AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Praca dyplomowa inżynierska

Pojazd gąsienicowy zdalnie sterowany przez aplikację okienkową

A tracked vehicle remotely controlled by dedicated window applicaiton

Autor: *Aleksander Filek*
Kierunek studiów: Informatyka Techniczna
Opiekun pracy: *dr inż. Adam Mrozek*

Kraków, 2023

Spis treści

Wstęp

Wraz z postępem cywilizacyjnym życie żołnierza stawało się coraz bardziej istotną kwestią. Na przestrzeni lat starano się opracować sposoby na zminimalizowanie ryzyka utraty życia lub odniesienia ran. Z myślą o tym wprowadzono pancerze i hełmy chroniące przez pociskami czy odłamkami (np. kamizelka kuloodporna). Pojazdy również zostały poddane modyfikacji w formie wytrzymalszego pancerza co pozwoliło na ukrycie się załogi w względnie bezpiecznym miejscu. Kolejnym naturalnym krokiem było sprawienie by żołnierz przebywał z dala od linii frontu, a zadania wykonywał przy pomocy urządzeń którymi może sterować z bezpiecznej odległości. W tym celu zaczęto eksperymentować z pojazdami zdalnie sterowanymi.

Początkowo takowe pojazdy były sterowane przy pomocy przewodów, którymi operator mógł wydawać polecenia pojazdowi, np. pojazd Goliath będący zdalnie sterowaną miną używaną w trakcie drugiej wojny światowej. Sprawiały one szereg problemów, ich zasięg operowania był bardzo ograniczony, przewód taki w skomplikowanym terenie mógł zostać uszkodzony co skutkowało utratą kontaktu z pojazdem. Przełom w tej technologii nastąpił dopiero po zastosowaniu komunikacji radiowej. Pozwoliło to na wysyłanie pojazdów na znacznie dalsze odległości przy zachowaniu stabilnego połączenia. Największy rozwój dotyczył segmentu lotniczego wykorzystywanych w zwiadzie jak i celach bojowych. Przykładem takiego drona jest amerykański MQ-1 Predator, który od 1994 roku brał udział w każdym większym konflikcie zbrojnym. W międzyczasie rozwijano pojazdy naziemne, których głównym zastosowaniem były prace saperskie, zwiadowcze oraz transport zasobów. W nowoczesnych konfliktach coraz częściej można spotkać drony bojowe, ale również te których zadaniem jest ewakuacja cywili czy żołnierzy z niebezpiecznych regionów. Rozwój tego segmentu pozwolił zminimalizować zagrożenie dla żołnierza, ale również koszty operowania, ponieważ, przy utracie pojazdu, wyszkolony żołnierz pozostaje żywy, a koszt pojazdy sterowane często jest znacznie niższy od maszyn które maja również chronić operatora.

System pojazdów zdalnie sterowanych można podzielić na trzy oddzielne segmenty: stanowisko operatora, pojazd oraz system komunikacji. Stanowisko musi zapewnić operatorowi wygodny mechanizm sterowania zawierający widok z kamery oraz interfejs sterowania. Mniejsze nowoczesne pojazdy korzystają z przenośnych dotykowych ekranów przez które operator wydaje polecenia, natomiast pojazdy bojowe często korzystają z

systemów o rozmiarach kokpitu myśliwca lub małego pokoju. Amerykańscy żołnierze korzystają chętnie z kontrolerów do gier, co znacznie ułatwia im sterowanie pojazdem oraz naukę z względu na ich wcześniejsze zaznajomienie się z tym rozwiązaniem. Sterowany pojazd musi być przystosowany do terenu w którym będzie operować, np. pojazdy saperskie często są wyposażone w podwozie gąsienicowe pozwalające na efektywne pokonywanie przeszkód terenowych. Montuje się na nich ramię pozwalające operatowi na manipulowanie obiektem stwarzającym zagrożeniem. Takowy pojazdu musi być również wyposażony w kamerę do obserwacji otoczenia. System komunikacji musi zapewnić stabilne oraz bezpieczne połączenie. W tym celu wykorzystuje się komunikacje radiową lub systemy satelitarne pozwalające na znacznie większy zasięg operowania. Sygnał sterujący jest szyfrowany w celu zabezpieczenia go przed wrogimi siłami. Rozszyfrowanie go naraziłoby operatora na niebezpieczeństwo jeśli znajduje się on w niedalekiej odległości od obszaru działania. Podczas wojny w Ukrainie do komunikacji z dronami zaczęto wykorzystywać internet dostarczany przez satelity Starlink firmy SpaceX. Pozwala to na względnie bezpieczne operowanie w obszarze działań oraz zapewnia szybkie i stabilne łącze, które w przypadku komunikacji radiowej często jest narażone na zakłócenia.

1 Cel pracy i zakres pracy

Celem niniejszej pracy jest zaprojektowanie oraz konstrukcja sterowanego radiowo pojazdu gąsienicowego, którego system sterowania oparty były na aplikacji okienkowej uruchamianej na komputerach z systemem Windows. Pojazd będzie wyposażony dodatkowo w termometr oraz żyroskop, którego dane będą przesyłane do aplikacji. Nadajnik podłączony do komputera będzie służył jako pośrednik pomiędzy aplikacją, a pojazdem. W celu realizacji pracy podzielono zadanie na 3 części:

- Pojazdu
- Nadajnik
- Aplikacja okienkowa

Na pierwszą część będzie się składać układ elektryczny sterujący silnikami, zbierający dane telemetryczne oraz układ komunikacji radiowej. W celu stworzenia komunikacji pomiędzy tymi systemami zostaną zaimplementowano protokoły takie jak SPI (ang. serial peripheral interface), I^2C (ang. two wire interface), które zostały zaprogramowane na bazie dokumentacji wykorzystywanego mikrokontrolera AVR.

Część nadajnika składa się z układu elektryczny z radiem oraz system komunikacji UART (ang. universal asynchronous receiver-transmitter), który przetwarza komunikaty aplikacji i przesyła je do pojazdu oraz na odwrót.

Ostatnim etapem jest aplikacji okienkowej zaprogramowana z wykorzystaniem środowiska WPF. Aplikacja będzie służyły jako system sterowania oraz zarządzania pojazdem, przez którą użytkownik może wydawać odpowiednie komendy.

2. Zdalnie sterowane pojazdy gąsienicowe

Zdalnie sterowane pojazdy gąsienicowe są powszechnie wykorzystywane w różnych dziedzinach w tym w wojsku, w przemyśle, w nauce, a nawet w rolnictwie. Pojazdy te wyposażone są w kamery oraz sensory umożliwiające operatorowi na efektywne wykonywanie operacji na odległość. Pojazdy te są szczególnie przydatne w sytuacjach, gdy zagrożenie dla ludzi jest zbyt duże lub gdy potrzebna jest duża mobilność i zwrotność w trudnym terenie.

2.1 Przegląd pojazdów

2.1.1 Foster-Miller Talon

Jest zdalnie sterowany pojazd o szerokim zastosowaniu w armii. Wyposażony jest w zestaw kamer oraz czujników. Posiada charakterystyczne ramię, którym operator może wykonywać precyzyjne operacje. Z względu na jego niewielki rozmiar oraz masę (około 27 kg w wersji podstawowej) jest bardzo mobilnym pojazdem. Wykorzystywany jest przez wojsko do przeprowadzania rozpoznania, drugim najpopularniejszym zastosowaniem jest wsparcie saperów, co pozawala im nie zbliżać się do niebezpiecznych materiałów. Operator dzięki niemu może w bezpieczny dla siebie sposób zneutralizować zagrożenie. Istnieje również możliwość zamontowania na nim karabinu maszynowe lub wyrzutni rakiet. Wiele armii świata w tym wojsko polskie posiada go na swoim wyposażeniu.

Robot Talon firmy Foster-Miller

Źródło: https://www.robotmilitar.org/robot-militar-talon/

2.1.2 Ripsaw

Ripsaw to rodzina zdalnie sterowanych pojazdów produkowanych przez firmę Textron Systems. Pierwszy pojazd tej serii Ripsaw M1, został opracowany w latach 90, jako lekki i zwrotny pojazd do przeprowadzania rozpoznania. Był on testowany przez wojsko w trakcie drugiej wojny w Zatoce Perskiej. W dalszych latach projekt był rozwijany co poskutkowało stworzeniem modelu Ripsaw M5 wykorzystywanego obecnie przez różne siły zbrojne na całym świecie do wykonywania różnych zadań na polu walki, takich jak rozpoznanie terenu, transport ładunków i wsparcie na polu walki.

