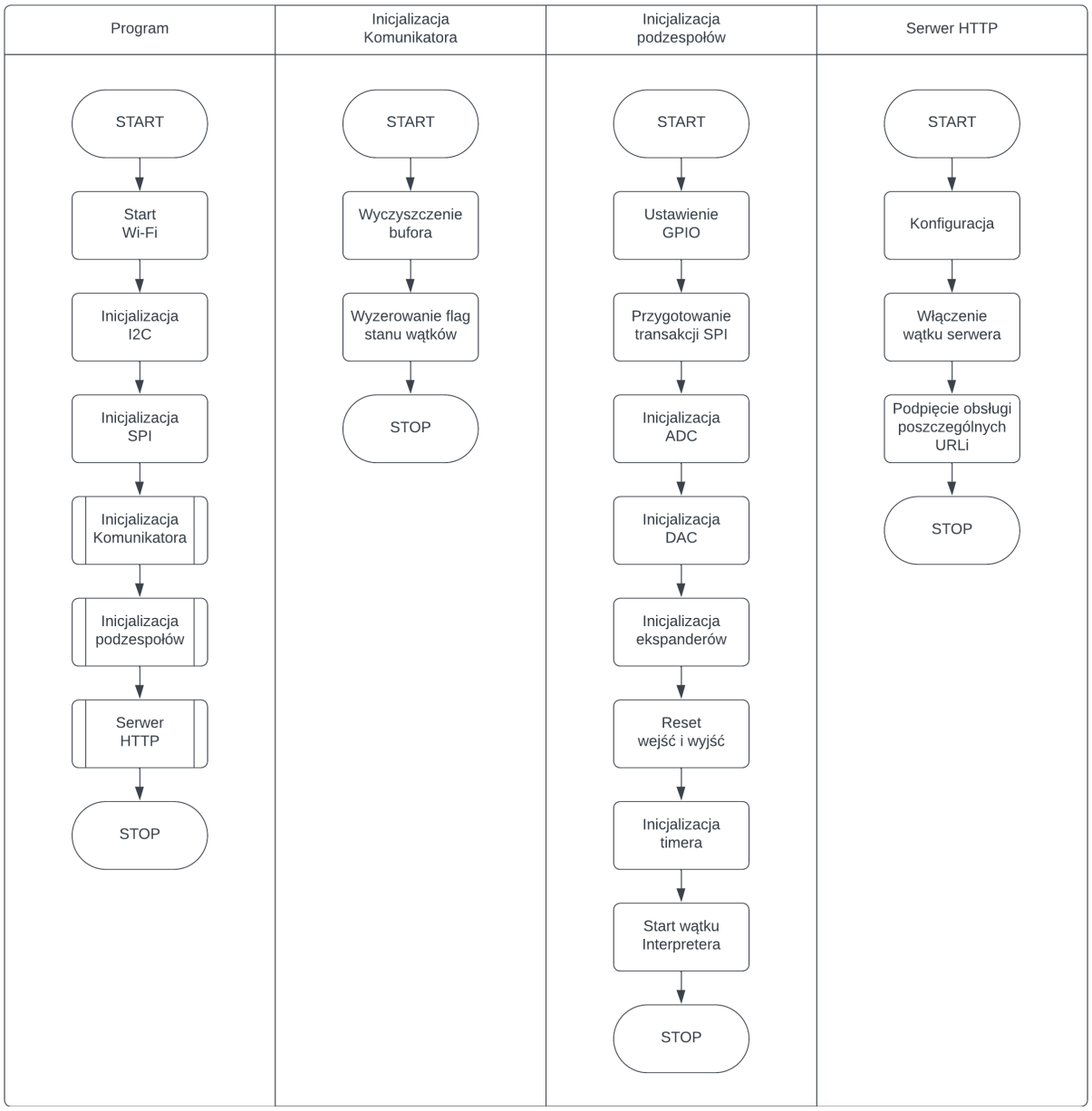
Program dzieli się na kilka części, przedstawionych niżej. Do działania wykorzystane są różne klasy i biblioteki, które pozwalają w pewnym stopniu ukryć niskopoziomową obsługę za różnymi interfejsami. Stworzone zostały biblioteki do obsługi przetwornika A/C (dodatek A.2.1), przetwornika C/A (dodatek A.2.2), ekspanderów GPIO (dodatek A.2.3).

Fragmenty kodu ważniejszych części wraz z opisami są przedstawione w dodatku A – Dokumentacji technicznej.

3.3.1 Inicjalizacja układu

Schemat blokowy inicjalizacji został przedstawiony na rysunku 3.18. Na samym początku zostają skonfigurowane: protokół Wi-Fi, interfejsy SPI do sterowania przetwornikami, interfejs I2C do sterowania ekspanderami oraz odpowiednie piny GPIO dla wyjść i wejść cyfrowych. Później następuje inicjalizacja podzespołów (dodatek A.56): utworzenie transakcji SPI, inicjalizacja przetworników (utworzenie jako urządzeń SPI), rezerwacja magistral, inicjalizacja ekspanderów, wyzerowanie wyjść analogowych i cyfrowych oraz

wyłączenie przekaźników wejść analogowych. Finalnie tworzony jest zegar (ang. *timer*), odpowiadający za synchronizację wewnątrz programu, oraz uruchamiany jest wątek Interpretera odpowiedzialny za wykonywanie zadań i obsługę całego sprzętu wejściowego i wyjściowego. Przygotowany jest również bufor na dane oraz zmienne służące do komunikacji między wątkami. Po tym wszystkim uruchamiany jest serwer `HTTP` odpowiedzialny za komunikację ze światem.



Rys. 3.18: Schemat blokowy inicjalizacji wszystkich komponentów

3.3.2 Praca ciągła

Po skończeniu inicjalizacji urządzenie wykonuje dwa główne zadania. Jedno to Interpreter, czyli wątek odpowiedzialny za wykonywanie wgrywanych zadań. Jest on na stałe

przypięty do drugiego rdzenia (CPU1, APP CPU), ma najwyższy priorytet i nie dopuszcza niczego innego poza przerwaniami. Z tego powodu *task watchdog*, odpowiedzialny m.in. za monitorowanie zużycia procesora i czasu przydzielanego zadaniom, jest wyłączony na tym rdzeniu.

Na pierwszym rdzeniu (CPU0, PRO CPU) wykonywana jest cała reszta zadań: serwer HTTP, obsługa stosu TCP/IP, serwera DNS, komunikacji przez Wi-Fi, oraz różnych innych wewnętrznych funkcji wymaganych do działania. Są one wszystkie zarządzane przez system czasu rzeczywistego – zmodyfikowany FreeRTOS.

Serwer odpowiada za całą komunikację ze światem. Obsługuje kilka adresów, będących jednocześnie punktami końcowymi API, które są opisane poniżej.

3.4 Punkty końcowe API

Serwer HTTP obsługuje dokładnie cztery adresy, z czego trzy to punkty końcowe API, zaś jeden odpowiada za przesył obrazka favicon.ico, który jest automatycznie żądany przez np. przeglądarki i został dodany w celu uniknięcia błędów 404.

Rys. 3.19: Obrazek favicon.ico przedstawiający symbol przetwornika

3.4.1 Powitanie, pomoc, ogólne informacje

Pierwszy punkt końcowy znajduje się pod domyślnym, głównym adresem: / i używa metody GET. Zwraca on stały obiekt JSON, który zawiera podstawowe informacje, jak data kompilacji, wersja kompilatora, wersję środowiska Espressif, adresy API, listę instrukcji wraz z ich opisami, argumentami i zwracanymi danymi, przykładowy generator, listę rodzajów przebiegów, przedziały i rozdzielczość wejść/wyjść analogowych i inne.

3.4.2 Wgrywanie zadania do urządzenia

Punkt końcowy odpowiedzialny za wgrywanie zadania i generatorów znajduje się pod adresem /settings i używa metody POST.

Ustawienia są przesyłane w treści zapytania jako obiekt zakodowany w formacie JSON (ang. *JavaScript Object Notation*). Jest to otwarty standardowy format plików i format wymiany danych, który wykorzystuje tekst czytelny dla człowieka do przechowywania i przesyłania wartości, które mogą być liczbami, ciągami znaków, wartościami logicznymi, tablicami (będącymi listą wartości) lub obiektami (tablicami tzw. asocjacyjnymi, to znaczy składającymi się z par {nazwa, wartość}). Jest to powszechny format danych

o różnorodnych zastosowaniach w elektronicznej wymianie danych, w tym w aplikacjach internetowych z serwerami.

Na ustawienia składają się (wymagane) zadanie do wykonania oraz (opcjonalna) lista generatorów przebiegów `DDS` (ang. *Direct Digital Synthesis*, Bezpośrednia synteza cyfrowa) dla wyjść analogowych. Obie części są dokładniej opisane niżej. Kiedy użytkownik wgrywa ustawienia, są one sprawdzane i parsowane (proces jest dokładniej opisany w dalszych sekcjach). Jeśli nie ma żadnych błędów, zostają one przeniesione i zapisane na 'stałe' (tzn. do momentu ich zmiany lub utraty zasilania). Pozwala to powtarzać pomiary bez konieczności wgrywania na nowo tego samego programu.

3.4.3 Wykonywanie zadania i odbiór danych

Punkt końcowy odpowiedzialny za faktyczne wykonanie zadania: sterowanie wyjściami, mierzenie wejść i przesyłanie wyników do użytkownika znajduje się pod adresem `/io` i używa metody `GET`. Przyjmuje on jeden parametr w *query* – `tb`, określający rozmiar w bajtach pomiarów czasu. Przyjmuje wartości od 0 do 8, gdzie 0 oznacza, że czas wykonania pomiaru nie jest zapisywany, zaś 8 zwraca pełną 64-bitową liczbę z timera synchronizującego. Przeważnie 4 lub nawet 2 bajty wystarczą, szczególnie dla krótkich zadań lub w przypadku, gdy użytkownik będzie we własnym programie pilnował ewentualnego przekroczenia zakresu tych wartości.

W momencie żądania wykonania zadania wątek Interpretera jest powiadamiany, przechodzi przez inicjalizację (reset wyjść, reset obiektu zadania, reset i start zegara) i zaczyna pobierać po kolei komendy z zadania, wykonując odpowiedni kod w celu ich realizacji oraz wstawiając zmierzone wartości do bufora. Wątek odpowiedzialny za serwer odczytuje dane z bufora i wysyła je do klienta. Cały proces trwa aż do skończenia programu lub zerwania połączenia.

Zamiast kodować zmierzone dane w celu bezpiecznego przesłania formatem `JSON`, wybrana została bardziej prymitywna, lecz jednocześnie szybsza i prostsza w użyciu metoda. Program odsyła surowe dane z nagłówkiem określającym typ zawartości jako `octet-stream` w postaci ciągu bajtów. Jest to, zależnie od rodzaju odczytywanych danych, liczba całkowita (ze znakiem lub bez) lub liczba zmiennoprzecinkowa oraz opcjonalnie obecny czas jako liczba całkowita o wybranej długości. Takie kodowanie pozwala też wysyłać i odbierać dane fragmentami (co byłoby trudne w przypadku wiadomości `JSON`, chyba że każdy fragment byłby kodowany osobno).

3.5 Parser i interpreter

Aby wykonać przesłane przez użytkownika zadanie, musi ono najpierw zostać odczytane. Oczywiście można by to robić od razu w trakcie wykonywania, jednak ze względu

na wybrany format:

1. przetwarzanie tekstu nie należy do najszybszych, szczególnie na mikrokontrolerach,
2. zadanie w wersji sparsowanej powinno zająć mniej miejsca w pamięci niż surowy tekst,
3. takie zadanie można wykonywać wielokrotnie, bez potrzeby kolejnego analizowania przy każdym uruchomieniu.

Z tego względu zadanie jest zamienione na drzewo instrukcji w momencie przesłania, co jednocześnie pozwala też wykryć błędy i poinformować użytkownika. Drzewo takie zostaje zapisane w pamięci i może być wielokrotnie wykonywane, wymaga jedynie zresetowania wskaźnika do obecnej instrukcji.

3.5.1 Zamiana tekstu na format binarny – parser zadania

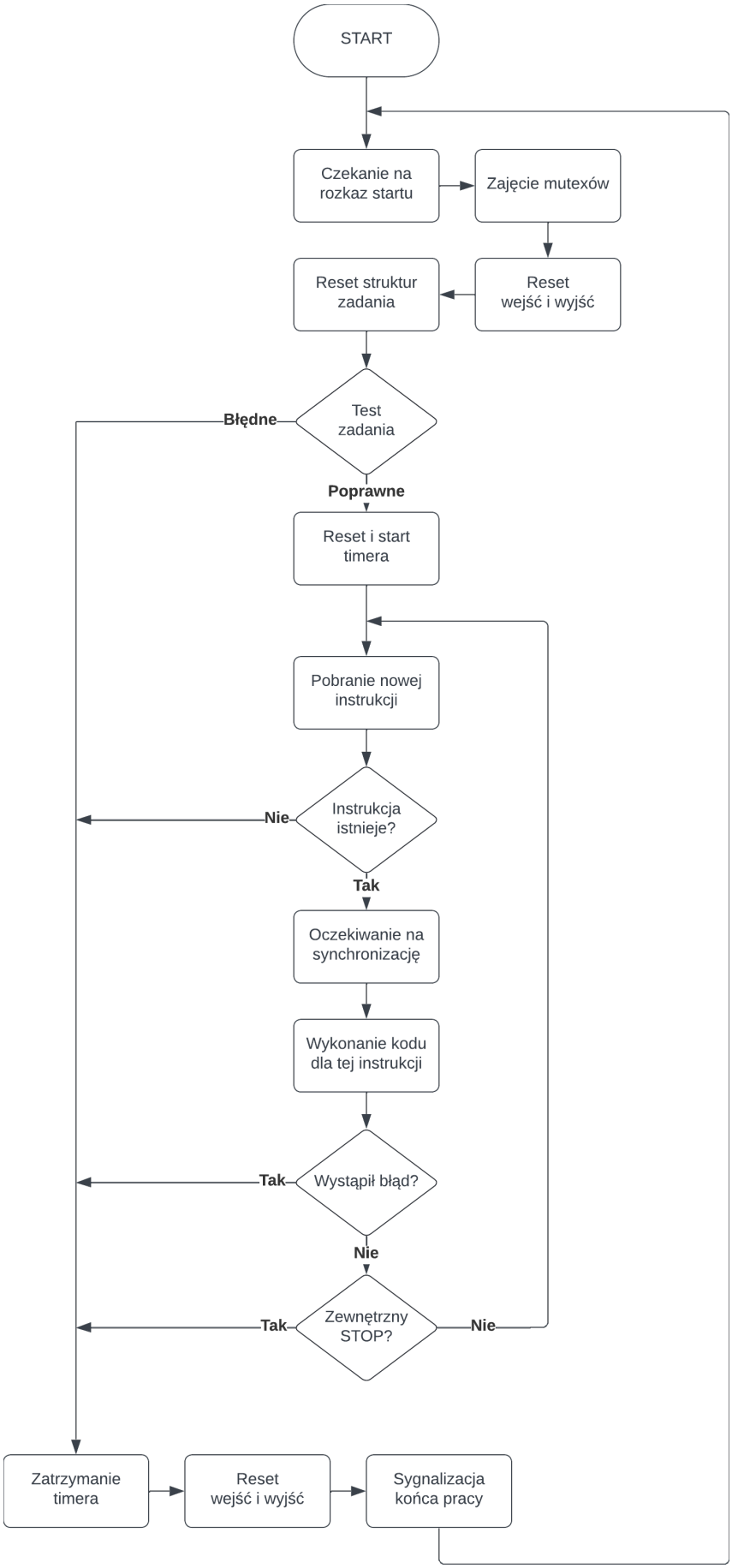
Analizator składniowy lub *parser* to program dokonujący analizy składniowej danych wejściowych w celu określenia ich struktury gramatycznej w związku z określoną gramatyką formalną. Analizator umożliwia przetworzenie tekstu czytelnego dla człowieka w strukturę danych przydatną dla oprogramowania. Wynikiem analizy składni dokonywanej przez parser najczęściej jest drzewo składniowe.

Dokładna budowa pomocniczych struktur oraz parsera opisana jest w dodatku A.2.5, jednak zostanie pokrótce przedstawiona tutaj. Najpierw stworzono strukturę, która reprezentuje pojedynczą instrukcję, tzn. zawiera wykonywaną operację oraz opcjonalnie port, którego ta operacja dotyczy i/lub dowolny argument w postaci liczby całkowitej lub zmiennoprzecinkowej. Później powstała klasa przechowująca listę instrukcji – `Scope` oraz klasa pozwalająca je powtarzać – `Loop`. Klasa `Program` zawiera jeden `Scope` z całym zadaniem, a także implementuje parser. Parser najpierw tworzy stos `Scope`'ów, gdzie do tego na górze stosu dopisywane są wyrażenia. W pętli odczytywane są kolejne instrukcje, dzielone następnie na operację i argumenty. Jeśli operacja to `LOOP`, odczytywana jest pożądana ilość iteracji, nowa pętla zostaje utworzona i dodana do stosu. Jeśli operacja to `END`, obecny `Scope` zostaje zamknięty i usunięty ze stosu, zaś kolejne instrukcje będą dopisywane do poprzedniego, niższego poziomu. W pozostałych przypadkach, operacja to instrukcja do sprzętu. Sprawdzana jest nazwa operacji, zapisywany jest odpowiedni `OPCode`, sprawdzana jest wymagana liczba argumentów i zostają one odczytane. Jeśli nie wystąpił żaden błąd, zadanie jest poprawne; zostaje zapisane w stałej części pamięci i może zostać wykonane.

3.5.2 Interpreter zadań

Interpreter to program komputerowy wykonujący inne programy [28]. Jest kluczowym elementem znacznej części implementacji języków skryptowych oraz języków kompilowa-

nych do kodu bajtowego [5]. Schemat blokowy został przedstawiony na rysunku 3.20, zaś dokładny opis kodu znajduje się w dodatku A.59. Wątek Interpretera oczekuje na sygnał startu od serwera `HTTP`. Po otrzymaniu go, inicjalizuje porty i zegar. Następnie w pętli odczytuje instrukcje z zadania i je wykonuje, czekając na synchronizację jeśli tak zażądano. Jeśli wystąpi błąd, klient rozłączy się lub zadanie zostanie skończone – ponownie ustawia stan domyślny portów, sygnalizuje koniec pracy i wraca do oczekiwania na zadanie.



Rys. 3.20: Schemat blokowy interpretera

3.6 Wykonywanie zadań

Zadanie to lista wyrażeń – komend wraz z ich argumentami. Każde wyrażenie kończy się znakiem `;`, zaś komenda i argumenty oddzielone są białymi znakami. Dla poprawienia czytelności, w całym programie mogą być swobodnie dodawane kolejne białe znaki (np. nowe linie, tabulacje), zarówno między argumentami jak i całymi wyrażeniami.

Wyrażenie może być bezpośrednią instrukcją do sprzętu lub służyć do zmiany przepływu programu. Z kilku możliwości wspomnianych wcześniej, została zaimplementowana tylko pętla, ze względu na jej prostotę. Pętla może zawierać kolejne wyrażenia, zarówno instrukcje jak i kolejne pętle. Takie zagnieżdżenie pozwala na łatwe stworzenie wielokrotnie powtarzających się pomiarów, sekwencyjnych przełączeń, itp. Poniżej przedstawione są dostępne wyrażenia wraz z ich funkcją.

3.6.1 Pętle

Deklaracja pętli składa się z dwóch wyrażeń – otwarcie, wraz z ilością iteracji, oraz zamknięcie. Ilość otwarć i zamknięć w programie musi być taka sama.

```
1 LOOP <iterations (uint32)>;
2     [...]
3 END;
```

3.6.2 Instrukcje

Reszta wyrażeń to instrukcje. Są one podzielone na kilka kategorii, zależnie od pełnionej funkcji. Same kategorie nie mają jednak żadnego znaczenia technicznego.

Instrukcje pomocnicze

- *No OPeration* – wartość domyślna, nic nie robi, nie należy jej używać.
  Składnia: `NOP`

- *DELAY* – opóźnienie, ustawia następny czas synchronizacji na podstawie poprzedniego.
  Składnia: `DELAY <microseconds (uint32)>`

- *GET TiMe* – jeśli nie czeka na synchronizację, ustawia następny czas synchronizacji na 'teraz'.
  Składnia: `GETTM`

- *ReSeT TiMe* – resetuje i restartuje zegar, tzn. pomiary czasowe odnoszą się do tego momentu.
  Składnia: `RSTTM`

Instrukcje do ustawiania wejść analogowych

- *Analog Input ENable* – łączy wejścia z dzielnikami i wzmacniaczami.
  Składnia: `AIEN`

- *Analog Input DISable* – odłącza wejścia.
  Składnia: `AIDIS`

- *Analog Input RaNGe* – ustawia przedział wartości portu wejściowego (wybiera dzielnik napięcia lub mnożnik wzmacniacza pomiarowego). Wyłączony/minimalny/średni/maksymalny.
  Składnia: `AIRNG <port (1|2|3|4)> <range (OFF|MIN|MED|MAX)>`

Instrukcje do pomiarów wejść analogowych

- *Analog Input ReaD Float* – zwraca pomiar (wykonany podaną ilość razy i uśredniony) w jednostkach V, A jako 32-bitowa liczba zmiennoprzecinkowa (`float`). W
  Składnia: `AIRDF <port (1|2|3|4)> <repetitions>`

- *Analog Input ReaD Milli* – zwraca pomiar (wykonany podaną ilość razy i uśredniony) w jednostkach mV, mA jako 32-bitowa liczba całkowita (`int32`). Wykonany podaną ilość razy i uśredniony.
  Składnia: `AIRDM <port (1|2|3|4)> <repetitions>`

- *Analog Input ReaD Micro* – zwraca pomiar (wykonany podaną ilość razy i uśredniony) w jednostkach µV, µA jako 32-bitowa liczba całkowita (`int32`). Wykonany podaną ilość razy i uśredniony.
  Składnia: `AIRDU <port (1|2|3|4)> <repetitions>`

Instrukcje do sterowania wyjść analogowych

- *Analog Output VALue* – wystawia na wyjście podaną wartość.
  Składnia: `AOVAL <port (1|2)> <voltage (float)>`

- *Analog Output GENerator* – wystawia na wyjście wartość obliczoną przez generator DDS.
  Składnia: `AOGEN <port (1|2)> <generator_idx (uint)>`

Instrukcje do pomiarów wejść cyfrowych

- *Digital Inputs ReaD* – zwraca stan pinów jako 4 bity w liczbie całkowitej (`uint32`).
  Składnia: `DIRD`

Instrukcje do sterowania wyjść cyfrowych

- *Digital Outputs WRite* – bezpośrednio ustawia stan pinów.
  Składnia: `DOWR <state (uint4 hex/oct/dec)>`
- *Digital Outputs SET* – załącza piny odpowiadające bitom w stanie wysokim (bitowy OR).
  Składnia: `DOSET <state (uint4 hex/oct/dec)>`
- *Digital Outputs ReSeT* – wyłącza piny odpowiadające bitom w stanie wysokim.
  Składnia: `DORST <state (uint4 hex/oct/dec)>`
- *Digital Outputs AND* – wyłącza piny odpowiadające bitom w stanie niskim.
  Składnia: `DOAND <state (uint4 hex/oct/dec)>`
- *Digital Outputs XOR* – wykonuje bitowo operację eXclusive OR.
  Składnia: `DOXOR <state (uint4 hex/oct/dec)>`

Opis argumentów komend

Typ `int` oznacza liczbę całkowitą, z przedrostkiem `u` jest ona bez znaku (*unsigned*) tj. bez wartości ujemnych. Dodatkowa liczba na końcu oznacza ilość bitów (a więc określa maksymalną wartość jaka może być podana). Typ `float` oznacza liczbę dziesiętną z opcjonalnym ułamkiem, która będzie zapisana jako zmiennoprzecinkowa.

Przedział dla wejść analogowych jest podawany jako cztery możliwe słowa: `OFF`, `MIN`, `MED`, `MAX`. `OFF` to oczywiście odłączenie wejścia - powoduje wyłączenie wszystkich przekaźników. W przypadku wejść napięciowych `MIN` powoduje bezpośrednie przekazanie wejścia na przetwornik, `MED` powoduje 10x pomniejszenie napięcia, zaś `MAX` to 100x pomniejszenie. Oznacza to, że maksymalne mierzalne napięcia na wejściu to odpowiednio $\pm 4\,\mathrm{V}$, $\pm 40\,\mathrm{V}$ i $\pm 400\,\mathrm{V}$. W przypadku wejścia prądowego mierzony jest spadek napięcia na boczniku $1\,\Omega$; `MIN` powoduje 1000x wzmocnienie, `MED` powoduje 100x wzmocnienie, zaś `MAX` to tylko 10x wzmocnienie. Oznacza to, że maksymalny prąd płynący przez wejście to odpowiednio $\pm 4\,\mathrm{mA}$, $\pm 40\,\mathrm{mA}$ i $\pm 400\,\mathrm{mA}$.

3.7 Generator sygnałów

Oprócz programu, w ustawieniach można dołączyć dane opisujące wirtualne generatory przebiegów. Generator to lista obiektów sygnałów, każdy taki obiekt zawiera para-

metry A – amplitudę (w woltach lub amperach) i S – obiekt poszczególnego sygnału. Obiekt sygnału zawiera parametr WF – jego nazwę-kod oraz odpowiednie parametry poszczególnych typów, opisane niżej. Podstawa czasowa to jedna mikrosekunda.

3.7.1 Typy przebiegów możliwych do użycia w generatorach

Istnieje kilka typów przebiegów, każdy w inny sposób oblicza wartości wyjściowe na podstawie obecnego czasu.

Wartość stała

Określany w kodzie jako `Const`. Zwraca wartość 1 niezależnie od czasu. Parametry: brak.

Impuls

Określany w kodzie jako `Impulse`. Zwraca wartość 1 dla czasu 0, 0 dla reszty. Parametry: brak.

Przebieg sinusoidalny

Określany w kodzie jako `Sine`. Zwraca wartość obliczoną za pomocą przybliżonej funkcji sinus. Parametry:

- `T` – okres w mikrosekundach (`int32`)

Przebieg prostokątny

Określany w kodzie jako `Square`. Zwraca wartość 1 jeśli czas wewnątrz okresu jest mniejszy niż *Duty*, 0 w przeciwnym przypadku. Parametry:

- `T` – okres w mikrosekundach (`int32`)
- `D` – czas trwania stanu wysokiego (`int32`)

Przebieg trójkątny

Określany w kodzie jako `Triangle`. Przebieg jest symetryczny, zarówno w czasie jak i wartościach. Zaczyna od zera i rośnie. Parametry:

- `T` – okres w mikrosekundach (`int32`)

Sygnał świergotowy

Określany w kodzie jako `Chirp`. Jest to funkcja sinusoidalna z częstotliwością zmieniającą się liniowo w czasie. Parametry:

- `T` – czas trwania fragmentu w mikrosekundach (`int32`)

- `FS` – częstotliwość startowa (`float`)
- `FD` – zmiana częstotliwości (koniec − początek) (`float`)

Sygnał świergotowy logarytmiczny

Określany w kodzie jako `ChirpLog`. Jest to funkcja sinusoidalna z częstotliwością zmieniającą się wykładniczo w czasie, a więc liniowo w skali logarytmicznej. Parametry:

- `T` – czas trwania fragmentu w mikrosekundach (`int32`)
- `FS` – częstotliwość startowa (`float`)
- `FR` – stosunek częstotliwości ($\frac{\text{koniec}}{\text{początek}}$) (`float`)

Sygnał losowy

Określany w kodzie jako `Random`. Generuje wartości losowe rozłożone liniowo w przedziale $[-1, 1)$. Parametry: brak.

3.7.2 Modyfikatory przebiegów

Modyfikatory przebiegów również są klasą dziedziczącą z głównej klasy `Signal`, jednak różnią się od poprzednich tym, że przyjmują one przynajmniej jeden inny sygnał jako parametr.

Opóźnienie

Określany w kodzie jako `Delay`. Przesuwa w czasie dowolny inny sygnał. Parametry:

- `D` – czas przesunięcia w mikrosekundach (`int32`)
- `S` – obiekt sygnału

Wartość bezwzględna

Określany w kodzie jako `Absolute`. Zwraca wartość bezwzględną dowolnego innego sygnału. Parametry:

- `S` – obiekt sygnału

Clamp / minmax

Określany w kodzie jako `Clamp`. Zwraca wartość dowolnego innego sygnału, chyba że wykracza poza podaną wartość minimalną lub maksymalną – w takim przypadku zwraca przekroczoną granicę. Parametry:

- `L` – dolna granica (`float`)
- `H` – górna granica (`float`)
- `S` – obiekt sygnału

Funkcja liniowa

Określany w kodzie jako `LinearMap`. Zwraca wartość dowolnego innego sygnału przekształconą za pomocą funkcji liniowej $y = A \cdot x + B$. Parametry:

- `A` – współczynnik kierunkowy (`float`)
- `B` – wyraz wolny (`float`)
- `S` – obiekt sygnału

Iloczyn sygnałów

Określany w kodzie jako `Multiply`. Generuje dwa podane sygnały i zwraca iloczyn ich wartości. Parametry:

- `S1` – obiekt sygnału 1
- `S2` – obiekt sygnału 2

3.7.3 Opis przykładowych generatorów

Poniżej przedstawione są struktury `JSON` reprezentujące kilka generatorów, tworzących różne przebiegi.

