POLITECHNIKA ŚLĄSKA
WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI

Projekt z przedmiotu Systemy Wbudowane i Systemy Czasu Rzeczywistego

Line follower

Autorzy:
inż. Maksymilian Skibiński
inż. Paweł Kaźmieruk
rok 2020/2021, sem. letni, gr. Rob, sekcja nr 1

Kierujący pracą:
dr inż. Krzysztof Jaskot

Gliwice, lipiec 2021

Spis treści

1 Wstęp

Celem projektu była budowa i zaprogramowanie mobilnego robota poruszającego się wzdłuż linii, tzw. Line followera. W dużym skrócie, taki pojazd jest wyposażony w odpowiednie czujniki, które informują mikrokontroler o jego położeniu w stosunku do linii, a ten korzystając z tej wiedzy generuje odpowiednie sygnały modyfikujące prędkości jego kół. W ten sposób robot potrafi podążać za swoją trasą.

Sposobów na realizację tego zadania jest wiele i zależy to już w głównej mierze od autorów projektu. Treścią tego raportu jest omówienie jednego z takich pomysłów, który został zrealizowany przez autorów.

Jako etapy realizacji projektu można wymienić:

- projekt części elektronicznej projektu: wybór czujników, mikrokontrolera, wybór silników oraz kół, okablowanie,
- projekt podwozia wraz z instalacją całego osprzętu,
- propozycja systemu sterowania i jego implementacja w systemie wbudowanym oraz systemie czasu rzeczywistego,
- wnioski końcowe, pomysły na przyszłość.

2 Harmonogram

Zajęcia odbywały się w trzecim bloku zajęć w semestrze. Składało się na nie 5 cotygodniowych wirtualnych spotkań, podczas których pokazywane były postępy w pracy. Projekt był wykonywany pomiędzy tymi spotkaniami.

Tydzień 1

Na pierwsze spotkanie przygotowany został robot, który potrafił się poruszać, ale wykonywał tylko zaprogramowane ruchy. Na tym etapie nie posiadał żadnej autonomii i nie potrafił jeździć wzdłuż trasy.

Tydzień 2

Z powodów godzin dziekańskich te zajęcia się nie odbyły.

Tydzień 3

Tym razem, robot potrafił już jeździć wzdłuż linii. Zainstalowane zostały dwa analogowe sensory na przodzie pojazdu informujące o jego położeniu względem linii oraz zaimplementowane zostało prawo sterowania, które korzystając z różnicy tych wskazań generowało odpowiedni (proporcjonalny do niej) sygnał sterujący.

Robot poruszał się płynnie i z zadowalającą prędkością wzdłuż trasy, potrafił pokonywać mniej lub bardziej ostre zakręty oraz nie było dla niego problematyczne skrzyżowanie na środku trasy, ale nie potrafił wykonać skrętu o 90 stopni. Dwa czujniki okazały się do tego niewystarczające (choć być może zdolniejsi nawet z nich umieją zrealizować to zadanie).

Na tym etapie powstał też całkiem ciekawy pomysł zainspirowany naturą – a dokładniej rybami głębinowymi – kupna jeszcze jednego czujnika analogowego, który celowo miał wystawać przed pojazd i informować o przyszłych zmianach na trasie. Ostatecznie jednak, ze względu na inne bardziej istotne problemy, ten pomysł nie został zrealizowany.

Materiały prezentujące działanie robota na tym etapie:

- https://youtu.be/KM7RJapF71Y
- https://youtu.be/R2dgWDf-asg

Tydzień 4

Dołożone zostały dwa kolejne czujniki, tym razem cyfrowe, na przód pojazdu, za pomocą których robot był w stanie wykonywać skręty o 90 stopni. Pewnej zmianie musiało ulec prawo sterowania (które dokładniej zostanie omówione później), dodany został człon D do regulacji (czyli na

tym etapie działało równanie PD), ale co najważniejsze dodanie kolejnych czujników nie wpłynęło negatywnie na płynny ruch robota po trasie, a tylko pomogło przy ostrych skrętach.

Na tym etapie, pojazd już spełniał swój cel – potrafił podążać za linią.

Materiały prezentujące działanie robota na tym etapie:

- https://youtu.be/qdXwQmjnL98
- https://youtu.be/QG8gyqzkxBA

Tydzień 5

Ostatni tydzień poświęcony został głównie na zapoznanie się z systemem czasu rzeczywistego FreeRTOS oraz na implementację układu sterowania w tym systemie. Jak ostatnie zajęcia pokazały – cel został zrealizowany.

Pewnym zmianom uległy także zmiany nastaw regulatora PD oraz działanie trybu nawracania – robot nie robi teraz obrotu o 90 stopni w miejscu, tylko skręca oraz wraca do tyłu, by nadrobić to, że drobno wyjechał poza trasę.

Materiały prezentujące działanie robota pod koniec zajęć projektowych:

- https://youtu.be/4WTrk285v4s
- https://youtu.be/nwpjGGdDjxc

3 Elementy

3.1 Kosztorys

By projekt mógł zostać wykonany, konieczne było kupno wielu części. Te zostały spisane w podanej poniżej tabeli nr 1.

FORBOT – zestaw do budowy robota	**270 zł**
Arduino Uno Rev3	**95 zł**
Zestaw przewodów połączeniowych – żeńsko-żeńskie	**6 zł**
Czujnik odległości, odbiciowy 3,3V/5V - Iduino ST1140	2 x **7,70 zł**
Bateria AA (R6 LR06) alkaliczna Energizer Alkaine Power - 4szt.	2 x **8,50 zł**
Suma	**403,40 zł**

Tabela 1: Kosztorys

Robot był budowany według kursu dostępnego w internecie na stronie Forbota:

https://forbot.pl/blog/kurs-budowy-robotow-arduino-wstep-spis-tresci-id18935

Większość z wykorzystanych i kluczowych dla projektu części jest zawarta w zestawie pochodzącym z tego kursu. Warto także dodać, że w zestawie znajdują się także inne, nieistotne dla pracy elementy, więc teoretycznie koszt zestawu dla opisanego w tej pracy robota powinien być mniejszy, ale jako że nie wszystkie z tych części możemy kupić osobno podany został koszt całego zestawu.

Cała praca była wykonywana przez obu członków sekcji, zatem każdy z nich poniósł własne koszta i budował własny pojazd. W ten sposób łatwiej było wymieniać się pomysłami, uwagami i wspólnie pracować nad projektem.

Obaj członkowie sekcji posiadali już swoje Arduino Uno i wykorzystywali je w przeszłości, ale jako, że jest one niezbędne do działania projektu, zostało ono także wpisane do listy kosztów.

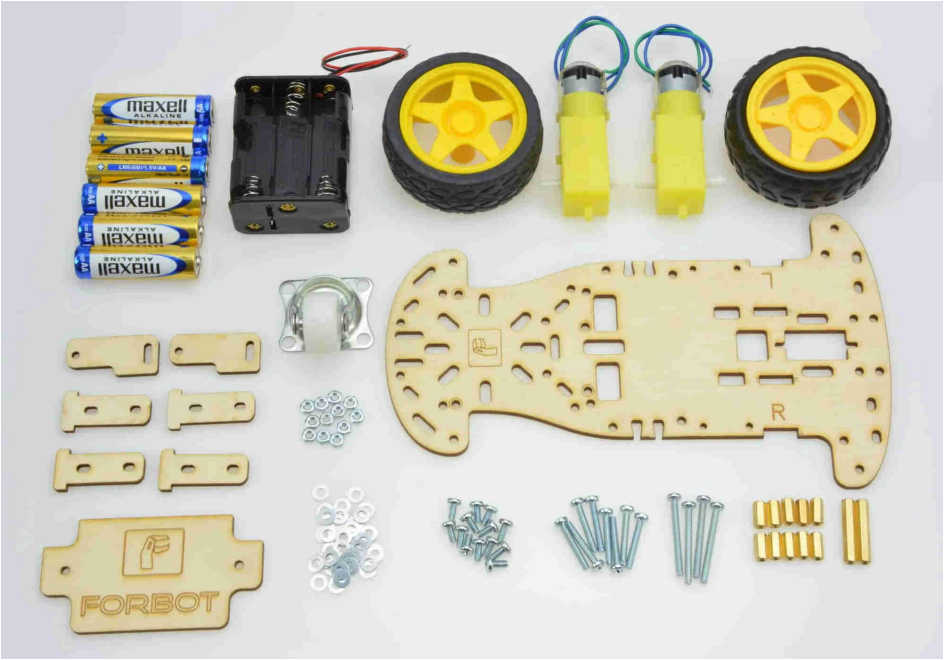
Wszystkie zakupy zostały zrobione w internetowym sklepie Botland.

3.2 Opis części

Na rysunku nr 1 została pokazana część elementów wchodzących w skład zakupionego zestawu (pozycja nr 1 w kosztorysie).

Podwozie

Drewniane elementy widoczne na rysunku nr 1 wchodzą w skład podwozia. Są to lekkie części wykonane ze sklejki, zawierające dużą liczbę otworów o różnej wielkości, co pozwala na odpowiednie przygotowanie robota do realizacji celu. 4 z mniejszych części na lewej stronie służą



Rysunek 1: Część elementów wchodzących w skład zestawu
Źródło: kurs Forbota

do montażu silników, a większy kawałek w dolnym lewym rogu służy do montażu koszyka z bateriami.

Największy element, który stanowi podwozie ma wymiary:

- długość: 19 cm,
- szerokość przodu: 12 cm,
- szerokość tyłu: 9 cm.

Podwozie – montaż

By przymocować części takie jak np. silniki, Arduino do podwozia wykorzystany został zestaw różnych śrubek, podkładek i tulei dystansowych:

- śrubki 30mm, 20mm i 6mm,
- tuleje dystansowe 25mm i 10mm,
- podkładki, podkładki sprężyste i nakrętki.

Koło swobodne

W skład pojazdu wchodzą w sumie 3 koła. Jednym z nich jest mniejsze koło, które zamontowane zostało z tyłu pojazdu i jest przez niego ciągnięte – takie kółko może się obracać w dowolnym kierunku. Koło działa na zasadzie podobnej do tej znanej z wózków sklepowych.

Koła i silniki

Pozostałymi dwoma kołami są większe, gumowe koła, które są napędzane przez silniki. Te które wchodzą w skład zestawu (i które zostały użyte w omawianej pracy) można kupić oddzielnie tutaj za 15 zł (koło + silnik). Użyte elementy są produkowane przez OEM.

Parametry koła:

- średnica opony: 65 mm,
- szerokość opony: 26 mm.