Pojazd bojowy Ripsaw M5 firmy Textron Systems

Źródło: https://www.textronsystems.com/products/ripsaw-m5#specs

3. Protokół komunikacji

System zdalnie sterowanego pojazdu gąsienicowego składa się z trzech podsystemów: pojazd, nadajnik oraz aplikacja. Komunikują się one pomiędzy sobą przy pomocy różnych interfejsów. W celu standaryzacji konstrukcji wiadomości zaprojektowano poniższy protokół.

Wiadomość protokołu składa się z 32 bajtów. Rozmiar ten został dobrany ze względu na maksymalny rozmiar wiadomości przesyłanej przez dobrany moduł radiowy, który wynosi dokładnie 32 bajty.

Zawartość	Typ	Status	Dane	Symbol zamykający
Liczba bajtów	1	1	29	1

Typ – wartość numeryczna typu komendy. System obsługuje domyślnie trzy różne wiadomości:

- Connect – służy do sprawdzenia poprawności działania komunikacji pomiędzy aplikacją, nadajnikiem oraz pojazdem.
- MotorControl – komenda sterowania pojazdem
- GetData – pobiera dane telemetryczne z pojazdu

Status - rezultat wykonania komendy, przyjmuje dwa stany:

- Success – poprawne wykonanie
- Failed – niepoprawne wykonanie

Dane – tablica 29 bajtów, która przechowuje dana przesyłane podsystemami.

Symbol zamykający – pojedynczy symbol ASCII końca linii, służący do zamknięcia wiadomości.

Protokół nie wymaga mechanizmu kontroli poprawności wiadomości, ponieważ wykorzystywane interfejsy komunikacji, implementują własne rozwiązania.

4. Projekt pojazdu

Do budowy układ elektrycznego dobrano komponenty o parametrach spełniających wymagania, którymi było napięcie pracy oraz interfejsy komunikacji. Dodatkowo uwzględniono ich dostępność na rynku. Komponenty musiały operować na prądzie poniżej 3.7V z względu na napięcie akumulatorów, więc zdecydowano się na stabilizacje napięcia do 3.3V. Zostały umieszczone i połączone na uniwersalnej płytce prototypowej, której rozmiar został dobrany do wolnego miejsca na podwoziu pojazdu.

4.1 Komponenty elektroniczne

4.1.1 Mikrokontroler

Głównym elementem składowym układu jest mikrokontroler Atmega 328P produkowany przez firmę Atmel. Jest to układ scalony zawierający 32-bitowy procesor AVR, 32 KB pamięci oraz różne peryferia jak interfejsy we/wy, liczniki oraz timery. Popularnie jest wykorzystywany w module Arduino UNO. W tym układzie operuje z częstotliwością taktowania 16MHz. Wykorzystywany jest jego interfejs SPI do komunikacji z radiem oraz I^2C do pobierania danych z żyroskopu. Natomiast prędkość silników sterowana jest poprzez sygnały PWM (ang. Pulse Width Modulation), pozwalające na zmianę napięcia z bardzo dużą częstotliwością. Funkcjonalności dla tego układu są tworzone zgodnie z dokumentacją [1].

Rys.3.1.1 .Mikrokontroler AVR – ATmega328P

Źródło: witryna sklepu internetowego [1]

4.1.2 Moduł radiowy

NRF24L01 to moduł radiowy produkowany przez firmę Nordic Semiconductor. Jest to układ scalony umożliwiający komunikację bezprzewodową za pomocą transmisji radiowej

w paśmie 2,4 GHz. Moduł ten pozwala na wysyłanie jak i odbieranie danych. Komunikacja z nim następuje poprzez interfejs SPI. Wykorzystywany jest w robotach, systemach automatyki domowej oraz urządzeniach przemysłowych.

Rys.3.1.2. Moduł radiowy nRF24L01

Źródło: witryna sklepu internetowego [2]

4.1.3 Układ żyroskopu, akcelerometru i termometru

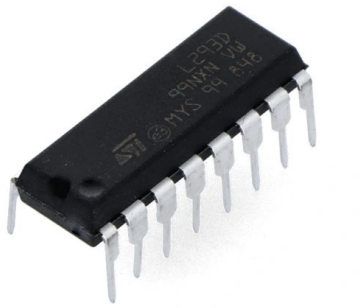
MPU-6050 jest to układ zawierający akcelerometr, żyroskop firmy InvenSense. Służy on do pomiaru przyśpieszenia i rotacji w trzech osiach. Wyposażony jest on dodatkowo w termometr o zakresach -40°C do +85°C. Do komunikacji wykorzystuje protokół I^2C.

Rys.3.1.3. Układ MPU-6050

Źródło: witryna sklepu internetowego [3]

4.1.4 Sterownik silników

L293D to układ scalony firmy STMicroelectronics, służący do sterowania silnikami elektrycznymi prądu stałego. Posiada wejścia umożliwiające wybór kierunków obrotów silników oraz prędkości ich obrotów przy pomocy sygnału PWM. Obsługuje on napięcie od 3V do 30V, przy czym napięcie logiczne maksymalnie do 5V.

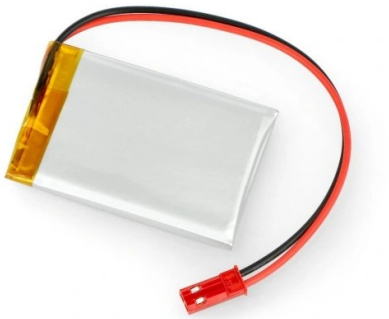


Rys.3.1.4. Sterownik silników L293D

Źródło: witryna sklepu internetowego [4]

4.1.5 Akumulator LiPo

Jako źródło zasilania wykorzystano dwa akumulatory litowo-polimerowe (LiPo). Charakteryzują się dobrym stosunkiem gęstości energii do masy przez co są często są stosowane w smartfonach, aparatach fotograficznych czy w modelach RC. Wykorzystane w projekcie akumulatory posiadają napięcie 3.7V oraz pojemność 750mAh. Połączone zostały w sposób równoległy co poskutkowało podwojeniem ich pojemności. Ładownie ich następuje poprzez dedykowaną ładowarkę przez co nie było potrzeby implementacji systemu ładowania do układu.

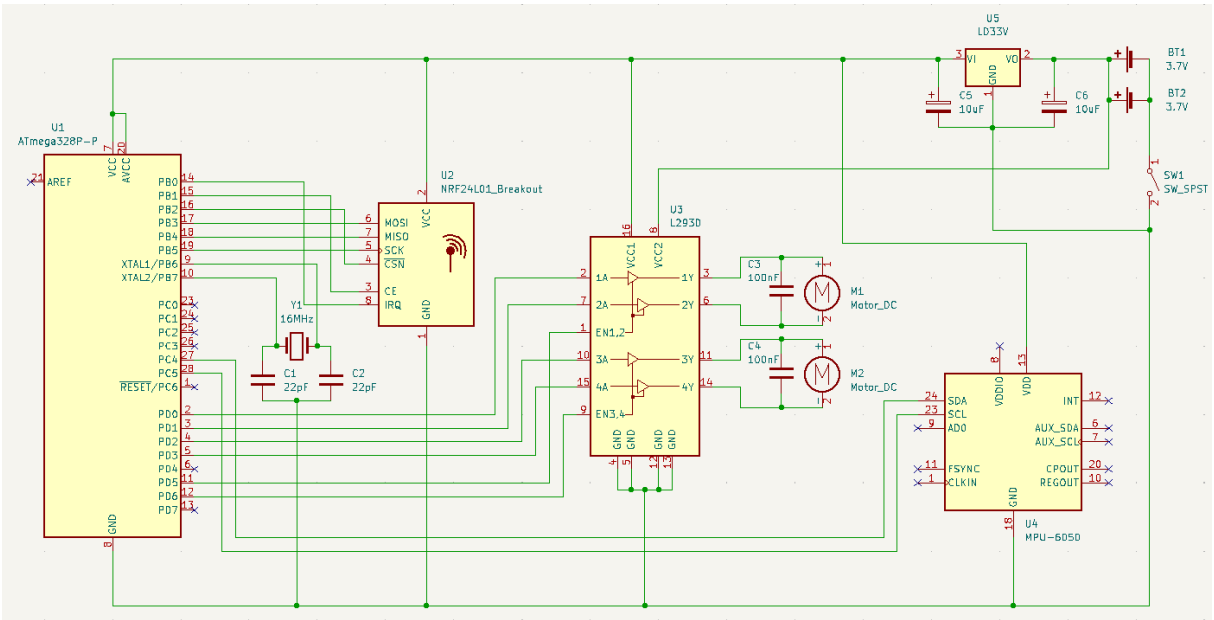


Rys.3.1.5. Akumulator Li-Pol 3.7V 750mAh

Źródło: witryna sklepu internetowego [5]

4.2 Schemat elektryczny

Układ elektryczny został zaprojektowany w programie KiCad 6.0. Układ zasilany jest z dwóch akumulatorów o napięciu 3.7V, połączonych równolegle, co pozwala podwoić ich sumaryczną pojemność do 1500mAh. Napięcie jest obniżane do 3.3V przy pomocy stabilizatora liniowego LD33V, który jest wspierany przez dwa kondensatory elektrolityczne 10uF, których zadaniem jest filtrowanie napięcia. Przełącznik dwu pozycyjny przesuwny odpowiada za przerywanie obwodu i odłączanie zasilania od układów.