Obiekt ustawień generatora posiadającego jeden sygnał sinusoidalny o amplitudzie 1 V i okresie 1000 µs.

```
[
    {
        "A": 1,
        "S": {
            "WF": "Sine",
            "T": 1000
        }
    }
]
```

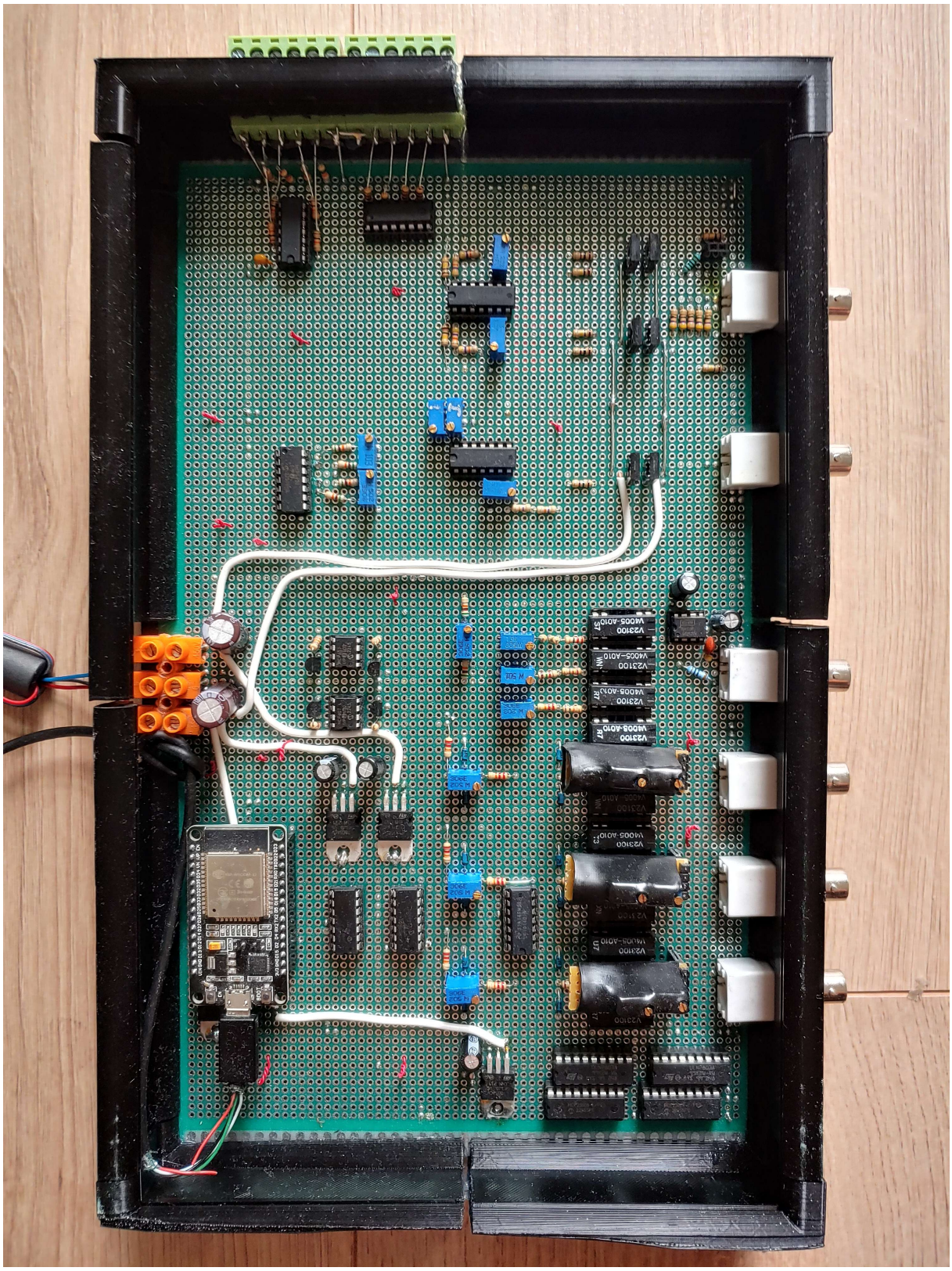
Obiekt ustawień generatora imitującego działanie inwertera - przybliżenie sinusa za pomocą dwóch sygnałów prostokątnych, o częstotliwości i wartości `RMS` równej zasilaniu sieciowemu.

```
[
    {
        "A": 1,
        "S": {
```

```
            "WF": "Square",
            "T": 20000,
            "D": 7100
        }
    },
    {
        "A": -1,
        "S": {
            "WF": "Delay",
            "D": 10000,
            "S": {
                "WF": "Square",
                "T": 20000,
                "D": 7100
            }
        }
    }
]
```

3.8 Prototyp bloku wejść-wyjść analogowych i cyfrowych

Poniżej przedstawione jest zdjęcie zbudowanego urządzenia. Całość ma wymiary 31 cm na 19 cm na 5.5 cm (32.5 cm na 20.5 cm na 5.5 cm uwzględniając zewnętrzne elementy złącz wejść i wyjść). Do stworzenia układu została wykorzystana płytka uniwersalna, na której umieszczono wszystkie komponenty. Po lewej stronie widać wejście zasilania ±12 V w postaci kostki elektrycznej. Obok znajdują się regulatory napięcia ±5 V oraz wzmacniacze napięć odniesienia ±4.096 V i ±2.048 V. Pod nimi (w lewym dolnym rogu) umieszczony jest mikrokontroler `ESP32` na swojej płytce ewaluacyjnej. Obok niego znajdują się przetwornik `A/C` oraz wzmacniacz, następnie sumatory napięcia i kolejny wzmacniacz. Po prawej stronie widać rząd przekaźników oraz sterujące nimi ekspandery i Darlingtony; obok nich znajdują się dzielniki napięcia i wzmacniacz pomiarowy oraz finalnie gniazda `BNC`. W górnej połowie płytki znajduje się przetwornik `C/A`, wzmacniacze oraz tranzystory mocy, tworzące wyjścia napięciowe i prądowe. W górnym lewym rogu umieszczone zostały układy (bufory i Darlingtony) obsługujące porty cyfrowe, oraz gniazda na terminale śrubowe stanowiące interfejs tychże. Ze względu na prototypową naturę budowanego urządzenia, działanie wielu części układu jest kalibrowane za pomocą potencjometrów wieloobrotowych.



Rys. 3.21: Zdjęcie zbudowanego prototypu urządzenia

Rozdział 4

Kalibracja i testy urządzenia

Do testów zostały w języku Python stworzone kody służące do wgrywania ustawień i przetwarzania pomiarów, ze względu na jego prostotę i szybkość tworzenia. Urządzenie najpierw zostało skalibrowane za pomocą kilku zadań (zostały ustawione potencjometry regulujące). Następnie zostały przeprowadzone trzy eksperymenty, mające na celu potwierdzenie możliwości wykorzystania zbudowanego urządzenia.

Aby ułatwić obsługę, została napisana niewielka klasa reprezentująca urządzenie. Pozwala ona w zwięzły sposób wgrywać ustawienia, pobierać dane na żywo oraz zapisywać i odczytywać pomiary w plikach za pomocą czterech metod.

```
1  import requests
2  import struct
3  import json
4
5  class IOBlock:
6      def __init__(self, addr="http://192.168.4.1"):
7          self.addr = addr
8
9      def settings(self, *, jsonstr=None, settings=None, task="",
       ↪  generators=[]):
10         if jsonstr is None:
11             if settings is None:
12                 settings = {"task": task, "generators": generators}
13             jsonstr = json.dumps(settings)
14         response = requests.post(self.addr + "/settings", jsonstr)
15         return json.loads(response.content)
16
17     def get_iter(self, format, tb=0):
```

```
18          with requests.get(self.addr + "/io?tb=" + str(tb), stream=True)
            ↪ as req:
19              req.raise_for_status()
20              for bytes in
                ↪ req.iter_content(chunk_size=struct.calcsize(format)):
21                  yield struct.unpack(format, bytes)
22
23      def write_file(self, fname, tb=0):
24          with requests.get(self.addr + "/io?tb=" + str(tb), stream=True)
            ↪ as req:
25              req.raise_for_status()
26              with open(fname, mode="wb") as fp:
27                  for chunk in req.iter_content(chunk_size=None):
28                      if chunk:
29                          fp.write(chunk)
30
31      @staticmethod
32      def read_file_iter(fname, format):
33          with open(fname, mode="rb") as fp:
34              for tpl in struct.iter_unpack(format, fp.read()):
35                  yield tpl
```

Listing 4.1: Klasa `IOBlock` upraszczająca obsługę urządzenia

Zadanie może być podane np. jako zmienna tekstowa lub odczyt z pliku, zaś generatory to po prostu tablica obiektów. Jest jednak możliwość podania gotowego obiektu ustawień lub nawet przygotowanego tekstu w formacie `JSON`.

4.1 Kalibracja wejść i wyjść

Najpierw zostały stworzone dwa mini-programy bazujące na przedstawionej wcześniej klasie. Jeden służy do wgrywania ustawień, zaś drugi do odbierania danych. Na samym początku należało ustawić potencjometry regulujące działanie wszystkich wejść i wyjść. Do kalibracji urządzenia potrzebne są dane "na żywo", dlatego strumień danych z urządzenia jest na bieżąco dzielony, konwertowany i wyświetlany w postaci liczbowej.

```
1 from IOBlock import IOBlock
2
3 iob = IOBlock()
```

```
fp = open("program.prg", "r")

ret = iob.settings(task=fp.read())
print(ret)
```

Listing 4.2: Program służący do wgrywania ustawień do urządzenia

```
from IOBlock import IOBlock

iob = IOBlock()

# depends on received data
format = "<iI"
format = "<fI"
format = "<BxxxI"

for tpl in iob.get_iter(format, 4):
    time = tpl[1]/1000000
    data = tpl[0]
    print("D:", data, "@:", time)
```

Listing 4.3: Program służący do odbierania pomiarów z urządzenia

4.1.1 Kalibracja wejść analogowych

Wejścia analogowe mają tylko jeden potencjometr, odpowiadający za ustawienie napięcia (lub prądu) zera. Zarówno zadanie jak i sposób kalibracji są bardzo proste; wejście zostaje zwarte do masy i potencjometr jest przestawiany tak długo, aż wartości zwracane przez urządzenie będą równe zero. Zadanie przez 60 sekund wykonuje pomiary 10 razy na sekundę i zwraca odczytaną wartość. W razie potrzeby może być powtórzone lub przerwane w dowolnym momencie. Dla wejścia prądowego zalecane może być wykorzystanie komendy `AIRDU`.

```
AIRNG <numer wejścia (1-4)> MIN;
AIEN;
LOOP 600;
   DELAY 100000;
```

```
5    AIRDM <numer wejścia (1-4)>;
6 END;
```

4.1.2 Kalibracja dzielników napięcia i wzmocnień

Teraz można ustawić dzielniki dla wejść napięciowych oraz wzmocnienie dla wejścia prądowego. W przypadku wejść napięciowych zakres `MIN` nie jest regulowany, w przypadku wejścia prądowego wszystkie trzy są. Na wejście podane jest jakieś znane napięcie, na przykład z któregoś regulatora, odniesienia wewnątrz urządzenia lub zewnętrznej baterii. Do porównania pomiarów został wykorzystany miernik uniwersalny `UT58D`. Potencjometry dzielników napięcia (lub regulujące wzmocnienie) są przestawiane, dopóki zwracane wartości nie zgadzają się z rzeczywistością.

```
1 AIRNG <numer wejścia (1-4)> MIN;
2 AIEN;
3 LOOP 600;
4    DELAY 100000;
5    AIRDM <numer wejścia (1-4)>;
6 END;
```

4.1.3 Kalibracja wyjścia napięciowego

Wyjście napięciowe ma dwa potencjometry. Jeden, podobnie jak w wejściach, odpowiada za ustawienie napięcia zera. Drugi odpowiada za dokładne ustawienie wzmocnienia.
Kalibracja pierwszego jest bardzo prosta: zadanie wpisuje do przetwornika kod odpowiadający zeru, następnie należy ustawić potencjometr tak, żeby napięcie faktyczne na wyjściu było równe zero.

```
1 AOVAL 1 0;
2 DELAY 100000000;
```

Drugi program posłużył do kalibracji wzmocnienia. Generuje on na zmianę napięcie bliskie granicznemu, tj. $\pm 10\,\mathrm{V}$. Należy przestawiać potencjometr w sprzężeniu tak długo aż napięcie na wyjściu dla obu polaryzacji zgadza się z wartością żądaną. Może być konieczne ponowne skorygowanie wcześniejszego potencjometru.

```
1 LOOP 100;
2    AOVAL 1 10;
```

```
3    DELAY 3000000;
4    AOVAL 1 -10;
5    DELAY 3000000;
6 END;
```

4.1.4 Kalibracja wyjścia prądowego

Wyjście prądowe ma aż pięć potencjometrów. Procedura jego kalibracji jest najbardziej skomplikowana. Jeden, podobnie jak w wyjściu napięciowym, odpowiada za ustawienie prądu zera. Dwa kolejne (po jednym na tor) odpowiadają za precyzyjne ustawienie transkonduktancji – zależności prądu wyjściowego od napięcia z przetwornika. Ostatnie dwa (po jednym na tor) odpowiadają za uniezależnienie prądu wyjściowego od podłączonego obciążenia (oporu).

Z tego względu ustawianie trwa trochę od końca. Najpierw niezależność od obciążenia jest osobno nastawiana dla obu kanałów. Polega na zadaniu jakiegoś niezbyt dużego prądu (jego wartość i tak będzie niedokładna). Do wyjścia podpięte są szeregowo opornik (np. 10–20 Ω) oraz miernik ustawiony na pomiar prądu.

Należy cyklicznie zwierać opornik i przestawiać potencjometr tak długo, aż płynący prąd przestanie się zmieniać przy zwieraniu lub rozwieraniu (a więc uniezależni się od obciążenia). Można taki test przeprowadzić również z większym oporem (np. 500 Ω) jednak prąd musi być odpowiednio niski, aby nie nastąpiła saturacja. Zadanie służące do tej kalibracji widnieje poniżej.

```
1 AOVAL 2 0.1;
2 DELAY 100000000;
```

Drugim ustawieniem jest prąd zera. Oba tory są ustawiane wspólnie, więc można mierzyć ten z mniejszym zakresem, gdyż to w nim błąd będzie lepiej widoczny. Należy przestawiać potencjometr tak długo, aż przez miernik przestanie płynąć prąd. Zadanie służące do tej kalibracji przedstawiono poniżej.

```
1 AOVAL 2 0;
2 DELAY 100000000;
```

Finalnie ustawiana jest transkonduktancja. Ponieważ boczniki nie mają idealnie dokładnej wartości (tak samo sprzężenia i same wzmacniacze), taka korekta jest wymagana. Teraz wartości mierzone muszą zgadzać się z zadanymi – należy tak ustawić potencjometr, aby było to prawdą. Powinno się to stać jednocześnie dla prądów dodatnich i ujemnych, jednak

jeśli tak nie jest – może być wymagana ponowna korekta prądu zera – aż obie wartości są symetryczne.

```
LOOP 100;
   AOVAL 2 0.1;
   DELAY 3000000;
   AOVAL 2 -0.1;
   DELAY 3000000;
END;
```

4.1.5 Test wejść cyfrowych

Wejścia cyfrowe nie są regulowane, więc zostało tylko przetestowane ich działanie. Ze względu na rezystory *pulldown*, stan jest domyślnie niski. Do każdego z wejść zostało podłączone napięcie (minimum 3.3 V) i sprawdzono, czy stan cyfrowy zmienia się poprawnie.
W programie odczytującym pomiary została wprowadzona minimalna zmiana, która powoduje wyświetlenie osobno bitów:

```
data = "{:08b}".format(tpl[0])
```

Zaś poniżej przedstawione jest zadanie odczytujące wejścia cyfrowe:

```
LOOP 10000;
   DIRD;
   DELAY 10000;
END;
```

4.1.6 Test wyjść cyfrowych

Wyjścia cyfrowe również nie są regulowane, więc zostało tylko przetestowane ich działanie. Ze względu na wykorzystanie tranzystorów Darlington, stan logiczny wysoki oznacza przyciągnięcie wyjścia do zera. Stan niski to wysoka impedancja.
Do wyjść zostały podłączone cztery diody o wspólnym zasilaniu. Program po kolei "odlicza" od 0 do 15, wyświetlając wynik w systemie dwójkowym.

```
LOOP 1000;
   DOWR 0; DELAY 10000;
```

```
3    DOWR 1; DELAY 10000;
4    DOWR 2; DELAY 10000;
5    DOWR 3; DELAY 10000;
6    DOWR 4; DELAY 10000;
7    DOWR 5; DELAY 10000;
8    DOWR 6; DELAY 10000;
9    DOWR 7; DELAY 10000;
10   DOWR 8; DELAY 10000;
11   DOWR 9; DELAY 10000;
12   DOWR A; DELAY 10000;
13   DOWR B; DELAY 10000;
14   DOWR C; DELAY 10000;
15   DOWR D; DELAY 10000;
16   DOWR E; DELAY 10000;
17   DOWR F; DELAY 10000;
18 ENDLOOP;
```

Wszystkie diody zaświecają się i gasną w poprawnej kolejności. Sprawdzona została również maksymalna częstotliwość – zmniejszając opóźnienia między instrukcjami, a finalnie całkiem je usuwając. Udało się uzyskać wydajność rzędu $< 10\,\mu s$ na przełączenie, a więc częstotliwość 100 kHz.

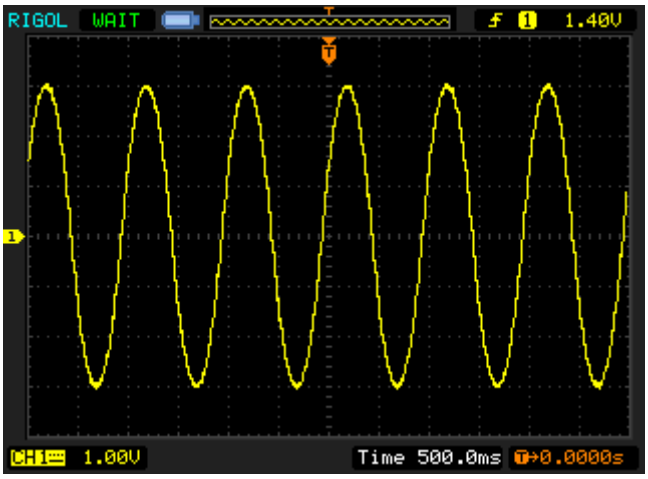
4.2 Sprawdzenie charakterystyki częstotliwościowej wyjścia

Aby określić rzeczywiste możliwości urządzenia, została zbadana charakterystyka częstotliwościowa toru wyjściowego napięciowego. Wyjście napięciowe zostało podłączone do oscyloskopu cyfrowego Rigol `DS1052E`. Za pomocą niżej przedstawionego programu zostały po kolei generowane sinusoidy o amplitudzie 3 V i różnych częstotliwościach: 1 Hz, 10 Hz, 100 Hz, 1 kHz, 10 kHz.

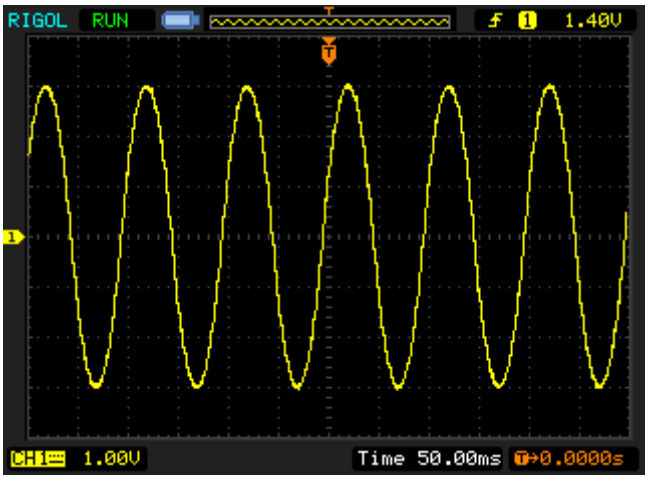
```
1 LOOP 20000;
2    AOGEN 1 0;
3    DELAY 25;
4 END;
```

Listing 4.4: Zadanie służące do generowania sinusoid

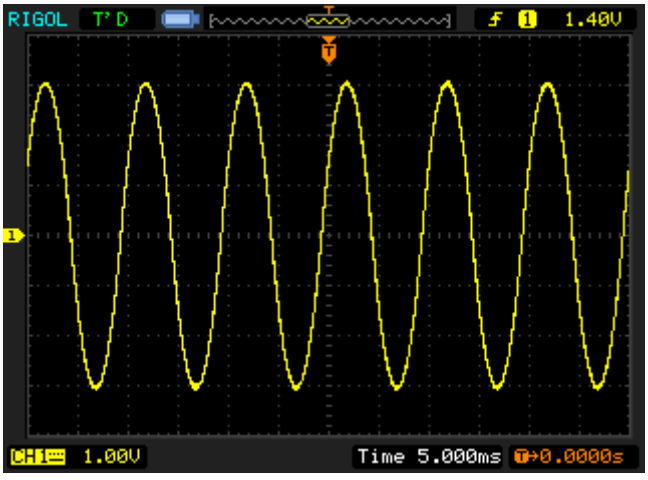
Każdy przebieg został zmierzony przez oscyloskop, czego wyniki są przedstawione poniżej:



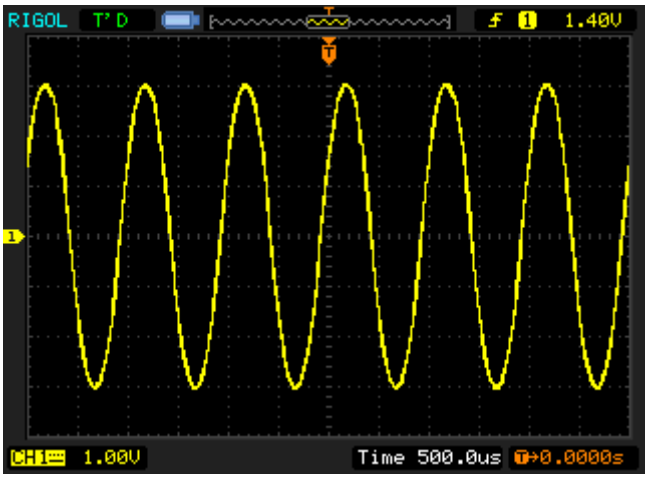
Rys. 4.1: Przebieg wygenerowanej sinusoidy o częstotliwości 1 Hz



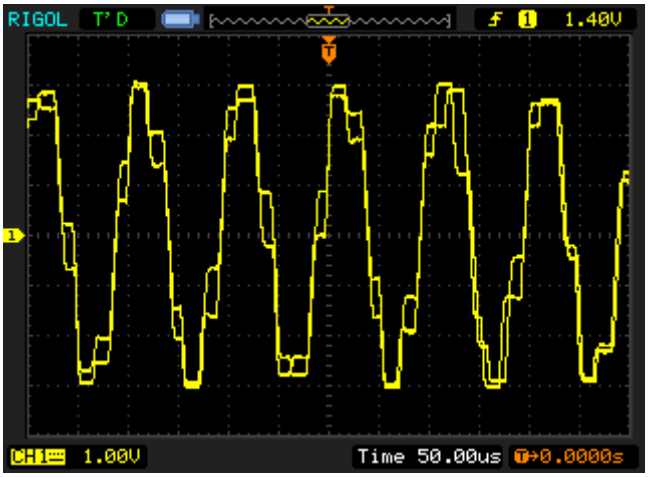
Rys. 4.2: Przebieg wygenerowanej sinusoidy o częstotliwości 10 Hz



Rys. 4.3: Przebieg wygenerowanej sinusoidy o częstotliwości 100 Hz

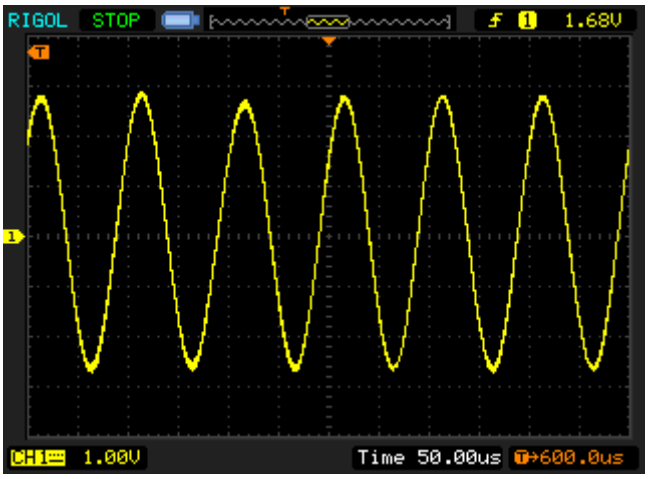


Rys. 4.4: Przebieg wygenerowanej sinusoidy o częstotliwości 1 kHz



Rys. 4.5: Przebieg wygenerowanej sinusoidy o częstotliwości 10 kHz

Tory wyjściowe nie posiadają filtrów dolnoprzepustowych, co widać na przedstawionych powyżej przebiegach. Z zarejestrowanych odpowiedzi czasowych można wyciągnąć kilka wniosków. Przy częstotliwości 10 kHz program pozwala na maksymalnie ok. 5-6 próbek na okres, tak więc przebieg jest wyraźnie zniekształcony. Napięcie wyjściowe potrafi osiągnąć zadaną wartość w czasie około 10 µs, nawet jeśli wymagana jest zmiana o kilka woltów.
Aby wygładzić przebieg napięcia wyjściowego, użytkownik może dołączyć własny filtr na wyjściu napięciowym. Pomiędzy urządzeniem a oscyloskopem został na próbę wpięty filtr aktywny model `3202R` firmy Krohn-Hite, przedstawiony na rysunku 4.7. Pozwoliło to na znaczną poprawę kształtu przebiegu, szczególnie przy najwyższej częstotliwości; zauważalny jest jednak niewielki spadek amplitudy. Filtr został ustawiony na dolnoprzepustowy z częstotliwością graniczą równą 20 kHz.



Rys. 4.6: Przebieg wygenerowanej sinusoidy o częstotliwości 10 kHz po filtrowaniu

Rys. 4.7: Panel przedni filtra model 3202R firmy Krohn-Hite

4.3 Przykładowe generatory – podstawowe sygnały

Następnie zostały przetestowane generatory funkcyjne. Wyjście napięciowe zostało podłączone do oscyloskopu, w celu wykonania pomiarów. Poniżej przedstawiony jest obiekt ustawień zawierający kilka testowanych generatorów oraz zadanie, które było odpowiedzialne za obsługę. Wszystkie przebiegi mają okres 1 ms oraz amplitudę 3 V. Przebieg prostokątny ma wypełnienie 50%. Przebieg trójkątny jest symetryczny, zaś piłokształtny – przechylony, z pionowym zboczem opadającym.

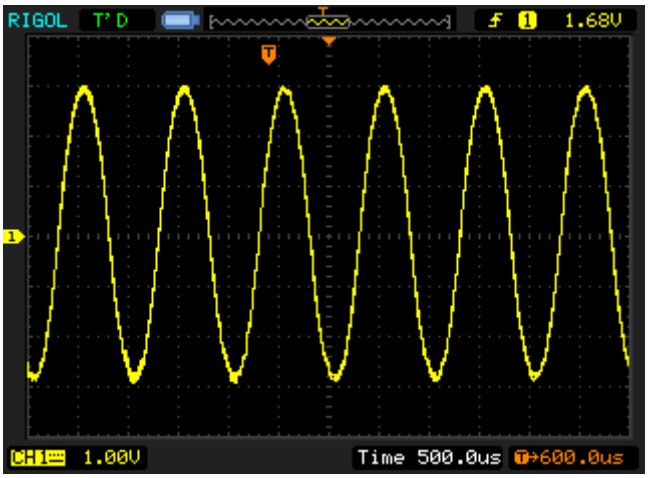
```
"generators": [
    [{"A": 3, "S": {"WF": "Sine", "T": 100000}}],
    [{"A": 3, "S": {"WF": "Square", "T": 100000, "D": 50000}}],
    [{"A": 3, "S": {"WF": "Triangle", "T": 100000, "P": 25000}}],
    [{"A": 3, "S": {"WF": "Triangle", "T": 100000, "P": 50000}}],
]
```

Listing 4.5: Przetestowane generatory

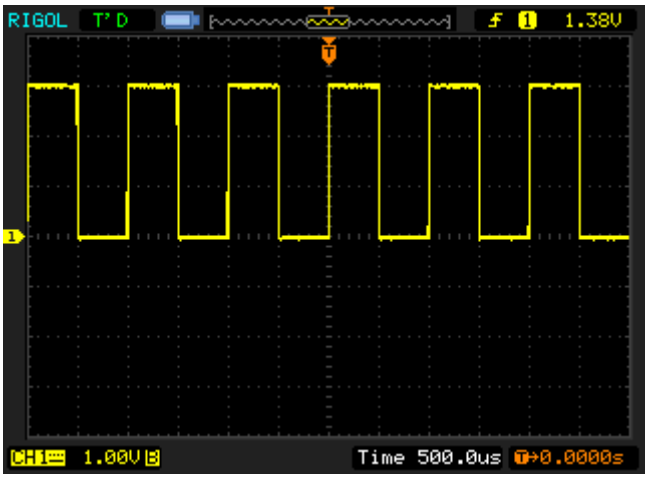
```
LOOP 20000;
   AOGEN 1 0;
   DELAY 25;
END;
```

Listing 4.6: Zadanie służące do generowania sinusoid

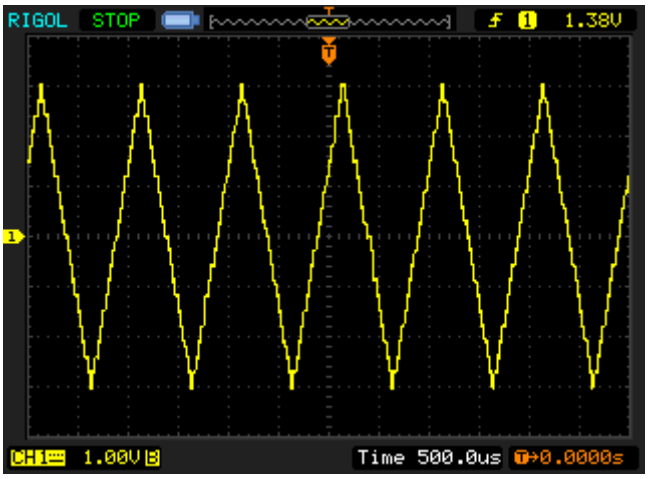
Wszystkie podstawowe przebiegi są generowane zgodnie z założeniami.



Rys. 4.8: Przebieg sygnału sinusoidalnego



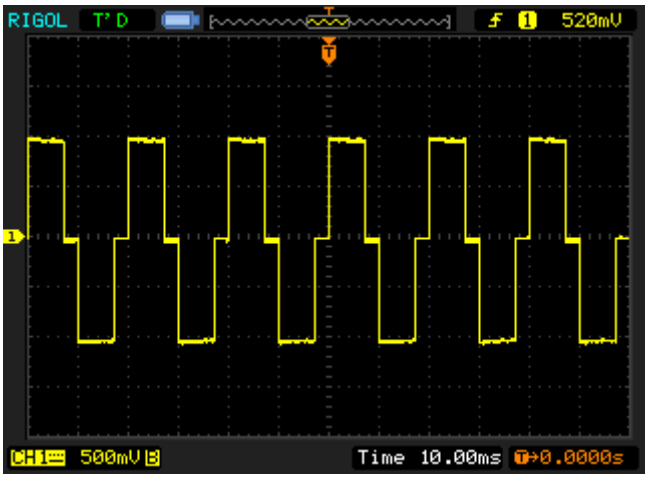
Rys. 4.9: Przebieg sygnału prostokątnego



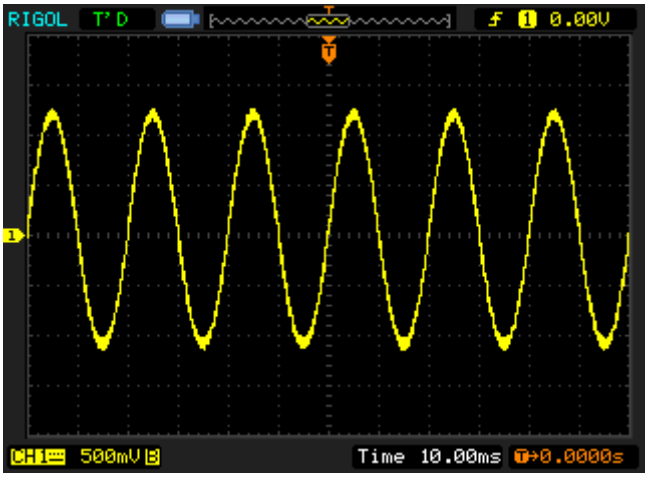
Rys. 4.10: Przebieg sygnału trójkątnego

Rys. 4.11: Przebieg sygnału piłokształtnego

Następnie sprawdzony został przykładowy przebieg, będący sumą dwóch przebiegów prostokątnych, przedstawiony w punkcie 3.7.3. Na drugim rysunku występuje w wersji przefiltrowanej wspomnianym wcześniej filtrem `2302R` ustawionym na dolnoprzepustowy z częstotliwością graniczną 100 Hz.

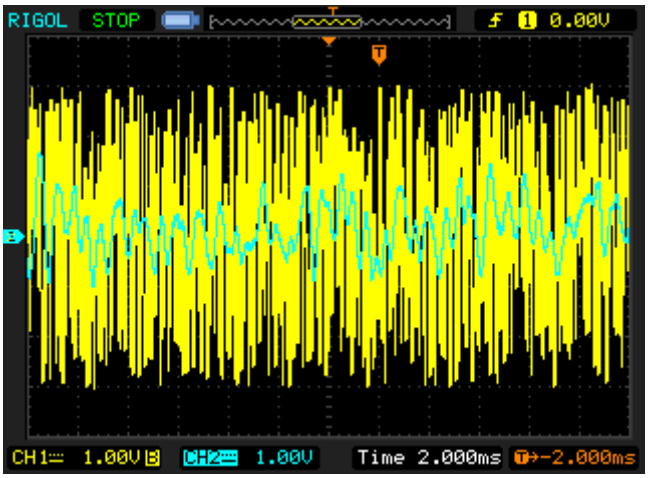


Rys. 4.12: Przebieg funkcji symulującej prosty inwerter

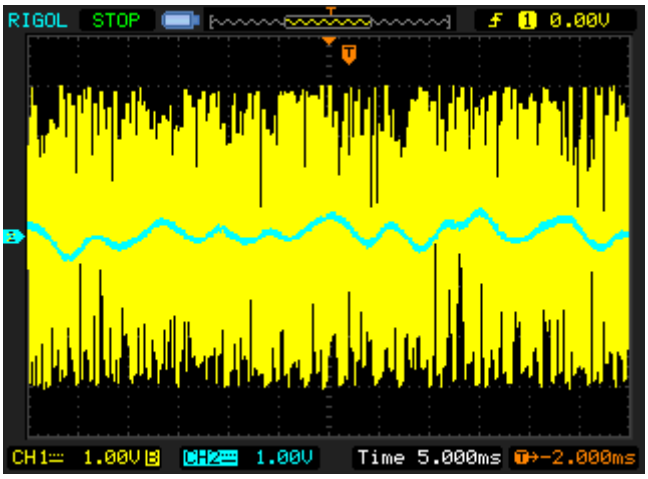


Rys. 4.13: Przebieg funkcji symulującej prosty inwerter po przefiltrowaniu

Finalnie, przetestowany został generator szumu – `Random`. Dodatkowo, na drugim kanale został przedstawiony ten sam sygnał, lecz po przefiltrowaniu z dwoma różnymi częstotliwościami granicznymi.