Parametry silnika z przekładnią:

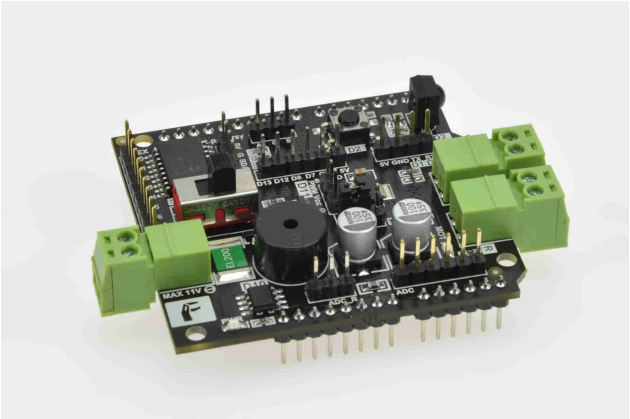
- napięcie zasilania: 5 V,
- pobór prądu: ok. 180 mA,
- przekładnia 48:1,
- prędkość obrotowa po przekładni: ok. 80 $\frac{\text{obr}}{\text{min}}$.

W pracy nie został wykorzystany enkoder.
Użyte elementy zostały pokazane na rysunku nr 2 poniżej.

Rysunek 2: Zestaw silnika z kołem
Źródło: sklep Botland

Nakładka do Arduino

W projekcie została użyta nakładka (inaczej *shield*) do Arduino, która przygotowuje je do pracy z elementami wchodzącymi w skład robota. Została ona pokazana na rysunku nr 3.



Rysunek 3: Nakładka
Źródło: kurs Forbota

Jak sama nazwa wskazuje, nakładkę „nakłada" się na Arduino, w ten sposób wszystkie połączenia z czujnikami czy silnikami wykonuje się poprzez piny na shieldzie.

Nakładka zawiera:

- mostek-h DRV8835,
- bezpiecznik polimerowy wielokrotnego użycia,
- włącznik zasilania,
- kondensatory ceramiczne do filtracji zakłóceń,
- zworka zasilania silników (w ten sposób można łączyć Arduino z komputerem nie uruchamiając silników),
- złącza dla czujników analogowych i cyfrowych,
- złącza zasilania.

Czujniki

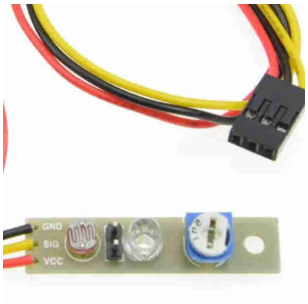
Do wykrywania linii wykorzystane zostały 4 czujniki (widoczne na rysunku nr 4 powyżej):

- 2 czujniki analogowe (fotorezystor + dioda LED),
- 2 czujniki cyfrowe odbiciowe.

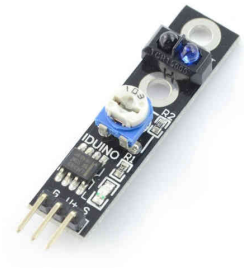
Oba typy czujników zawierają potencjometry: w przypadku sensorów analogowych regulujemy w ten sposób jasność diody, a w przypadku sensorów cyfrowych ustalamy wartość progowa dla której czujnik zwróci wartość logicznie prawdziwą.

Arduino

Płytka Arduino Uno (widoczna na rysunku nr 5 jest sercem robota – na podstawie napisanego kodu zajmuje się sterowaniem pojazdem. Wyposażona jest w:



(a) analogowy



(b) cyfrowy

Rysunek 4: Czujniki
Źródła: kurs Forbota, sklep Botland

- mikrokontroler AVR Tmega328,
- pamięć Flash i RAM,
- wejścia/wyjścia cyfrowe,
- wejścia analogowe,
- kanały PWM,

Rysunek 5: Płyta Arduino Uno
Źródło: sklep Botland

W omawianym projekcie na Arduino nałożony jest shield, przez który łączone są elementy robota. Jedyne połączenie jakie wykonywane było bezpośrednio przez Arduino dotyczyło połączenia go przez przewód USB z komputerem w celu wgrania kodu.

Baterie

Pojazd zasilany jest poprzez 6 połączonych szeregowo baterii AA. Takie połączenie daje zasilanie 9 V, które wystarcza do zasilenia elementów robota. Nie można zastosować w tym celu jednej baterii 9 V, gdyż takie nie nadają się do zasilania układów o większym zapotrzebowaniu na prąd.

Baterie umieszczone są w specjalnym koszyku, który położony jest na środku pojazdu, a ten zaś połączony jest z nakładką na Arduino.

Jeden komplet baterii wchodził w skład zestawu, ale w trakcie realizacji projektu rozładowowały się i należało kupić kolejne.

Przewody

Czujniki i piny na nakładce do Arduino należało połączyć korzystając z przewodów żeńsko-żeńskich.

4 Urządzenie wraz z aplikacją

Na wykonanie projektu składają się właściwie dwie części:

– budowa robota – łączenie elementów, przykręcanie ich do podwozia,

– napisanie kodu sterującego robotem.

4.1 Montaż

Montaż został rozpoczęty od przykręcenia kilku śrubek z podkładkami zwykłymi, sprężystymi i tulejami dystansowymi do podwozia (rys. nr 6a, 6b). W następnych krokach te połączenia zostały wykorzystane do montażu innych części.

Następnie, na przód platformy zamontowane zostały silniki. W tym celu zostały wykorzystane dłuższe śrubki (30 mm), nakrętki oraz 4 małe części ze sklejki (rys. nr 6c).

Z tyłu platformy, na środku, zamontowane zostało swobodne koło, które jest ciągnięte przez pojazd (rys. nr 6d). Większe koła z gumowymi oponami, które napędzane są przez silniki, zostały następnie nałożone na osie silników (rys. nr 6e).

Do koszyka zostały włożonych 6 baterii AA, a następnie koszyk został umieszczony na środku platformy (rys. nr 6f). Tak położony koszyk oczywiście nie ma prawa się utrzymywać, gdy pojazd się porusza, dlatego został on przykręcony przez element ze sklejki (rys. nr 6g).

Na przód podwozia, korzystając z 3 tulei dystansowych – zamontowanych w pierwszym kroku – przykręcona została płytka Arduino Uno (rys. nr 6h), a na nią nałożona została nakładka (rys. nr 6i). Do nakładki podłączone zostały przewody zasilające silniki, oraz przewód pochodzący od koszyka z bateriami.

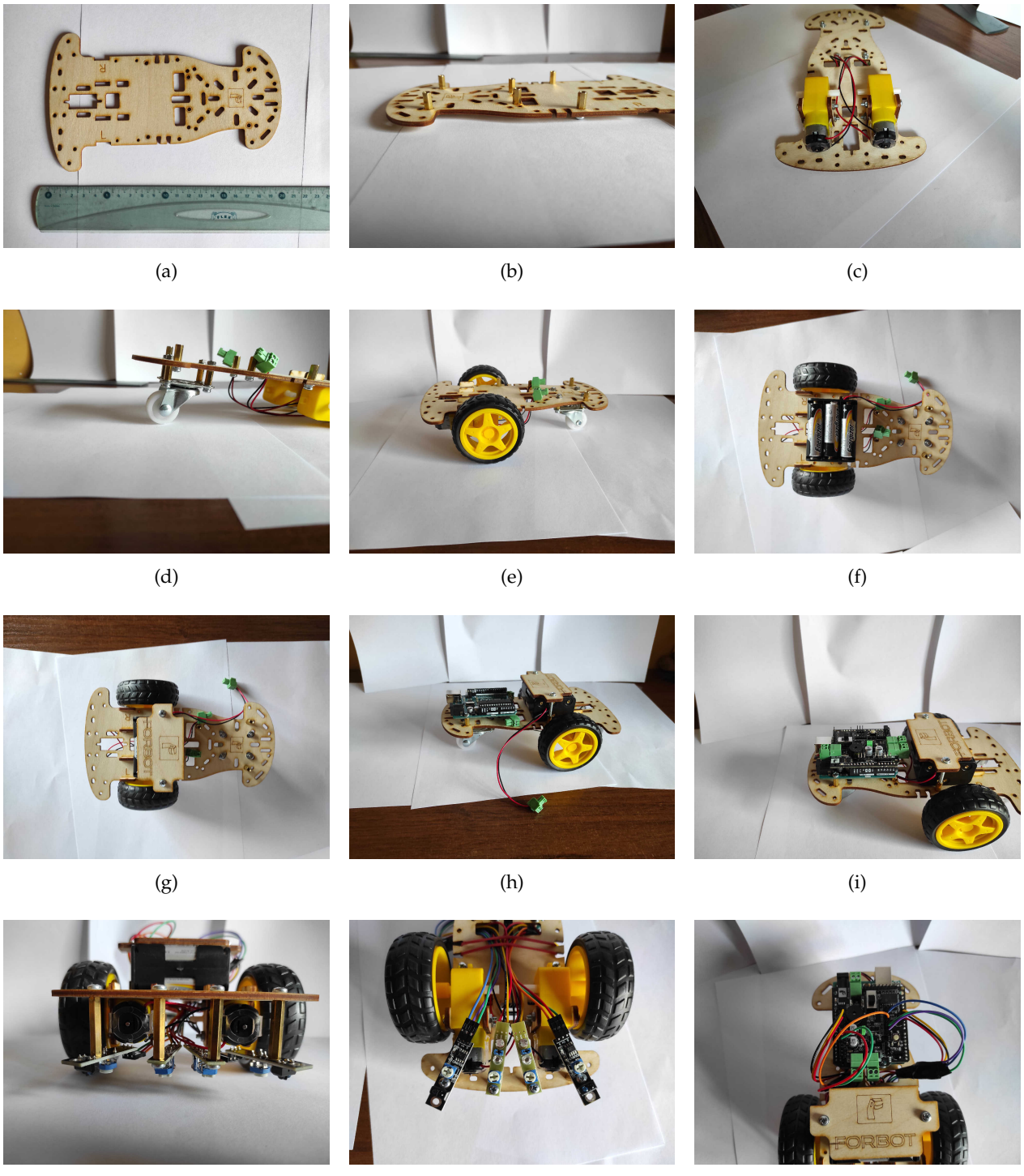
Na tym etapie, pojazd jest w stanie poruszać się w sposób nieautonomiczny, tzn. może wykonywać zaprogramowane ruchy. Oczywiście, w tym celu należy wpierw podłączyć Arduino z komputerem i wgrać odpowiedni kod.

Na samym końcu, zamontowane zostały czujniki. Sensory znajdują się na samym przodzie platformy, a przymocowanie ich za pomocą dłuższych (25 mm) tulei dystansowych powoduje, że są one umieszczone tuż nad ziemią, co pozwala im dobrze śledzić linię (rys. nr 6j). Czujniki analogowe i cyfrowe zostały rozmieszczone w trochę inny sposób (rys. nr 6k), co wynika z faktu, że pełnią trochę inne zadania:

– czujniki analogowe znajdują się blisko siebie, zostały ustawione pod pewnym kątem, tak by były w stanie objąć (oba naraz) linię na podłodze,

– czujniki cyfrowe znajdują się bardziej po bokach, na zewnętrznych stronach platformy, a dodatkowo są umieszczone dalej, bardziej do przodu niż sensory analogowe.