Rys. 3.2. Schemat elektryczny pojazdu

Źródło: opracowanie własne

Mikrokontroler Atmega328P służy jako jednostka sterująca układu. Działa z częstotliwością 16MHz wygenerowaną przy pomocy rezonatora kwarcowego oraz dwóch kondensatorów elektrolitycznych o pojemności 22pF.

Moduł radiowy nRF24L01 jest połączony do mikrokontrolera przy pomocy interfejsu SPI. Pin MOSI (ang. Master Out, Slave In) odpowiada za przesył informacji z mikrokontrolera do radia, natomiast pin MISO (ang. Master In, Slave Out) za odwrotny kierunek. Pin SCK (ang. Serial Clock) służy do synchronizacji sygnału zegara pomiędzy układami. Piny CSN (ang. SPI Chip Select) oraz CE (ang. Chip Enable) odpowiadają za przekazanie informacji, że dany moduł będzie wykorzystywany w komunikacji SPI oraz sterowanie komendami. Wykorzystywany jest również pin IRQ, który gdy przyjmuje stan wysoki informatuje mikrokontroler, że radio otrzymało wiadomość, która powinna zostać przetworzona.

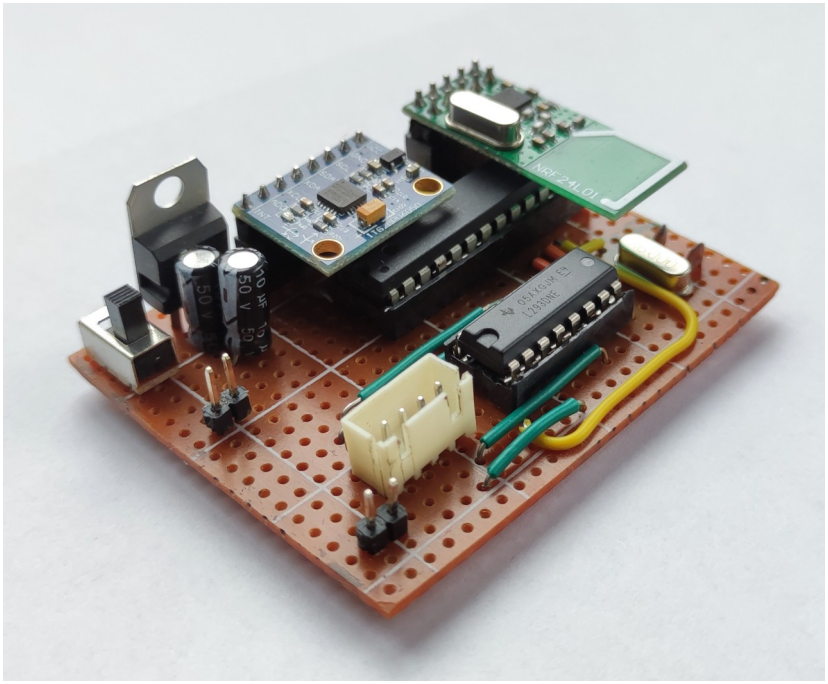
Silniki pojazdu są sterowane przy pomocy dwukanałowego sterownika silników L293D. Każdy z silników sterowany jest przy pomocy 3 połączeń. Połączenia typu A odpowiadają za określanie kierunku obrotu. Gdy tylko jeden z sygnałów A ma stan wysoki to silnik się obraca, w przypadku dwóch wysokich lub dwóch niskich stanów silnik zostaje natychmiastowo zatrzymany. Prędkość silnika ustalana jest poprzez połączenie EN. Wykorzystywany jest sygnał PWM, generowany z częstotliwością 20 833Hz. Silniki są zasilane napięciem nie stabilizowanym czyli 3.7V, ponieważ potrzebuję one operować na napięciu powyżej wykorzystywanego logicznego., dodatkowo ich zasilanie jest odszumiane przy pomocy kondensatorów ceramicznych 100nF.

Jako układ telemetryczny dodany żyroskop, akcelerometr i termometr w formie pojedynczego układu scalonego MPU-6050. Komunikuje się on z mikrokontrolerem przy pomocy interfejsu I^2C, który wymaga jedynie dwóch przewodów SDA (ang. Serial Data Line) i SCL (ang. Serial Clock Line). Dane z niego są pobierane a następnie w przetworzonej wersji wysyłane do aplikacji, gdzie są wyświetlone użytkownikowi.

4.3 Złożenie

Układ elektryczny został złożony na uniwersalnej płytce prototypowej, którą wycięto z większego arkusza. Ze względu na zastosowanie wpinanych układów pozwoliło to na

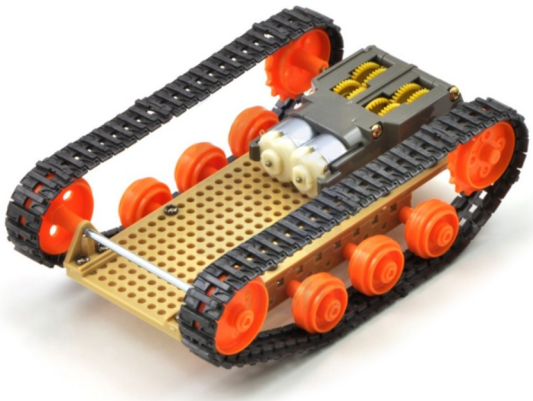
stworzenie płytki o niewielkim rozmiarze. Połączenia logiczne zostały umieszczone na górnej stronie płytki, na dolnej znajdują się przewody zasilające.



Rys. 4.3.1. Układ sterujący pojazdem na płytce prototypowej

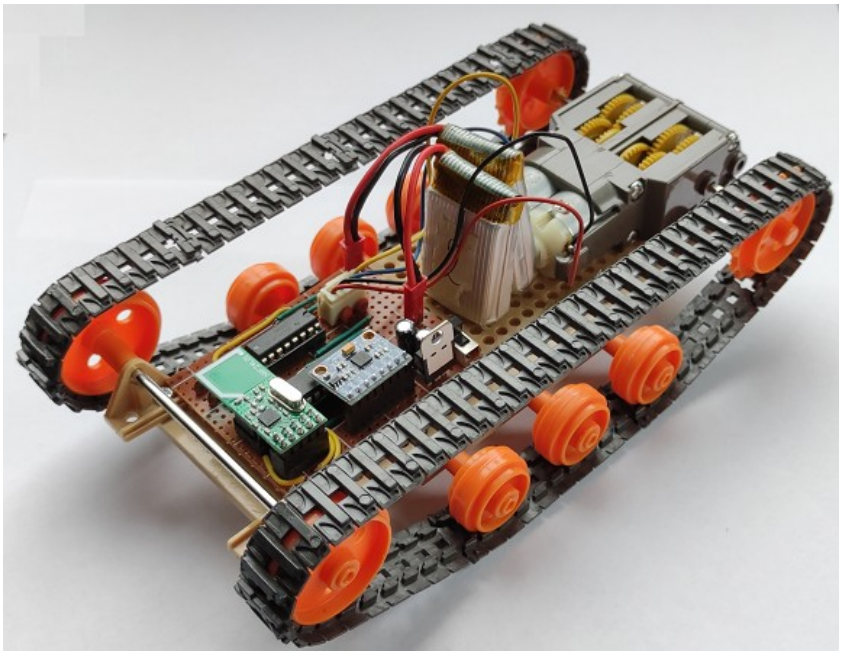
Źródło: opracowanie własne

Układ zamontowano na podwoziu wykonanym przez firmę Tamiya. Jest to popularnie stosowane w modelarstwie podwozie, które zawiera w zestawie przekładnie dwóch silników. Zawiera ono również otwory montażowe przy pomocy których przymocowano układ.