Rys. 4.14: Przebieg sygnału losowego i po lekkim przefiltrowaniu



Rys. 4.15: Przebieg sygnału losowego i po mocniejszym przefiltrowaniu

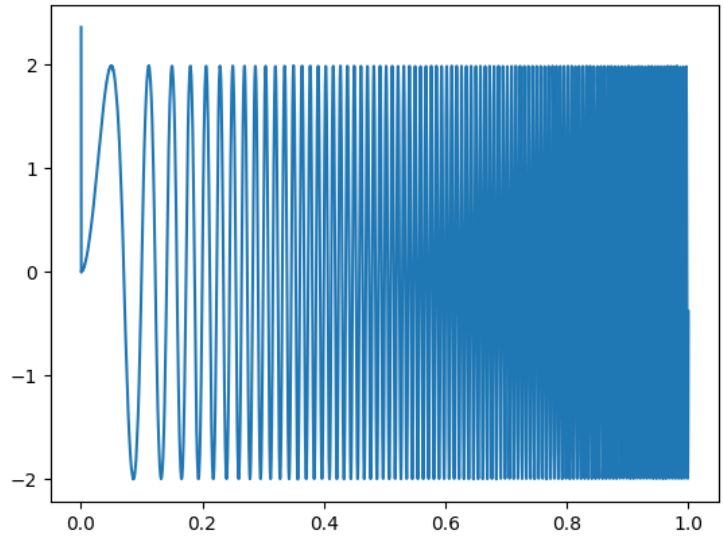
Przebieg świergotowy trwa pełną sekundę i zmienia się od 1 Hz do 100 Hz. Został on zmierzony poprzez wpięcie wyjścia do wejścia urządzenia, żeby w prosty sposób przechwycić całość sygnału.

```
"generators": [
    [{ "A": 3, "S": {"WF": "Chirp", "T": 1000000, "FS": 1 * 1e-6, "FD":
    ↪  99 * 1e-6} }],
    [{ "A": 3, "S": {"WF": "ChirpLog", "T": 1000000, "FS": 1 * 1e-6,
    ↪  "FR": 100}, }],
]
```

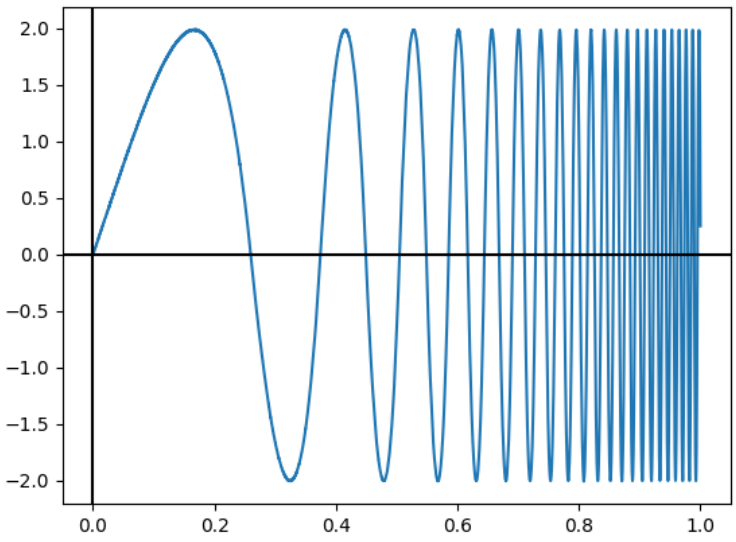
Listing 4.7: Przetestowane generatory sygnału świergotowego

```
AIRNG 1 MIN; AIEN; DELAY 100000; RSTTM;
LOOP 10000;
   AOGEN 1 0;
   AIRDF 1 1;
   DELAY 100;
END;
```

Listing 4.8: Zadanie generujące i mierzące sygnały



Rys. 4.16: Przebieg sygnału świergotowego



Rys. 4.17: Przebieg sygnału świergotowego logarytmicznego

4.4 Odpowiedź częstotliwościowa wejścia napięciowego

4.4.1 Badanie sygnałem sinusoidalnym

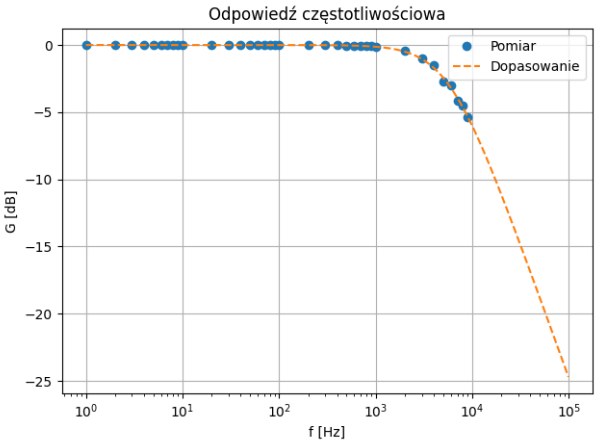
Aby określić rzeczywiste możliwości urządzenia, został przeprowadzony test odpowiedzi częstotliwościowej wejść. Wyjście napięciowe zostało podłączone do jednego z wejść napięciowych. Pełny kod programu odbierajacego i przetwarzającego pomiary dostępny jest w dodatku A.4.1.
Na wyjście został zadany sygnał sinusoidalny, gdzie częstotliwość była zwiększana dwukrotnie za każdym powtórzeniem programu. Jego napięcie szczytowe to ok. 3 V, zaś wartość RMS została zmierzona miernikiem `UT58D` jako minimalnie większa niż teoretyczna, tj. 2.22 V. Poniżej jest przedstawione zadanie odpowiedzialne za generację i pomiary:

```
1 AIRNG 1 MIN; AIEN; DELAY 100000; RSTTM;
2 LOOP 20000;
3    AOGEN 1 0;
4    AIRDF 1 1;
5    DELAY 50;
6 END;
```

Listing 4.9: Zadanie służące do badania odpowiedzi częstotliwościowej

Każdy przebieg został zmierzony przez urządzenie i wyliczona została jego wartość skuteczna napięcia. Następnie przeliczono ją na decybele względem sygnału wejściowego. Są one przedstawione na rysunkach niżej.
Do wzoru na odpowiedź amplitudową prostego filtru dolnoprzepustowego pierwszego stopnia wstawiono zmierzone częstotliwości i wzmocnienia i za pomocą biblioteki `SciPy` algorytmem minimalizacji została wyznaczona częstotliwość graniczna f_c.

$$G = 20\lg\left(\frac{1}{\sqrt{1+\left(\frac{f}{f_c}\right)^2}}\right) = -10\lg\left(1+\left(\frac{f}{f_c}\right)^2\right)$$

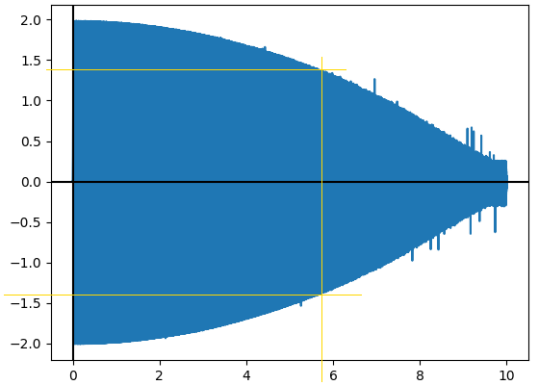


Rys. 4.18: Punkty pomiarowe wraz z dopasowaną funkcją

Częstotliwość graniczna została wyznaczona na około 5765 Hz. Tak niska częstotliwość graniczna wynika z kilku kwestii: ręcznej budowy na płytce uniwersalnej (a więc niezoptymalizowanych ścieżek), dużego rozmiaru dzielników i przekaźników oraz wysokiej rezystancji wejściowej (przez co nawet minimalna pojemność parazytowa mocno ogranicza częstotliwość).

4.4.2 Badanie sygnałem świergotowym

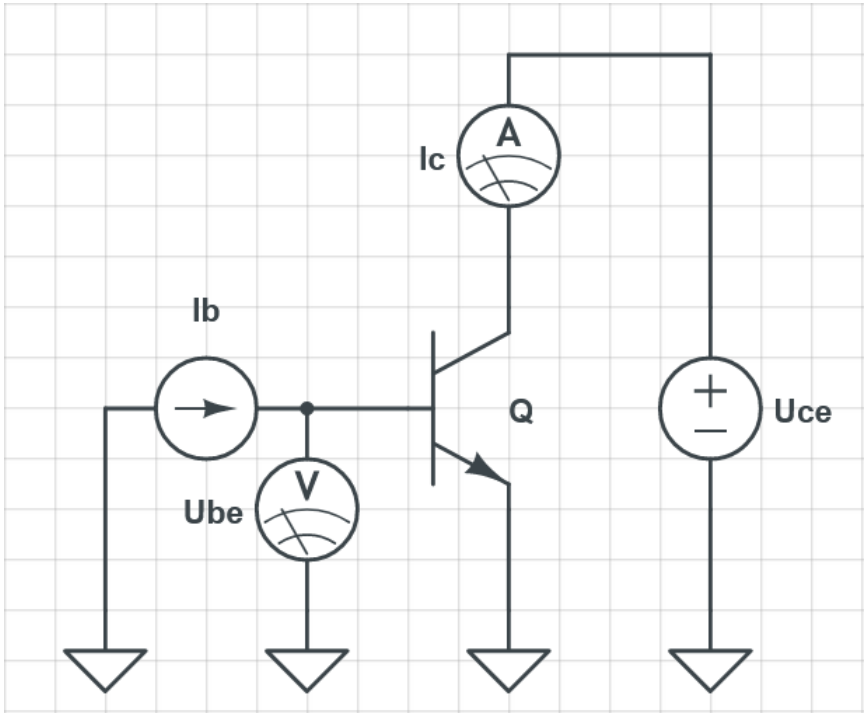
W celu weryfikacji, eksperyment został powtórzony używając sygnału `Chirp`. W ciągu 10 sekund przechodzi on od zera do 10 kHz, tak więc w tym przypadku czas i częstotliwość są równe (pomijając rzędy wielkości). Wizualnie z wykresu została odnaleziona amplituda mniejsza $\sqrt{2}$-krotnie od maksymalnej i zaznaczona jej częstotliwość. Wynik jest zbliżony do poprzedniej metody.



Rys. 4.19: Amplituda pomiarów sygnału świergotowego w funkcji czasu/częstotliwości

4.5 Eksperyment 1 – wyznaczenie charakterystyk tranzystora BJT

Zostały przetestowane trzy charakterystyki statyczne tranzystora bipolarnego. Są to: wejściowa, przejściowa i wyjściowa. Badania te zostały powtórzone dla dwóch tranzystorów: `2N3904` oraz `2N3055`.
Emiter tranzystora jest połączony z masą. Do bazy zostało podłączone wyjście prądowe oraz wejście napięciowe. Kolektor został zasilony z wyjścia napięciowego poprzez wejście prądowe. W ten sposób znamy (poprzez zadawanie lub mierzenie) wszystkie cztery potrzebne wielkości – prąd bazy, napięcie baza-emiter, prąd kolektora, napięcie kolektor-emiter. Schemat połączeń wykorzystany w celu przeprowadzenia doświadczenia jest przedstawiony poniżej.



Rys. 4.20: Schemat układu użytego do przeprowadzenia doświadczeń

4.5.1 Badanie charakterystyki wejściowej tranzystora

Do wyznaczenia tej charakterystyki należy dla kilku wartości U_{CE} przedstawić przebieg zależności U_{BE} od I_B. W tym celu został stworzony program w języku Python, który wgrywa ustawienia, zapisuje pomiary i przetwarza je. Zawiera on też zadanie i obiekt generatora. Wpisując wartość zmiennej `Uce` możemy w prosty sposób ustawiać generowane przez zadanie napięcie kolektor-emiter. Urządzenie wymusza przez bazę liniowo narastający prąd w przedziale 0–1 mA i mierzy napięcie baza-emiter. Pomiary te zostają następnie zapisane do pliku. Poniżej przedstawione jest zadanie, zaś pełny kod zarządzający odbio-

rem i analizą pomiarów dostępny jest w dodatku A.4.2.

```
AOVAL 1 {Uce};
AIRNG 1 MIN; AIEN; DELAY 100000; RSTTM;
LOOP 1000;
   AOGEN 2 0;
   AIRDF 1 50;
   DELAY 1000;
END;
```

4.5.2 Badanie charakterystyki przejściowej tranzystora

Do wyznaczenia tej charakterystyki należy dla kilku wartości U_{CE} przedstawić przebieg zależności I_C od I_B. W tym celu został lekko zmodyfikowany program oraz zadanie. Wpisując wartość zmiennej `Uce` możemy w prosty sposób ustawiać napięcie kolektor-emiter. Urządzenie wymusza przez bazę liniowo narastający prąd w przedziale 0–1 mA i mierzy prąd kolektora. Pomiary te zostają następnie zapisane do pliku. Poniżej przedstawione jest zadanie, zaś kod zarządzający odbiorem i analizą pomiarów dostępny jest w dodatku A.4.3.

```
AOVAL 1 {Uce};
AIRNG 4 MAX; AIEN; DELAY 100000; RSTTM;
LOOP 1000;
   AOGEN 2 0;
   AIRDF 4 50;
   DELAY 1000;
END;
```

4.5.3 Badanie charakterystyki wyjściowej tranzystora

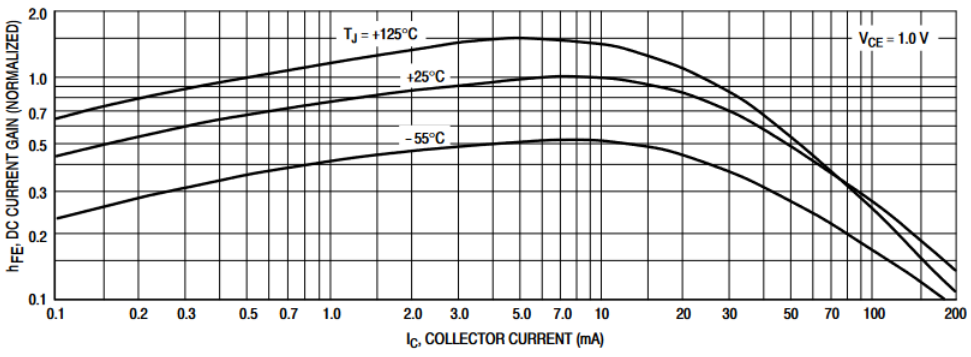
Do wyznaczenia tej charakterystyki należy dla kilku wartości I_B przedstawić przebieg zależności I_C od U_{CE}. W tym celu został lekko zmodyfikowany program oraz zadanie. Wpisując wartość zmiennej `Ib` możemy w prosty sposób ustawiać prąd bazy. Urządzenie zasila kolektor liniowo narastającym napięciem w przedziale 0–10 V i mierzy prąd kolektora. Pomiary te zostają następnie zapisane do pliku. Poniżej przedstawione jest zadanie, zaś kod zarządzający odbiorem i analizą pomiarów dostępny jest w dodatku A.4.4.

```
AOVAL 2 {Ib};
AIRNG 4 MAX; AIEN; DELAY 100000; RSTTM;
```

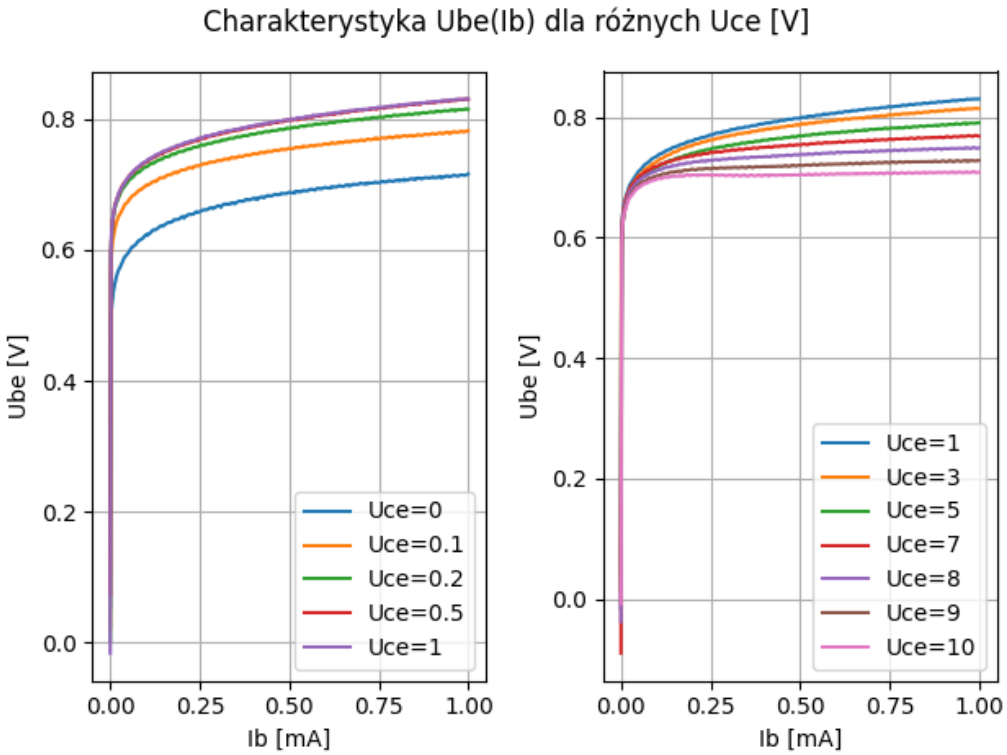
```
3 LOOP 1000;
4    AOGEN 1 0;
5    AIRDF 4 50;
6    DELAY 1000;
7 END;
```

4.5.4 Wykresy charakterystyk tranzystorów 2N3904 i 2N3055 wyznaczone za pomocą prototypu bloku wejść-wyjść

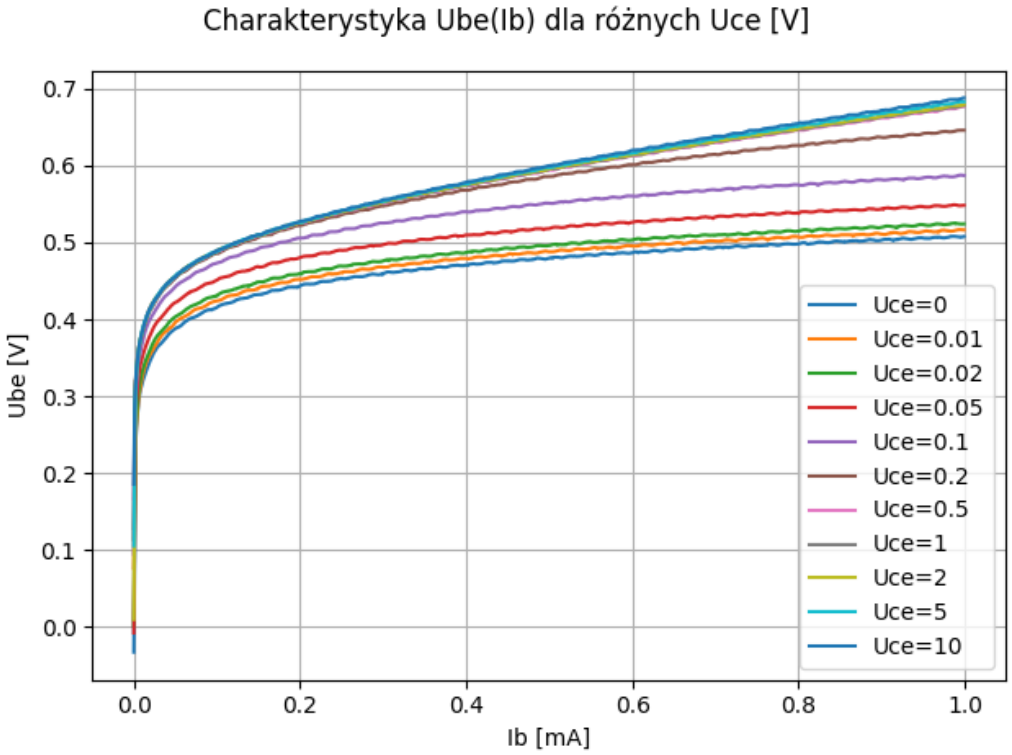
Zbadane zostały dwa tranzystory: 2N3904 [8] oraz 2N3055 [1]. Różnią się one parametrami, co widać też na charakterystykach. Rezultaty wszystkich pomiarów znajdują się poniżej, sześć rysunków przedstawiających trzy charakterystyki obu tranzystorów.
Jedyna nietypowość dostrzeżona w zmierzonych charakterystykach to kształt charakterystyk przejściowych. W tranzystorze 2N3904 występuje wyraźne zgięcie, co oznacza zmniejszenie wzmocnienia prądowego. Jest to jednak możliwe do wytłumaczenia na podstawie danych producenta [8] – na rysunku 4.21 (Fig. 15 w oryginalnym dokumencie) można zauważyć spadek wzmocnienia prądowego, występujący powyżej prądu kolektora 20 mA, nawet dwu- czy trzykrotny. Zakładając typowe wzmocnienie dla tego zakresu prądu około 100, spadek występuje przy prądzie bazy 0.2 mA, czyli tak jak na wykresach przedstawiających przeprowadzone pomiary.



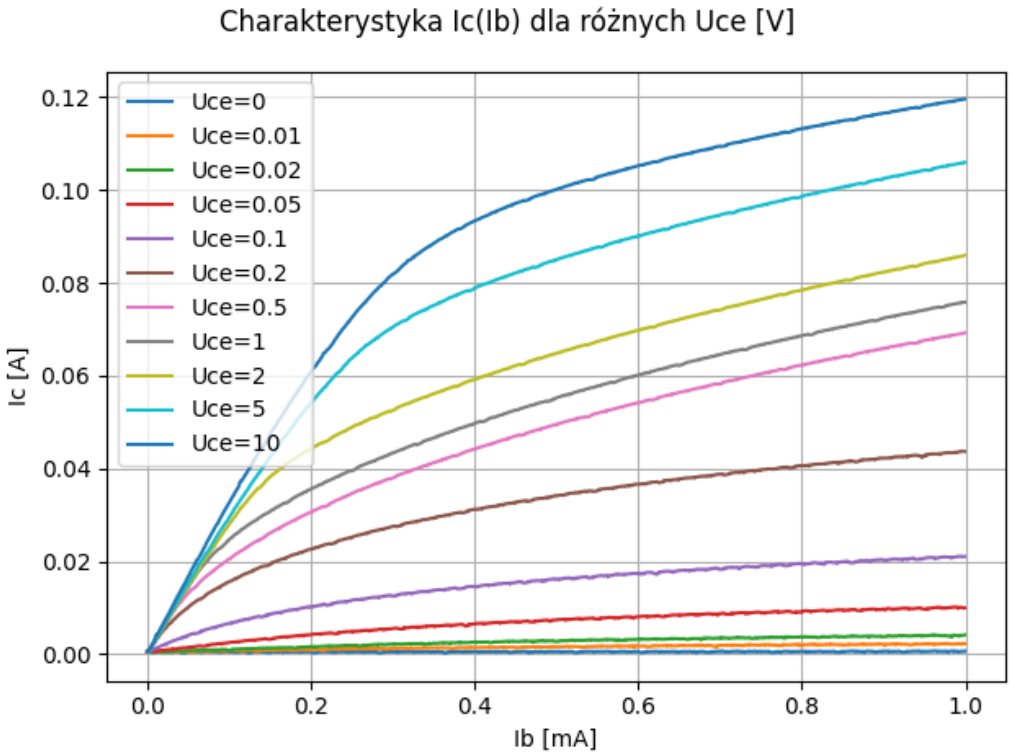
Rys. 4.21: Zależność wzmocnienia od prądu kolektora tranzystora 2N3904 [8]



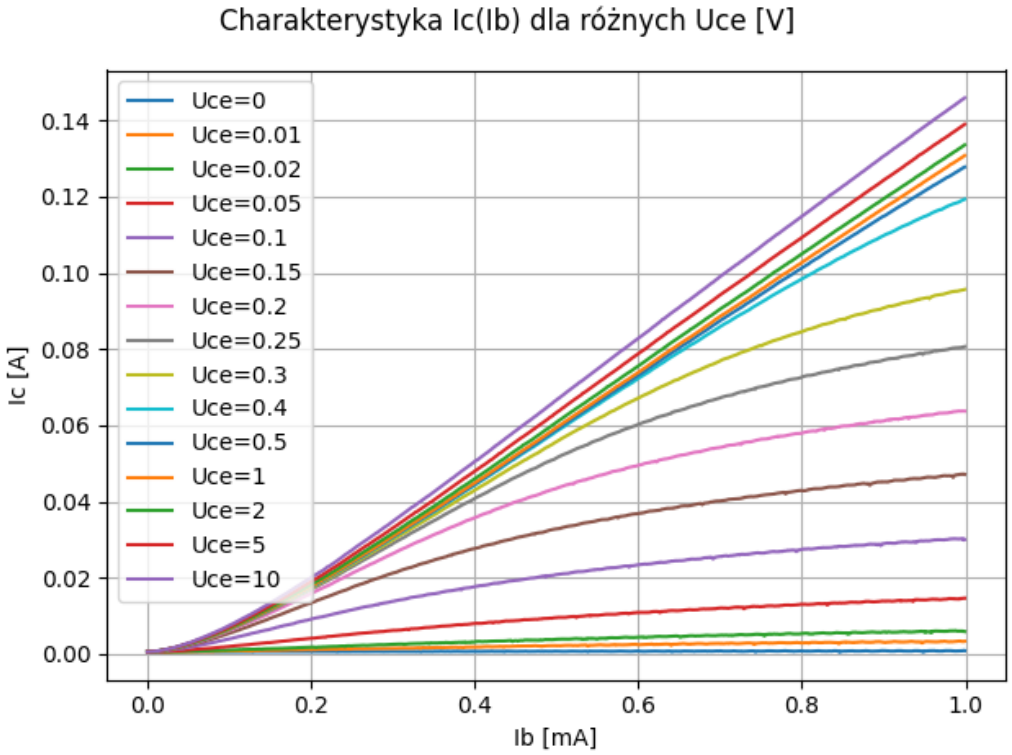
Rys. 4.22: Charakterystyka wejściowa tranzystora 2N3904



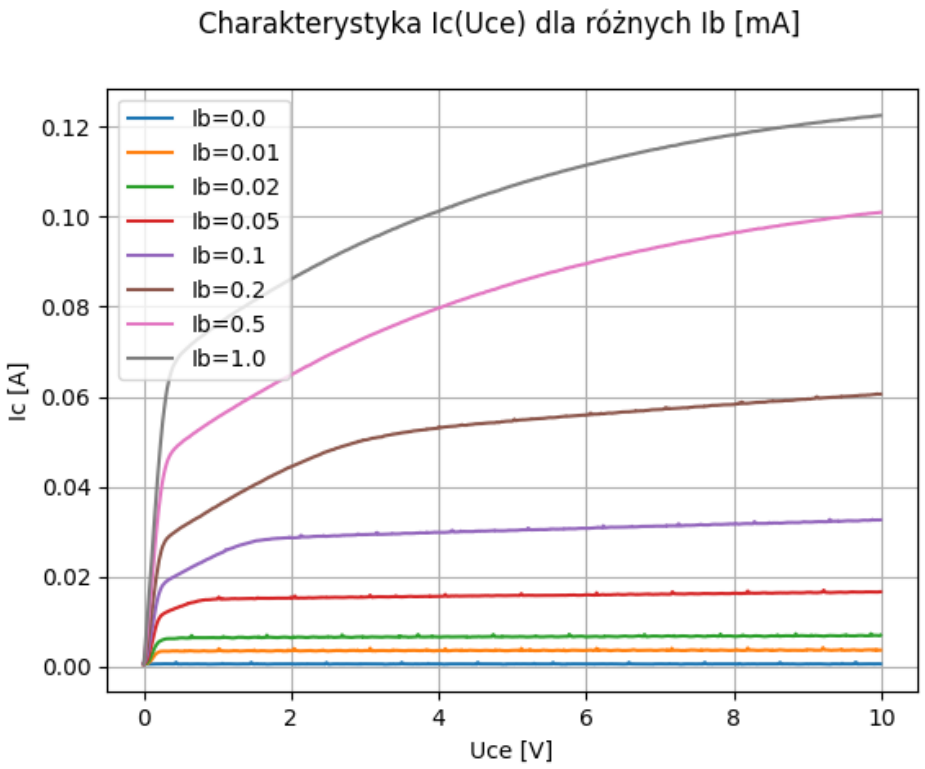
Rys. 4.23: Charakterystyka wejściowa tranzystora 2N3055



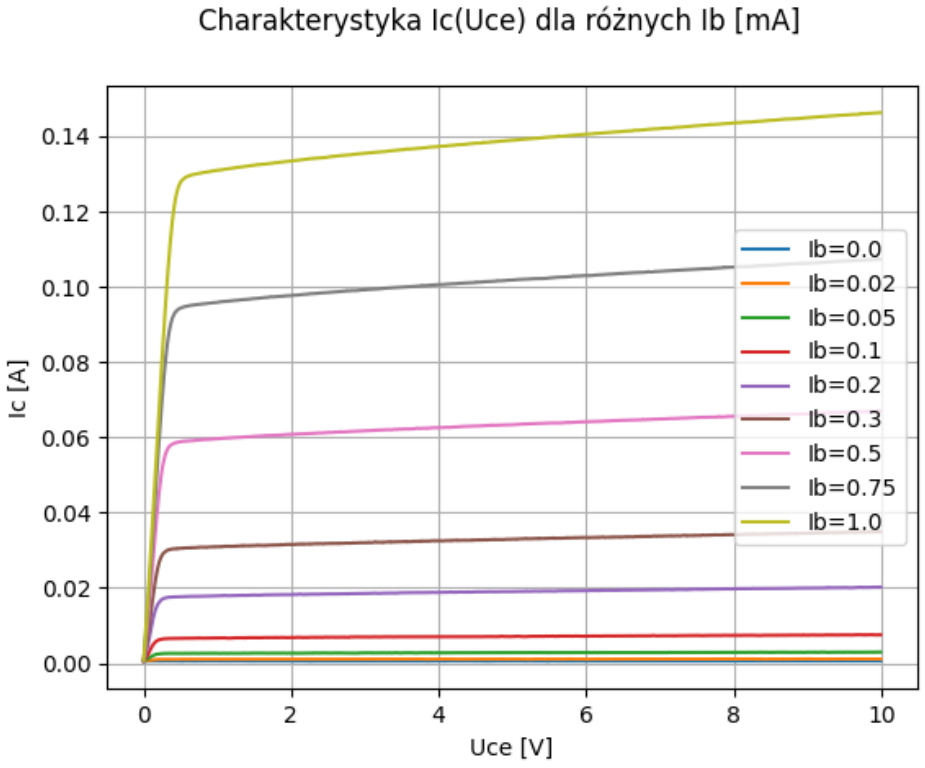
Rys. 4.24: Charakterystyka przejściowa tranzystora `2N3904`