Ze względu na fakt, że pod spodem pojazdu znajduje się wiele przewodów zostały one objęte przez gumkę recepturkę (rys. nr 6k). Połączenie przewodów z Arduino (a właściwie z nakładką) widać na ostatnim zdjęciu 6l.



(a) (b) (c)

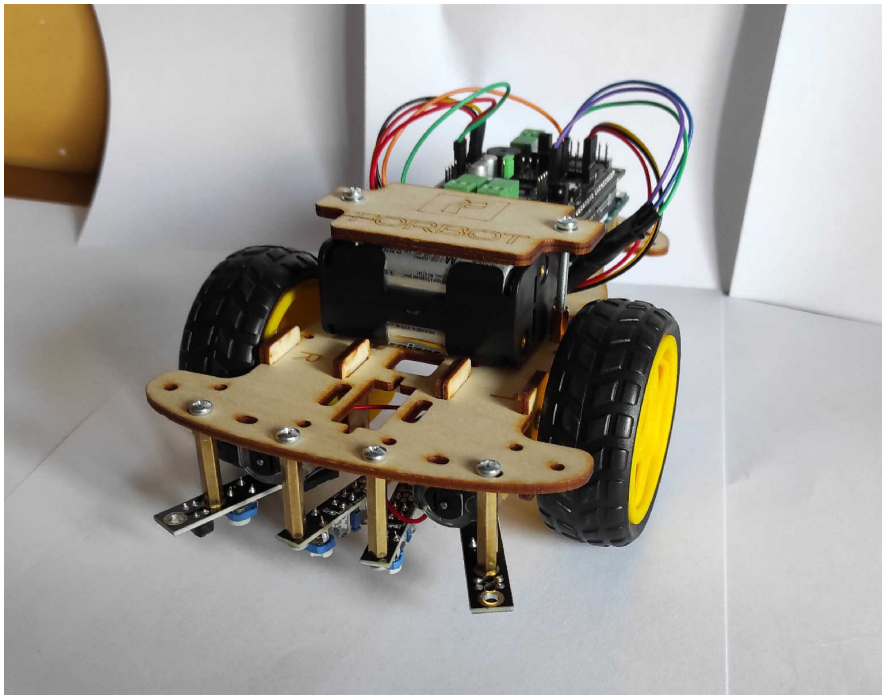
(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Rysunek 6: Montaż

Gotowy pojazd został przedstawiony na rysunku nr 7 poniżej.

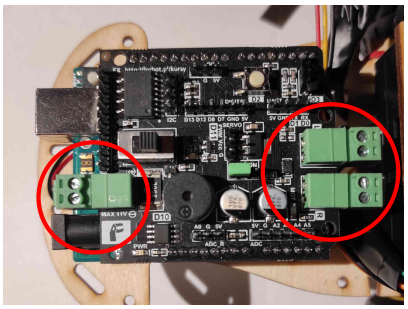


Rysunek 7: Gotowa konstrukcja robota

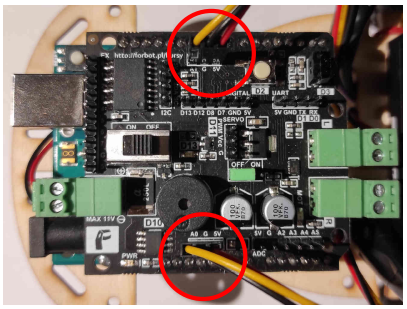
4.2 Okablowanie

Następnie połączono odpowiednie elementy z odpowiednimi pinami znajdującymi się na shieldzie Arduino (rys. nr 8). Połączenia są następujące:

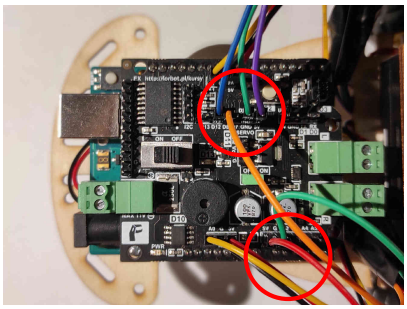
- Gniazda związane z zasilaniem silników oraz koszykiem z bateriami (rys. nr 8a), do tych gniazd doprowadzane są po dwa przewody. W przypadku silników (gniazda po prawej), ich kolejność nie ma znaczenia – na etapie pisania programu będzie można łatwo „naprawić" połączenie przewodów na odwrót. W przypadku koszyka z bateriami (gniazdo po lewej) polaryzacja jest ważna. W przypadku przypadkowego niepoprawnego połączenia przed spaleniem elektroniki powinien uratować nas bezpiecznik.
- Każdy z czujników analogowych (rys. nr 8b) ma 3 przewody: zasilania, masy i sygnału. Na shieldzie znajdują się specjalnie przygotowane do obsługi tych czujników miejsca (na górze i na dole). Poza tym, na płytce znajdują się inne piny do obsługi sygnałów analogowych.
- Podobnie jak w przypadku czujników analogowych, sensory cyfrowe (rys. nr 8c) także mają 3 takie same przewody. Różnica polega jednak na tym, że przewody „sygnałowe" trafiają na piny cyfrowe, a nie analogowe. Sygnał cyfrowy jest bardziej ubogi od analogowego, bo daje tylko dwie możliwe wartości: logicznie prawdziwa lub fałszywa, ale ich obsługa przebiega znacznie szybciej, bo konwertowanie sygnału analogowego na liczby zabiera dużo czasu.



(a) silniki, baterie



(b) czujniki analogowe



(c) czujniki cyfrowe

Rysunek 8: Połączenia

Przewody, które przenoszą informacje, zostają wykorzystane na etapie tworzenia aplikacji. Wykorzystane piny to:

- czujnik analogowy lewy: A1, czujnik analogowy prawy: A0,
- czujnik cyfrowy lewy: 12, czujnik cyfrowy prawy: 8.

4.3 Aplikacja

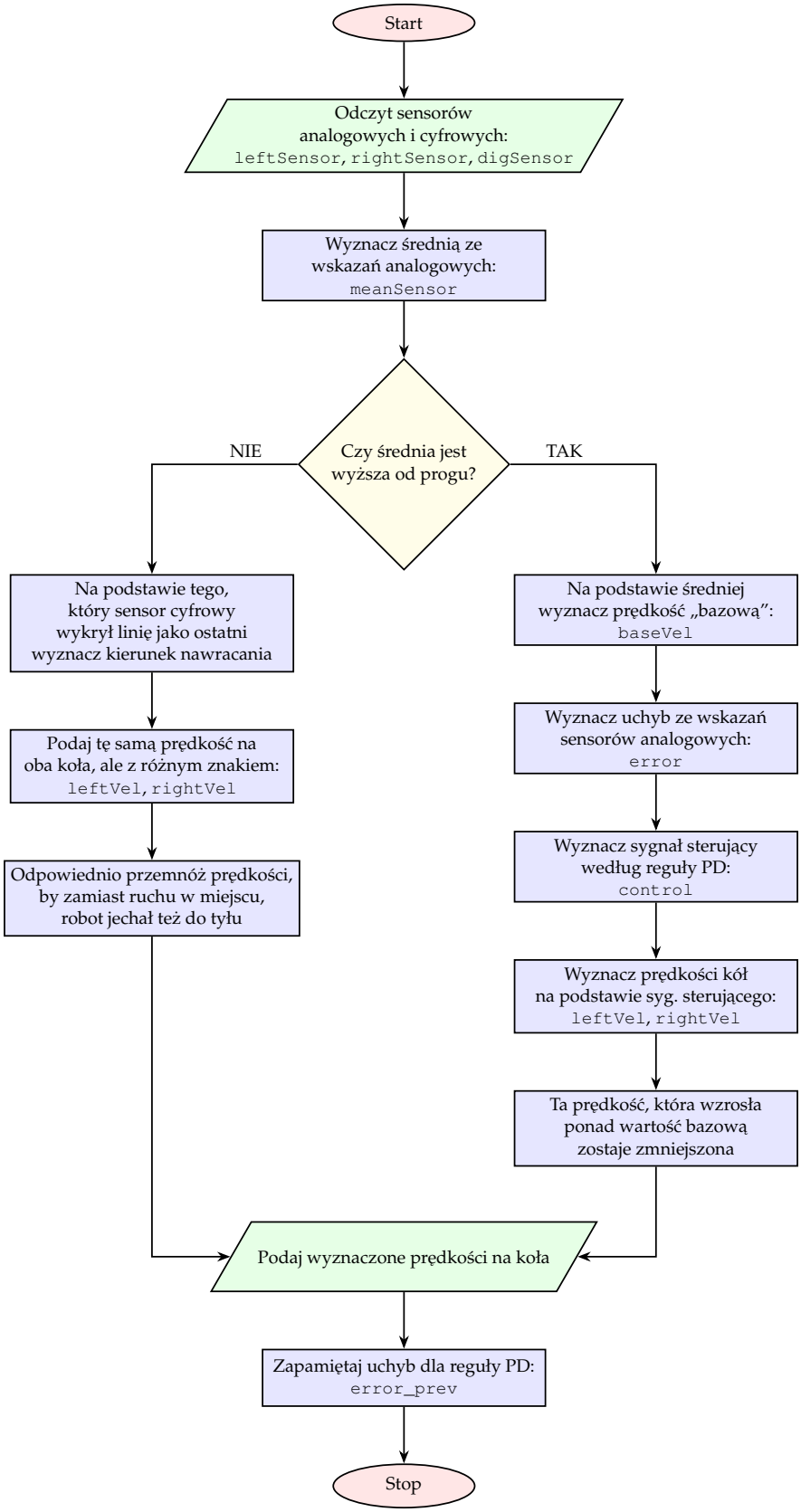
W tym punkcie omówiony zostanie sposób działania systemu sterowania. Podane zostaną listingi fragmentów kodu odpowiedzialnego za realizację sterowania. Kompletny kod został umieszczony na końcu raportu jako dodatek A. Kod ten jest bogaty w komentarze, ale w tym punkcie zostaną one usunięte jako że kluczowe części kodu zostaną osobno omówione, więc nie potrzeby dwukrotnie podawać tych samych informacji.

Algorytm odpowiadający za sterowanie pojazdem został pokazany na rysunku nr 9.

1. System sterowania rozpoczyna od odczytania informacji z czujników analogowych i cyfrowych. Te informacje zostają przypisane do odpowiednich zmiennych. Operacja wejścia/wyjścia są operacjami, które zajmują najwięcej czasu, dlatego w przypadku, w konkretnej iteracji algorytmu odczyt czujników musi występować tylko jeden raz. Jeśli kod kilkukrotnie korzysta z informacji sensorycznych, to odczytuje zmienne, a nie czyta na nowo wskazań czujników.