Rys. 4.3.2. Podwozie czołgu zbudowane z elementów firmy Tamiya

Źródło: witryna sklepu internetowego [6]



Rys. 4.3.3. Złożony pojazd

Źródło: opracowanie własne

4.4 Kod źródłowy

Kod został napisany z wykorzystaniem programu Microchip Studio, który pozwala na bezproblemowe skompilowanie i wgranie do mikrokontrolera programu. Do sterowania modułem radiowym wykorzystano sterownik przygotowany przez producenta układu.

4.4.1 Interfejs SPI

SPI to intefejs szeregowy, który umożliwia przesyłanie danych pomiędzy urządzeniami przy pomocy wspólnej szyny danych i synchronizacji. W tym projekcie

wykorzystywany jest do komunikacji z mikrokontrolera z modułem radiowym przez co został skonfigurowany w trybie dla pojedynczego zarządcy (ang. master).

W funkcji konfiguracyjnej w pierwszej kolejności ustawiane są piny DDB5 – MOSI oraz DDB3 – SCK jako piny wyjściowej. Następnie uruchamiany jest interfejs SPI – SPE, tryb zarządcy – MSTR, szybkość zegara – SPR0 oraz aktywowany jest system przerwań dla wiadomości przychodzących.

```
void SpiMasterInit(void)
{
	DDRB |= (1 << DDB5) | (1 << DDB3);
	SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0) | (1<<SPIE);
}
```

Fragment kodu 1. Funkcja inicjująca interfejs SPI

Źródło: opracowanie własne

Wysyłanie bajtu polega na przypisaniu do rejestru SPDR wysyłanego bajtu, a następnie kontynuacja funkcji blokowana jest poprzez pętle, aż do wystąpienia przerwania informującego o poprawnym przesłaniu wiadomości.

```
uint8_t SpiSend(uint8_t data)
{
	spiTxRxDone = 0;
	SPDR = data;

	while(spiTxRxDone == 0);

	return spiRxData;
}
```

Fragment kodu 2. Funkcja wysyłająca dane poprzez interfejs SPI

Źródło: opracowanie własne

Funkcja przerwania sprawdza czy rejestr SPSR, który przechowuje informacje o rezultacie operacji ma ustawiony bit SPIF oznaczający pomyślne zakończenie przesyłania. Następnie wartość rejestru SPDR zapisywana jest do zmiennej ponieważ może posiadać wartość zwrotną, a zmienna spiTxRxDone jest resetowana do wartości 1.W przypadku niepoprawnej operacji do zmiennej spiTxRxDone zostanie przypisana wartość 255.

```
ISR(SPI_STC_vect)
{
	if(SPSR & (1<<SPIF))
	{
		spiRxData = SPDR;
		spiTxRxDone = 1;
		return;
	}

	spiRxData = SPDR;
	spiTxRxDone = 255;
}
```

Fragment kodu 3. Funkcja przerwania interfejsu SPI

Źródło: opracowanie własne

4.4.2 Interfejs I²C

Interfejs ten to dwukierunkowa magistrala służąca do przesyłania danych w urządzeniach elektronicznych. W kodzie źródłowym nazywany jest TWI (ang. Two Wire Interface). Komunikacja z układem MPU-6050 została ustawiona na 100kHz, ponieważ prędkość przesyłu danych telemetrycznych w tym przypadku nie jest istotna, a pozwala ograniczyć zużycie energii.

Poniższa funkcja odpowiada za uruchomienie komunikacji. Najpierw obliczana jest prędkość, z którą operuje układ, a następnie jest przypisywana do rejestru TWBR odpowiadającego za przechowywanie współczynnika dzielenia potrzebnego dla dzielnika

częstotliwości, który generuje częstotliwość pracy. Następnie do rejestru TWCR ustawiany jest bit TWEN odpowiadający za uruchomienie komunikacji oraz TWIE odpowiadający za uruchomienie przerwań.

```
void TwiInit(uint32_t speed)
{
	uint32_t gen_t = 0;
	gen_t = (((F_CPU/speed) - 16)/2) & 0xFF;
	TWBR = gen_t & 0xFF;
	TWCR = (1 << TWEN) | (1 << TWIE);
}
```

Fragment kodu 4.4.1.1. Funkcji inicjująca interfejs USART

Źródło: opracowanie własne

Poniżej przedstawiono fragment kodu mający za zadanie obsługę błędów, czyli sprawdzenie czy poprzednia funkcja wykonała się pomyślnie i w przypadku niepowodzenia przerwanie komunikacji oraz aktualnie wykonywanej funkcji. Fragment ten został wyszczególniony osobno w celu skrócenia dalej opisywanych funkcji. Każde jego wystąpienie zostało opisane jako „//obsługa błedu”.

```
if(err != TWI_OK)
{
	TwiStop();
	return err;
}
```

Fragment kodu 4.4.1.1. Funkcji inicjująca interfejs USART

Źródło: opracowanie własne

Funkcja TwiRead ma za zadanie odczytywać dane z danego urządzenia. Pierwszy parametr addr to adres układu, z którym następuje komunikacja, każdy układ ma swój unikatowy adres. Parametr reg to rejestr z którego dane będą odczytywane, następnie podawany jest wskaźnik do tablicy – data oraz liczba bajtów, które mają zostać odczytane – length. Na początku funkcji zostaje wysłany sygnał mający rozpocząć komunikacji TwiStart, następnie do rejestru TWDR, który służy do przechowywania wysyłanych i odbieranych danych przypisywany jest adres układu, z którym następuje komunikacja. Funkcja TwiAddrWriteAck sprawdza status poprzedniej operacji. Następnie wysłany jest rejestr, z którego będą pobierane dane. Funkcja TwiRestart resetuje część rejestrów i przełącza

komunikacje w tryb odczytu. Pętla for pobiera zadaną liczbę danych i przypisuje ją do tablicy, na samym końcu komunikacje zostaje zamknięta TwiStop.

```
uint8_t TwiRead(uint8_t addr, uint8_t reg, uint8_t* data, uint16_t length)
{
	uint16_t i = 0;
	uint8_t err = TWI_OK;
	err = TwiStart();
	//obsługa błędu
	TWDR = (addr << 1) | 0;
	err = TwiAddrWriteAck();
	//obsługa błędu
	TWDR = reg;
	err = TwiDataWriteAck();
	//obsługa błędu
	err = TwiRestart();
	//obsługa błędu
	TWDR = (addr << 1) | 1;
	err = TwiAddrReadAck();
	//obsługa błędu
	for(i = 0; i < (length - 1); i++)
	{
		err = TwiDataReadAck(1);
		//obsługa błędu
		data[i] = TWDR;
	}
	err = TwiDataReadAck(0);
	//obsługa błędu
	data[i] = TWDR;
	TwiStop();
	return err;
}
```

Fragment kodu 4.4.1.1. Funkcji inicjująca interfejs USART

Źródło: opracowanie własne

4.4.3 Sterowanie silnikami

Funkcja inicjująca sterowanie silnikami ma za zadanie w pierwszej kolejności skonfigurować piny PWM do sterowania prędkością silników z częstotliwością 20 833Hz. W tym celu wykorzystywane są liczniki 8-bitowe. Ustawienie bitu WGM01 w rejestrze TCCR0A sprawia, że licznik po osiągnięciu wartości porównywalnej jest zerowany. Bity CS00 oraz CS01 w rejestrze TCCR0B ustawiają skale licznika (ang. prescaler) na 64. Sterownik został skonfigurowany do obsługi dwóch silników, które są dalej rozróżniane jako motorId o wartościach 0 oraz 1.

```
void MotorInit()
{
	TCCR0A = (1<<WGM01);
	TCCR0B = (1<<CS00) | (1<<CS01);
	TIMSK0 = (1<<OCIE0A);
	OCR0A = 5;
	OCR0B = 5;
	DDRD |= (1<<DDD6) | (1<<DDD5) | (1<<DDD3) | (1<<DDD2) | (1<<DDD1) | (1<<DDD0);

	factor[0] = factor[1] = 0;
	MotorSpeedSet(0, 0);
	MotorSpeedSet(1, 0);
	MotorDirectionSet(0, MD_STOP);
	MotorDirectionSet(1, MD_STOP);
}
```