Rys. 4.25: Charakterystyka przejściowa tranzystora `2N3055`



Rys. 4.26: Charakterystyka wyjściowa tranzystora `2N3904`

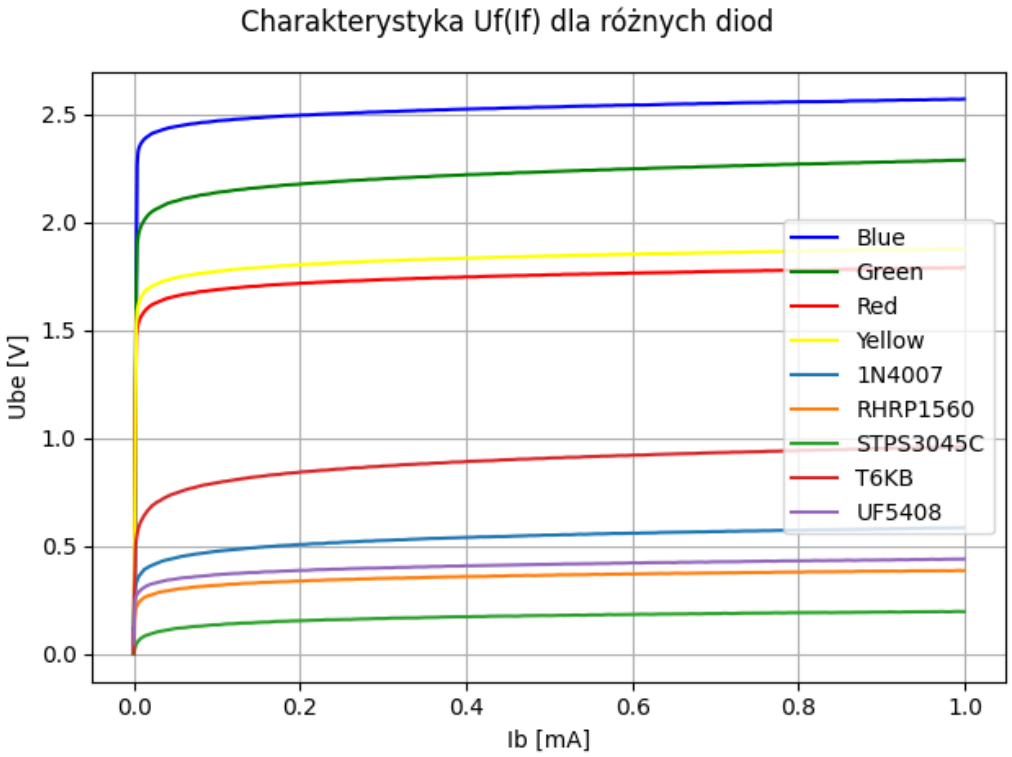


Rys. 4.27: Charakterystyka wyjściowa tranzystora `2N3055`

4.6 Eksperyment 2 – wyznaczenie charakterystyk diod półprzewodnikowych

Do wyznaczenia tej charakterystyki zostały użyte program oraz zadanie bardzo podobne do tych użytych do wyznaczenia charakterystyki wejściowej tranzystora. Urządzenie wymusza przez diodę liniowo narastający prąd w przedziale 0–1 mA i mierzy napięcie anoda-katoda. Pomiary te zostają następnie zapisane do pliku. Poniżej przedstawione jest zadanie:

```
AIRNG 1 MIN; AIEN; DELAY 100000; RSTTM;
LOOP 1000;
   AOGEN 2 0;
   AIRDF 1 50;
   DELAY 1000;
END;
```



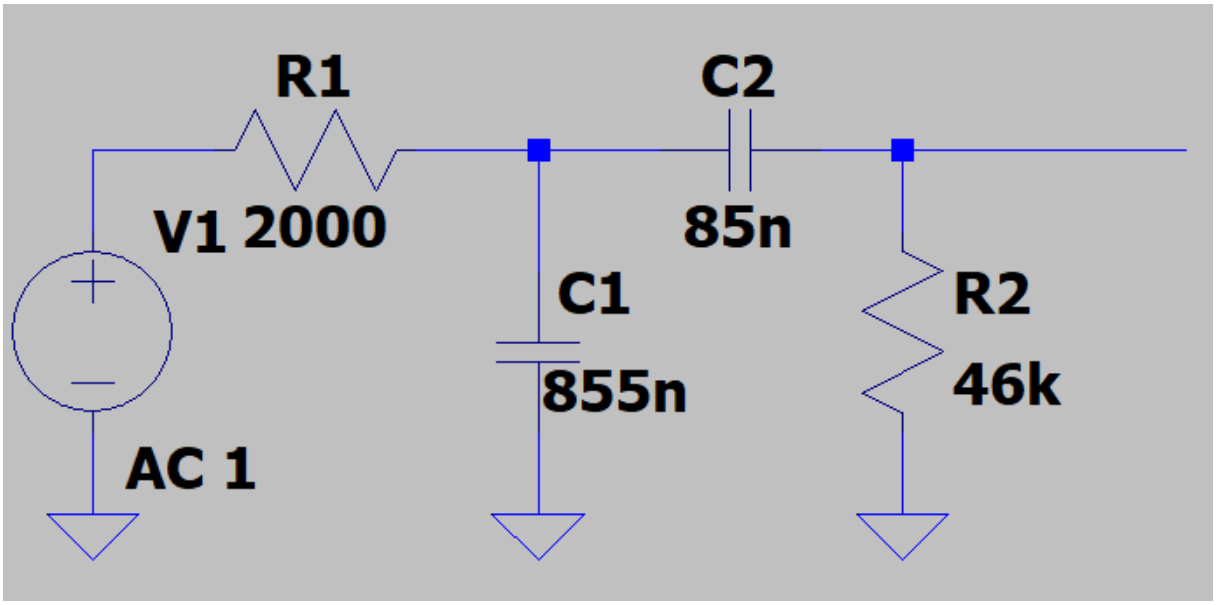
Rys. 4.28: Charakterystyki różnych diod

Można zauważyć, że największy spadek mają diody `LED`, tym większy im wyższa częstotliwość wytwarzanej fali światła. Pod nimi jest mostek prostowniczy `T6KB` (ze względu na podwójną diodę między `-` i `+`), później niewielka dioda krzemowa z popularnej rodziny

1N400X, po niej dioda prostownicza trochę wyższego prądu rodziny UF540X, za nią dioda RHRP1560 w formie TO-220, zaś na samym końcu wyspecjalizowana dioda prostownicza STPS3045C w formie TO-245 używana np. w zasilaczach komputerowych, z bardzo niskim spadkiem oraz wysokim maksymalnym prądem.

4.7 Eksperyment 3 – wyznaczenie odpowiedzi częstotliwościowej filtra

Została przetestowana odpowiedź częstotliwościowa prostego filtra RC typu pasmowoprzepustowego. Do pomiarów została dopasowana funkcja odpowiedzi – wyznaczona w ten sposób została częstotliwość graniczna. Dla porównania, filtr został również zasymulowany programem LTSpice.



Rys. 4.29: Schemat filtra oraz wartości komponentów

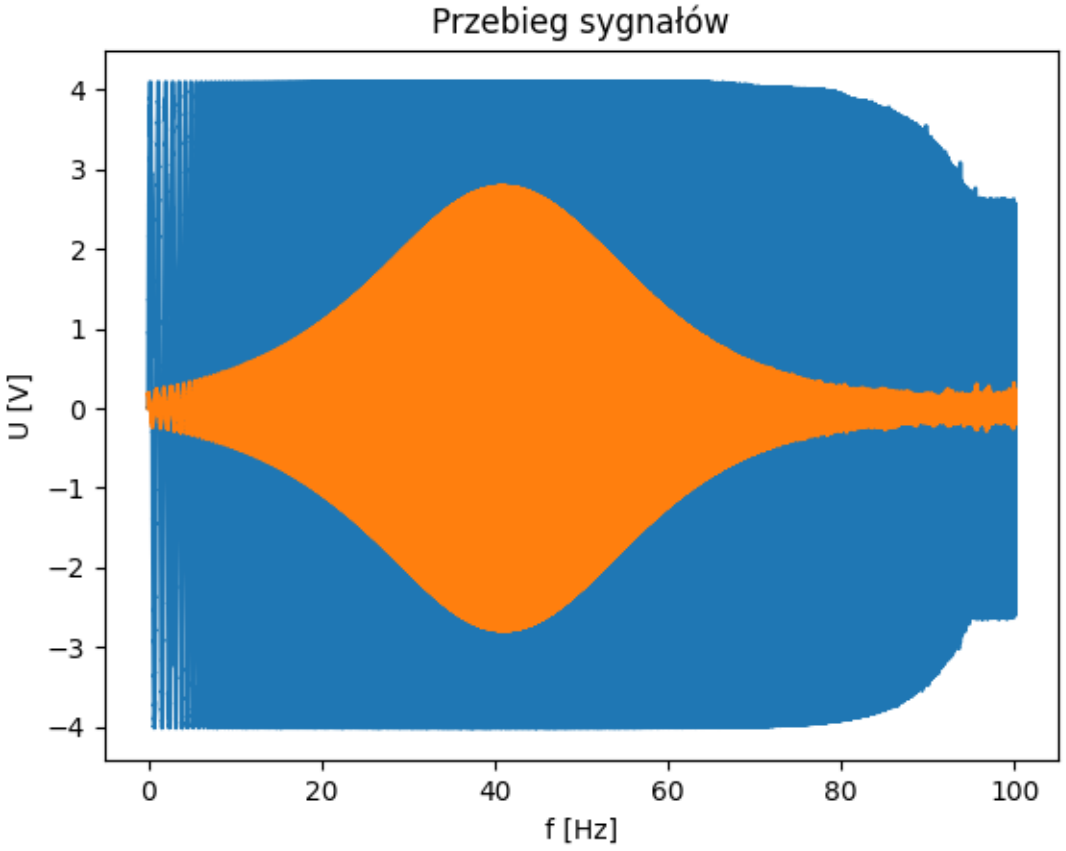
Na podstawie wartości komponentów wyznaczono: częstotliwość graniczna części górnoprzepustowej $f_H = 93.1\,\text{Hz}$, częstotliwość graniczna części dolnoprzepustowej $f_L = 40.7\,\text{Hz}$. Oczywiście to zakłada brak wzajemnego obciążania obu części, co nie jest prawdą, jednak obliczone częstotliwości służą jako punkt odniesienia.

Badanie sygnałem świergotowym

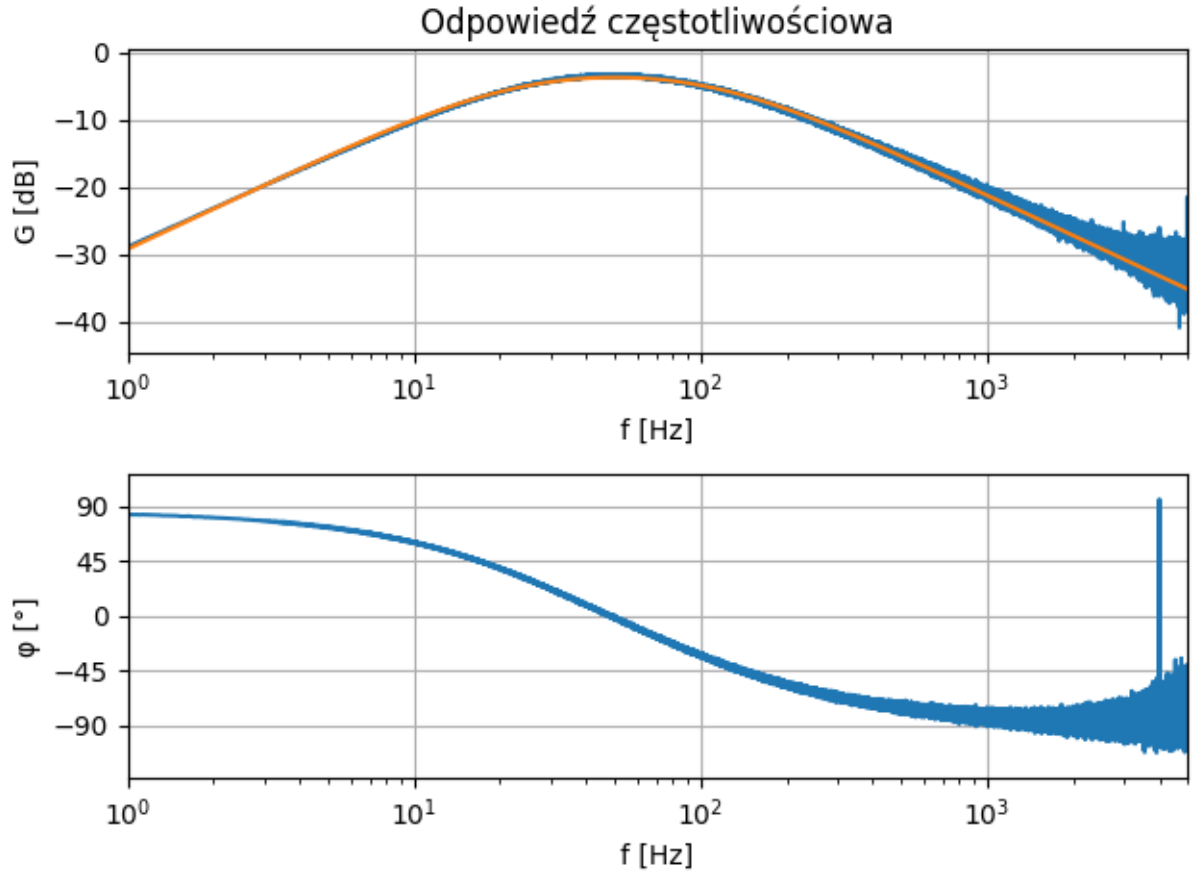
Pomiary zostały dokonane w następujący sposób: na wejście podany został sygnał świergotowy o częstotliwości zmieniającej się 1–10000 Hz. Został on dokładnie zmierzony. Następnie zadanie zostało wykonane ponownie, jednak tym razem mierząc wyjście. Oba sygnały zostały przetworzone transformatą Fouriera w celu otrzymania spektrum, zaś następnie

spektrum wyjścia zostało znormalizowane, dzieląc je przez spektrum wejścia. Na podstawie tego ilorazu można wyznaczyć charakterystykę częstotliwościową zarówno amplitudy jak i fazy.
Poniżej przedstawione jest zadanie dla urządzenia, zaś cały kod w języku Python odpowiedzialny za transmisję ustawień i pomiarów oraz obliczenia dostępny jest w dodatku A.4.5.

```
1 AIRNG 1 MIN; AIEN; DELAY 100000; RSTTM;
2 LOOP 2000000;
3    AOGEN 1 0;
4    AIRDF 1 1;
5    DELAY 50;
6 END;
```



Rys. 4.30: Przebieg sygnałów wejściowego i wejściowego

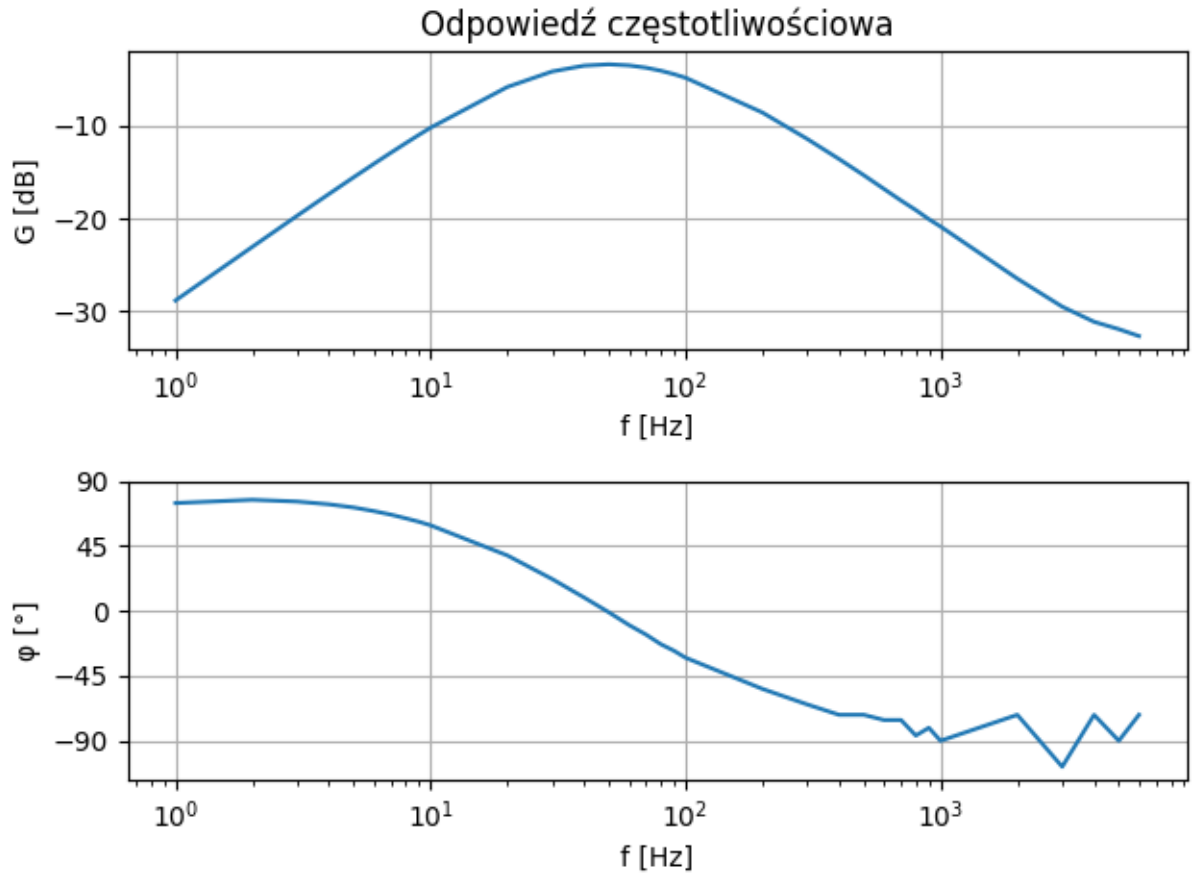


Rys. 4.31: Odpowiedź częstotliwościowa amplitudy i fazy

Parametry oszacowane za pomocą dopasowania mają następujące wartości: częstotliwość graniczna części górnoprzepustowej $f_H = 85.7\,\text{Hz}$, częstotliwość graniczna części dolnoprzepustowej $f_L = 30.2\,\text{Hz}$.

Badanie sygnałem sinusoidalnym

W celu weryfikacji, eksperyment został też przeprowadzony w inny sposób. Dla pewnego zbioru częstotliwości zostały wygenerowane fale sinusoidalne oraz wykonane pomiary wejścia i wyjścia. Dla każdej z częstotliwości zostały policzone wartości napięcia `RMS` wejścia i wyjścia w celu wyznaczenia tłumienia, zaś używając korelacji tych sygnałów zostało wyznaczone przesunięcie w fazie. W każdym przypadku została użyta tylko druga połowa pomiarów, w celu ustabilizowania działania filtra.
Zadanie jest takie samo, jedynie zmienił się generator; cały kod w języku Python odpowiedzialny za pomiary i obliczenia dostępny jest w dodatku A.4.6.

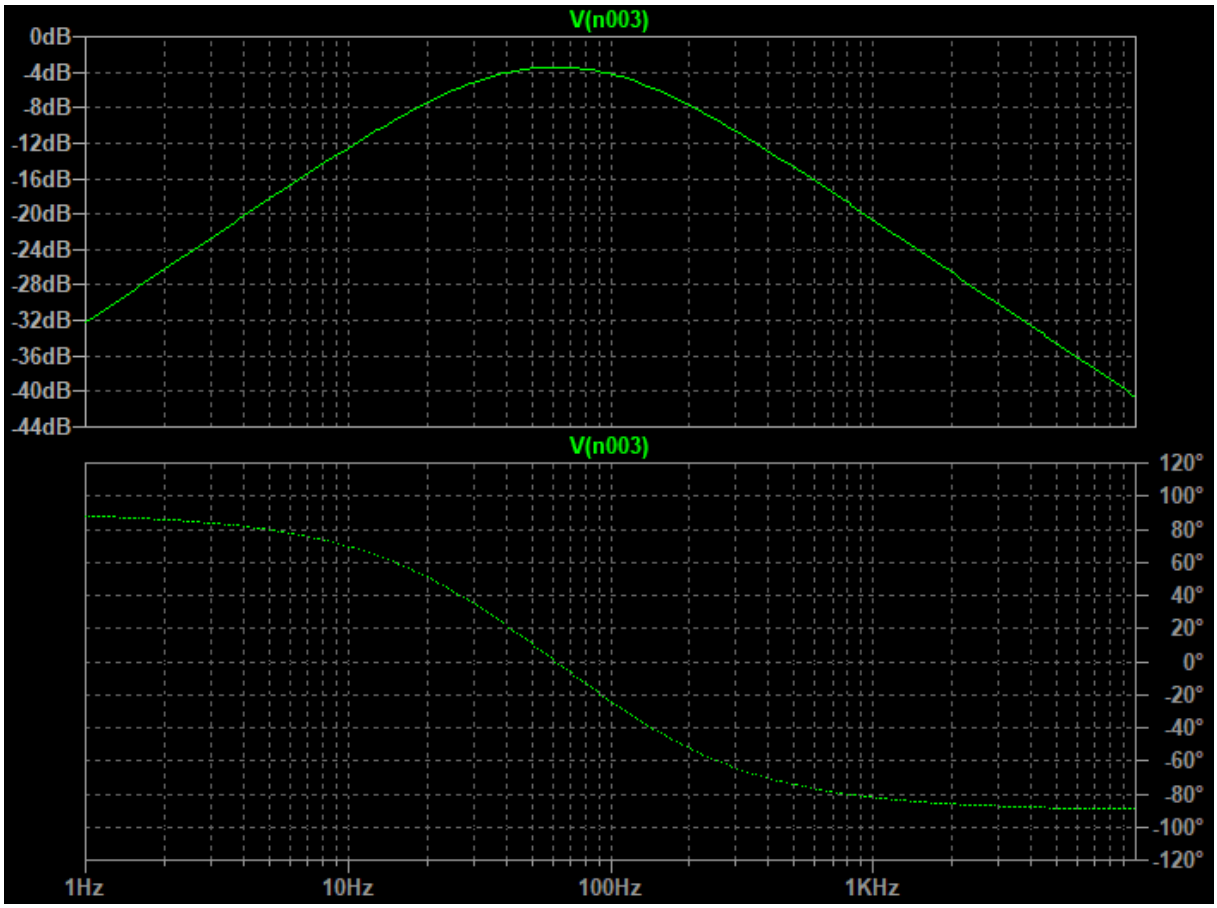


Rys. 4.32: Odpowiedź częstotliwościowa amplitudy i fazy

Łatwo zauważyć, że wyznaczona charakterystyka jest bardzo podobna do tej z poprzedniego punktu. pomijając zakłócenia w wyznaczonej fazie przy wyższych częstotliwościach.

Porównanie z symulacją

Filtr został też przetestowany za pomocą programu `LTSpice`. Wyniki są bardzo zbliżone, chociaż oczywiście program nie zawiera zakłóceń i innych błędów pomiaru, przez co jego wykres jest gładszy.



Rys. 4.33: Odpowiedź częstotliwościowa amplitudy i fazy

Rozdział 5

Podsumowanie

W ramach niniejszej pracy magisterskiej stworzono uniwersalny blok wejść-wyjść analogowych i cyfrowych, sterowany za pomocą komputera PC. Przeanalizowano rodzaje przetworników i wybrano najbardziej odpowiednie do tego typu zastosowania. Zaprojektowano tory wejść i wyjść oraz dobrano odpowiednie wzmacniacze i inne komponenty wspomagające. Sprawdzono różne sposoby komunikacji z urządzeniem, po czym wybrano taki, który spełniał wszystkie założenia projektu i oferował najlepszą wydajność. Stworzono prosty język, pozwalający na sterowanie pracą urządzenia przez użytkownika. Zbudowano prototyp bloku wejść-wyjść z wykorzystaniem płytki uniwersalnej. Mikrokontroler oprogramowano tak, żeby zarządzał układami znajdującymi się na płytce oraz prowadził dwukierunkową komunikację z użytkownikiem, w postaci przyjmowania ustawień oraz odsyłania dokonanych pomiarów. Przeprowadzono kalibrację działania torów wejść i wyjść. Przetestowano właściwości urządzenia pod względem charakterystyk częstotliwościowych oraz dokładności generowanych sygnałów. Przeprowadzono przykładowe eksperymenty demonstrujące możliwości zbudowanego układu oraz, w celu weryfikacji ich poprawności, porównano wyniki do obliczeń teoretycznych, symulacji lub danych z kart katalogowych. Uzyskane wyniki testów i eksperymentów pozwalają wnioskować, że cel pracy został zrealizowany. Urządzenie spełnia założenia projektowe, a wykonane eksperymenty wykazały jego funkcjonalność i precyzję. Sposób montażu urządzenia (płytka uniwersalna) spowodował niestety, że niektóre komponenty nie osiągają wydajności deklarowanej przez producenta. Jest to ograniczona częstotliwość zegara sterująca komunikacją mikrokontrolera z przetwornikami.

Możliwy jest dalszy rozwój pracy. Obejmuje on między innymi zbudowanie urządzenia na dedykowanej płytce drukowanej, z wykorzystaniem elementów o szerszym paśmie. Możliwa jest również optymalizacja oprogramowania, np. przez przetestowanie działania innych metod synchronizacji niż powiadamianie z przerwań, rozszerzenie funkcjonalności, jak implementację generatorów `DDS` oprócz funkcyjnych, czy klas pozwalających na stworzenie jakiegoś rodzaju wewnętrznego sprzężenia, jak kontrolery `PID` mierzące napięcie wejściowe i generujące napięcie wyjściowe. Kolejną opcją rozwoju oprogramowania jest

rozszerzenie stworzonego języka o instrukcje i pętle warunkowe, co pozwoliłoby między innymi na implementację funkcjonalności wyzwalania znanej z oscyloskopów. Mogłoby to wymagać nowych instrukcji, np. do obsługi pewnego rodzaju jednostki arytmetyczno-logicznej.

Bibliografia

[1] *2N3055(NPN), MJ2955(PNP) / Complementary Silicon Power Transistors*. ON Semiconductor Corp. 2005. URL: https://www.onsemi.com/pdf/datasheet/2n3055-d.pdf.

[2] *ADC DC Specifications*. Microchip Technology, Inc. 14 list. 2023. URL: https://microchipdeveloper.com/xwiki/bin/view/products/data-converters/adc-specs/adc-dc/.

[3] *An overview of HTTP*. MDN Web Docs. 3 mar. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[4] James Bryant i Walt Kester. *Data Converter Architectures*. Analog Devices, Inc. 30 sierp. 2017. URL: http://www.analog.com/media/en/training-seminars/design-handbooks/Data-Conversion-Handbook/Chapter3.pdf.

[5] *Compilers vs. Interpreters*. Ionos SE. 7 list. 2023. URL: https://www.ionos.com/digitalguide/websites/web-development/compilers-vs-interpreters/.

[6] Piyu Dhaker. *Introduction to SPI Interface*. Analog Devices, Inc. 2018. URL: https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf.

[7] *Digital Multimeter Measurement Fundamentals*. NATIONAL INSTRUMENTS CORP. URL: https://www.ni.com/en/shop/electronic-test-instrumentation/digital-multimeters/dmm-measurement-fundamentals.html.

[8] *General Purpose Transistors / NPN Silicon / 2N3903, 2N3904*. ON Semiconductor Corp. 2021. URL: https://www.onsemi.com/pdf/datasheet/2n3903-d.pdf.

[9] Rop Gonggrijp. *I2C Manager for ESP32*. 2022. URL: https://github.com/ropg/i2c_manager.

[10] *How delta-sigma ADCs work*. Texas Instruments Inc. Lip.–wrz. 2011. URL: https://www.ti.com/lit/an/slyt423a/slyt423a.pdf.

[11] *HTTP Server - ESP32*. Espressif Systems Inc. 10 sierp. 2023. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/protocols/esp_http_server.html.

[12] *INA122 Single Supply, MicroPower INSTRUMENTATION AMPLIFIER*. Texas Instruments Inc. 2024. URL: https://www.ti.com/lit/ds/symlink/ina122.pdf.

[13] Walt Jung, Walt Kester i James Bryant. *Data Converter Support Circuits*. Analog Devices, Inc. 30 sierp. 2017. URL: https://www.analog.com/media/en/training-seminars/design-handbooks/Data-conversion-handbook/Chapter7.pdf.

[14] Walt Jung, Walt Kester, James Bryant, Joe Buxton, Wes Freeman, Ethan Bordeax, Johannes Horvath, Catherine Redmond i Eva Murphy. *Hardware Design Techniques*. Analog Devices, Inc. 30 sierp. 2017. URL: https://www.analog.com/media/en/training-seminars/design-handbooks/Data-conversion-handbook/Chapter9.pdf.

[15] Walt Kester, Dan Sheingold i James Bryant. *Fundamentals of Sampled Data Systems*. Analog Devices, Inc. 30 sierp. 2017. URL: https://www.analog.com/media/en/training-seminars/design-handbooks/Data-conversion-handbook/Chapter2.pdf.

[16] *Key Parameters Of ADCs*. Monolithic Power Systems, Inc. 14 list. 2023. URL: https://www.monolithicpower.com/en/analog-to-digital-converters/introduction-to-adcs/key-parameters-of-adcs.

[17] *LM4040 Precision Micropower Shunt Voltage Reference*. Texas Instruments Inc. 2017. URL: https://www.ti.com/lit/ds/symlink/lm4040.pdf.

[18] *LMx24-N, LM2902-N Low-Power, Quad-Operational Amplifiers*. Texas Instruments Inc. 2015. URL: https://www.ti.com/lit/ds/symlink/lm2902-n.pdf.

[19] *LT1014, LT1014A, LT1014D QUAD PRECISION OPERATIONAL AMPLIFIERS*. Texas Instruments Inc. 2009. URL: https://www.ti.com/lit/ds/symlink/lt1014.pdf.

[20] *MCP23008/MCP23S08 8-Bit I/O Expander with Serial Interface*. Microchip Technology Inc. 2019. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/MCP23008-MCP23S08-Data-Sheet-20001919F.pdf.

[21] *MCP3204/3208 2.7V 4-Channel/8-Channel 12-Bit A/D Converters with SPI Serial Interface*. Microchip Technology Inc. 2008. URL: https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/21298e.pdf.

[22] *MCP4902/4912/4922 8/10/12-Bit Dual Voltage Output Digital-to-Analog Converter with SPI Interface*. Microchip Technology Inc. 2010. URL: https://ww1.microchip.com/downloads/en/devicedoc/22250a.pdf.

[23] *MCP6021/1R/2/3/4 Rail-to-Rail Input/Output, 10 MHz Op Amps*. Microchip Technology Inc. 2023. URL: https://ww1.microchip.com/downloads/aemDocuments/documents/MSLD/ProductDocuments/DataSheets/MCP6021-Data-Sheet-DS20001685.pdf.

[24] *MCP606/7/8/9 2.5V to 6.0V Micropower CMOS Op Amp*. Microchip Technology Inc. 2009. URL: https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/11177f.pdf.

[25] Dawid Najda. *ESP32 MCP23008 expander driver/library*. 2024. URL: https://github.com/herhor67/MCP23008.

[26] Dawid Najda. *ESP32 MCP3x0x ADC driver/library*. 2024. URL: https://github.com/herhor67/MCP3X0X.

[27] Dawid Najda. *ESP32 MCP4xxx DAC driver/library*. 2024. URL: https://github.com/herhor67/MCP4XXX.

[28] Robert Nystrom. *Crafting Interpreters*. 2021. URL: https://craftinginterpreters.com/a-map-of-the-territory.html.

[29] Don Peterson i B&K Precision. *B&K Precision Function Generator Guide*. B&K Precision. URL: https://bkpmedia.s3.amazonaws.com/downloads/guides/en-us/function-generator-awg-guide.pdf.

[30] Eric Peña i Mary Grace Legaspi. *UART: A Hardware Communication Protocol*. Analog Devices, Inc. 2020. URL: https://www.analog.com/media/en/analog-dialogue/volume-54/number-4/uart-a-hardware-communication-protocol.pdf.

[31] *SNx4HC125 Quadruple Buffers with 3-State Outputs*. Texas Instruments Inc. 2021. URL: https://www.ti.com/lit/ds/symlink/sn74hc125.pdf.

[32] *Socket programming*. IBM. 7 maj. 2024. URL: https://www.ibm.com/docs/en/i/7.5?topic=communications-socket-programming.

[33] Chris Stephens. *Embedded USB - a brief tutorial*. Computer Solutions Ltd. URL: https://www.computer-solutions.co.uk/info/Embedded_tutorials/usb_tutorial.htm.

[34] *ULN2801A, ULN2802A, ULN2803A, ULN2804A Eight Darlington arrays*. STMicroelectronics N.V. 2018. URL: https://www.st.com/resource/en/datasheet/uln2801a.pdf.