   W przypadku czujników analogowych zapamiętywane są oba wskazania. W przypadku czujników cyfrowych zapamiętywana jest tylko informacja o tym, który sensor wskazał linię jako ostatni.

```
78 leftSensor = analogRead(SENSOR_ANALOG_LEFT);
79 rightSensor = analogRead(SENSOR_ANALOG_RIGHT);
80
81 digSensor = digitalRead(SENSOR_DIGITAL_LEFT) ? SENSOR_DIGITAL_LEFT : digSensor;
82 digSensor = digitalRead(SENSOR_DIGITAL_RIGHT) ? SENSOR_DIGITAL_RIGHT : digSensor;
```



Rysunek 9: Algorytm sterowania

2. Ze wskazań czujników analogowych wyznaczana jest średnia arytmetyczna. Dzięki tej informacji system sterowania jest w stanie stwierdzić czy robot znajduje się na czarnej linii, czy na podłodze.

```
86 meanSensor = (rightSensor + leftSensor) / 2.0;
```

3. Następnie średnia ta jest porównywana względem pewnej wartości progowej. Ta informacja jest bezpośrednio wykorzystywana do stwierdzenia czy robot ma podążać wzdłuż linii według reguły PD, czy powinien nawracać.

```
91 if (meanSensor > TURN_MEAN) {
```

Podążanie za linią

Jeśli średnia jest wyższa od progu robot stara się podążać wzdłuż linii, wtedy:

4. Średnia arytmetyczna ze wskazań czujników analogowych wykorzystywana jest do wyznaczenia pewnej prędkości „bazowej" `baseVel`. Na tym etapie, ta prędkość bazowa może być tylko pomniejszona. Za pomocą `#define` w kodzie deklarujemy wartość stałą `BASE_VEL`, która jest pożądaną prędkością bazową. Jeśli wskazania czujników analogowych są niższe od pewnej wartości progowej, prędkość bazowa jest liniowo zmniejszana. Chodzi o to, że jeśli robot wyjechał zbyt mocno – ale cały czas będąc na niej – poza linię, to jego pożądana prędkość jest zmniejszana, by zapobiec całkowitemu wyjechaniu poza nią, i by „upłynnić" jego skręty.

```
94 baseVel = meanSensor <= DOLNA_GRANICA ?
95     map(meanSensor, PODLOGA_MEAN, DOLNA_GRANICA, PODLOGA_VEL, BASE_VEL) :
96     BASE_VEL;
```

5. Ze wskazań czujników analogowych wyznaczany jest uchyb. Następnie uchyb ten jest liniowo przekształcany na zmiany prędkości. Dobrane zostały tutaj wartości 150 jako maksymalne możliwe różnice pomiędzy wskazaniami sensorów analogowych, które są przekształcane na maksymalne możliwe zmiany prędkości, czyli wyznaczoną wcześniej `baseVel`. Tak naprawdę, to przekształcenie mogłyby wykonywać nastawy regulatora PD, ale pozostaliśmy przy tym wariancie.

```
101 error = map(rightSensor - leftSensor, -150, 150, -baseVel, baseVel);
```

6. Na podstawie uchybu wyznaczany jest sygnał sterujący `control`, który jest obliczany według reguły PD. Nastawy regulatora były dobierane metodą prób i błędów.

```
104 control = P * error + D * (error - errorPrev);
```

Jeśli algorytm obliczył zbyt duży sygnał sterujący to jest on pomniejszany, tak by nie przekraczał wartości `baseVel`. Chodzi o to, by w najgorszym przypadku jedno z kół dostało prędkość równą 0, a nie ujemną.

```
108 if (abs(control) > baseVel)
109     control = control > 0 ? baseVel : -baseVel;
```

7. Sygnał sterujący modyfikuje pożądane prędkości kół. Kluczowy jest znak sygnału sterującego.

```
112 leftVel = baseVel + control;
113 rightVel = baseVel - control;
```

8. Ostatecznie, tylko jedna prędkość ulega zmianie – ta która została zmniejszona. Prędkość która wzrosła, jest sprowadzana na poziom `baseVel`.

```
117 leftVel = leftVel > baseVel ? baseVel : leftVel;
118 rightVel = rightVel > baseVel ? baseVel : rightVel;
```

Nawracanie

Natomiast, jeśli średnia `meanSensor` była zbyt niska – co oznacza, że robot lekko wyjechał poza linię – system sterowania wyznacza prędkości, których celem jest nawrócenie, i powrót pojazdu na trasę.

4. Zmienna `digSensor` zapamiętuje, który sensor cyfrowy wykrył linię jako ostatni. Ta informacja jest wykorzystywana do wyznaczenia kierunku nawracania, a właściwie, wyznaczana jest pewna prędkość bazową nawracania `turnVel`[1], której znak jest zmieniany w zależności od wskazań sensorów cyfrowych.

```
122 turnVel = digSensor == SENSOR_DIGITAL_LEFT ? -TURN_VEL : TURN_VEL;
```

5. Prędkość nawracania `turnVel` jest wykorzystywana do wyznaczenia faktycznych prędkości kół `leftVel`, `rightVel`. Prędkości muszą mieć różny znak, by ruch robota był realizowany w miejscu.

6. Dodatkowo prędkości te są przemnażane, by pojazd poruszał się nie tyle w miejscu, co wykonywał ruch do tyłu w odpowiednim kierunku. W ten sposób uwzględniany jest fakt, że w tym momencie robot jest poza linią.

```
127 leftVel = turnVel < 0 ? turnVel * 1.5 : turnVel * 0.75;
128 rightVel = -turnVel < 0 ? -turnVel * 1.5 : -turnVel * 0.75;
```

[1]Ta zmienna jest nadmiarowa, równie dobrze w kodzie można było dla tego samego celu użyć zmiennej `baseVel`, która jest w tej gałęzi instrukcji `if` nieużywana.

Dalsza część algorytmu

Bez względu na to, która część algorytmu zadziałała, wyznaczone zostały pożądane prędkości kół `leftVel` i `rightVel`. Te prędkości są już podawane do osobnej funkcji `motor`, która zajmuje się generacją odpowiedniego sygnału PWM.

```
148 void motor(int vel, byte leftRight) {
149     byte dir = vel > 0 ? FORWARD : BACKWARDS;
150 
151     vel = map(abs(vel), 0, 100, 0, PWM_MAX);
152 
153     digitalWrite(leftRight == LEFT ? H_LEFT_DIR : H_RIGHT_DIR, dir);
154     analogWrite(leftRight == LEFT ? H_LEFT_PWM : H_RIGHT_PWM, vel);
155 }
```

Funkcja jako argumenty przyjmuje: pożądaną prędkość w procentach (`vel`), oraz który silnik został wybrany (`leftRight`). Znak prędkości jest uwzględniany do wyboru kierunku (`dir`), a sama prędkość jest następnie liniowo przeskalowana na odpowiednie wypełnienie sygnału PWM (teraz już przetwarzana jest tylko jest wartość bezwzględna). Ostatnie dwie linijki związane są z obsługą mostku H:

- najpierw na odpowiedni pin mostka, podawany jest sygnał logicznie prawdziwy/fałszywy w zależności od pożądanego kierunku jazdy,
- następnie na odpowiedni pin mostka, podawany jest sygnał PWM, który odpowiada za prędkość.

O co chodzi z `PWM_MAX`? Nie chcemy by silniki były zasilane zbyt dużym napięciem, chcemy by maksymalnie było to 5V. Dla zdefiniowanej wartość `PWM_MAX` sygnał o tym wypełnieniu dostarcza mniej więcej takie napięcie.

Jak widać, funkcja jest uniwersalna – może obsługiwać oba silniki oraz prędkości w obie strony. Z tego powodu jest tu wiele różnych stałych związanych z mostkiem H, uwzględnieniem kierunku prędkości i uwzględnieniem wybranego silnika[2]. Co najważniejsze, jest czytelna, i nie zajmuje wiele miejsca.

Na samym końcu zapamiętywany jest uchyb, by w następnej iteracji algorytmu został uwzględniony dla części D regulatora.

```
141 errorPrev = error;
```

Cały omówiony algorytm realizowany jest cyklicznie w funkcji `loop`.

[2]Dokładne wartości wszystkich wspomnianych stałych są widoczne w listingu kodu.

4.4 Uwagi do aplikacji

4.4.1 Prędkości bazowe

W kodzie znajdują się 3 zdefiniowane przez `#define` prędkości bazowe:

- `BASE_VEL` – to pożądana prędkość ruchu w linii prostej,
- `PODLOGA_VEL` – to pożądana prędkość ruchu w linii prostej, gdy robot nie znajduje się w całości na linii,
- `TURN_VEL` – to prędkość bazowa dla nawracania.

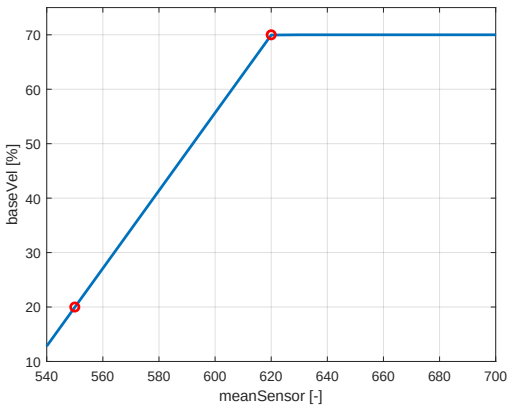
```
49 #define BASE_VEL 70
50 #define PODLOGA_VEL 20
51 #define TURN_VEL 40
```

Użycie prędkości `BASE_VEL` i `PODLOGA_VEL` znajdują się w tej samym miejscu w kodzie – jest to fragment odpowiedzialny za ustalenie prędkości bazowej `baseVel`[3].

```
94 baseVel = meanSensor <= DOLNA_GRANICA ?
95     map(meanSensor, PODLOGA_MEAN, DOLNA_GRANICA, PODLOGA_VEL, BASE_VEL) :
96     BASE_VEL;
```

Na rysunku nr 10 poniżej przedstawiony został kawałek wykresu przedstawiającego wpływ średniej z sensorów na prędkość bazową. Na wykresie zaznaczone zostały dwa punkty:

- (`DOLNA_GRANICA`, `BASE_VEL`) – punkt określający dla jakiego wskazania, utrzymana jest stała prędkość `baseVel` równa `BASE_VEL`,
- (`PODLOGA_MEAN`, `PODLOGA_VEL`) – punkt określający nachylenie linii profilu prędkości.



Rysunek 10: Prędkość `baseVel` w funkcji średniej z sensorów `meanSensor`

[3]Uwaga! `baseVel` to zmienna, której wartość ulega zmianie, a `BASE_VEL` to wartość stała.

Natomiast prędkość TURN_VEL wykorzystywana jest w przypadku dobierania prędkości nawracania turnVel.

```
122 turnVel = digSensor == SENSOR_DIGITAL_LEFT ? -TURN_VEL : TURN_VEL;
```

Wszystkie 3 wspomniane prędkości stanowią pewne punkty odniesienia, które dostrajają robota do pracy w jego środowisku. Skąd konkretne wartości? Zostały dobrane metodą prób i błędów. Gdyby robot miałby przejechać trasę w innym miejscu, być może uległyby one pewnym zmianom.