Fragment kodu 4.4.1.1. Funkcji inicjująca interfejs USART

Źródło: opracowanie własne

Funkcja MotorSpeedSet odpowiada za ustawienie wartości - value z zakresu 0-255 dla danego silnika. Tablica nextFactor przechowuje wartości wypełnienia sygnałów PWM dla silników. Poprzednia wartość jest nadpisywana po zakończeniu aktualnego cyklu.

```
void MotorSpeedSet(uint8_t motorId, uint8_t value)
{
	nextFactor[motorId] = value;
}
```

Fragment kodu 4.4.1.1. Funkcji inicjująca interfejs USART

Źródło: opracowanie własne

Do działania PWM potrzebne jest zaimplementowanie funkcji przerwania, która wywołuje się z zadaną częstotliwością. Wewnątrz niej zliczane są jej wywołania które pozwalają podzielić jej wywołania na dwa segmenty: stan wysoki i niski. W ten sposób uzyskiwany jest sygnał sterowania prędkością o zmiennym wypełnieniu.

```
ISR (TIMER0_COMPA_vect)
{
	static int counter = 0;
	counter++;
```

```
	uint8_t high = PORTD & (1<<DDD6);

	uint8_t realFactor = (high) ? factor[0] : 256 -  factor[0];
	if(counter == realFactor)
	{
		if(factor[0] != nextFactor[0])
		{
			factor[0] = nextFactor[0];
		}
		PORTD ^= (1<<DDD6);
		counter = 0;
	}
}
```

Fragment kodu 4.4.1.1. Funkcji inicjująca interfejs USART

Źródło: opracowanie własne

Funkcja MotorDirectionSet odpowiada za ustawianie kierunku obrotów silników, który jest ustawiany przy pomocy dwóch sygnałów. Ustawienie MD_Stop oznacza zatrzymanie silnika, czyli oba sygnały mają stan niski. MD_FORWARD obrót silników umożliwiających poruszanie pojazdu do przodu – pierwszy sygnał to stan wysoki, a drugi to stan niski. MD_BACKWARD działa odwrotnie do MD_FORWARD. Z racji, że piny sterujące są w tym samym rejestrze ułożone obok siebie, możliwe jest obliczenie ich wartości pin0, pin1 przy pomocy motorId oraz pierwszego pinu sterujacego DDD0.

```
void MotorDirectionSet(uint8_t motorId, MotorDirection direction)
{
	uint8_t pin0 = DDD0 + 2*motorId;
	uint8_t pin1 = pin0 + 1;
	switch(direction)
	{
		case MD_STOP:
			PORTD &= ~(1<<pin0);
			PORTD &= ~(1<<pin1);
			break;
		case MD_FORWARD:
			PORTD |= (1<<pin0);
			PORTD &= ~(1<<pin1);
			break;
		case MD_BACKWARD:
			PORTD &= ~(1<<pin0);
			PORTD |= (1<<pin1);
```

```
                    break;
        }
}
```

Fragment kodu 4.4.1.1. Funkcji inicjująca interfejs USART

Źródło: opracowanie własne

4.4.4 Obsługa komend

System sterowania komendami oparty jest strukturze danych zwanej mapa, która jako klucz używa wartość numeryczną komendy do której przyporządkowany jest wskaźnik do funkcji. Funkcja inicjująca CommadnInit ma za zadanie zarezerwować miejsce w pamięci na strukturę danych ma Przypisanie odpowiedniej funkcji następuje poprzez funkcje CommandRegisterFunc.

```
CommandInit(commandManagerRef);
CommandRegisterFunc(commandManagerRef, CMD_MOTORCONTROL, OnMotorControl);
CommandRegisterFunc(commandManagerRef, CMD_GETDATA, OnGetData);
```

Fragment kodu 4.4.1.3. Przerwanie odbierające dane

Źródło: opracowanie własne

Komenda obsługi silników ma za zadanie przekonwertować dane wejściowe użytkownika na prędkości i kierunki obrotów silników. Funkcja analizuje kilka przypadków. Pierwszy z nich to sytuacja, gdy dane wejściowe są równe 0, w takim przypadku zostaje wydane polecenie zatrzymania silników i ustawienia ich prędkości na 0. W przypadku ruchu do przodu lub do tyłu ustawiana jest maksymalna prędkość dla obu silników , a następnie dobrany kierunek obrotów. Jeśli zmienna forward równa jest 1 to porusza się do przodu, jeśli 2 to do tyłu. Następnie analizowany jest status danych skrętu: jeśli right równe 1 to skręca w lewo, jeśli 2 to w prawo. Komenda skrętu bez poruszania do przód/tył powoduje ustawienie jednego silnika na maksymalną wartość oraz wyłączenie drugie, w ten sposób otrzymywany jest skręt w miejscu. W przypadku ruchu przód/tył oraz skrętu zamiast zatrzymywać silnik, jego prędkość jest ustawiana na połowę, co skutkuje ruchem po promieniu.

```
void OnMotorControl(CommandType Type, CommandStatus Status, uint8_t* data)
{
```

```
	uint8_t forward = data[2];
	uint8_t right = data[3];
	uint8_t speed[2] = { 0, 0 };
	MotorDirection direction = MD_STOP;
	if(forward + right > 0)
	{
		direction = (forward == 2)? MD_BACKWARD: MD_FORWARD;
		speed[0] = speed[1] = 255;

		if(right > 0)
		{
			uint8_t turnMotorId = right - 1;
			speed[turnMotorId] = (forward > 0)? 128: 0;
		}
	}
	MotorSpeedSet(0, speed[0]);
	MotorSpeedSet(1, speed[1]);
	MotorDirectionSet(0, direction);
	MotorDirectionSet(1, direction);
}
```

Fragment kodu 4.4.4.1. Komenda sterowania silnikami

Źródło: opracowanie własne

Komenda pobierania danych telemetrycznych pobiera dane z układu MPU6050 przy pomocy interfejsu I^2C. Adres układu oraz rejestrów został odczytany z mapy rejestrów [8]. Obliczenie temperatury polega na przekonwertowaniu danych z rejestrów na wartość zmiennoprzecinkową, następnie dzielona jest przez wartość 340 oraz dodanie 36,53 podane w dokumentacji. W celu uzyskania danych akcelerometru należy pobrać dane z sześciu rejestrów. Następnie są one mapowane do przedziału -90 do 90, a końcowe wartości kątów są uzyskiwane po obliczeniu wartości funkcji arcus tangens oraz przekonwertowaniu ich do stopni z radianów. Tak uzyskane dane wysyłane są odpowiednią komendą z powrotem do nadajnika, a pojazd wraca do trybu odbioru.

```
#define MPU6050_ADDR 0b1101000

void OnGetData(CommandType Type, CommandStatus Status, uint8_t* data)
{
	uint8_t temperatureReg[2];
	TwiRead(MPU6050_ADDR, 0x41, temperatureReg, 2);
```

```
        int temperatureBig = (temperatureReg[0] << 7) | temperatureReg[1];
        float temperature = ((float)temperatureBig / 340.0f) + 36.53f;

        uint8_t accelerometerReg[6];
        TwiRead(MPU6050_ADDR, 0x3B, accelerometerReg, 6);
        float acX = (float)((accelerometerReg[0] << 8) | accelerometerReg[1]);
        float acY = (float)((accelerometerReg[2] << 8) | accelerometerReg[3]);
        float acZ = (float)((accelerometerReg[4] << 8) | accelerometerReg[5]);

        float mappedX = map(acX, 265, 402, -90.0f, 90.0f);
        float mappedY = map(acY, 265, 402, -90.0f, 90.0f);
        float mappedZ = map(acZ, 265, 402, -90.0f, 90.0f);

        float angleX = (atan2(-mappedY, -mappedZ) + M_PI) * 180.0f / M_PI;
        float angleY = (atan2(-mappedX, -mappedZ) + M_PI) * 180.0f / M_PI;
        float angleZ = (atan2(-mappedY, -mappedX) + M_PI) * 180.0f / M_PI;

        uint16_t newDataLength = 4 * sizeof(float);
        uint8_t newData[newDataLength];
        memcpy(newData, &temperature, sizeof(float));
        memcpy(newData + sizeof(float), &angleX, sizeof(float));
        memcpy(newData + 2*sizeof(float), &angleY, sizeof(float));
        memcpy(newData + 3*sizeof(float), &angleZ, sizeof(float));

        uint8_t* command = CommandCreate(CMD_GETDATA, CMDS_SUCCESS, newData,
newDataLength);

        Nrf24StopListening(radioRef);
        Nrf24OpenWritingPipe(radioRef, pipe);
        Nrf24Write(radioRef, command, COMMAND_LENGTH);
        free(command);

        Nrf24OpenReadingPipe(radioRef, pipe);
        Nrf24StartListening(radioRef);
}
```

Fragment kodu 4.4.4.2. Komenda pobierania danych telemetrycznych

Źródło: opracowanie własne

5. Projekt nadajnika

Głównym zadaniem nadajnika jest pośredniczenie w komunikacji pomiędzy komputerem, a pojazdem, przez co jest on wyposażony moduł radiowy oraz moduł do komunikacji poprzez USB. Dodatkowo umieszczono w nim brzęczyk, który dźwiękowo wspiera informowanie o postępach w działaniu.