[35] Jonathan Valdez i Jared Becker. *Understanding the I2C Bus*. Texas Instruments Inc. 2015. URL: https://www.ti.com/lit/an/slva704/slva704.pdf.

[36] *WebSocket*. Wikipedia. 16 maj. 2023. URL: https://pl.wikipedia.org/wiki/WebSocket.

[37] *What is HTTP?* Cloudflare, Inc. URL: https://www.cloudflare.com/en-gb/learning/ddos/glossary/hypertext-transfer-protocol-http/.

Dodatki

Dodatek A

Dokumentacja techniczna

A.1 Oprogramowanie

Program korzysta z biblioteki `ropg\i2c_manager`[9], która zastępuje domyślny interfejs `I2C`, daje bezpieczeństwo dla używania wielowątkowego i inne korzyści.
Konfiguracja `SPI` wymaga tylko podania numerów `GPIO` linii `MOSI`, `MISO` i `CLK` oraz flagi odpowiedzialnej za oznaczenie *master*a. Konfiguracja `I2C` również wymaga podania numerów `GPIO`, ale dla linii `SDA` i `SCL`; poza tym częstotliwości zegara oraz włączenia rezystorów *pull-up*.

A.2 Klasy

W przypadku obsługi elementów które mogą się powtarzać, zostały stworzone odpowiednie klasy. Są to np. przetworniki lub ekspandery. Oczywiście nie jest to jedyne zastosowanie klas w projekcie.

A.2.1 Obsługa przetwornika A/C

Do obsługi przetwornika analogowo-cyfrowego została stworzona biblioteka (z połączenia i modyfikacji kilku istniejących otwarto-źródłowych bibliotek) [26]. Klasa odpowiedzialna za to jest szablonem – w celu możliwości obsługi różnych przetworników tej rodziny.
Najpierw zdefiniowano enumeracje, określające jego parametry: liczbę kanałów wejściowych `mcp_adc_channels_t`, liczbę bitów `mcp_adc_bits_t`, oraz typ wyjścia (ze znakiem lub bez) `mcp_adc_signed_t`. Te enumeracje są używane przy tworzeniu szablonów.

```
1 template <mcp_adc_channels_t C, mcp_adc_bits_t B, mcp_adc_signed_t S =
  ↪  MCP_ADC_DATA_UNSIGNED>
2 class MCP3xxx
3 {
```

```
4  private:
5      spi_host_device_t spi_host;
6      gpio_num_t cs_gpio;
7      int clk_hz;
8      spi_device_handle_t spi_hdl;
9
10 public:
11     MCP3xxx(spi_host_device_t, gpio_num_t, int);
12     ~MCP3xxx();
13
14     esp_err_t init();
15     esp_err_t deinit();
16
17     esp_err_t acquire_spi(TickType_t) const;
18     esp_err_t release_spi() const;
19
20     inline esp_err_t send_trx(spi_transaction_t &trx) const;
21     inline esp_err_t recv_trx(TickType_t timeout = portMAX_DELAY) const;
22     inline out_t parse_trx(const spi_transaction_t &trx) const;
23
24     static spi_transaction_t make_trx(uint8_t, mcp_adc_read_mode_t);
25 };
```

Listing A.1: Szablon definicji klasy zarządzającej `ADC`

Klasa zawiera następujące zmienne, podawane w konstruktorze: użyty host `SPI` (`ESP32` posiada dwa), pin `GPIO` odpowiadający za `chip select` – włączenie komunikacji z układem, oraz częstotliwość zegara podczas komunikacji. Funkcja `init()` wykorzystuje podane parametry, tworzy 'urządzenie', zapisuje je do zmiennej `spi_hdl` oraz podłącza do wybranego hosta.
W celu komunikacji z przetwornikiem, została stworzona statyczna metoda `make_trx`. Zwraca ona odpowiednio przygotowany obiekt reprezentujący transakcję `SPI` – jest to specjalna struktura, zawierająca wysyłane polecenie i dane oraz miejsce na odbierane dane. Może ona być użyta wiele razy, bez ponownego tworzenia, co poprawia wydajność. W przypadku użytego przetwornika `A/C` podaje się 5 bitów: 1 bit startowy, 1 bit odpowiedzialny za wybór trybu pomiaru oraz 3 bity wyboru wejścia. Komenda ta zostaje na żądanie programu wysłana do chipa. Następnie, przez 2 cykle zegara odpowiednie wejście jest próbkowane. Finalnie, uruchamia się sekwencyjna praca przetwornika, produkując po kolei 12 bitów przez 12 cykli zegara.
Tak reprezentowaną transakcję należy następnie wykonać za pomocą metody `send_trx`.

Przekazuje ona dane do sterownika SPI, który obsługuje warstwę fizyczną. Aby poczekać i odebrać dane, należy użyć metody recv_trx. Zostały one rozdzielone, ponieważ w ten sposób w międzyczasie można wykonywać inny kod, zamiast spinowania bez celu. Jeśli obie zostaną wykonane bez zwrócenia błędu, można następnie odczytać zmierzoną wartość, zapisaną w obiekcie transakcji, za pomocą metody parse_trx. Zwraca ona odczytany pomiar jako liczbę całkowitą, ze znakiem lub bez (zależnie od typu przetwornika).
Metody acquire_spi oraz release_spi służą do zajęcia całego hosta SPI przez jedno urządzenie – poprawia to wydajność, gdyż sterownik nie musi sprawdzać czy inne urządzenia nie próbują się komunikować na tych samych liniach oraz nie musi przełączać kontekstu.

```
using MCP3002 = MCP3xxx<MCP_ADC_CHANNELS_2, MCP_ADC_BITS_10>;

using MCP3004 = MCP3xxx<MCP_ADC_CHANNELS_4, MCP_ADC_BITS_10>;
using MCP3008 = MCP3xxx<MCP_ADC_CHANNELS_8, MCP_ADC_BITS_10>;

using MCP3202 = MCP3xxx<MCP_ADC_CHANNELS_2, MCP_ADC_BITS_12>;

using MCP3204 = MCP3xxx<MCP_ADC_CHANNELS_4, MCP_ADC_BITS_12>;
using MCP3208 = MCP3xxx<MCP_ADC_CHANNELS_8, MCP_ADC_BITS_12>;

using MCP3302 = MCP3xxx<MCP_ADC_CHANNELS_4, MCP_ADC_BITS_13,
    MCP_ADC_DATA_SIGNED>;
using MCP3304 = MCP3xxx<MCP_ADC_CHANNELS_8, MCP_ADC_BITS_13,
    MCP_ADC_DATA_SIGNED>;
```

Listing A.2: Nazwane specjalizacje szablonu dla różnych układów z rodziny MCP3xxx

A.2.2 Obsługa przetwornika C/A

Do obsługi przetwornika cyfrowo-analogowego została stworzona od zera biblioteka, bazująca na bibliotece dla ADC [27]. Są one bardzo zbliżone, główna różnica to enumeracje specjalizujące szablon oraz kierunek komunikacji.
Najpierw zdefiniowano enumeracje, określające jego parametry: liczbę kanałów wejściowych mcp_dac_channels_t oraz liczbę bitów mcp_dac_bits_t.

```
template <mcp_dac_channels_t C, mcp_dac_bits_t B>
class MCP4xxx;
```

Listing A.3: Szablon definicji klasy zarządzającej `DAC`

Większość metod jest nazwana tak samo i pełni dokładnie te same funkcje. Jedyna różnica wynika z kierunku przepływu danych – nie odczytujemy żadnych informacji zwrotnych, więc metoda `parse_trx` znika. Zastąpiona jest metodą `write_trx`, którą należy wykonać *przed* wysłaniem transakcji. Wpisuje ona dane do wysłania do obiektu transakcji.
Dla wykorzystanego przetwornika `C/A` komenda składa się z 4 bitów: 1 bit wyboru kanału, 1 bit włączający wewnętrzny bufor napięcia odniesienia, 1 bit włączający dodatkowe dwukrotne wzmocnienie, oraz 1 bit pozwalający na wyłączenie kanału (wysoka impedancja, ang. *high-impendance*, `Hi-Z`). Następnie przesyłane jest 12 bitów reprezentujących wartość cyfrową, która po ostatnim cyklu zegara zostaje wpisana równolegle do pamięci przetwornika i przekonwertowana na wartość analogową. Przetwornik ten nie wykorzystuje linii `MISO`.

```
using MCP4801 = MCP4xxx<MCP_DAC_CHANNELS_1, MCP_DAC_BITS_8>;
using MCP4811 = MCP4xxx<MCP_DAC_CHANNELS_1, MCP_DAC_BITS_10>;
using MCP4821 = MCP4xxx<MCP_DAC_CHANNELS_1, MCP_DAC_BITS_12>;

using MCP4802 = MCP4xxx<MCP_DAC_CHANNELS_2, MCP_DAC_BITS_8>;
using MCP4812 = MCP4xxx<MCP_DAC_CHANNELS_2, MCP_DAC_BITS_10>;
using MCP4822 = MCP4xxx<MCP_DAC_CHANNELS_2, MCP_DAC_BITS_12>;

using MCP4901 = MCP4xxx<MCP_DAC_CHANNELS_1, MCP_DAC_BITS_8>;
using MCP4911 = MCP4xxx<MCP_DAC_CHANNELS_1, MCP_DAC_BITS_10>;
using MCP4921 = MCP4xxx<MCP_DAC_CHANNELS_1, MCP_DAC_BITS_12>;

using MCP4902 = MCP4xxx<MCP_DAC_CHANNELS_2, MCP_DAC_BITS_8>;
using MCP4912 = MCP4xxx<MCP_DAC_CHANNELS_2, MCP_DAC_BITS_10>;
using MCP4922 = MCP4xxx<MCP_DAC_CHANNELS_2, MCP_DAC_BITS_12>;
```

Listing A.4: Nazwane specjalizacje szablonu dla różnych układów z rodziny `MCP4xxx`

A.2.3 Obsługa ekspandera GPIO

Do obsługi ekspandera `GPIO` została stworzona biblioteka (po zmodyfikowaniu i zmodernizowaniu innej dostępnej publicznie biblioteki) [25].

```
class MCP23008
{
   i2c_port_t port; // I2C_NUM_0 or I2C_NUM_1
   uint8_t address; // Hardware address of the device

public:
   uint8_t gpio = 0;

private:
   enum class Register : uint8_t
   {
      IODIR = 0x00,
      IPOL = 0x01,
      GPINTEN = 0x02,
      DEFVAL = 0x03,
      INTCON = 0x04,
      IOCON = 0x05,
      GPPU = 0x06,
      INTF = 0x07,
      INTCAP = 0x08,
      GPIO = 0x09,
      OLAT = 0x0A,
   };

public:
   MCP23008(i2c_port_t p, uint8_t a = 0b000);
   ~MCP23008();

   esp_err_t init(bool out = false);
   esp_err_t deinit();

   esp_err_t set_pins(uint8_t val);
   esp_err_t get_pins(uint8_t &val);

   esp_err_t set_pins();
   esp_err_t get_pins();

   //

```

```
40    esp_err_t set_direction(uint8_t val);
41    esp_err_t set_input_polarity(uint8_t val);
42
43    esp_err_t set_interrupt_enabled(uint8_t val);
44    esp_err_t set_interrupt_control(uint8_t val);
45
46    esp_err_t set_default(uint8_t val);
47
48    esp_err_t set_config(bool s, bool d, bool o, bool i);
49    esp_err_t set_config(uint8_t val);
50
51    esp_err_t set_pullup(uint8_t val);
52
53    esp_err_t read_interrupt_flag(uint8_t &val);
54    esp_err_t read_interrupt_captured(uint8_t &val);
55
56    // Single bit manipulation
57    void set_bit(uint8_t b);
58    void clear_bit(uint8_t b);
59    void flip_bit(uint8_t b);
60
61 private:
62    esp_err_t read_reg(Register reg, uint8_t &d);
63    esp_err_t write_reg(Register reg, uint8_t d);
64 };
```

Listing A.5: Definicja klasy zarządzającej ekspanderem `GPIO`

Klasa zawiera następujące zmienne, podawane w konstruktorze: użyty port `I2C` (`ESP32` posiada dwa), oraz 7-bitowy adres urządzenia. Protokół ten nie posiada obiektów reprezentujących urządzenia, każda komenda jest całkowicie niezależna. Funkcja `init()` pozwala więc jedynie na szybkie ustalenie kierunku wszystkich pinów.
Układ posiada kilka rejestrów, których adresy są wypisane w enumeracji. Prywatne metody `read_reg` i `write_reg` służą do wysyłania poleceń po `I2C`, tzn. na odpowiednim interfejsie przesyła kierunek komunikacji, adres urządzenia, adres rejestru i finalnie jego wartość. Sterownik następnie wysyła sygnał zegara i poszczególne bity do odbiornika, lub odczytuje produkowane przez niego dane.
Kilka stworzonych metod pozwala na ustawienie kierunku pinów (wejścia lub wyjścia), odczyt wartości, ustawianie wartości, włączanie przerwań, zmianę polaryzacji i inne, rzadziej użyteczne funkcje układu.

A.2.4 Odczytywanie ustawień

Dane dotyczące ustawień są przesyłane w formacie `JSON`. Aby uniknąć kilkukrotnego kopiowania danych i niepotrzebnego zużycia pamięci (najpierw dane trafiają przez serwer, a więc gniazdo do wewnętrznego bufora, potem z bufora są kopiowane do własnego bufora, skąd są parsowane przez bibliotekę na strukturę obiektu `JSON`) została stworzona specjalna klasa. Wyposażona jest ona w dwa interfejsy, jeden współpracuje z parserem `JSON`, drugi z serwerem/gniazdem. Posiada mały wewnętrzny bufor oraz rozmiar wpisanych danych i obecną pozycję.
Parser `JSON` dopuszcza kilka sposobów podania ciągu znaków. W tym przypadku użyteczny jest ten korzystający z iteratorów i właśnie w takim stylu został stworzony interfejs. Każdy iterator posiada wskaźnik do obiektu do którego należy (istnieje tylko jeden unikalny iterator dla każdego obiektu). Odczytuje on po kolei znaki z bufora rodzica, aż do wyczerpania. Wtedy wywołuje metodę `recv` która pobiera nowe dane z gniazda do bufora. Jeśli nowe dane nie istnieją – proces kończy się. Jeśli gdziekolwiek wystąpi błąd, jest on zapisywany do późniejszego sprawdzenia. Iterator końcowy to specjalny przypadek; taki, który nie posiada rodzica. To do niego porównywane są te zwykłe, wraz ze specjalną implementacją kilku możliwych kombinacji (`operator==`).

```
1  class SocketReader
2  {
3     static constexpr int BUF_SIZE = 1024;
4
5  public:
6     esp_err_t err = ESP_OK;
7
8  private:
9     char buffer[BUF_SIZE];
10    size_t dataLen = 0;
11    size_t ptr = 0;
12    httpd_req_t *req;
13
14    void recv()
15    {
16       int ret = httpd_req_recv(req, buffer, BUF_SIZE);
17
18       if (ret < 0) // error
19       {
20          err = ret;
21          return;
```

```
22        }
23        dataLen = ret;
24        ptr = 0;
25    }
26
27 public:
28    SocketReader(httpd_req_t *r) : req(r)
29    {}
30    ~SocketReader() = default;
31
32    class iterator;
33
34 public:
35    iterator begin()
36    {
37        recv();
38        return iterator(this);
39    }
40
41    iterator end()
42    {
43        return iterator(nullptr);
44    }
45 };
```

Listing A.6: Kod klasy `SocketReader` służącej do odczytu przesyłanych danych fragmentami

```
1  class iterator
2  {
3      friend class SocketReader;
4
5  public:
6      using iterator_category = std::input_iterator_tag;
7      using value_type = char;
8      using difference_type = std::ptrdiff_t;
9      using pointer = char *;
10     using reference = char &;
```

```
11
12 private:
13     SocketReader *parent;
14
15     iterator(SocketReader *p) : parent(p) {}
16
17 public:
18     ~iterator() = default;
19
20 public:
21     char operator*() const
22     {
23         assert(parent->ptr < parent->dataLen);
24         return parent->buffer[parent->ptr];
25     }
26
27     iterator &operator++()
28     {
29         ++parent->ptr;
30         if (parent->ptr >= parent->dataLen)
31             parent->recv();
32         return *this;
33     }
34
35     bool operator==(const iterator &other) const
36     {
37         if (parent == other.parent) // If the same owner, then the same
38             return true;
39
40         if (other.parent == nullptr) // I am begin, other is end
41             return parent->ptr >= parent->dataLen;
42
43         if (parent == nullptr) // I am end (who TF uses this order?)
44             return other.parent->ptr >= other.parent->dataLen;
45
46         return false; // Completely different owners
47     }
48 };
```

Listing A.7: Kod wewnętrznej klasy `iterator` służącej jako interfejs dla parsera

A.2.5 Zadania

W celu implementacji wykonywania zadań musiały zostać stworzone klasy pozwalające na przechowanie odpowiednio zagnieżdżonej struktury zadania w pamięci, klasy przechowujące instrukcje, kody operacji, argumenty i inne dane.
Z głównej klasy przechowującej całe zadanie musi dać się w prosty sposób pobierać kolejne instrukcje, zgodnie z przepływem programu. Oznacza to, że każda klasa która jakoś przechowuje instrukcje, musi implementować taką metodę. Do tego klasy te muszą mieć jakiś wewnętrzny stan, wskaźnik, indeks, itp. aby pamiętać obecną pozycję czy ilość wykonań dla pętli. Z tego samego powodu, klasy te muszą być wyposażone w funkcję pozwalającą na rekursywne zresetowanie całego programu w celu umożliwienia wielokrotnego wykonywania.
Najpierw została stworzona enumeracja, zawierająca dostępne operacje:

```
enum class OPCode : uint8_t
{
    NOP = 0,
    AIRDF, AIRDM, AIRDU,
    AIEN, AIDIS, AIRNG,
    AOVAL, AOGEN,
    DIRD,
    DOWR, DOSET, DORST, DOAND, DOXOR,
    DELAY, GETTM,
};
```

Listing A.8: Enumeracja `OPCode` istniejących operacji

Potem stworzono strukturę która reprezentuje pojedynczą instrukcję, tzn. zawiera wykonywaną operację, oraz opcjonalnie port której ta operacja dotyczy oraz dowolny argument w postaci liczby całkowitej lub zmiennoprzecinkowej. Zmienne zostały ułożone tak, aby rozmiar klasy był możliwie mały – uwzględniając tzw. *padding* jest to 8 bajtów.

```
struct Instruction
{
    OPCode opc = OPCode::NOP;
    uint8_t port = 0;
```

```
5       union
6       {
7           uint32_t u = 0;
8           float f;
9       } arg;
10  };
11
12  using InstrPtr = const Instruction *;
13  constexpr InstrPtr nullinstr = nullptr;
```

Listing A.9: Klasa `Instruction` reprezentująca instrukcję

Następnie zostały utworzone klasy i typy, wymagane do implementacji zagnieżdżonych list, używanych do przechowywania pętli i instrukcji. Wstępna deklaracja klasy `Loop`, jej inteligentny wskaźnik, oraz klasa `Statement` (wyrażenie) bazująca na szablonie `std::variant` która może być albo instrukcją albo pętlą.

```
1  class Loop;
2  using LoopPtr = std::unique_ptr<Loop>;
3  using Statement = std::variant<std::monostate, Instruction, LoopPtr>;
```

Listing A.10: Deklaracje klas i typów

Mając to, można było przystąpić do stworzenia klasy `Scope` (zasięg widoczności) która przechowuje listę wyrażeń oraz posiada indeks do następnego zwracanego wyrażenia. Posiada ona metodę służącą do pobrania kolejnych instrukcji, metodę sprawdzającą czy jakieś instrukcje jeszcze pozostają, metodę do rekursywnego resetu, metodę do restartu (powrót do początkowej instrukcji) oraz dwie metody do tworzenia – jedna dodaje pętlę, druga dodaje instrukcję.

```
1  class Scope
2  {
3      std::vector<Statement> statements;
4      mutable size_t index = 0;
5
6  public:
7      DEFAULT_CTOR(Scope);
8      DEFAULT_MV_CTOR(Scope);
9
```

```
10      InstrPtr getInstr() const;
11      bool finished() const;
12      void reset() const;
13      void restart() const;
14
15      Instruction& appendInstr(const Instruction& = Instruction());
16      Loop& appendLoop(size_t);
17  };
```

Listing A.11: Klasa `Scope` reprezentująca listę operacji (zasięg widoczności)

Klasa ta jest następnie użyta do stworzenia klasy `Loop` (pętla), która przechowuje własny `Scope` i żądaną ilość powtórzeń oraz posiada licznik iteracji. Ma ona podobny interfejs co `Scope`: metodę służącą do pobrania kolejnych instrukcji, metodę sprawdzającą czy jakieś instrukcje jeszcze pozostają, metodę do rekursywnego resetu, metodę do restartu (wyzerowanie iteracji) oraz metodę pomocniczą – prosty getter – aby było możliwe wpisywanie do niej instrukcji bez *zbytniego* duplikowania interfejsu poprzedniej klasy.

```
1  class Loop
2  {
3      Scope scope;
4      size_t max_iter = 0;
5      mutable size_t iter = 0;
6
7  public:
8      Loop(size_t = 0);
9      ~Loop();
10
11     InstrPtr getInstr() const;
12     bool finished() const;
13
14     void reset() const;
15     void restart() const;
16
17     Scope &getScope();
18 };
```

Listing A.12: Definicja klasy `Loop` przechowującej powtarzany `Scope`

Finalnie powstała główna klasa `Program` reprezentująca całe zadanie. Przechowuje ona główny `Scope` oraz implementuje metodę która parsuje tekst na drzewo i ewentualnie zwraca błędy.

```
class Program
{
    Scope scope;

public:
    DEFAULT_CTOR(Program);
    DEFAULT_MV_CTOR(Program);

    bool parse(const std::string &, std::vector<std::string> &);

    InstrPtr getInstr() const;
    void reset() const;

    size_t size() const;
    bool isValid() const;
};
```

Listing A.13: Definicja klasy `Program` przetrzymującej zadanie

Metody służące do pobierania instrukcji są proste, `Scope` pobiera obecne wyrażenie, sprawdza czy jest instrukcją, jeśli tak to zwraca ją. Jeśli nie, czyli jest pętlą, wywołuje metodę klasy `Loop`. Klasa ta sprawdza ilość iteracji i wywołuje metodę własnego `Scope`. Taka rekursja będzie się powtarzać aż do dotarcia do zwykłej instrukcji.
Najciekawszą metodą jest metoda `parse` klasy `Program`, która jest opisana poniżej.

A.2.6 Parser

Przygotowania do implementacji

Najpierw został stworzony typ, który przechowuje pewną funkcję. Pozwala to oddzielić parsowanie poszczególnych instrukcji od samego kodu parsera. Funkcja ta przyjmuje listę słów i ma zamienić je na argumenty w podanym obiekcie klasy `Instruction`. Zwraca wartość prawda/fałsz na podstawie tego, czy udało się poprawnie odczytać wszystkie argumenty.
Oprócz tego powstała struktura `InstrLUTRow`, który przechowuje dane potrzebne do parsowania instrukcji oraz informacje o niej. Są to: kod operacji, tekstowa reprezentacja

operacji, tekstowa lista wymaganych argumentów, opis działania instrukcji, wymagana ilość argumentów, oraz *callback* do funkcji parsującej argumenty.
Finalnie, łącząc wszystko w całość, stworzona została tablica, tzw. *lookup table* która zawiera tak sparametryzowane wszystkie dostępne instrukcje. Część instrukcji posiada te same argumenty, więc w celu duplikowania ich opisów oraz funkcji parsujących, zostały wykorzystane makra preprocesora, które mogą być wstawione samodzielnie lub w połączeniu z innymi fragmentami tekstu czy funkcjami.

```
1  using ParseCb = std::function<bool(const std::vector<std::string>&,
   ↪  Instruction&)>; // in args, inout instr
2
3  struct InstrLUTRow
4  {
5      Interpreter::OPCode opc;
6      const char *namestr;
7      const char *argstr;
8      const char *descstr;
9      size_t argcnt;
10     ParseCb parser;
11 };
12
13 #define CB_HELP(COND) [](const std::vector<std::string>& args,
   ↪  Instruction& cs) { return (COND); }
14 #define READ_IP (try_parse_integer(args[0], cs.port) && (cs.port >= 1 &&
   ↪  cs.port <= 4))
15 #define READ_OP (try_parse_integer(args[0], cs.port) && (cs.port >= 1 &&
   ↪  cs.port <= 2))
16 #define READ_DO (try_parse_integer(args[0], cs.arg.u, 0) && (cs.arg.u <=
   ↪  0b1111))
17
18 const std::vector<InstrLUTRow> CS_LUT = {
19     [...]
20 };
```

Listing A.14: Przygotowanie tablicy zawierającej spis instrukcji i informacji o nich

Właściwy parser

Metoda `Program::parse` przyjmuje dwa argumenty, jeden to ciąg znaków stanowiący program do odczytania, zaś drugi pozwala na zwrócenie ewentualnych napotkanych błędów. Najpierw stworzony jest stos, przechowujący obecny`Scope` do którego dopisywane są wyrażenia. W pętli odczytywane są kolejne linie (między znakami `;`), wyrażenie to jest kopiowane i dzielone na słowa, przy okazji czyszcząc białe znaki dookoła. Jeśli nie ma żadnych słów - linia była pusta, można pominąć. Pierwsze słowo jest przenoszone do osobnej zmiennej, reszta listy to argumenty.
Następnie rozpoczyna się faktyczne parsowanie wyrażenia. Najpierw sprawdzane są dwa specjalne przypadki, które wymagają innego traktowania (nie są instrukcjami). Jeśli operacja to `LOOP`, odczytywana jest pożądana ilość iteracji, nowa pętla zostaje dodana do obecnego poziomu, i do stosu `Scope`ów zostaje dodany nowo utworzony, do którego będą pisane dalsze wyrażenia. Jeśli operacja to `END`, obecny `Scope` zostaje zamknięty i usunięty ze stosu, tak więc kolejne instrukcje będą dopisywane do poprzedniego, niższego poziomu. W przeciwnym przypadku, można założyć że operacja to instrukcja do sprzętu (albo jakiś nieistniejące polecenie). W tym momencie wykorzystywana jest wcześniej przygotowana tablica opisująca instrukcje. Sprawdzana jest nazwa operacji, jeśli się zgadza to zapisywany jest odpowiedni `OPCode`, sprawdzana jest wymagana liczba argumentów, i finalnie uruchamiany jest parser argumentów. Jeśli w którymkolwiek momencie wystąpił błąd, jest on zgłaszany do zwracanej listy. Tak samo w przypadku, jeśli cała tablica zostanie sprawdzona i operacja nie została znaleziona. Po odczytaniu całego programu, następuje sprawdzenie czy wszystkie `Scope` zostały zamknięte, jeśli nie to również zwracany jest błąd. Jeśli nie został zwrócony żaden błąd, program jest poprawny i może zostać wykonany.

```
1  #define PARSE_ERR(rsn) do {                                          \
2     prgValid = false;                                                 \
3     err.push_back("Stmt #"s + std::to_string(line) + ": " + rsn);  \
4  } while (0)
5
6  static const char *expstx = "expected syntax: ";
7
8  #define PARSE_ERR_SNTX(syntax) PARSE_ERR(expstx + syntax)
9
10 bool Program::parse(const std::string& str, std::vector<std::string>&
   ↪  err)
11 {
12     prgValid = true;
13
```

```
14      std::stack<Scope *, std::vector<Scope *>> scopes;
15      scopes.push(&scope); // main
16
17      size_t beg = 0;
18      size_t end = 0;
19      size_t line = 0;
20
21      while (beg < str.length())
22      {
23          ++line;
24
25          end = str.find(';', beg);
26          if (end == std::string::npos)
27              end = str.length();
28
29          if (end - beg > 32) // sanity check
30          {
31              beg = end + 1;
32              PARSE_ERR("malformed (too long, max 32)!");
33              continue;
34          }
35
36          std::string stmtstr = str.substr(beg, end - beg);
37          beg = end + 1;
38
39          std::vector<std::string> args = str_split_on_whitespace(stmtstr);
40
41          if (args.empty()) // no command
42              continue;
43
44          std::string cmd = std::move(args[0]);
45          args.erase(args.begin());
46
47          if (cmd == "LOOP")
48          {
49              size_t iters = 0;
50
51              if (args.size() != 1 || !try_parse_integer(args[0], iters))
52                  PARSE_ERR_SNTX("LOOP <iterations (uint32)>");
53
```

```
54              Loop &l = scopes.top()->appendLoop(iters);
55              scopes.push(&l.getScope());
56          }
57          else if (cmd == "END")
58          {
59              if (args.size() != 0)
60                  PARSE_ERR_SNTX("END <no args>");
61
62              if (scopes.size() == 1)
63                  PARSE_ERR("END: no Scope to end!");
64
65              scopes.pop();
66          }
67          else // regular command
68          {
69              Instruction cs;
70
71              for (const auto &lut : CS_LUT)
72              {
73                  if (cmd == lut.namestr)
74                  {
75                      cs.opc = lut.opc;
76
77                      if ((args.size() != lut.argcnt) || (lut.parser &&
                        ↪  !lut.parser(args, cs)))
78                          PARSE_ERR_SNTX(lut.namestr + ' ' + lut.argstr);
79
80                      break;
81                  }
82              }
83
84              if (cs.opc == OPCode::NOP)
85                  PARSE_ERR("unknown command!");
86
87              scopes.top()->appendInstr(cs);
88          }
89      }
90
91      scopes.pop(); // main
92
```

```
93      line = -1;
94      for (size_t i = 0; i < scopes.size(); ++i)
95          PARSE_ERR("Scope has not been terminated (missing END)!");
96
97      return prgValid;
98  }
```

Listing A.15: Kod metody `Program::parse` odpowiedzialnej za konwersję z tekstu na drzewo poleceń

A.2.7 Generatory

Klasa `Generator`

W celu stworzenia generatorów DDS, stworzona została specjalna klasa `Generator`. Przechowuje ona listę składowych sygnałów, wraz z ich amplitudami. Posiada ona metodę służącą do dodawania sygnałów oraz dwie metody do pobierania danych: jedna zwiększa obecny numer próbki o jeden i generuje sygnał, druga zaś pozwala na podanie arbitralnego punktu czasowego.

```
1  class Generator
2  {
3  public:
4     using index_t = Signal::index_t;
5
6  private:
7     using Signals = std::vector<std::pair<float, SignalHdl>>;
8
9     Signals signals;
10    index_t current_step;
11
12 public:
13    DEFAULT_CTOR(Generator);
14    DEFAULT_MV_CTOR(Generator);
15
16    void add(float a, SignalHdl &&s)
17    {
18       signals.emplace_back(a, std::move(s));
19    }
```

```
20    float get(index_t i)
21    {
22       current_step = i;
23       return calculate();
24    }
25    float forward()
26    {
27       ++current_step;
28       return calculate();
29    }
30
31 private:
32    inline float calculate() const
33    {
34       float sum = 0;
35       for (const auto &s : signals)
36          sum += s.first * s.second->get(current_step);
37       return sum;
38    }
39 };
```

Listing A.16: Kod klasy `Generator` używanej do produkcji przebiegów

Klasa `Signal` i pomocnicze

Następnie została stworzona wirtualna klasa `Signal`, której klasy dziedziczące generują kilka z najpopularniejszych przebiegów. Implementuje ona interfejs do generowania, pomocniczą metodę w celu konwersji z i do `JSON` oraz własną wersję sinusa (zamiana kąta na wartość), z minimalnie pogorszoną dokładnością, ale dużo szybszą niż domyślna implementacja języka.

```
1 class Signal
2 {
3    friend SignalHdl;
4
5 protected:
6    static inline float ang_to_sine(float ang) // Full sine cycle on <0,
     ↪  1>, duplicated to negatives for simplicity
7    {
```

```
8        bool flip = std::signbit(ang);
9        ang = std::abs(ang);
10       if (ang > 0.5f)
11       {
12          ang -= 0.5f;
13          flip = !flip;
14       }
15
16       float smpl = 16.0f * ang * (0.5f - ang);
17       return flip ? -smpl : smpl;
18       float bhskr = 4.0f * smpl / (5.0f - smpl);
19       return flip ? -bhskr : bhskr;
20    }
21
22 public:
23    using index_t = int32_t;
24
25    Signal() = default;
26    virtual ~Signal() = default;
27
28    virtual float get(index_t) const
29    {
30       return 0;
31    }
32    virtual SignalType type() const
33    {
34       return SignalType::Virtual;
35    }
36 };
```