4.4.2 Charakterystyczne wskazania sensorów analogowych

Podobnie jak w przypadku prędkości, zdefiniowane zostały 3 charakterystyczne wskazania czujników analogowych:

- DOLNA_GRANICA – dolne wskazanie czujnika analogowego, które cały czas oznacza przebywania ponad linią,
- TURN_MEAN – średnia ze wskazań czujników, która przekroczona z dołu oznacza przebywanie poza linią i aktywny tryb nawracania,
- PODLOGA_MEAN – średnia ze wskazań czujników charakterystyczna dla „podłogi" (robot poza linią).

```
58 #define DOLNA_GRANICA 620
59 #define TURN_MEAN 600
60 #define PODLOGA_MEAN 550
```

Pomiędzy tymi wartościami istnieje ścisła relacja:

$$\texttt{PODLOGA\_MEAN} < \texttt{TURN\_MEAN} < \texttt{DOLNA\_GRANICA}$$

- Jeśli średnia jest ponad DOLNA_GRANICA robot znajduje się na linii (dosyć centralnie ponad nią);
- Jeśli średnia jest pomiędzy DOLNA_GRANICA, a TURN_MEAN robot znajduje się na linii, a zaczyna z niej wyjeżdżać, zatem zmniejszeniu ulega prędkość bazowa baseVel;
- Jeśli średnia jest poniżej TURN_MEAN robot znajduje się poza linią, więc aktywowany zostaje tryb nawracania;
- Punkt PODLOGA_MEAN wykorzystywany jest przy liniowym zmniejszaniu prędkości (poprzedni punkt).

Te punkty dobierane są na etapie kalibracji robota (osobny punkt rozdziału).

4.4.3 Nastawy regulatora

Do wyznaczania sygnału sterującego wykorzystany został algorytm PD. Jego nastawy zostały dobrane metodą prób i błędów.

```
73 float P = 0.9;
74 float D = 0.3;
```

Te wartości ulegały także zmianom, jeśli dobierane były inne prędkości bazowe. Dodatkowo warto wspomnieć o dwóch rzeczach:

1. Linijkę przed wyznaczaniem sygnału sterującego `control`, sygnał uchybu `error` jest liniowo przekształcany na pewien zakres prędkości. W klasycznym regulatorze, to nastawy odpowiadają za wszelkie przekształcanie uchybu, więc zastosowane nastawy uległy by pewnym zmianom. Nie miałoby to jednak większego znaczenia na działanie układu (zakładając, że zostaną dobrane odpowiednie nastawy).

2. W przypadku równania PD, człon D pomija wartość kroku czasowego, tzn. wzmocnienie D jest wymnażane przez różnicę pomiędzy uchybem aktualnym, a poprzednim, ale nie jest on dzielony przez krok czasowy dt. W zależności od „sytuacji", kod może wykonywać się krócej lub dłużej, więc nie możemy tu wpisać pewnej stałej wartości czasu. Za pomocą pewnych funkcji można wyznaczyć ile wykonuje się fragment kodu, ale nie ma to większego znaczenia, bo może on ulegać zmianom. Krok czasowy został zatem uwzględniony we wzmocnieniu części D.

4.4.4 Sensory analogowe a cyfrowe

Jak można zauważyć po punkcie analizującym kod systemu sterowania, sensory analogowe i cyfrowe są traktowane w inny sposób. Nie chodzi tutaj o samą oczywisty różnicę pomiędzy sygnałami, które zwracają Arduino, a o ich zadania w systemie sterowania:

– czujniki cyfrowe umieszczone są *dalej* niż czujniki analogowe, ich zadaniem jest zapamiętywanie który czujnik wykrył linię jako ostatni i kluczowe jest to, że analizują przyszłe zdarzenia w stosunku do sensorów analogowych, które reagują na „teraźniejszość",

– informacja pochodząca z czujników analogowych jest wykorzystywana dużo częściej, można powiedzieć „cały czas", bo na ich bazie wyznaczane są sygnały sterujące sprowadzające robota na trasę.

Czujniki cyfrowe są wykorzystywane tylko w przypadku gdy robot zjechał z linii, i należy wyznaczyć kierunek nawracania.

4.5 Kalibracja

Zanim robot będzie mógł realizować swoje zadania podążania wzdłuż linii musi zostać odpowiednio dostrojony do tego. Chodzi tutaj o:

– potencjometry czujników analogowych i cyfrowych,

– dokładne wartości pewnych zdefiniowanych stał występujących w kodzie.

4.5.1 Potencjometry

Obsługa potencjometrów czujników cyfrowych jest dużo prostsza. Służą one do wyznaczenia pewnej wartości progowej odpowiedzialnej za różnicę pomiędzy zwracaniem wartości logicznie prawdziwej, a fałszywej. Prawdę mówiąc, po zainstalowaniu sensorów dokładne ustawienie tych potencjometrów nie ulegało zmianie. Różnice pomiędzy linią, a podłogą, są na tyle duże, że poza wstępnym ustawieniem tych potencjometrów, dalsze dostrajanie nie miało sensu.

Potencjometry czujników analogowych są dużo ważniejsze. Za ich pomocą możemy regulować jasność z jaką świeci dioda, coś zaś wpływa na inny zakres wartości liczbowych zwracanych do kodu Arduino. Ten zakres musi zostać odpowiednio dobrany do stałych występujących w kodzie (omówionych w poprzednich rozdziałach), ale dużo bardziej kluczowa jest inna sprawa – oba czujniki muszą wskazywać mniej więcej to samo. Właśnie w kalibracji tych sensorów chodzi przede wszystkim o to, by oba czujniki wskazywały około te same wartości gdy znajdują się ponad podłogą albo ponad linią. Przyjmujemy tutaj pewien minimalny zakres dla różnic pomiędzy czujnikami, który nie wpływa znacząco na działanie układu[4], ale ważne jest żeby zwracać uwagę, że przy „tyrpnięciu" robota ustawienie może ulec zmianie, przez co trzeba to naprawić.

Do kalibracji służy pewna funkcja w kodzie `sensorPrint`.

```
164 void sensorPrint(void) {
165     Serial.print("l: ");
166     Serial.print(leftSensor);
167     Serial.print(",  r: ");
168     Serial.print(rightSensor);
169     Serial.print(" ==> mean: ");
170     Serial.println(meanSensor);
171     delay(250);
172 }
```

Jej jedynym zadaniem jest wypisywanie na monitorze portu szeregowego wskazań sensorów analogowych oraz ich średniej co okres 250 ms. Należy przy tym także pamiętać o odkomentowaniu dwóch linijek w kodzie:

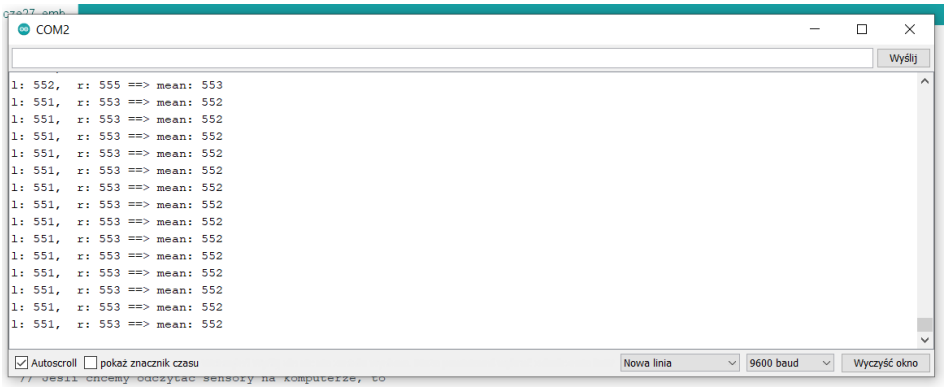
```
43 //Serial.begin(9600);
```

[4]Dla małych różnic i tak wyznaczone sygnały sterujące zostaną następnie przez konwersję na liczby całkowitoliczbowe sprowadzone do zera.

```
138 //sensorPrint();
```

Pierwsza z nich znajduje się w funkcji `setup`, która uruchamia się przy włączeniu Ardunio czy wgraniu nowego kodu i odpowiada za przygotowanie Arduino do przesyłu danych do monitora portu szeregowego. Druga linijka znajduje się w funkcji `loop` i jest to po prostu wywołanie wspomnianej wcześniej funkcji `sensorPrint`.

Przy kalibracji należy pamiętać o tym, by zworka zasilania silników była umieszczona w pozycji OFF (albo zupełnie ściągnięta), a Arduino pozostawało połączone poprzez kabel USB z komputerem. Następnie należy umieszczać pojazd w kilku różnych miejscach na podłodze i sprawdzać czy wskazania są podobne dla tych samych fragmentów podłoża. Jeśli tak nie jest, należy którymś z potencjometrów zmienić jasność diody. Na rysunku nr 11 widać ekran w trakcie kalibracji.



Rysunek 11: Monitor portu szeregowego

4.5.2 Stałe występujące w kodzie

Gdy poprzedni etap kalibracji jest zakończony przechodzimy do kolejnej części. Teraz chodzi o dobranie poprawnych wartości stałych: `PODLOGA_MEAN`, `TURN_MEAN` i `DOLNA_GRANICA`, które poprawnie charakteryzują środowisku w którym będzie poruszał się robot. Cały czas korzystamy tutaj z poprzednio włączonego monitora portu szeregowego.

Stałe należy dobrać w następujący sposób:

1. Należy położyć robota w kilka różnych miejscach na podłodze, ale nie na linii. Podłogę będzie charakteryzował pewien zakres niższych wartości. Stała `PODLOGA_MEAN` to po prostu około wartość w środku tego zakresu.

2. Stała `TURN_MEAN` powinna być górną granicą zakresu charakteryzującego podłogę.

3. Należy położyć robota w kilku różnych miejscach ponad linią, by zbadać zakres wartości charakteryzujący sytuację gdy robot znajduje się ponad linią. Tym razem nie patrzymy na wartość średnią, a na wskazania obu sensorów. Jako stałą `DOLNA_GRANICA` powinno dobrać się wartość dolnej granicy omawianego zakresu.