5.1 Komponenty elektroniczne

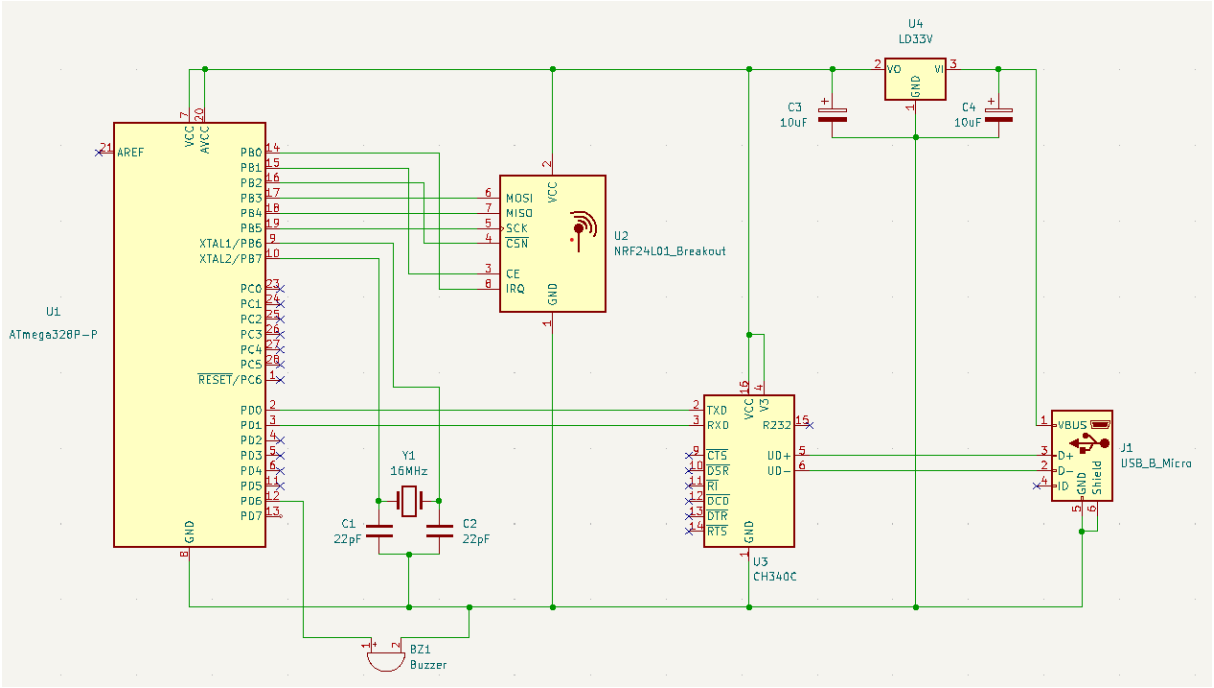
Układ mikrokontrolera oraz modułu radiowego są takie same jak w układzie pojazdu. Do komunikacji poprzez USB wykorzystano układ scalony CH340C, który konwertuje sygnał z mikrokontrolera na sygnał komputera, a następnie ten sygnał jest wyprowadzany przez złącze USB mikro.

Rys.4.1 Układ scalony CH340C

Źródło: witryna sklepu internetowego [7]

5.2 Schemat elektryczny

Układ zasilany jest napięciem 5V z komputera poprzez złącze USB, które następnie jest stabilizowane do 3.3V. Mikrokontroler oraz radio są podłączone z sobą w identycznie jak w układzie pojazdu. Do pinu PD6 został podłączony dodatkowo brzęczyk. Układ CH340C został podłączony do mikrokontrolera poprzez piny PD0 oraz PD1, które są wykorzystywane w sprzętowej komunikacji interfejsu USART.

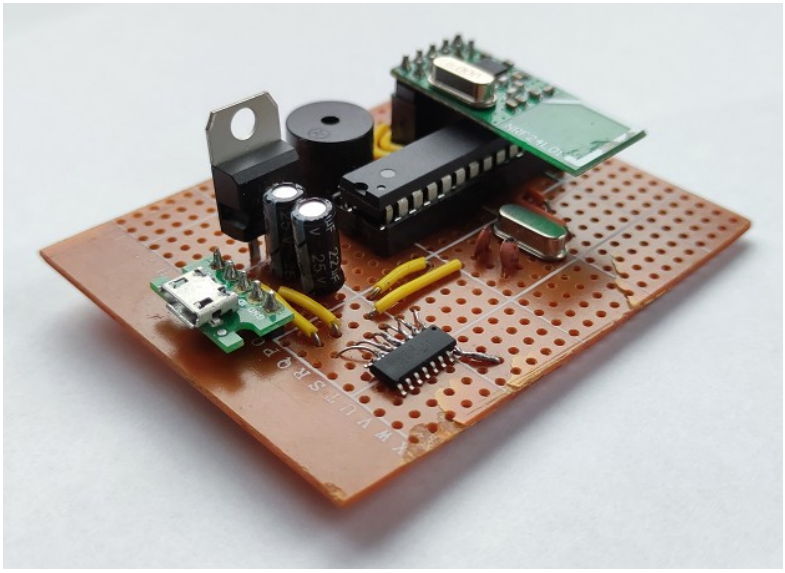


Rys.4.2. Schemat elektryczny nadajnika

Źródło: opracowanie własne

5.3 Złożenie

Układ elektryczny został umieszczony na uniwersalnej płytce prototypowej. Połączenia logiczne zostały umieszczone na górnej stronie płytki, na dolnej znajdują się przewody zasilające.



Rys. 5.3. Układ nadajnika na płytce prototypowej

Źródło: opracowanie własne

W trakcie wycinania płytki uniwersalnej doszło do uszkodzenia na jeden z krawędzi. Ze względu na brak materiałów nie wykonano nowej wersji. Uszkodzenie nie wpływa na działanie układu.

5.4 Kod źródłowy

5.4.1 Interfejs USART

Interfejs USART jest interfejsem szeregowym, który służy do komunikacji pomiędzy dwoma urządzeniami za pomocą sygnałów cyfrowych. Może być używany w trybie synchronizacji, jak i asynchronicznym.

Interfejs został zaimplementowany w konfiguracji asynchronicznej działającego z prędkością przesyłania wynoszącą 9600 bod (ang. baud). Poprzez ustawienie odpowiednich bitów rejestru UCSR0B uruchomiono opcje wysyłania - TXEN0 i odbierania danych - RXEN0 oraz uruchomiono dla tych opcji przerwania, odpowiednio: TXCIE0, RXCIE0.

```
void UsartInit(uint32_t baud)
{
	uint8_t speed = 16;
	baud = (F_CPU/(speed*baud)) - 1;
	UBRR0H = (baud & 0xF00) >> 8;
	UBRR0L = (baud & 0x00FF);
	UCSR0B |= (1 << TXEN0) | (1 << RXEN0) | (1 << RXCIE0) | (1 << TXCIE0);
}
```

Fragment kodu 4.4.1.1. Funkcji inicjująca interfejs USART

Źródło: opracowanie własne

Podstawową funkcją wysyłającą bajt jest UsartByteSend, która oczekuje w pętli, dopóki poprzednia operacja wysłania nie zostanie zakończona przerwaniem informującym o pomyślnym przesłaniu bajtu i zmienna usartTxBusy zostanie ustawiona na 1. Po tym do rejestru UDR0 przypisany jest kolejny bajt i natychmiastowo wysłany.

```
void UsartByteSend(uint8_t byte)
{
	while(usartTxBusy == 0);
	usartTxBusy = 0;
	UDR0 = byte;
}

ISR(USART_TX_vect)
{
	usartTxBusy = 1;
}
```

Fragment kodu 4.4.1.2. Funkcji wysyłająca bajt

Źródło: opracowanie własne

Funkcje do przesyłania ciągów znaków opierają się na wyżej opisanej funkcji, przekazując jako jej parametr kolejny bajt łańcucha, aż zostanie napotkany znak końca linii, który oznacza koniec wiadomości.