Listing A.17: Kod wirtualnej klasy `Signal` służącej do generowania podstawowych kształtów fal

Do tego powstała klasa pomocnicza do przechowywania sygnałów. Nie wystarczył zwykły inteligentny wskaźnik, gdyż do serializacji wymagane było też stworzenie funkcji fabryki, która tworzy odpowiednią klasę na podstawie podanego typu sygnału, natomiast jest ona po prostu jego wrapperem.

```
class SignalHdl
{
   using SignalPtr = std::unique_ptr<Signal>;
   SignalPtr ptr;

public:
   DEFAULT_CTOR(SignalHdl);
   DELETE_CP_CTOR(SignalHdl);
   DEFAULT_MV_CTOR(SignalHdl);

   SignalHdl(SignalPtr &&p) : ptr(std::move(p)) {}

   Signal &operator*() const
   {
      return *ptr.get();
   }
   Signal *operator->() const
   {
      return ptr.get();
   }
};

template <typename T>
SignalHdl make_signal(const json &j)
{
   std::unique_ptr<Signal> sp = std::make_unique<T>(j.get<T>());
   return SignalHdl(std::move(sp));
}

template <typename T, typename... Args>
SignalHdl make_signal(Args &&...args)
{
   std::unique_ptr<Signal> sp =
    ↪  std::make_unique<T>(std::forward<Args>(args)...);
   return SignalHdl(std::move(sp));
}
```

Listing A.18: Klasa `SignalHdl` do przechowywania klasy `Signal` oraz fabryki tejże

Klasy dziedziczące z `Signal`

Poniżej przedstawione są kody klas odpowiedzialnych za generowanie poszczególnych kształtów oraz manipulację nimi.

```
class SignalConst : public Signal
{
public:
   SignalConst() = default;
   ~SignalConst() = default;

   float get(index_t) const override
   {
      return 1;
   }
   SignalType type() const override
   {
      return SignalType::Const;
   }
};
```

Listing A.19: Klasa `SignalConst` generująca przebieg o stałej wartości

```
class SignalImpulse : public Signal
{
public:
   SignalImpulse() {}
   ~SignalImpulse() = default;

   float get(index_t i) const override
   {
      return i == 0;
   }
   SignalType type() const override
   {
      return SignalType::Impulse;
   }
};
```

Listing A.20: Klasa `SignalImpulse` generująca impuls w czasie 0

```
class SignalSine : public Signal
{
    index_t T;

public:
    SignalSine(index_t t = 1) : T(t) {}
    ~SignalSine() = default;

    float get(index_t i) const override
    {
        i %= T;
        float ang = static_cast<float>(i) / T;
        return ang_to_sine(ang);
    }
    SignalType type() const override
    {
        return SignalType::Sine;
    }
};
```

Listing A.21: Klasa `SignalSine` generująca sinusoidę o podanym okresie

```
class SignalSquare : public Signal
{
    index_t T;
    index_t D;

public:
    SignalSquare(index_t t = 1, index_t d = 0) : T(t), D(d) {}
    ~SignalSquare() = default;

    float get(index_t i) const override
    {
        i = i % T;
```

```
13          return (i < D) ? 1 : 0;
14      }
15      SignalType type() const override
16      {
17          return SignalType::Square;
18      }
19  };
```

Listing A.22: Klasa `SignalSquare` generująca prostokąt (typu `PWM`) o podanym okresie i wypełnieniu

```
1  class SignalTriangle : public Signal
2  {
3      index_t T;
4      index_t P;
5
6  public:
7      SignalTriangle(index_t t = 1, index_t p = 1) : T(t), P(p) {}
8      ~SignalTriangle() = default;
9
10     float get(index_t i) const override
11     {
12         i = i % T;
13         if (i == 0) // to avoid division when P=0
14             return 0.0f;
15         if (i <= P) // rising pos edge
16             return static_cast<float>(i) / P;
17         if (i >= T - P) // rising neg edge
18             return static_cast<float>(i - T) / P;
19         // falling edge
20         return static_cast<float>(T / 2 - i) / (T / 2 - P);
21     }
22     SignalType type() const override
23     {
24         return SignalType::Triangle;
25     }
26 };
```

Listing A.23: Klasa `SignalTriangle` generująca trójkąt, piłę i pośrednie o podanym okresie i czasie szczytu

```
class SignalChirp : public Signal
{
    index_t T;
    float FS;
    float FD;

public:
    SignalChirp(index_t t = 1, float fs = 0, float fd = 0) : T(t), FS(fs),
    ↪  FD(fd) {}
    ~SignalChirp() = default;

    float get(index_t i) const override
    {
        i %= T;
        float fr = static_cast<float>(i) / T;
        float phs = i * (FS + FD * fr); // 0.5f *
        float ang = std::fmod(phs, 1.0f);
        return ang_to_sine(ang);
    }
    SignalType type() const override
    {
        return SignalType::Chirp;
    }
};
```

Listing A.24: Klasa `SignalChirp` generująca świergot o podanym czasie, początkowej częstotliwości i zmianie

```
class SignalChirpLog : public Signal
{
    index_t T;
    float FS;
```

```
    float FR;

public:
    SignalChirpLog(index_t t = 1, float fs = 0, float fr = 0) : T(t),
    ↪  FS(fs), FR(fr) {}
    ~SignalChirpLog() = default;

    float get(index_t i) const override
    {
        i %= T;
        float fr = static_cast<float>(i) / T;

        float phs = FS * T / std::log(FR) * (std::pow(FR, fr) - 1.0f);

        float ang = std::fmod(phs, 1.0f);
        return ang_to_sine(ang);
    }
    SignalType type() const override
    {
        return SignalType::ChirpLog;
    }
};
```

Listing A.25: Klasa `SignalChirpLog` generująca świergot zmieniający się wykładniczo w czasie (liniowo w skali logarytmicznej) o podanym czasie, początkowej częstotliwości i mnożniku zmiany

```
class SignalRandom : public Signal
{
public:
    SignalRandom() = default;
    ~SignalRandom() = default;

    float get(index_t) const override
    {
        constexpr float mult = -1.0 / std::numeric_limits<int32_t>::min();
        int32_t val = std::bit_cast<int32_t>(esp_random());
        return static_cast<float>(val) * mult;
```

```
12    }
13    SignalType type() const override
14    {
15       return SignalType::Random;
16    }
17
18    friend void to_json(json &j, const SignalRandom &o) {}
19    friend void from_json(const json &j, SignalRandom &o) {}
20 };
```

Listing A.26: Klasa `SignalRandom` generująca wartości losowe w przedziale $[-1, 1)$

```
1  class SignalDelay : public Signal
2  {
3     index_t D;
4     SignalHdl S;
5
6  public:
7     SignalDelay(index_t d = 0, SignalHdl &&s = SignalHdl()) : D(d),
       ↪  S(std::move(s)) {}
8     ~SignalDelay() = default;
9
10    DEFAULT_MV_CTOR(SignalDelay);
11
12    float get(index_t i) const override
13    {
14       if (i < D)
15          return 0;
16       return S->get(i - D);
17    }
18    SignalType type() const override
19    {
20       return SignalType::Delay;
21    }
22 };
```

Listing A.27: Klasa `SignalDelay` generująca opóźnienie innego sygnału

```
class SignalAbsolute : public Signal
{
   SignalHdl S;

public:
   SignalAbsolute(SignalHdl &&s = SignalHdl()) : S(std::move(s)) {}
   ~SignalAbsolute() = default;

   DEFAULT_MV_CTOR(SignalAbsolute);

   float get(index_t i) const override
   {
      return std::abs(S->get(i));
   }
   SignalType type() const override
   {
      return SignalType::Absolute;
   }
};
```

Listing A.28: Klasa `SignalAbsolute` generująca wartość bezwzględną innego sygnału

```
class SignalClamp : public Signal
{
   float L;
   float H;
   SignalHdl S;

public:
   SignalClamp(float l = -1, float h = 1, SignalHdl &&s = SignalHdl()) :
    ↪  L(l), H(h), S(std::move(s)) {}
   ~SignalClamp() = default;

   DEFAULT_MV_CTOR(SignalClamp);

   float get(index_t i) const override
   {
      return std::clamp(S->get(i), L, H);
```

```
16    }
17    SignalType type() const override
18    {
19       return SignalType::Clamp;
20    }
21 };
```

Listing A.29: Klasa `SignalClamp` generująca inny sygnał ograniczony minimalną i maksymalną wartością

```
1  class SignalLinearMap : public Signal
2  {
3     float A;
4     float B;
5     SignalHdl S;
6
7  public:
8     SignalLinearMap(float a = 1, float b = 0, SignalHdl &&s = SignalHdl())
      ↪  : A(a), B(b), S(std::move(s)) {}
9     ~SignalLinearMap() = default;
10
11    DEFAULT_MV_CTOR(SignalLinearMap);
12
13    float get(index_t i) const override
14    {
15       return A * S->get(i) + B;
16    }
17    SignalType type() const override
18    {
19       return SignalType::LinearMap;
20    }
21 };
```

Listing A.30: Klasa `SignalLinearMap` generująca inny sygnał przetworzony przez funkcję liniową $A \cdot x + B$

```
class SignalMultiply : public Signal
{
   SignalHdl S1;
   SignalHdl S2;

public:
   SignalMultiply(SignalHdl &&s1 = SignalHdl(), SignalHdl &&s2 =
    ↪  SignalHdl()) : S1(std::move(s1)), S2(std::move(s2)) {}
   ~SignalMultiply() = default;

   DEFAULT_MV_CTOR(SignalMultiply);

   float get(index_t i) const override
   {
      return S1->get(i) * S2->get(i);
   }
   SignalType type() const override
   {
      return SignalType::Multiply;
   }
};
```

Listing A.31: Klasa `SignalMultiply` generująca iloczyn wartości dwóch innych sygnałów

A.3 Przestrzenie nazw

W przypadku obsługi całego urządzenia, nie została tworzona klasa, lecz odpowiednia przestrzeń nazw (ang. *namespace*). Ze względu na fakt, że istnieje tylko jedno "całe urządzenie", oraz "zmienne" wymagane do konstrukcji takiego obiektu tak naprawdę są stałymi, pozwala to kompilatorowi na dodatkową optymalizację kodu.

A.3.1 Komunikacja między wątkami

Przestrzeń nazw `Communicator` odpowiedzialna jest za zarządzanie komunikacją między dwoma rdzeniami/wątkami – wątek serwera `HTTP` oraz wątek obsługujący sprzęt.
Interferjs służący do takiej komunikacji jest przedstawiony poniżej:

```
namespace Communicator
{
    constexpr size_t buf_len = 128 * 1024;

    esp_err_t cleanup();
    esp_err_t init();
    esp_err_t deinit();

    esp_err_t time_settings(size_t);

    bool write_4bytes(const int64_t &, const uint32_t &);

    template <typename T>
    bool write_data(const int64_t &, const T &);

    etl::span<char> get_read();
    void commit_read();

    bool is_running();
    bool has_data();

    void start_running();
    void ask_to_exit();

    bool should_exit();
    void confirm_exit();
};
```

Listing A.32: Interfejs przestrzeni `Communicator` używanej do komunikacji miedzy wątkami

Zaś jej implementacja posiada dodatkowo następujące zmienne:

```
1 etl::bip_buffer_spsc_atomic<char, buf_len> bipbuf;
2 etl::span<char> current_read;
3
4 std::atomic_bool please_exit;
5 std::atomic_bool producer_running;
6
7 size_t time_bytes = 0;
8 size_t res_wrt_4b = 0;
```

Listing A.33: Zmienne "prywatne" przestrzeni `Communicator`

Funkcja `time_settings` pozwala na ustawienie liczby bajtów pomiarów czasu. Kilka funkcji/szablonów jest odpowiedzialnych za wpisywanie danych do bufora. Dwie funkcje służą jako interfejs do odczytu danych z bufora. Ostatnia grupa funkcji pozwala na zarządzanie stanem wątku interpretera. Funkcje `is_running` i `has_data` sprawdzają status wątku i bufora; wraz ze startowaniem i zatrzymywaniem stanowią interfejs dla wątku serwera. Funkcje `should_exit` oraz `confirm_exit` to interfejs wątku interpretera do zarządzania swoją pracą.
W celu przechowywania danych została wykorzystana klasa `bip_buffer` biblioteki `ETL` w wersji `spsc` – *single producer, single consumer* co pozwala na pozbycie się mechanizmów zabezpieczających dostęp typu semafory czy mutex i oparcie na niepodzielności operacji zapisu oraz odczytu – *atomic*. Przyspiesza to znacznie działanie oraz pozwala na dosłownie jednoczesną pracę obu wątków. Nazwa `bip_buffer` jest skrótem od *bipartite buffer* – bufor dwuczęściowy. Oznacza to, że w zarezerwowanej pamięci znajdują się jedna lub dwie logiczne części. Kiedy pierwsza część dotrze do końca pamięci (lub pozostały fragment jest niewystarczający), druga część zostaje rozpoczęta na początku pamięci. Kiedy cała pierwsza część zostanie skonsumowana, druga część staje się pierwszą i proces powtarza się. Taka reprezentacja pozwala na zapis i odczyt fragmentów o dowolnej długości, nawet jeśli nie zapełniają równo całej pamięci do końca. Odczyt większego kawałka pamięci jest wymagany do przesyłu danych ze względu na optymalizację (otoczka wiadomości zajmuje miejsce i czas), zaś dostęp do wewnętrznej pamięci w celu uniknięcia kopiowania ich przed wysłaniem.

Funkcje piszące rezerwują pożądaną ilość miejsca, wstawiają tam dane i potwierdzają zapis. Funkcje odczytujące pobierają możliwą ilość danych do odczytania, odczytują je (przesyłają) i potwierdzają odczyt, co kasuje dane z bufora.

A.3.2 Serwer HTTP

WebServer (lub HTTP Server) to gotowy komponent w środowisku ESP-IDF pozwalający na szybkie, proste i wygodne uruchomienie serwera obsługującego protokół HTTP na mikrokontrolerze. Automatycznie zajmuje się on kodowaniem i dekodowaniem danych w obie strony.

Funkcje pomocnicze

Została utworzona pomocnicza funkcja konwertująca ciąg znaków tzw. *query* przekazywane w URL na słownik/mapę w postaci klucz →wartość. Odczytuje ona *query* z wewnętrznej pamięci dotyczącej danego zapytania, dzieli ją na pary, dzieli je na klucz i wartość (jeśli możliwe) i wpisuje do słownika.

```
1  auto parse_query(httpd_req_t *r)
2  {
3     std::map<std::string, std::string> ret;
4     size_t qr_len = httpd_req_get_url_query_len(r) + 1;
5     std::string query(qr_len, '\0');
6     httpd_req_get_url_query_str(r, query.data(), qr_len);
7     query.erase(query.find('\0'));
8     size_t pos = 0;
9     do
10    {
11       size_t maxlen = query.find('&', pos); // find end of current param
12       if (maxlen == std::string::npos)
13          maxlen = query.length();
14       size_t middle = query.find('=', pos); // find break of current
         ↪  param
15       if (middle > maxlen)              // no value, key only
16       {
17          std::string key = query.substr(pos, maxlen - pos);
18          ret.emplace(std::move(key), "");
19       }
20       else // key and value
21       {
22          std::string key = query.substr(pos, middle - pos);
```

```
23          std::string val = query.substr(middle + 1, maxlen - middle - 1);
24          ret.emplace(std::move(key), std::move(val));
25       }
26       pos = maxlen + 1;
27    } //
28    while (pos <= query.length());
29    // ret.erase("");
30    return ret;
31 }
```

Listing A.34: Funkcja `parse_query` tworząca słownik parametrów

Konfiguracja i start serwera

Najpierw przedstawione jest tworzenie i konfiguracja serwera. W zmiennych przechowywane są wszystkie dostępne adresy, metoda dostępu do nich oraz funkcja odpowiedzialna za przetworzenie zapytania. Funkcja uruchamiająca startuje serwer i przypina wszystkie te adresy do wykonania przez niego.

```
1  static constexpr httpd_uri_t favicon_uri = {
2     .uri = "/favicon.ico",
3     .method = HTTP_GET,
4     .handler = favicon_handler,
5     .user_ctx = nullptr,
6  };
7
8  static constexpr httpd_uri_t welcome_uri = {
9     .uri = "/",
10    .method = HTTP_GET,
11    .handler = welcome_handler,
12    .user_ctx = nullptr,
13 };
14
15 static constexpr httpd_uri_t io_uri = {
16    .uri = "/io",
17    .method = HTTP_GET,
18    .handler = io_handler,
19    .user_ctx = nullptr,
20 };
```

```
21
22 static constexpr httpd_uri_t settings_uri = {
23     .uri = "/settings",
24     .method = HTTP_POST,
25     .handler = settings_handler,
26     .user_ctx = nullptr,
27 };
28
29 static httpd_handle_t server = nullptr;
30
31 esp_err_t start_webserver()
32 {
33     httpd_config_t config = HTTPD_DEFAULT_CONFIG();
34     [...]
35
36     // Start the httpd server
37     ESP_LOGI(TAG, "Starting HTTP server on port: %d", config.server_port);
38
39     ESP_RETURN_ON_ERROR(
40         httpd_start(&server, &config),
41         TAG, "Failed to httpd_start!");
42
43     ESP_LOGI(TAG, "Registering URI handlers...");
44
45     ESP_RETURN_ON_ERROR(
46         httpd_register_uri_handler(server, &welcome_uri),
47         TAG, "Failed to httpd_register_uri_handler!");
48
49     ESP_RETURN_ON_ERROR(
50         httpd_register_uri_handler(server, &io_uri),
51         TAG, "Failed to httpd_register_uri_handler!");
52
53     ESP_RETURN_ON_ERROR(
54         httpd_register_uri_handler(server, &settings_uri),
55         TAG, "Failed to httpd_register_uri_handler!");
56
57     ESP_RETURN_ON_ERROR(
58         httpd_register_uri_handler(server, &favicon_uri),
59         TAG, "Failed to httpd_register_uri_handler!");
60
```

```
61     return ESP_OK;
62 }
```

Listing A.35: Funkcja `start_webserver` ustawiająca i uruchamiająca serwer oraz zmienne pomocnicze

Przetwarzanie zapytań

Poniżej przedstawione są funkcje przetwarzające zapytania.
Najprostszą jest funkcja zwracająca ikonę – zawiera obrazek zakodowany jako bajty od razu w pamięci i tylko wysyła je.

```
1  static esp_err_t favicon_handler(httpd_req_t *req)
2  {
3      httpd_resp_set_status(req, HTTPD_200);
4      httpd_resp_set_type(req, "image/x-icon");
5      httpd_resp_set_hdr(req, "Cache-Control", "max-age=31536000,
       ↪  immutable");
6
7      static constexpr char favicon[661] ={ [...] };
8
9      return httpd_resp_send(req, favicon, 661);
10 }
```

Listing A.36: Funkcja `favicon_handler` wysyłająca plik `favicon.ico`

Później jest funkcja która wysyła wiadomość powitalną – tworzy obiekt `JSON`, wpisuje do niego wiadomość, dane, opis instrukcji, przykładowy generator, istniejące sygnały, itd.

```
1  static esp_err_t welcome_handler(httpd_req_t *req)
2  {
3      httpd_resp_set_status(req, HTTPD_200);
4      httpd_resp_set_type(req, "application/json");
5
6      ordered_json doc = create_ok_response();
7
8      doc["message"] = "Welcome!\n"
9                       "Last compilation time: ${data.cmpl.date}
                        ↪  ${data.cmpl.time}.\n"
```

```
10                 [...];
11
12     doc["data"]["cmpl"]["date"] = __DATE__;
13     doc["data"]["cmpl"]["time"] = __TIME__;
14     [...]
15
16     doc["data"]["prg"]["cmds"] = ordered_json::array();
17     for (const auto &lut : CS_LUT)
18     {
19         auto cmd = ordered_json::object();
20         cmd["sntx"] = ""s + lut.namestr + ' ' + lut.argstr;
21         cmd["desc"] = lut.descstr;
22         doc["data"]["prg"]["cmds"].push_back(cmd);
23     }
24
25     Generator gen;
26     gen.add(1.0f, make_signal<SignalSine>(1000));
27     doc["data"]["prg"]["gnrtr"] = static_cast<json>(gen);
28
29     doc["data"]["prg"]["wvfrms"] = ordered_json::array();
30     doc["data"]["prg"]["wvfrms"].push_back(SignalType::Const);
31     [...]
32
33     std::string out = doc.dump();
34
35     ESP_LOGI(TAG, "Handler done.");
36     return httpd_resp_send(req, out.c_str(), out.length());
37 }
```

Listing A.37: Funkcja `welcome_handler` wysyłająca odpowiedź powitalną w formacie `JSON`

Następnie przechodzimy do jednego z dwóch głównych punktów końcowych, odpowiedzialnego za wgrywanie ustawień do urządzenia. Parsuje on przesłany tekst na obiekt `JSON`, z którego następnie wyciąga generatory i zadanie do wykonania, sprawdza je, wgrywa do pamięci i odsyła wiadomość ze statusem i ewentualnymi napotkanymi błędami.

```
1 static esp_err_t settings_handler(httpd_req_t *req)
2 {
```

```
3    // Make sure that the producer is *not* running
4    if (Communicator::check_if_running())
5       return httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
        ↪  "Device is busy");
6
7    ESP_LOGV(TAG, "Req len: %" PRIu16, req->content_len);
8
9    Interpreter::Program program;
10   std::vector<Generator> generators;
11   std::vector<std::string> errors;
12
13   ESP_LOGD(TAG, "Reading JSON...");
14   SocketReader reader(req);
15   ordered_json q = ordered_json::parse(reader.begin(), reader.end(),
     ↪  nullptr, false, true);
16
17   if (reader.err) // failed to read from socket
18   {
19      ESP_LOGW(TAG, "Reader error");
20      if (reader.err == HTTPD_SOCK_ERR_TIMEOUT)
21         httpd_resp_send_408(req);
22      return reader.err;
23   }
24
25   if (!q.is_discarded()) // if JSON is valid
26   {
27      if (q.contains("generators"))
28      {
29         ESP_LOGD(TAG, "Generators exists, trying...");
30         if (q.at("generators").is_array())
31         {
32            size_t count = q.at("generators").size();
33            generators.reserve(count);
34
35            for (auto &[key, val] : q.at("generators").items())
36            {
37               try
38               {
39                  Generator g = val.get<Generator>();
40                  generators.push_back(std::move(g));
```

```
41                }
42                catch (ordered_json::exception &e)
43                {
44                    generators.emplace_back();
45                    errors.push_back("Generator #"s + key + " failed to
                      ↪  parse: " + e.what());
46                    ESP_LOGW(TAG, "Generator failed to parse: %s",
                      ↪  e.what());
47                }
48                val = nullptr;
49            }
50        }
51        else
52            errors.push_back("\"generators\" is not an array!");
53        q.erase("generators");
54    }
55    if (q.contains("task"))
56    {
57        ESP_LOGD(TAG, "Task exists, trying...");
58        if (q.at("task").is_string())
59        {
60            const std::string &prg = q.at("task").get_ref<const
                  ↪  std::string &>();
61            program.parse(prg, errors);
62            ESP_LOGD(TAG, "Task has %u statements", program.size());
63        }
64        else
65            errors.push_back("\"task\" is not a string!");
66        q.erase("task");
67    }
68 }
69 else
70    errors.push_back("JSON is invalid!");
71
72 q.clear();
73
74 ESP_LOGD(TAG, "Moving configs...");
75 if (Board::move_config(program, generators) != ESP_OK)
76    return httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
       ↪  "Device is busy");
```

```
77
78      //
79      ESP_LOGD(TAG, "Responding...");
80
81      httpd_resp_set_type(req, "application/json");
82
83      if (!errors.empty())
84      {
85          ordered_json res = create_err_response(errors);
86          errors.clear();
87          std::string out = res.dump();
88          res.clear();
89          httpd_resp_set_status(req, HTTPD_400);
90          return httpd_resp_send(req, out.c_str(), out.length());
91      }
92
93      ordered_json res = create_ok_response();
94      res["message"] = "Settings have been validated. No errors found.";
95
96      std::string out = res.dump();
97      res.clear();
98
99      ESP_LOGI(TAG, "Handler done.");
100     return httpd_resp_send(req, out.c_str(), out.length());
101 }
```

Listing A.38: Funkcja `settings_handler` odpowiedzialna za wgrywanie ustawień

Najważniejszym (choć niekoniecznie najbardziej skomplikowanym) punktem końcowym jest ten odpowiedzialny za odsyłanie zmierzonych danych do użytkownika. Odczytuje on żądaną liczbę bajtów dla pomiarów czasu i przekazuje ją do komunikatora. Czyści bufor danych, przygotowuje się do konsumowania danych, sygnalizuje wątkowi interpretera że może zacząć produkować dane, i zaczyna je odczytywać i wysyłać do użytkownika. Pętla trwa dopóki interpreter nie zakończył pracy lub pozostają jakieś dane do wysłania. W tym czasie sprawdzana jest dostępność danych w buforze, jeśli istnieją to są one wysyłane i usuwane. Jeśli nie, sprawdzane jest połączenie z klientem. W obu przypadkach, jeśli nastąpiło niepowodzenie tzn. rozłączenie, interpreter jest powiadamiany i kończy przedwcześnie swoją pracę. Po każdej iteracji, zadanie serwera na moment zwalnia rdzeń dla innych zadań, np. stosu `TCP/IP`, komunikacji `Wi-Fi`, itd.

```
static esp_err_t io_handler(httpd_req_t *req)
{
    // Make sure that the producer is *not* running
    if (Communicator::is_running())
        return httpd_resp_send_err(req, HTTPD_500_INTERNAL_SERVER_ERROR,
        ↪  "Device is busy");

    auto qr = parse_query(req);

    size_t time_bytes = 0;
    if (auto it = qr.find("tb"); it != qr.end())
        try_parse_integer(it->second, time_bytes);

    if (time_bytes > 8)
        time_bytes = 8;

    // Apply changes
    ESP_LOGI(TAG, "Preparing Communicator...");
    Communicator::time_settings(time_bytes);
    Communicator::cleanup();

    // Start consumer
    ESP_LOGI(TAG, "Running consumer...");
    httpd_resp_set_type(req, "application/octet-stream");

    size_t total_sent = 0;
    esp_err_t ret = ESP_OK;

    // Start producer
    ESP_LOGI(TAG, "Notifying producer...");
    Communicator::start_running();

    while (Communicator::is_running() || Communicator::has_data())
    {
        auto rsvd = Communicator::get_read();

        if (rsvd.size() != 0) // if something to send, send
        {
            ESP_LOGI(TAG, "Sending %zu bytes...", rsvd.size());
```

```
39
40          ret = httpd_resp_send_chunk(req, rsvd.data(), rsvd.size());
41          total_sent += rsvd.size();
42       }
43       else // if no new data
44       {
45          if (!httpd_req_check_live(req)) // check if dead, ask to stop
46          {
47             ESP_LOGW(TAG, "Client disconnected...");
48             ret = HTTPD_SOCK_ERR_TIMEOUT;
49          }
50       }
51
52       Communicator::commit_read();
53
54       if (ret != ESP_OK)
55          break;
56
57       taskYIELD();
58    }
59
60    Communicator::ask_to_exit();
61
62    while (Communicator::is_running())
63       vTaskDelay(pdMS_TO_TICKS(10));
64
65    ESP_LOGW(TAG, "Total sent: %zu bytes...", total_sent);
66
67    if (ret != ESP_OK)
68       return ret;
69
70    ESP_LOGI(TAG, "Handler done.");
71    return httpd_resp_send_chunk(req, nullptr, 0);
72 }
```

Listing A.39: Funkcja `io_handler` uruchamiająca interpreter i przesyłająca pomiary do użytkownika

A.3.3 Obsługa sprzętu i interpreter

Główną częścią programu jest oczywiście połączona obsługa wszystkich komponentów. Została w tym celu stworzona przestrzeń `Board` (oznaczająca całą płytkę).
Zewnętrzny interfejs jest bardzo niewielki. Zostały stworzone enumeracje reprezentujące analogowe porty wejściowe i wyjściowe, oraz enumeracja reprezentująca zakres wejścia, a więc dzielniki lub wzmacniacz. Do tego kilka stałych numerycznych, jak napięcie odniesienia, transrezystancja (Ω, $\frac{\mathrm{V}}{\mathrm{A}}$) wejścia prądowego czy transkonduktancja (S, $\frac{\mathrm{A}}{\mathrm{V}}$) wyjścia prądowego, ze względu na fakt, że przetworniki operują napięciami; wartości dzielników wejść i wzmacniacza prądowego.

```
1  enum class Input : uint8_t
2  {
3      None = 0,
4      In1 = 1,
5      In2 = 2,
6      In3 = 3,
7      In4 = 4,
8      Inv = 5,
9  };
10
11 enum class Output : uint8_t
12 {
13     None = 0,
14     Out1 = 1,
15     Out2 = 2,
16     Inv = 3,
17 };
18
19 enum class AnIn_Range : uint8_t
20 {
21     OFF = 0,
22     Min = 1,
23     Med = 2,
24     Max = 3,
25 };
26
27 namespace Board
28 {
29     constexpr const char *const TAG = "IOBoard";
```

```
30
31      constexpr double u_ref = 4.096;
32      constexpr double out_ref = u_ref / 2 / 2 * 10;
33
34      constexpr int32_t ItoU_input = 1;
35      constexpr int32_t UtoI_output = 100; // 100mA per 10V => 1V = 0.01A
36
37      // Divider settings:        Min: 1V=>1V, Med: 10V=>1V, Max: 100V=>1V
38      constexpr int32_t volt_divs[4] = {0, 1, 10, 100}; // ratios of
        ↪  dividers | min range => min attn
39      // Gains settings: R=1Ohm; Min: 1mA=>1V, Med: 10mA=>1V, Max:
        ↪  100mA=>1V
40      constexpr int32_t curr_gains[4] = {5, 1000, 100, 10}; // gains of
        ↪  instr.amp | min range => max gain
41
42      esp_err_t init();
43      esp_err_t deinit();
44
45      esp_err_t move_config(Interpreter::Program &, std::vector<Generator>
        ↪  &);
46      esp_err_t give_sem_emergency();
47      esp_err_t test();
48  };
```