4.5.3 Stałe związane z prędkością i nastawy regulatora

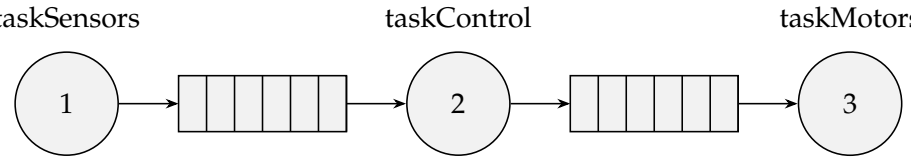
Należy także dobrać odpowiednie wartości stałych względem których dobierane są prędkości, czyli: `TURN_VEL`, `BASE_VEL` i `PODLOGA_VEL` oraz nastawy regulatora PD, ale tutaj już nie ma ogólnych reguł. Jak zostało wspomniane w poprzednich punktach, wartości tych stałych były dobierane metodą prób i błędów. Ogólnie:

- zależy nam na tym, by line follower jak najszybciej pokonywał trasę, więc prędkości powinny być jak największe, ale ...
- czym większe prędkości bazowe, tym trudniej jest układowi regulacji płynnie wyznaczać odpowiednie sygnały sprowadzające pojazd na trasę,
- dodatkowo czym większe prędkości tym bardziej prawdopodobne, że robot opuści trasę, a wtedy już układ sterowania będzie starał się sprowadzać pojazd za pomocą trybu nawracania co znacznie spowalnia ruch.

4.6 System FreeRTOS

Ostatnim krokiem pracy była implementacja systemu sterowania przy pomocy systemu czasu rzeczywistego FreeRTOS. Kompletny kod zaimplementowany na tym systemie został umieszczony, wraz z komentarzami, na końcu raportu jako dodatek B. Podobnie jak w poprzednich punktach, przy listingach fragmentów komentarze będą pomijane.

Schemat przepływu zaproponowanego rozwiązania został przedstawiony na rysunku nr 12.



Rysunek 12: Schemat przepływu

Kod, który poprzednio wykonywał się cyklicznie w funkcji `loop` został podzielony na 3 zadania:

- `taskSensors` – zajmuje się odczytem wskazań czujników, pakuje je do struktury danych i wysyła poprzez kolejkę do `taskControl`,
- `taskControl` – na podstawie danych sensorycznych wykonuje obliczenia odpowiedzialne za wyznaczenie pożądanych prędkości obu kół: `leftVel`, `rightVel`, a następnie umieszcza te prędkości w strukturze danych i przesyła za pomocą kolejki do `taskMotors`,
- `taskMotors` – zadanie, które zajmuje się generacją odpowiednich sygnałów PWM wynikających z zadanych prędkości kół otrzymanych od poprzedniego zadania.

Zadania różnia się priorytetem: `taskSensors` ma najniższy (1), `taskControl` wyższy (2), a `taskMotors` najwyższy (3). Definicja zadań w systemie widoczna jest w listingu poniżej.

```
xTaskCreate(
    taskSensors, "Obsluga czujnikow",
    128, NULL, 1, NULL
);
xTaskCreate(
    taskControl, "Regulacja",
    128, NULL, 2, NULL
);
xTaskCreate(
    taskMotors, "Obsluga silnikow",
    128, NULL, 3, NULL
);
```

Wszystkie 3 zadania otrzymują stos o rozmiarze 128 słów.

Jak zostało wspomniane, zadania komunikują się ze sobą poprzez kolejki. W tym celu zostały utworzone dwie kolejki:

– `sensorQueue` – wymiana danych sensorycznych:

`taskSensors` (nadawca) → `taskControl` (odbiorca)

– `controlQueue` – wymiana danych sterowania:

`taskControl` (nadawca) → `taskMotors` (odbiorca)

Kolejki mogą pomieścić 10 wiadomości, a te są przesyłane poprzez konkretne struktury: `sensorStruct`, `controlStruct`. Ich nazwy jasno wskazują do jakich kolejek trafiają takie paczki danych.

```
struct sensorStruct {
    int leftSensor, rightSensor;
    byte digSensor;
};

struct controlStruct {
    int leftVel, rightVel;
};
```

Definicja kolejek:

```
sensorQueue = xQueueCreate(10, sizeof(struct sensorStruct));
controlQueue = xQueueCreate(10, sizeof(struct controlStruct));
```

Sam algorytm sterowania nie uległ zmianie – zaproponowane rozwiązanie korzystające z systemu FreeRTOS jest „tłumaczeniem" tego co zostało napisane dla systemu wbudowanego na wspomniany system czasu rzeczywistego. Z tego powodu, nie został wykorzystany cały potencjał Fre-

eRTOSa i robot po zastosowaniu takiego kodu nie działa lepiej, a po prostu tak samo. Trzeba mieć na uwadze 2 kwestie:

1. Zaproponowany system sterowania nie jest szczególnie skomplikowany są tylko 4 czujniki, 2 silniki i pewne obliczenia „pomiędzy". Do wykorzystania takiego sprzętu spokojnie wystarcza system wbudowany. Co prawda, system czasu rzeczywistego pozwala dużo lepiej panować nad tym, kiedy działają które fragmenty kodu i prawdopodobnie można by usprawnić nawet zaproponowane rozwiązanie, które nie jest szczególnie skomplikowane, ale...

2. Na wykorzystanie systemu FreeRTOS wykorzystany zostało tylko ostatni tydzień prac nad projektem. Było to zbyt mało czasu, by dobrze zapoznać się z możliwościami systemu.

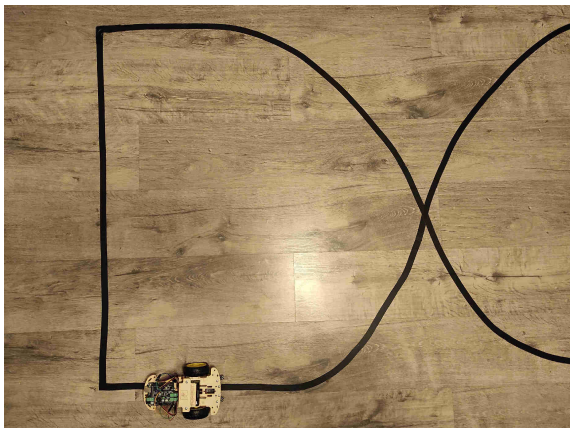
5 Rezultat

Ogólnie rzecz mówiąc, robot poprawnie radzi sobie z zadaniem, które miał wykonywać, czyli podążać za linią. Co więcej, skromnym zdaniem autorów robot jeździ znacznie płynniej i szybciej od „konkurencji", które prezentowała swoje rozwiązania w trakcie ostatnich zajęć projektowych[5].

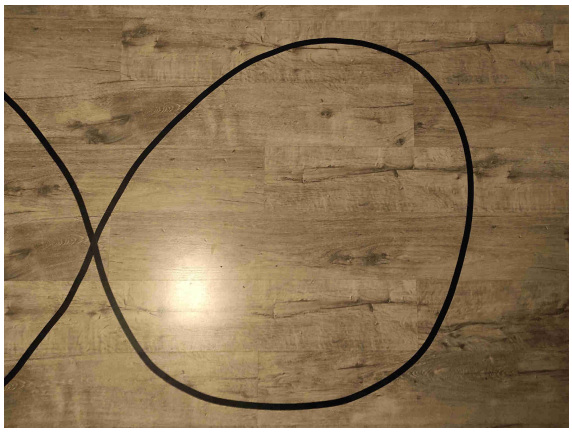
Materiały wideo (te same co rozdziale „Harmonogram"), pokazujące finalne działanie robota umieszczone są w linkach poniżej:

- https://youtu.be/4WTrk285v4s
- https://youtu.be/nwpjGGdDjxc

Trasa, którą line follower miał pokonywać była znana od początku zajęć projektowych. Jest ona widoczna na rysunku 13.



(a)



(b)

Rysunek 13: Trasa

Można tu dostrzec kilku kluczowych punktów, z którym robot musiał sobie poradzić:

- przecięcie na środku trasy,
- ostre skręte o 90 stopni,
- długi skręt po łuku,
- poza tym, musi także umieć pokonać ją w obu kierunkach.

Omawiany robot potrafi poradzić sobie ze wspomnianymi punktami:

[5]Co więcej, w porównaniu do większości wspomnianej konkurencji, autorzy prezentowali wyniki na żywo, a nie w postaci nagrań, które równie dobrze mogły zostać przygotowane dla mniej pewnych i bezpiecznych nastaw, dających lepsze rezultate (ale z pewnym prawdopodobieństwem, że coś może się nie udać

– Przecięcie na środku trasy nie robi większego problemu. Co prawda, po przejechaniu tego punktu robot przez chwilę oscyluje wokół trasy, ale są one momentalnie tłumione i nie powodują większych problemów. Podczas testów nie zdarzyło się ani razu, by pojazd wjechał na nie właściwą część trasy na skrzyżowaniu.
– Skręt po łuku także nie jest większym problemem, robot płynnie pokonuje tę część trasy.
– Przez pewien (krótki) czas, problem stanowił skręt o 90 stopni. Na początku prac, zaproponowany został robot wyposażony w dwa czujniki analogowe na przodzie, które w zupełności wystarczają do pokonania całej trasy, poza tymi dwoma punktami ostrych skrętów. Po kupnie i instalacji dwóch czujników cyfrowych (i napisaniu odpowiedniego kodu), te fragmenty także przestały stanowić problem. Algorytm sterowania stara się podążać wzdłuż linii – korzystając z czujników analogowych – a gdy lekko za nią wyjeżdża, bo nie jest w stanie za pomocą tych dwóch czujników zrealizować płynne skrętu o taki kąt, czujniki cyfrowe informują o tym, który jako ostatni wykrył linię i robot zaczyna nawracać w odpowiednim kierunku. Następnie, gdy znajduje już się na trasie, aktywowany ponownie jest tryb sterowania według reguły PD.
– Wykonanie trasy w obie strony nie stanowi żadnego problemu.

Trasa nie stanowi żadnego problemu dla omawianego pojazdu. Zakładamy tu oczywiście, że poprawnie został przeprowadzony etap kalibracji (poprzedni punkt rozdziału), gdyż w przypadku dobrania nieodpowiednich wartości dla stałych występujących w kodzie i nastaw regulatora, robot nie wykonuje już swojego zadania tak płynnie jak powinien.

– Przy większych pożądanych prędkościach (stałe prędkości bazowych) robotowi może zdarzyć się wyjechanie podczas trasy. Wtedy aktywowany jest tryb nawracania, który poza ostrymi skrętami w narożnikach nie powinien być aktywowany, bo znacznie zwiększa to czas pokonania trasy.
– Przy niewłaściwych nastaw regulatora PD robot może albo w ogóle nie nadążać za zmianami na trasie (i ją opuszczać) albo może reagować zbyt gwałtownie. Jeśli za cel stawiamy sobie pokonać trasę jak najszybciej, to lepiej jest by reagował zbyt gwałtownie na zmiany trasy, ale robot wtedy porusza się dosyć nieelegancko – wykonuje bardzo gwałtowne skręty.