Odbieranie bajtu zostało skonstruowane na bazie przerwania, które wywołuje się co otrzymany bajt. W momencie, gdy liczba otrzymanych bajtów rxCount będzie równa desiredLength to zostaje wywołana funkcja callback, która otrzymuje długość oraz tablice bajtów wiadomości. System komunikacji korzysta z stałej wielkości wiadomości, która wynosi 32 znaki.

```
ISR(USART_RX_vect)
{
	volatile static uint16_t rxWritePos = 0;

	rxBuffer[rxCount] = UDR0;
	rxCount++;

	if(rxCount == desiredLength)
	{
		uint8_t message[desiredLength];
		memcpy(message, rxBuffer, desiredLength);
		callback(message, desiredLength);
		rxCount= 0;
		return;
	}
}
```

Fragment kodu 4.4.1.3. Przerwanie odbierające dane

Źródło: opracowanie własne

5.4.2 Sterowanie brzęczykiem

Sterowanie brzęczyka opiera się na przełączaniu stanu jednego pinu, stan wysoki – wydaje dźwięk, stan niski – nie wydaje dźwięku. W celu inicjalizacji, należy ustawić pint DDC4 w tryb wyjściowy, a następnie ustawić startową wartość wyjściową na stan niski.

```
void AudioInit()
{
	DDRC |= (1<<DDC4);
	PORTC &= ~(1<<DDC4);
}
```

Fragment kodu 4.4.1.3. Przerwanie odbierające dane

Źródło: opracowanie własne

Funkcja AudioBeep ma za zadanie kilku krotonie uruchomić brzęczyk. Składa się z pętli wykonującej instrukcji liczbę równą parametrowi number. W pierwszej kolejności ustawiany jest stan wysoki na 200 milisekund, a następnie stan niski. W przypadku gdy iteracja nie jest ostatnią to dodawane jest dodatkowe opóźnienie 100 milisekund.

```
void AudioBeep(int number)
{
	for(int i = 0; i < number; i++)
	{
		PORTC |= (1<<DDC4);
		_delay_ms(200);
		PORTC &= ~(1<<DDC4);
		if(i < number - 1)
			_delay_ms(100);
	}
}
```

Fragment kodu 4.4.1.3. Przerwanie odbierające dane

Źródło: opracowanie własne

5.4.3 Obsługa komend

Nadajnik wykorzystuje tą samą implementacje systemu komend co pojazd. Natomiast dodatkowo obsługiwana jest komenda Connect.

```
CommandInit(commandManagerRef);
CommandRegisterFunc(commandManagerRef, CMD_CONNECT, OnConnect);
CommandRegisterFunc(commandManagerRef, CMD_MOTORCONTROL, OnMotorControl);
CommandRegisterFunc(commandManagerRef, CMD_GETDATA, OnGetData);
```

Fragment kodu 4.4.1.3. Przerwanie odbierające dane

Źródło: opracowanie własne

Komenda Connect ma za zadanie sprawdzić poprawność komunikacji pomiędzy podsystemami. W pierwszej kolejności brzęczyk wydaje pojedynczy sygnał informujący o poprawnym odebraniu wiadomości z aplikacji. Następnie wysyłana jest ta sama komenda do pojazdu. Jeśli funkcja Nrf24Write zwróci wartość 0 to oznacza, że nie udało się nawiązać połączenia z pojazdem. Wysyłana jest komenda zwrotna z statusem Failed. Zwracana wartość 1 oznacza poprawną komunikacje, następnie zwracana jest do aplikacji komenda z statusem Success.

```
void OnConnect(CommandType Type, CommandStatus Status, uint8_t* data)
{
	AudioBeep(1);
	if(Nrf24Write(radioRef, data, COMMAND_LENGTH) == 0)
	{
		data[1] = CMDS_FAILED;
		UsartArraySend(data, COMMAND_LENGTH);
		return;
	}
	data[1] = CMDS_SUCCESS;
	UsartArraySend(data, COMMAND_LENGTH);
	AudioBeep(1);
}
```

Fragment kodu 4.4.1.3. Przerwanie odbierające dane

Źródło: opracowanie własne

Komenda MotorControl na poziomie nadajnika ma jedynie za zadanie pośredniczyć w przesłaniu wiadomości do pojazdu oraz zwrócenie statusu tej operacji.

```
void OnMotorControl(CommandType Type, CommandStatus Status, uint8_t* data)
{
	if(Nrf24Write(radioRef, data, COMMAND_LENGTH) == 0)
	{
		data[1] = CMDS_FAILED;
		UsartArraySend(data, COMMAND_LENGTH);
		return;
	}
	data[1] = CMDS_SUCCESS;
	UsartArraySend(data, COMMAND_LENGTH);
}
```

Fragment kodu 4.4.1.3. Przerwanie odbierające dane

Źródło: opracowanie własne

Komenda GetData ma za zadanie przesłać do pojazdu żądanie o dane telemetryczne. Po przesłaniu komendy i obsługi błędów. Nadajnik przechodzi w tryb odbierania wiadomości. W przypadku, gdy upłynie czas na odebranie wiadomości, do aplikacji zostanie zwrócona informacja o błędzie, a nadajnik powróci do trybu nadawania. W przypadku otrzymania wiadomości z pojazdu, nadajnik powróci w tryb nadawania oraz przekaże otrzymaną wiadomość do apliakcji.

```
void OnGetData(CommandType Type, CommandStatus Status, uint8_t* data)
{
	if(Nrf24Write(radioRef, data, COMMAND_LENGTH) == 0)
	{
		data[1] = CMDS_FAILED;
		UsartArraySend(data, COMMAND_LENGTH);
		return;
	}

	Nrf24OpenReadingPipe(radioRef, pipe);
	Nrf24StartListening(radioRef);

	uint16_t timeoutCounter = 0;
	while(Nrf24Available(radioRef) == 0)
	{
		timeoutCounter++;
		if(timeoutCounter >= 1600)
		{
			data[1] = CMDS_FAILED;
			Nrf24StopListening(radioRef);
			Nrf24OpenWritingPipe(radioRef, pipe);
			UsartArraySend(data, COMMAND_LENGTH);
			return;
		}
	}
	uint8_t receivedMsg[COMMAND_LENGTH];
	Nrf24Read(radioRef, receivedMsg, COMMAND_LENGTH);

	Nrf24StopListening(radioRef);
	Nrf24OpenWritingPipe(radioRef, pipe);

	UsartArraySend(receivedMsg, COMMAND_LENGTH);
}
```