Listing A.40: Interfejs przestrzeni `Board`

W implementacji dostępne są również przedstawione niżej "prywatne" zmienne (a raczej stałe). Określają one liczbę wejść i wyjść analogowych, cyfrowych, piny `GPIO` odpowiedzialne za wejścia i wyjścia cyfrowe. Zostają wstępnie stworzone obiekty ekspanderów i przetworników. Inicjalizowane są zmienne stanu wyjść cyfrowych oraz przedziałów wejść analogowych. Rezerwowane jest miejsce do przechowywania programu i generatorów. Następna część to zmienne nie dotyczące bezpośrednio sprzętu, lecz implementacji różnych rozwiązań – wątek interpretera, mutex dostępu do danych, timer synchronizujący wykonywanie, semafor który jest przez niego przełączany, zmienne pomocnicze do zarządzania czasem i synchronizacją. Finalnie przygotowane są zmienne pomocnicze przechowujące transakcje do przetworników, maski bitowe pinów i tablice pomocnicze do zapisu wyjść.

```
1  namespace
2  {
```

```
3     // SPECIFICATION
4     constexpr size_t an_in_num = 4;
5     constexpr size_t an_out_num = 2;
6
7     constexpr size_t dg_in_num = 4;
8     constexpr size_t dg_out_num = 4;
9
10    // HARDWARE SETUP
11    constexpr std::array<gpio_num_t, dg_in_num> dig_in = {GPIO_NUM_34,
      ↪  GPIO_NUM_35, GPIO_NUM_36, GPIO_NUM_39};
12    constexpr std::array<gpio_num_t, dg_out_num> dig_out = {GPIO_NUM_4,
      ↪  GPIO_NUM_25, GPIO_NUM_26, GPIO_NUM_27};
13
14    MCP23008 expander_a(I2C_NUM_0, 0b000);
15    MCP23008 expander_b(I2C_NUM_0, 0b001);
16
17    MCP3204 adc(SPI3_HOST, GPIO_NUM_5, 2'000'000);
18    MCP4922 dac(SPI2_HOST, GPIO_NUM_15, 20'000'000);
19
20    // STATE MACHINE
21    uint32_t dg_out_state = 0;
22    std::array<AnIn_Range, an_in_num> an_in_range;
23
24    // CONFIG
25    std::vector<Generator> generators;
26    Interpreter::Program program;
27
28    //================================//
29    //            HELPERS             //
30    //================================//
31
32    // SOFTWARE SETUP
33    TaskHandle_t execute_task = nullptr;
34    std::mutex data_mutex;
35
36    gptimer_handle_t sync_timer = nullptr;
37    gptimer_alarm_config_t sync_alarm_cfg = {};
38
39 #if SYNC_USE_NOTIF_NOT_SEM
40    constexpr UBaseType_t notif_idx = 0;
```

```
#else
    DRAM_ATTR SemaphoreHandle_t sync_semaphore;
#endif

    // EXECUTION
    uint64_t time_now = 0;
    uint64_t &time_sync = sync_alarm_cfg.alarm_count;
    DRAM_ATTR bool wait_for_sync = false;

    // ADC/DAC TRANSACTIONS
    std::array<spi_transaction_t, an_in_num> trx_in;
    std::array<spi_transaction_t, an_out_num> trx_out;

    // CONSTANTS
    constexpr uint32_t dg_out_mask = (1 << dg_out_num) - 1;
    constexpr uint32_t dg_in_mask = (1 << dg_in_num) - 1;

    // LUT
    constexpr size_t dg_out_lut_sz = 1 << dg_out_num;
    constexpr std::array<uint32_t, dg_out_lut_sz> dg_out_lut_s = []()
    {
        std::array<uint32_t, dg_out_lut_sz> ret = {};
        for (size_t in = 0; in < dg_out_lut_sz; ++in)
            for (size_t b = 0; b < dg_out_num; ++b)
                if (in & BIT(b))
                    ret[in] |= BIT(dig_out[b]);
        return ret;
    }();
    constexpr std::array<uint32_t, dg_out_lut_sz> dg_out_lut_r = []()
    {
        std::array<uint32_t, dg_out_lut_sz> ret = {};
        for (size_t in = 0; in < dg_out_lut_sz; ++in)
            for (size_t b = 0; b < dg_out_num; ++b)
                if (~in & BIT(b))
                    ret[in] |= BIT(dig_out[b]);
        return ret;
    }();

    // I/O conversion
    constexpr int32_t halfrangein = MCP3204::ref / 2;
```

```
81      constexpr int32_t halfrangeout = MCP4922::ref / 2;
82  }
```

Listing A.41: Zmienne “prywatne” przestrzeni `Board`

Funkcje obsługujące sprzęt (*backend*)

Teraz zostaną omówione poszczególne ważniejsze funkcje, również te które są dostępne tylko w implementacji. Interfejs zostanie przedstawiony na końcu, gdyż przeważnie używa tych od implementacji.
Funkcje konwertujące unipolarny kod z przetworników na bipolarną wartość lub odwrotnie – proste przesunięcie.

```
1  static inline constexpr int32_t adc_offset(MCP3204::out_t val)
2  {
3      return static_cast<int32_t>(val) - halfrangein;
4  }
5
6  static inline constexpr MCP4922::in_t dac_offset(int32_t val)
7  {
8      return val + halfrangeout;
9  }
```

Listing A.42: Funkcje konwertujące kod na wartość symetryczną lub odwrotnie

Funkcja-szablon do konwersji pomiaru wejścia na wartość rzeczywistą. Przyjmuje typ numeryczny i opcjonalny mnożnik, w celu uniknięcia powtarzania kodu. Przesuwa wartość ze względu na bipolarność wejść, mnoży przez napięcie odniesienia, mnożnik oraz współczynnik dzielnika lub wzmacniacza.

```
1  template <typename num_t, int32_t mul = 1>
2  static num_t bin_to_phy(Input in, int32_t sum, int32_t cnt)
3  {
4      constexpr num_t ratio = u_ref * mul / halfrangein;
5
6      if (in == Input::None || in == Input::Inv) [[unlikely]]
7          return 0;
8
9      size_t inidx = static_cast<size_t>(in) - 1;
```

```
10      size_t rngidx = static_cast<size_t>(an_in_range[inidx]);
11      switch (in)
12      {
13      case Input::In1:
14      case Input::In2:
15      case Input::In3:
16          return sum * ratio * volt_divs[rngidx] / cnt;
17      case Input::In4:
18          return sum * ratio / curr_gains[rngidx] / cnt * ItoU_input;
19      default:
20          return 0;
21      }
22  }
```

Listing A.43: Funkcja-szablon `bin_to_phy` konwertująca kod binarny na wartość rzeczywistą

Ta funkcja działa analogicznie, lecz w drugą stronę. Zamienia wartość rzeczywistą na kod dla przetwornika, biorąc pod uwagę sposób implementacji wyjść.

```
1  static MCP4922::in_t phy_to_dac(Output out, float val)
2  {
3      constexpr float ratio = halfrangeout / out_ref;
4
5      if (out == Output::None || out == Output::Inv) [[unlikely]]
6          return 0;
7
8      if (out == Output::Out2)
9          val *= UtoI_output;
10
11     if (val >= out_ref) [[unlikely]]
12         return MCP4922::max;
13     if (val <= -out_ref) [[unlikely]]
14         return MCP4922::min;
15
16     return dac_offset(std::round(val * ratio));
17 }
```

Listing A.44: Funkcja `phy_to_dac` zamieniająca wartość rzeczywistą na kod przetwornika

Teraz zaczynają się implementacje poszczególnych instrukcji, używane w interpreterze. Funkcja ustawiająca zakresy wejść analogowych. Przyjmuje wejście i zakres. Zapisuje do tablicy.

```
static esp_err_t analog_input_range(Input in, AnIn_Range r)
{
    if (in == Input::None || in >= Input::Inv) [[unlikely]]
        return ESP_ERR_INVALID_ARG;

    uint8_t pos = static_cast<uint8_t>(in) - 1;
    an_in_range[pos] = r;
    return ESP_OK;
}
```

Listing A.45: Funkcja `analog_input_range` ustawiająca zakres wejść portu

Funkcja wyłączająca wejścia analogowe. Wysyła do obu ekspanderów same zera, co powoduje rozłączenie wszystkich styczników.

```
static esp_err_t analog_inputs_disable()
{
    ESP_RETURN_ON_ERROR(
        expander_a.set_pins(0x00),
        TAG, "Failed to expander_a.set_pins!");
    ESP_RETURN_ON_ERROR(
        expander_b.set_pins(0x00),
        TAG, "Failed to expander_a.set_pins!");

    return ESP_OK;
}
```

Listing A.46: Funkcja `analog_inputs_disable` wyłączająca wejścia analogowe

Funkcja `analog_inputs_enable` włączająca wejścia analogowe. Ustawia styczniki za pomocą ekspanderów zgodnie z ustawionymi wcześniej zakresami.

```
static esp_err_t analog_inputs_enable()
```

```
{
    uint8_t lower = range_to_expander_steering(an_in_range[0]) |
    ↪  range_to_expander_steering(an_in_range[1]) << 4;
    uint8_t upper = range_to_expander_steering(an_in_range[2]) |
    ↪  range_to_expander_steering(an_in_range[3]) << 4;

    ESP_RETURN_ON_ERROR(
        expander_a.set_pins(lower),
        TAG, "Failed to expander_a.set_pins!");
    ESP_RETURN_ON_ERROR(
        expander_b.set_pins(upper),
        TAG, "Failed to expander_b.set_pins!");

    return ESP_OK;
}
```

Listing A.47: Funkcja `analog_inputs_enable` włączająca wejścia analogowe

Funkcja wykonująca pomiar wejścia. Przyjmuje wejście, zwraca do podanej zmiennej odczytany kod.

```
static esp_err_t analog_input_read(Input in, MCP3204::out_t &out)
{
    if (in == Input::None || in == Input::Inv) [[unlikely]]
        return ESP_ERR_INVALID_ARG;

    spi_transaction_t &trx = trx_in[static_cast<size_t>(in) - 1];

    ESP_RETURN_ON_ERROR(
        adc.send_trx(trx),
        TAG, "Failed to ADC send_trx!");
    ESP_RETURN_ON_ERROR(
        adc.recv_trx(),
        TAG, "Failed to ADC recv_trx!");

    out = adc.parse_trx(trx);
    return ESP_OK;
}
```

Listing A.48: Funkcja `analog_input_read` wykonująca pomiar wejścia

Funkcja wykonująca przeciwną operację – wpisuje słowo do przetwornika na podane wyjście. Przyjmuje wyjście i kod.

```
static esp_err_t analog_output_write(Output out, MCP4922::in_t val)
{
    if (out == Output::None || out == Output::Inv) [[unlikely]]
        return ESP_ERR_INVALID_ARG;

    spi_transaction_t &trx = trx_out[static_cast<size_t>(out) - 1];
    dac.write_trx(trx, val);

    ESP_RETURN_ON_ERROR(
        dac.send_trx(trx),
        TAG, "Failed to dac.send_trx!");
    ESP_RETURN_ON_ERROR(
        dac.recv_trx(),
        TAG, "Failed to dac.recv_trx!");

    return ESP_OK;
}
```

Listing A.49: Funkcja `analog_output_write` wpisująca kod do przetwornika

Jest jeszcze jedna funkcja pisząca do wyjść analogowych. Służy ona do początkowego wyzerowania ich – domyślnie po włączeniu urządzenia są one w stanie wysokiej impedancji, a więc dzielnik powoduje *pulldown* i przyjęcie skrajnej ujemnej wartości. Tutaj wystawiana jest wartość odpowiadająca fizycznemu zeru. Używana jest też do sprzątania po skończeniu wykonywania programu.

```
static esp_err_t analog_outputs_reset()
{
    constexpr MCP4922::in_t midpoint = MCP4922::ref / 2;
    ESP_RETURN_ON_ERROR(
        analog_output_write(Output::Out1, midpoint),
        TAG, "Failed to analog_output_write 1!");

    ESP_RETURN_ON_ERROR(
        analog_output_write(Output::Out2, midpoint),
```

```
10          TAG, "Failed to analog_output_write 2!");
11
12      return ESP_OK;
13  }
```

Listing A.50: Funkcja `analog_outputs_reset` zerująca wyjścia analogowe.

Funkcja ustawiająca wyjścia cyfrowe. Włącza i wyłącza odpowiednie piny pisząc do rejestrów *write-1-to-set* oraz *write-1-to-clear*.

```
1  static void digital_outputs_to_registers()
2  {
3      dg_out_state &= dg_out_mask;
4      REG_WRITE(GPIO_OUT_W1TS_REG, dg_out_lut_s[dg_out_state]);
5      REG_WRITE(GPIO_OUT_W1TC_REG, dg_out_lut_r[dg_out_state]);
6  }
```

Listing A.51: Funkcja `digital_outputs_to_registers` ustawiająca wyjścia cyfrowe.

Kilka funkcji służących do zarządzania zmienną stanu wyjść cyfrowych.

```
1  static void digital_outputs_wr(uint32_t in)
2  {
3      dg_out_state = in;
4      digital_outputs_to_registers();
5  }
6  static void digital_outputs_set(uint32_t in)
7  {
8      dg_out_state |= in;
9      digital_outputs_to_registers();
10 }
11 static void digital_outputs_rst(uint32_t in)
12 {
13     dg_out_state &= ~in;
14     digital_outputs_to_registers();
15 }
16 static void digital_outputs_and(uint32_t in)
17 {
18     dg_out_state &= in;
```

```
19      digital_outputs_to_registers();
20  }
21  static void digital_outputs_xor(uint32_t in)
22  {
23      dg_out_state ^= in;
24      digital_outputs_to_registers();
25  }
```

Listing A.52: Funkcje zarządzające wyjściami cyfrowymi

Funkcja zwracająca stan wejść cyfrowych. Odczytuje stan pinów z rejestru. Po kolei sprawdza wartość bitów i kompresuje je koło siebie w 4-bitową liczbę całkowitą. Rozwinięcie pętli pozwala na maksymalną optymalizację (również dzięki zastosowaniu wielu wartości `constexpr`).

```
1  #pragma GCC push_options
2  #pragma GCC optimize("unroll-loops")
3  static void digital_inputs_read(uint32_t &out)
4  {
5      uint32_t in = REG_READ(GPIO_IN1_REG);
6      out = 0;
7      for (size_t b = 0; b < dg_in_num; ++b)
8          out |= !!(in & BIT(dig_in[b] - 32)) << b;
9  }
10 #pragma GCC pop_options
```

Listing A.53: Funkcja `digital_inputs_read` odczytująca stan wejść cyfrowych.

Ostania funkcja do zarządzania portów służy do resetu stanu wejść i wyjść. Czyści tablicę zakresów, wyłącza styczniki, wyłącza wyjścia cyfrowe, zeruje wyjścia analogowe. Używana do inicjalizacji i sprzątania po skończeniu wykonywania programu.

```
1  static esp_err_t port_cleanup()
2  {
3      for (size_t i = 0; i < an_in_num; ++i)
4          an_in_range[i] = AnIn_Range::OFF;
5
6      digital_outputs_wr(0);
7
```

```
8       ESP_RETURN_ON_ERROR(
9           analog_inputs_disable(),
10          TAG, "Failed to analog_inputs_disable!");
11
12      ESP_RETURN_ON_ERROR(
13          analog_outputs_reset(),
14          TAG, "Failed to analog_outputs_reset!");
15
16      return ESP_OK;
17  }
```

Listing A.54: Funkcja `port_cleanup` inicjalizująca/resetująca wejścia i wyjścia.

Poniżej przedstawiony jest *callback* timera – funkcja wykonywana jako przerwanie w momencie kiedy napotka on oczekiwaną wartość czasu. Jeśli interpreter czeka na synchronizację, daje semafor/powiadomienie i oddaje wykonywanie wątkowi.

```
1  static IRAM_ATTR bool sync_callback(gptimer_handle_t timer, const
   ↪  gptimer_alarm_event_data_t *edata, void *user_ctx)
2  {
3      BaseType_t high_task_awoken = pdFALSE;
4
5      if (edata->alarm_value == time_sync)
6  #if SYNC_USE_NOTIF_NOT_SEM
7          vTaskNotifyGiveIndexedFromISR(execute_task, notif_idx,
           ↪  &high_task_awoken);
8  #else
9          xSemaphoreGiveFromISR(sync_semaphore, &high_task_awoken);
10 #endif
11
12     return high_task_awoken == pdTRUE;
13 }
```

Listing A.55: Funkcja `sync_callback` uruchamiana jako przerwanie przez timer

Następnie stworzone zostały pomocnicze makra, wymuszające synchronizację jeśli na nią czekamy, oraz wyłączające synchronizację.

```
1  #if SYNC_USE_NOTIF_NOT_SEM
2
```

```
#define WAIT_FOR_SYNC        \
   do                        \
   {                         \
      if (wait_for_sync)  \
         while (ulTaskNotifyTakeIndexed(notif_idx, pdTRUE,
         ↪  portMAX_DELAY) != pdTRUE) \
            ;              \
   } while (0)

#define CLEAR_SYNC ulTaskNotifyTakeIndexed(notif_idx, pdTRUE, 0)

#else

#define WAIT_FOR_SYNC        \
   do                        \
   {                         \
      if (wait_for_sync)  \
         while (xSemaphoreTake(sync_semaphore, portMAX_DELAY) != pdTRUE)
         ↪  \
            ;              \
   } while (0)

#define CLEAR_SYNC xSemaphoreTake(sync_semaphore, 0)

#endif
```

Listing A.56: Makra pomocnicze do synchronizacji wykonywania zadania

Funkcje interfejsu (*frontend*)

Teraz przedstawione zostaną funkcje stanowiące interfejs.
Funkcja inicjalizuje wszystkie podzespoły i inne elementy. Ustawia piny wejść i wyjść cyfrowych, tworzy transakcje do przetworników, inicjalizuje przetworniki, inicjalizuje ekspandery, czyści porty. Przygotowuje i włącza timer, uruchamia wątek interpretera. Istnieje również jej przeciwieństwo – funkcja `deinit` która zatrzymuje interpreter, wyłącza timer, deinicjalizuje wszystkie podzespoły.

```
esp_err_t init()
{
```

```
3     ESP_LOGI(TAG, "Initing Board...");
4
5     // GPIO
6     for (size_t i = 0; i < dg_in_num; ++i)
7     {
8         gpio_set_direction(dig_in[i], GPIO_MODE_INPUT);
9         gpio_pulldown_en(dig_in[i]);
10    }
11
12    for (size_t i = 0; i < dg_out_num; ++i)
13        gpio_set_direction(dig_out[i], GPIO_MODE_OUTPUT);
14
15    // ADC/DAC
16    for (size_t i = 0; i < an_in_num; ++i)
17        trx_in[i] = MCP3204::make_trx(i);
18
19    for (size_t i = 0; i < an_out_num; ++i)
20        trx_out[i] = MCP4922::make_trx(an_out_num - i - 1); // reversed,
          ↪ because rotated chip
21
22    ESP_RETURN_ON_ERROR(
23        adc.init(),
24        TAG, "Error in adc.init!");
25
26    ESP_RETURN_ON_ERROR(
27        dac.init(),
28        TAG, "Error in dac.init!");
29
30    ESP_RETURN_ON_ERROR(
31        adc.acquire_spi(),
32        TAG, "Error in adc.acquire_spi!");
33
34    ESP_RETURN_ON_ERROR(
35        dac.acquire_spi(),
36        TAG, "Error in dac.acquire_spi!");
37
38    // EXPANDER
39    ESP_RETURN_ON_ERROR(
40        expander_a.init(true),
41        TAG, "Error in expander_a.init!");
```

```
42
43      ESP_RETURN_ON_ERROR(
44          expander_b.init(true),
45          TAG, "Error in expander_b.init!");
46
47      // CLEANUP BOARD
48      ESP_RETURN_ON_ERROR(
49          port_cleanup(),
50          TAG, "Error in port_cleanup!");
51
52      // TIMER
53  #if SYNC_USE_NOTIF_NOT_SEM
54  #else
55      ESP_RETURN_ON_FALSE(
56          (sync_semaphore = xSemaphoreCreateBinary()),
57          ESP_ERR_NO_MEM, TAG, "Error in xSemaphoreCreateBinary!");
58  #endif
59
60      constexpr gptimer_config_t timer_config = {
61          .clk_src = GPTIMER_CLK_SRC_DEFAULT,
62          .direction = GPTIMER_COUNT_UP,
63          .resolution_hz = 1 * 1000 * 1000, // 1MHz, 1 tick = 1us
64          .flags = {
65              .intr_shared = false,
66          },
67      };
68
69      sync_alarm_cfg.reload_count = 0;
70      sync_alarm_cfg.alarm_count = 0;
71      sync_alarm_cfg.flags.auto_reload_on_alarm = false;
72
73      constexpr gptimer_event_callbacks_t evt_cb_cfg = {
74          .on_alarm = sync_callback,
75      };
76
77      ESP_RETURN_ON_ERROR(
78          gptimer_new_timer(&timer_config, &sync_timer),
79          TAG, "Error in gptimer_new_timer!");
80
81      ESP_RETURN_ON_ERROR(
```

```
82          gptimer_set_alarm_action(sync_timer, &sync_alarm_cfg),
83          TAG, "Error in gptimer_set_alarm_action!");
84
85      ESP_RETURN_ON_ERROR(
86          gptimer_register_event_callbacks(sync_timer, &evt_cb_cfg,
            ↪  nullptr),
87          TAG, "Error in gptimer_register_event_callbacks!");
88
89      ESP_RETURN_ON_ERROR(
90          gptimer_enable(sync_timer),
91          TAG, "Error in gptimer_enable!");
92
93      // SPAWN EXE TASK
94      ESP_RETURN_ON_FALSE(
95          xTaskCreatePinnedToCore(execute, "BoardTask", BOARD_MEM, nullptr,
            ↪  BOARD_PRT, &execute_task, CPU1),
96          ESP_ERR_NO_MEM, TAG, "Error in xTaskCreatePinnedToCore!");
97
98      ESP_LOGI(TAG, "Done!");
99      return ESP_OK;
100 }
```

Listing A.57: Funkcja `init` ustawiająca wszystkie podzespoły po otrzymaniu zasilania

Funkcja przyjmuje program i generatory i zapisuje je poprzez przeniesienie (niszczy oryginał).

```
1  esp_err_t move_config(Interpreter::Program &p, std::vector<Generator> &g)
2  {
3      if (!data_mutex.try_lock())
4          return ESP_ERR_INVALID_STATE;
5
6      program = std::move(p);
7      generators = std::move(g);
8
9      data_mutex.unlock();
10     return ESP_OK;
11 }
```

Listing A.58: Funkcja `move_config` zapisująca program i generatory

Funkcja pozwala dać semafor synchronizujący z zewnątrz. Używana jest przez `Communicator` w celu przerwania oczekiwania, jeśli użytkownik się rozłączył w trakcie wykonywania programu.

```
esp_err_t give_sem_emergency()
{
#if SYNC_USE_NOTIF_NOT_SEM
    if (xTaskNotifyGiveIndexed(execute_task, notif_idx) != pdTRUE)
#else
    if (xSemaphoreGive(sync_semaphore) != pdTRUE)
#endif
        return ESP_FAIL;
    return ESP_OK;
}
```

Listing A.59: Funkcja `give_sem_emergency` dająca semafor z zewnątrz

Funkcja – wątek interpretera

Ostatnią, najważniejszą funkcją jest ta stanowiąca główny kod interpretera. Jest ona uruchamiana jako osobne zadanie na własnym, całkowicie przydzielonym rdzeniu. Wykorzystuje większość wyżej opisanych funkcji, i praktycznie cały kod jaki istnieje w programie (poza samym serwerem `HTTP`).
Na początku czeka w pętli aż dostanie zezwolenie na start od wątku serwera. Przygotowuje wszystko, tzn. zamyka dostęp do zmiennych, czyści porty, sprawdza program. Następnie uruchamia timer i zaczyna interpretację programu. W pętli pobierana jest nowa instrukcja, odczytywany jest port i następuje wykonanie operacji. `DELAY` dodaje opóźnienie do wcześniejszego końca timera, sprawdza czy już nie minął, i kontynuuje. `GETTM` pobiera obecny rzeczywisty czas, jeśli jest późniejszy niż oczekiwana synchronizacja to ją nadpisuje. Reszta operacji oczekuje na synchronizację, po czym po prostu wykonuje jedną (lub kilka) z wcześniej opisanych funkcji. Po wykonaniu synchronizowanej operacji, synchronizacja jest wyłączana – kolejne operacje wykonają się tak szybko jak to możliwe. Tylko `DELAY` i `GETTM` mogą włączać ją ponownie, jeśli oczekiwany czas jeszcze nie minął. Jeśli gdzieś wystąpił błąd – np. funkcja nie wykonała się poprawnie, nie ma miejsca w buforze lub serwer zasygnalizował rozłączenie użytkownika – interpreter kończy pracę. Po skończeniu zadania, zatrzymywany jest timer, resetowane są porty i wątek oczekuje na ponowne uruchomienie.

```
1  static void interpreter_task(void *arg)
2  {
3      __attribute__((unused)) esp_err_t ret; // used in on_false macros
4
5      Input in;
6      Output out;
7
8      ESP_LOGI(TAG, "Starting the Board executor...");
9      while (true)
10     {
11         ESP_LOGI(TAG, "Waiting for task...");
12         while (!Communicator::is_running())
13             vTaskDelay(1);
14
15         ESP_LOGI(TAG, "Dispatched...");
16
17         // lock access
18         std::lock_guard<std::mutex> lock(data_mutex);
19
20         // prepare hardware
21         ESP_GOTO_ON_ERROR(
22             port_cleanup(),
23             label_fail, TAG, "Failed to port_cleanup!");
24
25         // prepare software
26         ESP_GOTO_ON_FALSE(
27             program.isValid(),
28             ESP_ERR_INVALID_STATE, label_fail, TAG, "Program is
               ↪  invalid!");
29
30         program.reset();
31
32         // letsgooo
33         time_now = 0;
34         time_sync = -1;
35         wait_for_sync = false;
36
37         ESP_GOTO_ON_ERROR(
38             gptimer_set_raw_count(sync_timer, time_now),
```

```
39              label_fail, TAG, "Failed to gptimer_set_raw_count!");
40
41          ESP_GOTO_ON_ERROR(
42              gptimer_set_alarm_action(sync_timer, &sync_alarm_cfg),
43              label_fail, TAG, "Failed to gptimer_set_alarm_action!");
44
45          ESP_GOTO_ON_ERROR(
46              gptimer_start(sync_timer),
47              label_fail, TAG, "Failed to gptimer_start!");
48
49          time_sync = 0;
50
51          while (true)
52          {
53              bool comm_ok = true;
54
55              Interpreter::InstrPtr stmt = program.getInstr();
56
57              if (stmt == Interpreter::nullinstr) [[unlikely]]
58                  break;
59
60              in = static_cast<Input>(stmt->port);
61              out = static_cast<Output>(stmt->port);
62
63              switch (stmt->opc)
64              {
65              case OPCode::DELAY:
66                  time_sync += stmt->arg.u;
67                  wait_for_sync = true;
68                  CLEAR_SYNC;
69                  ESP_GOTO_ON_ERROR(
70                      gptimer_set_alarm_action(sync_timer,
                        ↪ &sync_alarm_cfg),
71                      label_fail, TAG, "Failed to gptimer_set_alarm_action
                        ↪ in OPCode::DELAY!");
72                  continue;
73
74              case OPCode::GETTM:
75                  time_sync = std::max(time_sync, get_now());
76                  wait_for_sync = true;
```

```
77                  CLEAR_SYNC;
78                  ESP_GOTO_ON_ERROR(
79                      gptimer_set_alarm_action(sync_timer,
                        ↪  &sync_alarm_cfg),
80                      label_fail, TAG, "Failed to gptimer_set_alarm_action
                        ↪  in OPCode::GETTM!");
81                  continue;
82
83              case OPCode::RSTTM:
84                  WAIT_FOR_SYNC;
85                  time_sync = -1;
86                  wait_for_sync = false;
87                  time_now = 0;
88                  CLEAR_SYNC;
89                  ESP_GOTO_ON_ERROR(
90                      gptimer_set_alarm_action(sync_timer,
                        ↪  &sync_alarm_cfg),
91                      label_fail, TAG, "Failed to gptimer_set_alarm_action
                        ↪  in OPCode::RSTTM!");
92                  time_sync = 0;
93                  ESP_GOTO_ON_ERROR(
94                      gptimer_set_raw_count(sync_timer, time_now),
95                      label_fail, TAG, "Failed to gptimer_set_raw_count in
                        ↪  OPCode::RSTTM!");
96                  continue;
97
98              case OPCode::DIRD:
99              {
100                 WAIT_FOR_SYNC;
101                 uint32_t val;
102                 digital_inputs_read(val);
103                 comm_ok = Communicator::write_data(get_now(), val);
104                 break;
105             }
106
107             case OPCode::DOWR:
108                 WAIT_FOR_SYNC;
109                 digital_outputs_wr(stmt->arg.u);
110                 break;
111             case OPCode::DOSET:
```

```
112                 WAIT_FOR_SYNC;
113                 digital_outputs_set(stmt->arg.u);
114                 break;
115             case OPCode::DORST:
116                 WAIT_FOR_SYNC;
117                 digital_outputs_rst(stmt->arg.u);
118                 break;
119             case OPCode::DOAND:
120                 WAIT_FOR_SYNC;
121                 digital_outputs_and(stmt->arg.u);
122                 break;
123             case OPCode::DOXOR:
124                 WAIT_FOR_SYNC;
125                 digital_outputs_xor(stmt->arg.u);
126                 break;
127 
128             case OPCode::AIRDF:
129             {
130                 WAIT_FOR_SYNC;
131                 int32_t sum = 0;
132                 MCP3204::out_t rd;
133                 for (size_t r = stmt->arg.u; r; --r)
134                 {
135                     ESP_GOTO_ON_ERROR(
136                         analog_input_read(in, rd),
137                         label_fail, TAG, "Failed to analog_input_read in
                            ↪ OPCode::AIRDF!");
138                     sum += adc_offset(rd);
139                 }
140                 float val = bin_to_phy<float>(in, sum, stmt->arg.u);
141                 comm_ok = Communicator::write_data(get_now(), val);
142                 break;
143             }
144             case OPCode::AIRDM:
145             {
146                 WAIT_FOR_SYNC;
147                 int32_t sum = 0;
148                 MCP3204::out_t rd;
149                 for (size_t r = stmt->arg.u; r; --r)
150                 {
```

```
151                     ESP_GOTO_ON_ERROR(
152                         analog_input_read(in, rd),
153                         label_fail, TAG, "Failed to analog_input_read in
                            ↪ OPCode::AIRDF!");
154                     sum += adc_offset(rd);
155                 }
156                 int32_t val = bin_to_phy<int32_t, 1'000>(in, sum,
                    ↪ stmt->arg.u);
157                 comm_ok = Communicator::write_data(get_now(), val);
158                 break;
159             }
160             case OPCode::AIRDU:
161             {
162                 WAIT_FOR_SYNC;
163                 int32_t sum = 0;
164                 MCP3204::out_t rd;
165                 for (size_t r = stmt->arg.u; r; --r)
166                 {
167                     ESP_GOTO_ON_ERROR(
168                         analog_input_read(in, rd),
169                         label_fail, TAG, "Failed to analog_input_read in
                            ↪ OPCode::AIRDF!");
170                     sum += adc_offset(rd);
171                 }
172                 int32_t val = bin_to_phy<int32_t, 1'000'000>(in, sum,
                    ↪ stmt->arg.u);
173                 comm_ok = Communicator::write_data(get_now(), val);
174                 break;
175             }
176
177             case OPCode::AOVAL:
178             {
179                 MCP4922::in_t outval = phy_to_dac(out, stmt->arg.f);
180                 WAIT_FOR_SYNC;
181                 ESP_GOTO_ON_ERROR(
182                     analog_output_write(out, outval),
183                     label_fail, TAG, "Failed to analog_output_write in
                        ↪ OPCode::AOVAL!");
184                 break;
185             }
```

```
186             case OPCode::AOGEN:
187             {
188                 MCP4922::in_t outval = phy_to_dac(out, (stmt->arg.u <
                    ↪  generators.size()) ?
                    ↪  generators[stmt->arg.u].get(time_sync) : 0);
189                 WAIT_FOR_SYNC;
190                 ESP_GOTO_ON_ERROR(
191                     analog_output_write(out, outval),
192                     label_fail, TAG, "Failed to analog_output_write in
                        ↪  OPCode::AOGEN!");
193                 break;
194             }
195
196             case OPCode::AIEN:
197                 WAIT_FOR_SYNC;
198                 ESP_GOTO_ON_ERROR(
199                     analog_inputs_enable(),
200                     label_fail, TAG, "Failed to analog_inputs_enable in
                        ↪  OPCode::AOVAL!");
201                 break;
202             case OPCode::AIDIS:
203                 WAIT_FOR_SYNC;
204                 ESP_GOTO_ON_ERROR(
205                     analog_inputs_disable(),
206                     label_fail, TAG, "Failed to analog_inputs_disable in
                        ↪  OPCode::AIDIS!");
207                 break;
208             case OPCode::AIRNG:
209                 WAIT_FOR_SYNC;
210                 ESP_GOTO_ON_ERROR(
211                     analog_input_range(in,
                        ↪  static_cast<AnIn_Range>(stmt->arg.u)),
212                     label_fail, TAG, "Failed to analog_input_range in
                        ↪  OPCode::AIRNG!");
213                 break;
214
215             case OPCode::NOP:
216                 ESP_GOTO_ON_FALSE(
217                     false, ESP_FAIL,
218                     label_fail, TAG, "Failed in OPCode::NOP!");
```

```
219                 break;
220             case OPCode::INV:
221                 ESP_GOTO_ON_FALSE(
222                     false, ESP_FAIL,
223                     label_fail, TAG, "Failed in OPCode::INV!");
224                 break;
225             }
226 
227             wait_for_sync = false;
228 
229             ESP_GOTO_ON_FALSE(
230                 comm_ok,
231                 ESP_ERR_NO_MEM, label_fail, TAG, "Communicator fail - no
                    ↪  buffer space!");
232 
233             if (Communicator::should_exit()) [[unlikely]]
234             {
235                 ESP_LOGW(TAG, "Communicator requests to exit!");
236                 goto label_fail;
237             }
238         }
239 
240         WAIT_FOR_SYNC;
241 
242     label_fail:
243         gptimer_stop(sync_timer);
244 
245         port_cleanup();
246 
247         ESP_LOGI(TAG, "Execution took %" PRIu64 "us", get_now());
248         ESP_LOGI(TAG, "Exiting...");
249         Communicator::confirm_exit();
250     }
251     // never ends
252 }
```

Listing A.60: Funkcja `interpreter_task` będąca zadaniem/wątkiem interpretera

A.4 Programy służące do pomiarów

Tutaj przedstawione są dłuższe kody źródłowe programów wykorzystywanych do eksperymentów.