6 Podsumowanie

Podsumowując, projekt skończył się sukcesem – robot dobrze wykonywał swoje zadanie. Można się jednak przyczepić do kilku fragmentów pracy:

- Wykorzystane w pracy silniki z kołami nie zawierają enkoderów. Powoduje to, że system nie zna prędkości z jakimi faktycznie kręcą się kołami, co ma wpływ na dokładność systemu. Założono jednak, że ta nieznajomość będzie traktowana jako pewne zakłócenie działające na obiekt podlegający regulacji, które jest skutecznie tłumione przez sygnały sterujące. Prawdopodobnie, robot stracił na tym pewną dokładność, ale i tak działał zadowalająco dobrze.
- Przez większość czasu robot korzysta tylko z dwóch sensorów analogowych do wyznaczania odpowiednich poprawek prędkości. Można cieszyć się z tego, że tak mała ilość informacji starcza do wyznaczania poprawnych sterowań, ale z drugiej strony gdyby zastosować większą ilość sensorów, można by mocniej wpłynąć na poprawne sterowanie. Nawet dodanie choćby jednego sensora analogowego (np. na środek dziobu platformy, pomiędzy dwa pozostałe czujniki) mogłoby znacząco usprawnić układ regulacji. Z drugiej jednak strony dodanie kolejnych sensorów analogowych znacząco zwiększa czas wykonywania się kodu, więc to też należałoby mieć na uwadze...
- Być może zatem, najlepiej byłoby skorzystać z większej ilości czujników cyfrowych, ale wtedy dużej zmianie musiałby ulec zaproponowany system sterowania.
- Czujniki cyfrowe wykorzystywane są tylko i wyłącznie do wyznaczenia kierunku nawracania. Być może, umieszczenie ich w innym miejscu, pozwoliłoby na wykorzystywanie ich razem z czujnikami analogowymi do wyznaczania sygnałów sterujących w trakcie podążania za linią.

Wymienione wyżej uwagi należy jednak bardziej rozumieć jako „pomysły na przyszłość", które zostałyby zrealizowane gdyby autorzy ponownie wykonywali swoją pracę.

Zajęcia projektowe stanowiły ważny punkt w semestrze. Projekty uczą najlepiej, bo pokazują na ile teorie omawiane w trakcie wykładów mogą zostać zastosowane do rozwiązań prawdziwych rzeczywistych problemów. Projekty pokazują także, że wspomniane teorie stanowią pewne uproszczone modele rzeczywistych zagadnień i nie zawsze sprawdzą się w całości w rzeczywistości. Jako przykład można choćby podać ruch pojazdu do przodu – należy po prostu podać na oba koła (zakładając napęd różnicowy) tę samą prędkość, by robot poruszał się do przodu ze stałą prędkością. Łatwo można się przekonać, że cała konstrukcja mechaniczna ma duży wpływ na ten ruch, i ruch idealnie po linii prostej jest bardzo trudno uzyskać.

Projekt pokazuje także, że rozwiązań jednego problemu jest mnóstwo i wybrane zależy głównie od preferencji autorów: pojazd może mieć różną ilość kół, platforma może mieć inny kształt, czujniki mogą zostać zamontowane w różnych miejscach podwozia, jak i same czujniki mogą dzia-

łać na różnych zasadach czy wreszcie sam kod może zostać napisany na wiele różnych sposobów korzystając z systemu wbudowanego lub systemu czasu rzeczywistego – jest tu dużo miejsca na kreatywność.

Last but not least, projekt był ważny, bo w czasach chińskiego wirusa, gdy cała nauka działa w trybie zdalnym, i większość zajęć to głównie siedzenie przed komputerem, każdy kontakt z rzeczywistym sprzętem zyskuje dwukrotnie na swojej wartości.

Dodatek A Pełny kod – system wbudowany

```
// Line follower -- system wbudowany
// Sekcja nr 1:
//    Maksymilian Skibiński,
//    Paweł Kaźmieruk.
// AiR S2-I/Rob, 2021 r.

// Obsługa mostku H.
#define H_LEFT_PWM 5
#define H_LEFT_DIR 4
#define H_RIGHT_PWM 6
#define H_RIGHT_DIR 9

// Oznaczenie kierunku ruchu na bazie znaku prędkości.
#define FORWARD 0
#define BACKWARDS 1

// Oznaczenie położenia silnika.
#define LEFT 0
#define RIGHT 1

// Maksymalne dozwolone wypełnienie sygnału PWM.
#define PWM_MAX 165

// Piny sensorów analogowych.
#define SENSOR_ANALOG_LEFT A1
#define SENSOR_ANALOG_RIGHT A0

// Piny sensorów cyfrowych.
#define SENSOR_DIGITAL_LEFT 12
#define SENSOR_DIGITAL_RIGHT 8

void setup() {
    // Deklaracja pinów mostka H.
    pinMode(H_LEFT_DIR, OUTPUT);
    pinMode(H_RIGHT_DIR, OUTPUT);
    pinMode(H_LEFT_PWM, OUTPUT);
    pinMode(H_RIGHT_PWM, OUTPUT);

    // W przypadku, gdy chcemy komunikacji robota z komputerem,
    // by odczytać sensory na ekranie, aktywujemy taką komunikację
    // linijką poniżej.

    //Serial.begin(9600);
}


// Prędkości względem których algorytm sterowania
// oblicza używane prędkości.
#define BASE_VEL 70
#define PODLOGA_VEL 20
#define TURN_VEL 40

// Kluczowe wskazania sensorów analogowych. Zależą one,
// głównie od warunków ,,podłogowych'':
//    - DOLNA_GRANICA to troche ponad dolna granica wskazania na lini,
//    - TURN_MEAN to troche ponad wskazanie podlogowe,
```

```
57 //    - PODLOGA_MEAN to srednia ze wskazan na podlodze.
58 #define DOLNA_GRANICA 620
59 #define TURN_MEAN 600
60 #define PODLOGA_MEAN 550
61
62 // Zmienne:
63 //    - digSensor -- zapamiętuje, który sensor cyfrowy wskazał jako ostatni,
64 //    - wskazania czujników analogowych i ich średnia,
65 //    - zmienne zapamiętujące wyznaczane prędkości,
66 //    - zmienne algorytmu sterowania.
67 byte digSensor = 0;
68 int leftSensor, rightSensor, meanSensor;
69 int baseVel, leftVel, rightVel, turnVel;
70 int control, error, errorPrev = 0;
71
72 // Nastawy regulatora PD.
73 float P = 0.9;
74 float D = 0.3;
75
76 void loop() {
77     // Odczyt sensorów analogowych.
78     leftSensor = analogRead(SENSOR_ANALOG_LEFT);
79     rightSensor = analogRead(SENSOR_ANALOG_RIGHT);
80
81     // Zmienna digSensor zapamiętuje, który sensor cyfrowy wykrył linię jako ostatni.
82     digSensor = digitalRead(SENSOR_DIGITAL_LEFT) ? SENSOR_DIGITAL_LEFT : digSensor;
83     digSensor = digitalRead(SENSOR_DIGITAL_RIGHT) ? SENSOR_DIGITAL_RIGHT : digSensor;
84
85     // Średnia z pomiarów czujników analogowych.
86     meanSensor = (rightSensor + leftSensor) / 2.0;
87
88     // W zależności od wskazania średniego sensorów analogowych robot:
89     //    - realizuje sterowanie według regulatora PD,
90     //    - realizuje nawracanie.
91     if (meanSensor > TURN_MEAN) {
92         // ,,Prędkość bazowa'' -- jeśli średnia z sensorów jest za niska to pr. bazowa
93         // zostaje zmniejszona, bo oznacza to, że pojazd nie jest ponad linią.
94         baseVel = meanSensor <= DOLNA_GRANICA ?
95             map(meanSensor, PODLOGA_MEAN, DOLNA_GRANICA, PODLOGA_VEL, BASE_VEL)
96             : BASE_VEL;
97
98         // Uchyb z sensorów analogowych, który jest od razu liniowo przekształcany
99         // na zmiany prędkości (równie dobrze nastawy regulatora
100        // mogłyby się tym zająć).
101        error = map(rightSensor - leftSensor, -150, 150, -baseVel, baseVel);
102
103        // Sygnał sterujący regulatora PD.
104        control = P * error + D * (error - errorPrev);
105
106        // Ograniczenie górne -- jeśli sygnał sterujący jest
107        // zbyt duży to stosujemy gorną granicę.
108        if (abs(control) > baseVel)
109            control = control > 0 ? baseVel : -baseVel;
110
111        // Sygnał sterujący trafia na oba koła z innym znakiem.
112        leftVel = baseVel + control;
113        rightVel = baseVel - control;
114
115        // Chcemy jedynie zmniejszyć jedną z prędkości, zatem jeśli, któraś
```

```
116         // wzrosła to wraca ona na poziom prędkości ,,bazowej''.
117         leftVel = leftVel > baseVel ? baseVel : leftVel;
118         rightVel = rightVel > baseVel ? baseVel : rightVel;
119     } else {
120         // W zależności od tego, który sensor cyfrowy wykrył linię jako ostatni
121         // wyznaczany jest odpowiedni kierunek zwrotu.
122         turnVel = digSensor == SENSOR_DIGITAL_LEFT ? -TURN_VEL : TURN_VEL;
123 
124         // Robot nie wykonuje obrotu w miejscu, bo znajduje się trochę za linią
125         // dlatego podawane prędkości są odpowiednio przemnażane, by
126         // obrócił się oraz wrócił do tyłu.
127         leftVel = turnVel < 0 ? turnVel * 1.5 : turnVel * 0.75;
128         rightVel = -turnVel < 0 ? -turnVel * 1.5 : -turnVel * 0.75;
129     }
130 
131     // Ostatecznie, prędkości trafiają do silników.
132     motor(leftVel, LEFT);
133     motor(rightVel, RIGHT);
134 
135     // Jeśli chcemy odczytać sensory na komputerze, to
136     // wykorzystujemy zakomentowaną funkcję.
137 
138     //sensorPrint();
139 
140     // Uchyb poprzedni, dla następnej iteracji.
141     errorPrev = error;
142 }
143 
144 // Funkcja do przekazywania sygnału sterowania na silniki.
145 // Przyjmuje dwa argumenty:
146 //     - vel -- prędkość zadana w procentach, uwzględniany jest znak,
147 //     - leftRight -- który silnik.
148 void motor(int vel, byte leftRight) {
149     // Uwzględnienie kierunku.
150     byte dir = vel > 0 ? FORWARD : BACKWARDS;
151 
152     // Przekształcenie prędkości w procentach,
153     // na użytkowany zakres sygnału PWM.
154     vel = map(abs(vel), 0, 100, 0, PWM_MAX);
155 
156     // Ostatecznie, przypisanie prędkości wraz z obsługą mostka H.
157     digitalWrite(leftRight == LEFT ? H_LEFT_DIR : H_RIGHT_DIR, dir);
158     analogWrite(leftRight == LEFT ? H_LEFT_PWM : H_RIGHT_PWM, vel);
159 }
160 
161 // Funkcja zajmująca się wypisywaniem poprzez monitor portu szeregowego
162 // wskazań sensora, i ich średniej.
163 // Funkcja jest istotna na etapie ,,strojenia'' potencjometrów czujników analogowych.
164 void sensorPrint(void) {
165     Serial.print("l: ");
166     Serial.print(leftSensor);
167     Serial.print(",  r: ");
168     Serial.print(rightSensor);
169     Serial.print(" ==> mean: ");
170     Serial.println(meanSensor);
171     delay(250);
172 }
```