Fragment kodu 4.4.1.3. Przerwanie odbierające dane

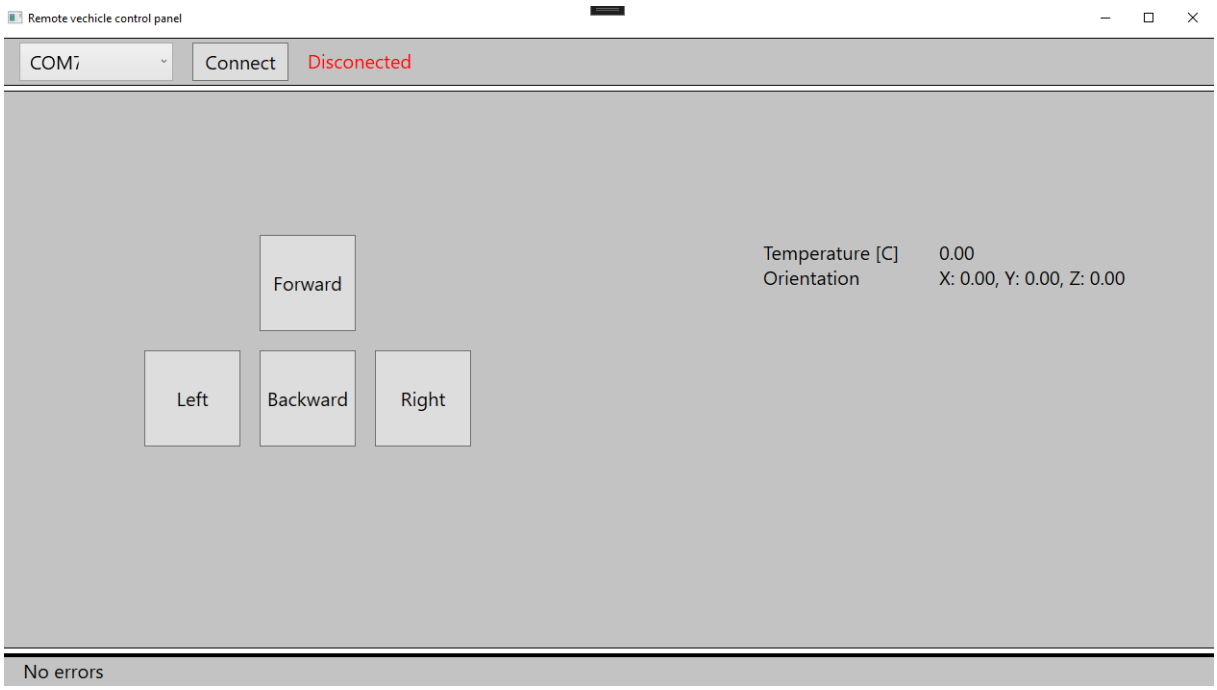
Źródło: opracowanie własne

6. Aplikacja okienkowa

Aplikacja okienkowa służy do zarządzania pojazdem. Zadaniem jej jest udostępnienie użytkownikowi prostego interfejsu sterowania i monitorowania pojazdu. Aplikacja została utworzona przy użyciu środowiska .Net Core w języku C#. Zdecydowano się na to środowisko z względu na łatwość tworzenia aplikacji na system operacyjny Windows oraz możliwość kompilacji na Linux oraz MacOS sprawiając, że aplikacja jest dostępna dla dużego grona odbiorców. Jako silnik graficzny wykorzystano WPF (ang. Windows Presentation Foundation), które pozwala tworzyć interfejs użytkownika w formie tekstowej XAML (ang. Extensible Aplication Markup Language) oraz poprzez graficzne układanie kafelków, którymi są elementy interfejsu, np. przyciski, tekst.

6.1 Interfejs użytkownika

Interfejs składa się z trzech paneli. Górny panel odpowiada za połączenie z nadajnikiem. Użytkownik wybiera port USB, do którego chce się podłączyć, następnie wciska przycisk Connect, który powoduje wysłanie komendy Connect do nadajnika. W przypadku pomyślnego połączenia napis Disconnected zostanie zamieniony na zielony napis Connected.



Rys. 6.1. Interfejs użytkownika aplikacji sterującej pojazdem

Źródło: opracowanie własne

Drugi panel służy do sterowania pojazdem. Po pomyślnym połączeniu można rozpocząć wydawanie poleceń poruszania. Przyciski Forward, Backward, Left, Right służą do wydawania komendy MotorControl do sterowania silnikami. Istnieje możliwość sterowania pojazdem przy pomocy klawiszy W, S, A, D , które układem odpowiadają przyciskom w oknie aplikacji. W momencie wciśnięcia którego z nich zostają one podświetlone na zielono. Po prawej stronie panelu znajdują się dane telemetryczne aktualizowane co pewien okres. Dolny panel służy do wypisywania błędów, np. o komendzie, która została wykonana niepoprawnie, za długim czasie oczekiwania na reakcje lub przerwaniu połączenia z pojazdem.

6.2 Diagram klas

6.3 Kod źródłowy

6.3.1 Menadżer komunikacji

Po wciśnięciu przycisku Connect aplikacja wywołuje metodę klasy CommunicationManager, która ma za zadanie nawiązanie połączenia z wskazanym portem szeregowym. W tym celu wykorzystywana jest klasa z biblioteki System.IO.Ports – SerialPort, która umożliwia wysyłanie oraz odczyt danych z portu. Po otworzeniu połączenia z prędkością 9600 bod dodawane jest zdarzenie, które asynchronicznie odbiera przychodzące wiadomości. W przypadku niepoprawnego połączenia metoda przerywa wywoływanie.

```
public bool Start(string portName)
{
    try
    {
        serialPort = new SerialPort(portName, 9600);
        serialPort.Open();
        serialPort.DataReceived += SerialPort_DataReceived;
    }
    catch(ObjectDisposedException)
    {
        return false;
    }
```

```
    return true;
}
```

Fragment kodu 6.3.1.1 Metoda rozpoczynająca komunikacje

Źródło: opracowanie własne

Metoda wywoływana jest w momencie wystąpienia zdarzenia pojawienia się nowej wiadomości dostępnej do odczytu na porcie szeregowym. Pierwszą instrukcją jest odczyt całej linii z portu, ponieważ systemy operacyjne implementuje komunikacje poprzez system plików, więc nowe wiadomości pojawiają się w pliku portu szeregowego. Następnie usuwany jest pierwszy znak dodawany automatycznie w procesie komunikacji poprzez USB. Na końcu funkcja wywołuje zdarzenie, do którego jest podłączona metoda z menadżera komend, która przetwarza informacje.

```
void SerialPort_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
      string result = serialPort.ReadLine().Remove(0, 1);
      OnReceived?.Invoke(result);
}
```

Fragment kodu 6.3.1.2 Metoda odbierające dane z portu szeregowego

Źródło: opracowanie własne

6.3.2 Menadżer komend

7. Testowanie systemu

8. Podsumowanie i wnioski

Celem niniejszej pracy inżynierskiej było zaprojektowanie, wykonanie oraz zaprogramowanie systemu zdalnie sterowanego pojazdu gąsienicowego, na który składają się trzy podsystemy: pojazd, nadajnik oraz aplikacja okienkowa. Pracy przyświecał cel stworzenia prostego do wykonania i obsługi rozwiązania do sterowania pojazdami zdalnymi, przez co nadajnik jest oddzielnym układem, który można podłączyć do dowolnego komputera z systemem operacyjnym Windows. Aplikacja wykonana w środowisku Net Core pozwala na korzystanie z niej na różnych wersjach systemu. W pracy zrealizowano wszystkie założone cele. W pracy przedstawiono opis wykonanie każdego z podsystemów jak dobór części elektronicznych, schemat elektryczny oraz programowanie. Zaprogramowanie aplikacji okienkowej, która była niezależnym tworem z pozostałymi podsystemami, co wymusiło stworzenie wspólnego protokołu komunikacji.

Ważnym elementem pojazdów zdalnie sterowanych, a który nie został zaimplementowany jest kamera wideo, przesyłająca obraz do jednostki sterującej. Zagadnienie to wymaga znacznego nakładu pracy, przez co zrezygnowano z wykonania go, jednakże jest element, który należałoby dodać w ramach prac rozwojowych opracowanego systemu. Zagadnieniem wartym uwagi jest również system zarządzania ruchem gąsienic poprzez wyliczanie trajektorii ruchu na bazie danych wejściowych i dobór prędkości w celu osiągnięcia najbardziej optymalnego wykonania komendy. Kolejnym krokiem rozwoju projektu byłoby zaprojektowanie dedykowanego podwozia pozwalającego na swobodne poruszanie w trudnym terenie.

9. Bibliografia

[1] Dokumentacja Atmega328P, tutaj link

[1]. Mikrokontroler AVR - ATmega328P-U DIP, https://botland.com.pl/avr-w-obudowie-tht/1264-mikrokontroler-avr-atmega328p-u-dip-5903351249928.html

[2] NRF24L01 Wireless Data Transmission Module, http://naradaelectronics.rw/product/nrf24l01-wireless-data-transmission-modulegreen-10pin/

[3] Akcelerometr 3-osiowy MPU-6050 /GY-521, https://abc-rc.pl/product-pol-6572-Akcelerometr-3-osiowy-MPU-6050-GY-521-zyroskop-na-I2C.html?query_id=1

[4] L293D - dwukanałowy sterownik silników, L293D - dwukanałowy sterownik silników 36V/0,6A - 5szt. Sklep Botland

[5] Akumulator Li-Pol 3.7V, https://botland.com.pl/akumulatory-li-pol-1s-37v/15607-akumulator-li-pol-akyga-750mah-1s-37v-zlacze-jst-bec-gniazdo-50x34x44mm-5904422324193.html

[6] Tamiya – podwozie czołgu, Tamiya 70098 - uniwersalna płytka montażowa Sklep Botland

[7] Układ scalony CH340, https://abc-rc.pl/product-pol-9910-Uklad-scalony-CH340-uklad-komunikacji-USB.html?query_id=2

[8] Mapa rejestrów MPU-6050 https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6500-Register-Map2.pdf