A.4.1 Wyznaczanie charakterystyki częstotliwościowej wejścia

Program ten służy do wykonania pomiarów i wyznaczenia odpowiedzi częstotliwościowej wejścia napięciowego na podstawie sygnałów sinusoidalnych.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.optimize as opt

from IOBlock import IOBlock


def parse_file(fname):
    times = []
    vals = []
    for tpl in IOBlock.read_file_iter(fname, "<fI"):
        times.append(tpl[1] / 1000000)
        vals.append(tpl[0])
    return [np.array(vals), np.array(times)]


iob = IOBlock()

task = """
AIRNG 1 MIN;
AIEN;
DELAY 100000;
RSTTM;
LOOP {iter};
   AOGEN 1 0;
   AIRDF 1 1;
   DELAY 50;
END;
"""


```

```
frqs = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90,
    100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000,
    5000, 6000, 7000, 8000, 9000]

fs = 1000000 / 50  # 1/(50us) = 20kHz
periods = 10

for ft in frqs:
    T = np.floor(1000000 / ft)
    ttltime = min(2, periods * T / 1000000)
    iter = int(ttltime * fs)

    generators = [[{"A": 3, "S": {"WF": "Sine", "T": T}}]]
    ret = iob.settings(task=task.format(iter=iter),
        generators=generators)
    print(ret)
    if ret["status"] != "ok":
        exit()

    iob.write_file("input/datas/" + str(ft) + ".bin", 4)

exit()

Gs = np.full_like(frqs, 0, dtype=float)

for idx, ft in enumerate(frqs):
    vo, t = parse_file("input/datas/" + str(ft) + ".bin")

    t = np.array_split(t, 2)[1]
    vo = np.array_split(vo, 2)[1]

    rmsi = 3.1 / np.sqrt(2)
    rmso = np.sqrt(np.mean(vo**2))

    Gs[idx] = rmso / rmsi

print(Gs)

Gs = 20 * np.log10(Gs)

```

```
69 print(Gs)
70
71
72 # Define the low-pass filter function with insertion loss
73 def lowpass(f, fL):
74     return 20 * np.log10(1 / np.sqrt(1 + (f / fL) ** 2))
75
76
77 # Initial guesses for RC and L
78 initial_guess = (1e3)
79 mins = (1)
80 maxs = (np.inf)
81
82 # Curve fitting
83 params, params_covariance = opt.curve_fit(
84     lowpass,
85     frqs,
86     Gs,
87     p0=initial_guess,
88     bounds=(mins, maxs),
89 )
90 # Fitted RC and L values
91 fLf = params
92
93 print("Fitted fc1:", fLf)
94
95 frqs_fitted = np.logspace(0, 5, 1000, endpoint=False)
96 fitted_curve = lowpass(frqs_fitted, fLf)
97
98 plt.figure()
99
100 plt.title("Odpowiedź częstotliwościowa")
101 plt.plot(frqs, Gs, "o", label="Pomiar")
102 plt.plot(frqs_fitted, fitted_curve, "--", label="Dopasowanie")
103 plt.xlabel("f [Hz]")
104 plt.ylabel("G [dB]")
105 plt.xscale("log")
106 plt.legend()
107 plt.grid()
108
```

```
plt.tight_layout()
plt.show()
```

Listing A.61: Program do wyznaczenia odpowiedzi częstotliwościowej wejścia.

A.4.2 Wyznaczanie charakterystyki wejściowej tranzystora

Program ten służy do wykonania i zapisania pomiarów z urządzenia oraz wygenerowania wykresu charakterystyki wejściowej tranzystora.

```
import matplotlib.pyplot as plt
import numpy as np
from autoscale import *
import os

from IOBlock import IOBlock


def parse_file(fname):
    vals = []
    for tpl in IOBlock.read_file_iter(fname, "<f"):
        vals.append(tpl[0])
    return np.array(vals)


task = """
AIRNG 1 MIN;
AIRNG 4 MAX;
AIEN;
AOVAL 1 {Uce};
DELAY 100000;
RSTTM;
LOOP 1000;
   AOGEN 2 0;
   AIRDF 1 50;
   DELAY 1000;
END;
"""

```

```
generators = [[{"A": 0.1, "S": {"WF": "Triangle", "T": 2000000, "P": 1000000}}]]

# =========
iob = IOBlock()
Uce = "1"
ret = iob.settings(task=task.format(Uce=Uce), generators=generators)
print(ret)
if ret["status"] != "ok":
    exit()
iob.write_file("tran/datas/ibub/inc/" + Uce + ".bin")
exit()

ibs = np.linspace(0, 1, num=1000, endpoint=False)

# Plot the magnitude and phase of the frequency response
plt.figure()
plt.suptitle("Charakterystyka Ube(Ib) dla różnych Uce [V]")

plt.subplot(1, 2, 1)
for entry in sorted(
    os.scandir("tran/datas/ibub/inc/"), key=lambda e: float(e.name.rsplit(".", 1)[0])
):
    vo = parse_file(entry.path)
    plt.plot(ibs, vo, label="Uce=" + entry.name.rsplit(".", 1)[0])

plt.xlabel("Ib [mA]")
plt.ylabel("Ube [V]")
plt.grid()
plt.legend()

plt.subplot(1, 2, 2)
for entry in sorted(
    os.scandir("tran/datas/ibub/dec/"), key=lambda e: float(e.name.rsplit(".", 1)[0])
):
    vo = parse_file(entry.path)
    plt.plot(ibs, vo, label="Uce=" + entry.name.rsplit(".", 1)[0])

```

```
plt.xlabel("Ib [mA]")
plt.ylabel("Ube [V]")
plt.grid()
plt.legend()

plt.tight_layout()
plt.show()
```

Listing A.62: Program do wygenerowania charakterystyki wejściowej tranzystora

A.4.3 Wyznaczanie charakterystyki przejściowej tranzystora

Program ten służy do wykonania i zapisania pomiarów z urządzenia oraz wygenerowania wykresu charakterystyki przejściowej tranzystora.

```
import matplotlib.pyplot as plt
import numpy as np
from autoscale import *
import os

from IOBlock import IOBlock


def parse_file(fname):
    vals = []
    for tpl in IOBlock.read_file_iter(fname, "<f"):
        vals.append(tpl[0])
    return np.array(vals)


task = """
AIRNG 1 MIN;
AIRNG 4 MAX;
AIEN;
AOVAL 1 {Uce};
DELAY 100000;
RSTTM;
LOOP 1000;
   AOGEN 2 0;
```

```
25    AIRDF 4 50;
26    DELAY 1000;
27 END;
28 """
29
30 generators = [[{"A": 0.1, "S": {"WF": "Triangle", "T": 2000000, "P":
   ↪  1000000}}]]
31
32 # =========
33 iob = IOBlock()
34 Uce = "0.5"
35 ret = iob.settings(task=task.format(Uce=Uce), generators=generators)
36 print(ret)
37 if ret["status"] != "ok":
38     exit()
39 iob.write_file("tran/datas/ibic/" + Uce + ".bin")
40 exit()
41
42 ibs = np.linspace(0, 1, num=1000, endpoint=False)
43
44 # Plot the magnitude and phase of the frequency response
45 plt.figure()
46 plt.suptitle("Charakterystyka Ic(Ib) dla różnych Uce [V]")
47
48 for entry in sorted(
49     os.scandir("tran/datas/ibic/"), key=lambda e:
       ↪  float(e.name.rsplit(".", 1)[0])
50 ):
51     vo = parse_file(entry.path)
52     plt.plot(ibs, vo, label="Uce=" + entry.name.rsplit(".", 1)[0])
53
54 plt.xlabel("Ib [mA]")
55 plt.ylabel("Ic [A]")
56 plt.grid()
57 plt.legend()
58
59 plt.tight_layout()
60 plt.show()
```

Listing A.63: Program do wygenerowania charakterystyki przejściowej tranzystora

A.4.4 Wyznaczanie charakterystyki wyjściowej tranzystora

Program ten służy do wykonania i zapisania pomiarów z urządzenia oraz wygenerowania wykresu charakterystyki wyjściowej tranzystora.

```
import matplotlib.pyplot as plt
import numpy as np
from autoscale import *
import os

from IOBlock import IOBlock


def parse_file(fname):
    vals = []
    for tpl in IOBlock.read_file_iter(fname, "<f"):
        vals.append(tpl[0])
    return np.array(vals)


task = """
AIRNG 1 MIN;
AIRNG 4 MAX;
AIEN;
AOVAL 2 {Ib};
DELAY 100000;
RSTTM;
LOOP 1000;
   AOGEN 1 0;
   AIRDF 4 50;
   DELAY 1000;
END;
"""

generators = [[{"A": 10, "S": {"WF": "Triangle", "T": 2000000, "P":
    1000000}}]]

# =========
```

```
iob = IOBlock()
Ib = "0.05"
ret = iob.settings(task=task.format(Ib=Ib), generators=generators)
print(ret)
if ret["status"] != "ok":
    exit()
iob.write_file("tran/datas/uceic/" + Ib + ".bin")
exit()

ucs = np.linspace(0, 10, num=1000, endpoint=False)

# Plot the magnitude and phase of the frequency response
plt.figure()
plt.suptitle("Charakterystyka Ic(Uce) dla różnych Ib [mA]")

for entry in sorted(
    os.scandir("tran/datas/uceic/"), key=lambda e:
    ↪ float(e.name.rsplit(".", 1)[0])
):
    vo = parse_file(entry.path)
    plt.plot(ucs, vo, label="Ib=" + str(float(entry.name.rsplit(".",
    ↪ 1)[0]) * 10))

plt.xlabel("Uce [V]")
plt.ylabel("Ic [A]")
plt.grid()
plt.legend()

plt.show()
```

Listing A.64: Program do wygenerowania charakterystyki wyjściowej tranzystora

A.4.5 Wyznaczanie odpowiedzi częstotliwościowej świergotem

Program ten służy do wykonania pomiarów i wyznaczenia odpowiedzi częstotliwościowej filtra na podstawie sygnału świergotowego.

```
import matplotlib.pyplot as plt
import numpy as np
```

```
from scipy.fftpack import fft
import scipy.optimize as opt
from autoscale import *

from IOBlock import IOBlock


def parse_file(fname, format):
    times = []
    vals = []
    for tpl in IOBlock.read_file_iter(fname, format):
        times.append(tpl[1] / 1000000)
        vals.append(tpl[0])
    return [np.array(vals), np.array(times)]


iob = IOBlock()

task = """
AIRNG 1 MIN;
AIEN;
DELAY 100000;
RSTTM;
LOOP 2000000;
   AOGEN 1 0;
   AIRDF 1 1;
   DELAY 50;
END;
"""

generators = [
    [
        {
            "A": 3,
            "S": {"WF": "ChirpLog", "T": 100000000, "FS": 1 * 1e-6, "FR":
             ↪  10000},
        }
    ]
]

```

```
ret = iob.settings(task=task, generators=generators)
print(ret)
if ret["status"] != "ok":
    exit()

# iob.write_file("filt/dataswv/chirplog/in.bin", 4)
# iob.write_file("filt/dataswv/chirplog/out.bin", 4)
exit()

vi, t = parse_file("filt/dataswv/chirplog/in.bin", "<fI")
vo, _ = parse_file("filt/dataswv/chirplog/out.bin", "<fI")

N = len(t)
fs = 20000  # (N) / (t[-1] - t[1])

print("Fs:", fs)

plt.figure()
plt.plot(t, vi, label="Input")
plt.plot(t, vo, label="Output")
plt.title("Przebieg sygnałów")
plt.xlabel("f [Hz]")
plt.ylabel("U [V]")
plt.show()

# Compute FFT of input and output signals
input_fft = fft(vi)
output_fft = fft(vo)

# Frequency axis
freqs = np.fft.fftfreq(N, 1 / fs)

# Calculate frequency response
frequency_response = output_fft / input_fft

frqs = freqs[: N // 2]
dbs = 20 * np.log10(np.abs(frequency_response)[: N // 2])
phs = np.angle(frequency_response, deg=True)[: N // 2]


```

```
# Define the low-pass filter function with insertion loss
def bandpass(f, fL, fH, L):
    return (
        -L
        + 20 * np.log10(1 / np.sqrt(1 + (f / fL) ** 2))
        + 20 * np.log10((f / fH) / np.sqrt(1 + (f / fH) ** 2))
    )


# Initial guesses for RC and L
initial_guess = (1e1, 1e3, -10)
mins = (1, 1, -np.inf)
maxs = (np.inf, np.inf, 0)

fftkeys = np.nonzero((frqs >= 1) & (frqs <= 1000))
fftfrs = frqs[fftkeys]
fftdbs = dbs[fftkeys]

# Curve fitting
params, params_covariance = opt.curve_fit(
    bandpass,
    fftfrs,
    fftdbs,
    p0=initial_guess,
    bounds=(mins, maxs),
    sigma=np.log10(fftfrs + 1),
)
# Fitted RC and L values
fLf, fHf, Lf = params

print("Fitted fc1:", fLf)
print("Fitted fc2:", fHf)
print("Fitted insertion loss (dB):", Lf)

fitted_curve = bandpass(frqs, fLf, fHf, Lf)

# Plot the magnitude and phase of the frequency response
plt.figure()

plt.subplot(2, 1, 1)
```

```
122 plt.title("Odpowiedź częstotliwościowa")
123 plt.plot(frqs, dbs)
124 plt.plot(frqs, fitted_curve)
125 plt.xlabel("f [Hz]")
126 plt.ylabel("G [dB]")
127 plt.xscale("log")
128 plt.xlim(1, 5000)
129 autoscale()
130 plt.grid()
131
132 plt.subplot(2, 1, 2)
133 plt.plot(frqs, phs)
134 plt.xlabel("f [Hz]")
135 plt.ylabel("φ [°]")
136 plt.xscale("log")
137 plt.xlim(1, 5000)
138 autoscale()
139 plt.gca().set_yticks(np.arange(-90, 90 + 1, 45))
140 plt.grid()
141
142 plt.tight_layout()
143 plt.show()
```

Listing A.65: Program do wyznaczenia odpowiedzi częstotliwościowej filtra. Sposób 1.

A.4.6 Wyznaczanie odpowiedzi częstotliwościowej sinusoidami

Program ten służy do wykonania pomiarów i wyznaczenia odpowiedzi częstotliwościowej filtra na podstawie odpowiedzi na sygnały sinusoidalne.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy.fftpack import fft
4 import scipy.optimize as opt
5 import os
6 from autoscale import *
7
8 from IOBlock import IOBlock
9
```

```
10
11 def parse_file(fname):
12     times = []
13     vals = []
14     for tpl in IOBlock.read_file_iter(fname, "<fI"):
15         times.append(tpl[1] / 1000000)
16         vals.append(tpl[0])
17     return [np.array(vals), np.array(times)]
18
19
20 iob = IOBlock()
21
22 task = """
23 AIRNG 1 MIN;
24 AIEN;
25 DELAY 100000;
26 RSTTM;
27 LOOP {iter};
28    AOGEN 1 0;
29    AIRDF 1 1;
30    DELAY 50;
31 END;
32 """
33
34
35 frqs = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90,
   ↪  100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000,
   ↪  5000, 6000]
36
37 fs = 1000000 / 50  # 1/(50us) = 20kHz
38 periods = 10
39
40 for ft in frqs:
41     T = np.floor(1000000 / ft)
42     ttltime = min(2, periods * T / 1000000)
43     iter = int(ttltime * fs)
44
45     generators = [[{"A": 3, "S": {"WF": "Sine", "T": T}}]]
46     ret = iob.settings(task=task.format(iter=iter),
       ↪  generators=generators)
```

```
47      print(ret)
48      if ret["status"] != "ok":
49          exit()
50
51      # iob.write_file("filt/datassine/in/" + str(ft) + ".bin", 4)
52      # iob.write_file("filt/datassine/out/" + str(ft) + ".bin", 4)
53
54  exit()
55
56  Gs = np.full_like(frqs, 0, dtype=float)
57  Ps = np.full_like(frqs, 0, dtype=float)
58
59  for idx, ft in enumerate(frqs):
60      vi, t = parse_file("filt/datassine/in/" + str(ft) + ".bin")
61      vo, _ = parse_file("filt/datassine/out/" + str(ft) + ".bin")
62
63      t = np.array_split(t, 2)[1]
64      vi = np.array_split(vi, 2)[1]
65      vo = np.array_split(vo, 2)[1]
66
67      vi -= np.mean(vi)
68      vo -= np.mean(vo)
69
70      rmsi = np.sqrt(np.mean(vi**2))
71      rmso = np.sqrt(np.mean(vo**2))
72
73      Gs[idx] = rmso / rmsi
74
75      correlation = np.correlate(vi, vo, mode="full")
76      lag = np.argmax(correlation) - (len(vi) - 1)
77
78      phsh = ((lag / fs * ft) % 1) * 360
79      if phsh > 180:
80          phsh -= 360
81
82      Ps[idx] = phsh
83
84
85  print(Gs)
86  print(Ps)
```

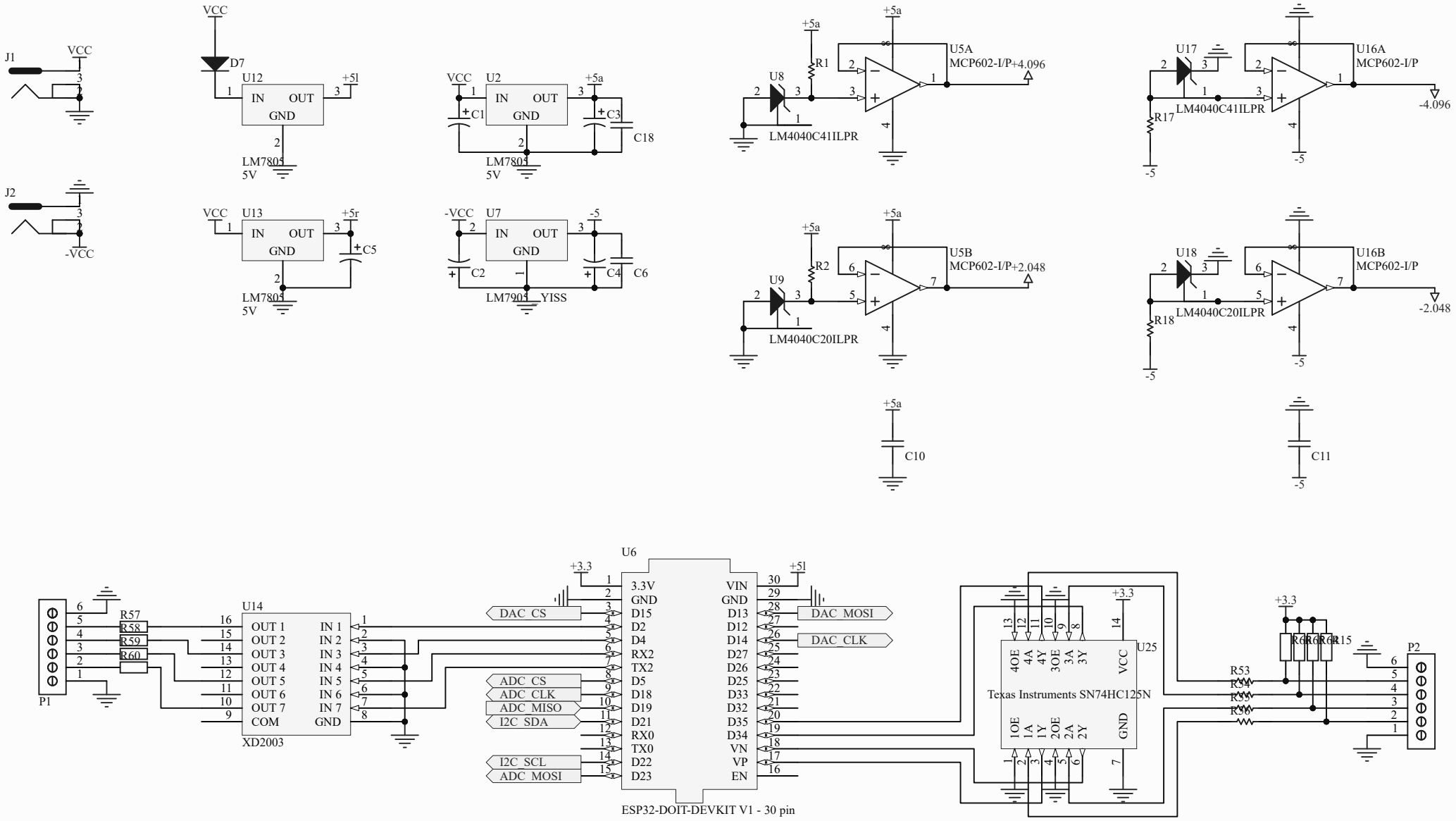
```
87
88 plt.figure()
89
90 plt.subplot(2, 1, 1)
91 plt.title("Odpowiedź częstotliwościowa")
92 plt.plot(frqs, 20*np.log10(Gs))
93 # plt.plot(frqs, fitted_curve)
94 plt.xlabel("f [Hz]")
95 plt.ylabel("G [dB]")
96 plt.xscale("log")
97 # plt.yscale("log")
98 # plt.xlim(1, 5000)
99 # autoscale()
100 plt.grid()
101
102 plt.subplot(2, 1, 2)
103 plt.plot(frqs, Ps)
104 plt.xlabel("f [Hz]")
105 plt.ylabel("φ [°]")
106 plt.xscale("log")
107 # plt.xlim(1, 5000)
108 # autoscale()
109 plt.gca().set_yticks(np.arange(-90, 90 + 1, 45))
110 plt.grid()
111
112 plt.tight_layout()
113 plt.show()
```

Listing A.66: Program do wyznaczenia odpowiedzi częstotliwościowej filtra. Sposób 2.

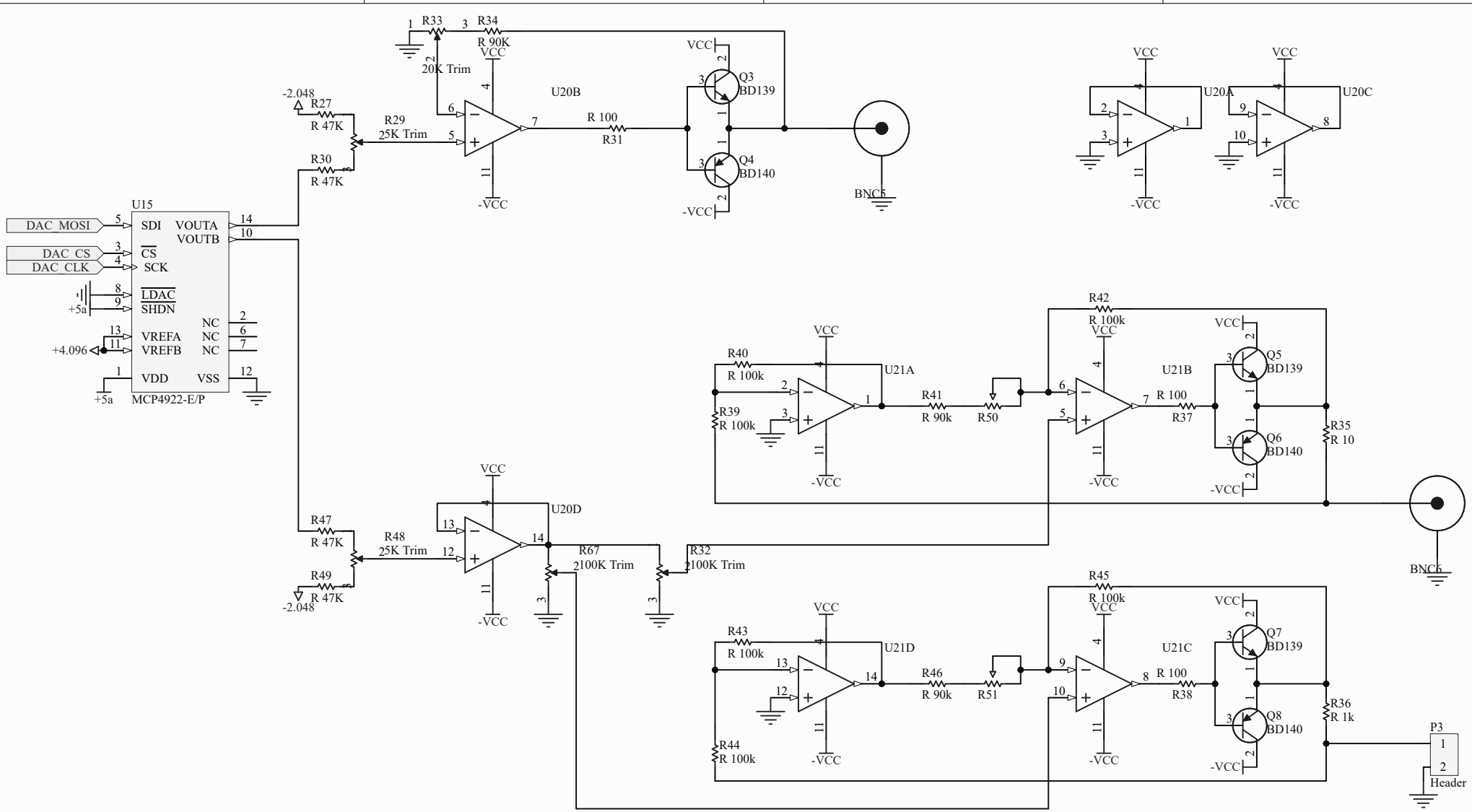
Dodatek B

Schemat ideowy całego bloku wejść-wyjść analogowych i cyfrowych

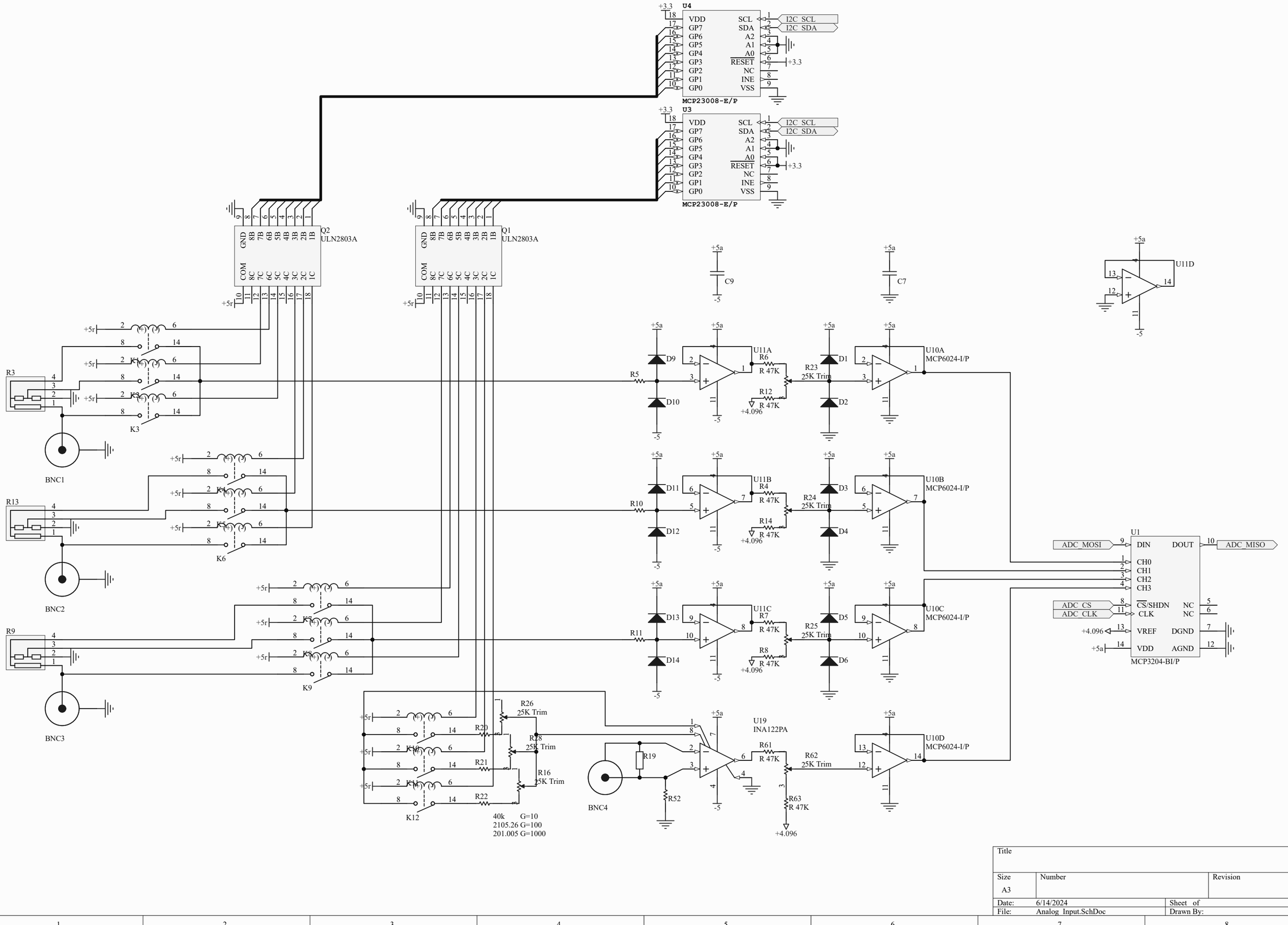
Poniżej przedstawione są schematy ideowe całego bloku wejść-wyjść analogowych i cyfrowych. Pierwszy schemat zawiera mikrokontroler, sekcję zasilania oraz porty cyfrowe. Drugi schemat zawiera wyjścia analogowe. Trzeci schemat przedstawia wejścia analogowe.



Title		
Size: A4	Number	Revision
Date: 6/14/2024		Sheet of
File: Power_Digital.SchDoc		Drawn By:



Title		
Size: A4	Number	Revision
Date: 6/14/2024		Sheet of
File: Analog_Output.SchDoc		Drawn By:



Title			
Size A3	Number		Revision
Date:	6/14/2024	Sheet of	
File:	Analog Input.SchDoc	Drawn By:	

Dodatek C

Spis skrótów i symboli

AC *alternating current*, prąd przemienny

DC *direct current*, prąd stały

A/C analogowo-cyfrowy

C/A cyfrowo-analogowy

A/D *analog/digital*, analogowo-cyfrowy

D/A *digital/analog*, cyfrowo-analogowy

ADC *Analog-Digital Converter*, przetwornik `A/C`

DAC *Digital-Analog Converter*, przetwornik `C/A`

PWM *Pulse-Width Modulation*, modulacja szerokości impulsów

PDM *Pulse-Density Modulation*, modulacja gęstości impulsów

INL *integral nonlinearity*, nieliniowość całkowa

DNL *differential nonlinearity*, nieliniowość różniczkowa

MMIO *memory-mapped input/output*, sposób ułatwienia obsługi dostępu do i wykonywania operacji na urządzeniach wejścia/wyjścia w systemach komputerowych

UART *universal asynchronous receiver-transmitter*, uniwersalny asynchroniczny nadajnik-odbiornik

I2C *Inter-Integrated Circuit*, szeregowa, dwukierunkowa magistrala służąca do przesyłania danych

SPI *Serial Peripheral Interface*, szeregowy interfejs urządzeń peryferyjnych

SQNR *signal-to-quantization-noise ratio*, stosunek mocy sygnału do zakłócenia kwantyzacji

DR *Dynamic Range*, rozpiętość tonalna, możliwa różnicaa między minimalną a maksymalną wartością

Spis rysunków

Spis kodów