Dodatek B Pełny kod – system FreeRTOS

```
// Line follower -- system FreeRTOS
// Sekcja nr 1:
//    Maksymilian Skibiński,
//    Paweł Kaźmieruk.
// AiR S2-I/Rob, 2021 r.

// Ładujemy FreeRTOSa oraz obsługę kolejek.
#include <Arduino_FreeRTOS.h>
#include <queue.h>

// W kodzie tworzymy 3 zadania:
//    - taskSensors -- obsługa sensorów,
//    - taskControl -- generowanie sterowania na podstawie danych z sensorów,
//    - taskMotors -- podawanie sygnałów sterujących na silniki.
void taskSensors(void *pvParameters);
void taskControl(void *pvParameters);
void taskMotors(void *pvParameters);

// Pomiędzy zadaniami są 2 kolejki.
QueueHandle_t sensorQueue;
QueueHandle_t controlQueue;

// Paczka danych wysyłana przez taskSensors do taskControl.
struct sensorStruct {
    int leftSensor, rightSensor;
    byte digSensor;
};

// Paczka danych wysyłana przez taskControl do taskMotors.
struct controlStruct {
    int leftVel, rightVel;
};

void setup() {
    // Tworzymy dwie kolejki do wymiany danych o odpowiednich rozmiarach (w bajtach)
    // i założyliśmy kolejki o długości 10 wiadomości.
    sensorQueue = xQueueCreate(10, sizeof(struct sensorStruct));
    controlQueue = xQueueCreate(10, sizeof(struct controlStruct));

    // Jeśli kolejki zostały utworzone, tworzymy zadania.
    // Zadania różnią się priorytetem, gdzie 3 to najwyższy.
    if (sensorQueue != NULL && controlQueue != NULL) {
        xTaskCreate(
            taskSensors, "Obsluga czujnikow",
            128, NULL, 1, NULL
        );
        xTaskCreate(
            taskControl, "Regulacja",
            128, NULL, 2, NULL
        );
        xTaskCreate(
            taskMotors, "Obsluga silnikow",
            128, NULL, 3, NULL
        );
    }
}
```

```
57
58  // Funkcja ,,loop'' jest teraz oczywiście pusta.
59  void loop() { }
60
61  // Piny sensorów analogowych.
62  #define SENSOR_ANALOG_LEFT A1
63  #define SENSOR_ANALOG_RIGHT A0
64
65  // Piny sensorów cyfrowych.
66  #define SENSOR_DIGITAL_LEFT 12
67  #define SENSOR_DIGITAL_RIGHT 8
68
69  // Zadanie taskSensors zajmuje się odczytem wskazań sensorów
70  // oraz przekazaniem ich poprzez kolejkę do następnego zadania.
71  void taskSensors(void *pvParameters) {
72
73      for (;;) {
74          struct sensorStruct sensorData;
75
76          // Odczyt sensorów analogowych.
77          sensorData.leftSensor = analogRead(SENSOR_ANALOG_LEFT);
78          sensorData.rightSensor = analogRead(SENSOR_ANALOG_RIGHT);
79
80          // Zmienna digSensor zapamiętuje, który
81          // sensor cyfrowy wykrył linię jako ostatni.
82          sensorData.digSensor =
83              digitalRead(SENSOR_DIGITAL_LEFT) ? SENSOR_DIGITAL_LEFT
84              : sensorData.digSensor;
85          sensorData.digSensor =
86              digitalRead(SENSOR_DIGITAL_RIGHT) ? SENSOR_DIGITAL_RIGHT
87              : sensorData.digSensor;
88
89          // Dane zostają przekazane dalej.
90          xQueueSend(sensorQueue, &sensorData, portMAX_DELAY);
91      }
92  }
93
94  // Prędkości względem których algorytm sterowania
95  // oblicza używane prędkości.
96  #define BASE_VEL 70
97  #define PODLOGA_VEL 20
98  #define TURN_VEL 40
99
100 // Kluczowe wskazania sensorów analogowych. Zależą one,
101 // głównie od warunków ,,podłogowych'':
102 //    - DOLNA_GRANICA to troche ponad dolna granica wskazania na lini,
103 //    - TURN_MEAN to troche ponad wskazanie podlogowe,
104 //    - PODLOGA_MEAN to srednia ze wskazan na podlodze.
105 #define DOLNA_GRANICA 620
106 #define TURN_MEAN 600
107 #define PODLOGA_MEAN 550
108
109 // Zadanie taskControl zajmuje się wygenerowaniem odpowiednich sygnałów
110 // sterujących, czyli prędkości dla obu silników.
111 // Następnie, te prędkości zostają przekazane do zadania taskMotors.
112 void taskControl(void *pvParameters) {
113     // Zmienne:
114     //    - meanSensor -- średnia z dwóch sensorów analogowych,
115     //    - baseVel, turnVel -- prędkość bazowa dla ruchu wzdłuż linii, i obrotu,
```

```
116     //    - zmienne algorytmu sterowania.
117     int meanSensor;
118     int baseVel, turnVel;
119     int control, error, error_prev;
120     float P, D;
121     struct sensorStruct sensorData;
122 
123     // Inicjalizacja uchyba poprzedniego oraz nastaw regulatora.
124     error_prev = 0;
125     P = 0.9;
126     D = 0.3;
127 
128     for (;;) {
129         // Jeśli zadania otrzymało paczkę danych od sensora to kod jest realizowany.
130         // W przeciwnym razie zadanie czeka.
131         if (xQueueReceive(sensorQueue, &sensorData, portMAX_DELAY) == pdPASS) {
132             struct controlStruct controlData;
133 
134             // Średnia z pomiarów czujników analogowych.
135             meanSensor = (sensorData.rightSensor + sensorData.leftSensor) / 2.0;
136 
137             // W zależności od wskazania średniego sensorów analogowych robot:
138             //    - realizuje sterowanie według regulatora PD,
139             //    - realizuje nawracanie.
140             if (meanSensor > TURN_MEAN) {
141                 // ,,Prędkość bazowa'' -- jeśli średnia z sensorów jest za niska
142                 // to pr. bazowa zostaje zmniejszona, bo oznacza to,
143                 // że pojazd nie jest ponad linią.
144                 baseVel = meanSensor <= DOLNA_GRANICA ?
145                     map(meanSensor, PODLOGA_MEAN, DOLNA_GRANICA, PODLOGA_VEL, BASE_VEL)
146                     : BASE_VEL;
147 
148                 // Uchyb z sensorów analogowych, który jest od razu
149                 // liniowo przekształcany na zmiany prędkości
150                 // (równie dobrze nastawy regulatora mogłyby się tym zająć).
151                 error = map(sensorData.rightSensor - sensorData.leftSensor,
152                     -150, 150, -baseVel, baseVel);
153 
154                 // Sygnał sterujący regulatora PD.
155                 control = P * error + D * (error - error_prev);
156 
157                 // Ograniczenie górne -- jeśli sygnał sterujący
158                 // jest zbyt duży to stosujemy gorną granicę.
159                 if (abs(control) > baseVel)
160                     control = control > 0 ? baseVel : -baseVel;
161 
162                 // Sygnał sterujący trafia na oba koła z innym znakiem.
163                 controlData.leftVel = baseVel + control;
164                 controlData.rightVel = baseVel - control;
165 
166                 // Chcemy jedynie zmniejszyć jedną z prędkości, zatem jeśli, któraś
167                 // wzrosła to wraca ona na poziom prędkości ,,bazowej''.
168                 controlData.leftVel =
169                     controlData.leftVel > baseVel ?
170                     baseVel : controlData.leftVel;
171                 controlData.rightVel =
172                     controlData.rightVel > baseVel ?
173                     baseVel : controlData.rightVel;
174             } else {
```

```
                // W zależności od tego, który sensor cyfrowy wykrył linię jako ostatni
                // wyznaczany jest odpowiedni kierunek zwrotu.
                turnVel =
                    sensorData.digSensor == SENSOR_DIGITAL_LEFT ?
                    -TURN_VEL : TURN_VEL;

                // Robot nie wykonuje obrotu w miejscu, bo znajduje się trochę za linią
                // dlatego podawane prędkości są odpowiednio przemnażane, by
                // obrócił się oraz wrócił do tyłu.
                controlData.leftVel = turnVel < 0 ? turnVel * 1.5 : turnVel * 0.75;
                controlData.rightVel = -turnVel < 0 ? -turnVel * 1.5 : -turnVel * 0.75;
            }

            // Dane zostają przekazane do następnego zadania.
            xQueueSend(controlQueue, &controlData, portMAX_DELAY);

            // Uchyb poprzedni, dla następnej iteracji.
            error = error_prev;
        }
    }
}


// Obsługa mostku H.
#define H_LEFT_PWM 5
#define H_LEFT_DIR 4
#define H_RIGHT_PWM 6
#define H_RIGHT_DIR 9

// Maksymalne dozwolone wypełnienie sygnału PWM.
#define PWM_MAX 165

// Oznaczenie kierunku ruchu na bazie znaku prędkości.
#define FORWARD 0
#define BACKWARDS 1

void taskMotors(void *pvParameters) {
    // Deklaracja pinów mostka H.
    pinMode(H_LEFT_DIR, OUTPUT);
    pinMode(H_RIGHT_DIR, OUTPUT);
    pinMode(H_LEFT_PWM, OUTPUT);
    pinMode(H_RIGHT_PWM, OUTPUT);

    struct controlStruct controlData;
    byte dir;

    for (;;) {
        // Jeśli zadanie otrzymało paczkę danych od taskControl, to kod
        // jest realizowany. W przeciwnym wypadku zadanie oczekuje.
        if (xQueueReceive(controlQueue, &controlData, portMAX_DELAY) == pdPASS) {
            // Najpierw silnik lewy.
            // Uwzględnienie kierunku.
            dir = controlData.leftVel > 0 ? FORWARD : BACKWARDS;

            // Przekształcenie prędkości w procentach,
            // na użytkowany zakres sygnału PWM.
            controlData.leftVel = map(abs(controlData.leftVel), 0, 100, 0, PWM_MAX);
            digitalWrite(H_LEFT_DIR, dir);
            analogWrite(H_LEFT_PWM, controlData.leftVel);
```

```
234
235             // Teraz to samo dla silnika prawego.
236             dir = controlData.rightVel > 0 ? FORWARD : BACKWARDS;
237
238             controlData.rightVel = map(abs(controlData.rightVel), 0, 100, 0, PWM_MAX);
239             digitalWrite(H_RIGHT_DIR, dir);
240             analogWrite(H_RIGHT_PWM, controlData.rightVel);
241         }
242     }
243 }
```