Teknisk Dokumentation

Tävlingsbil

Gustaf Sjögemark

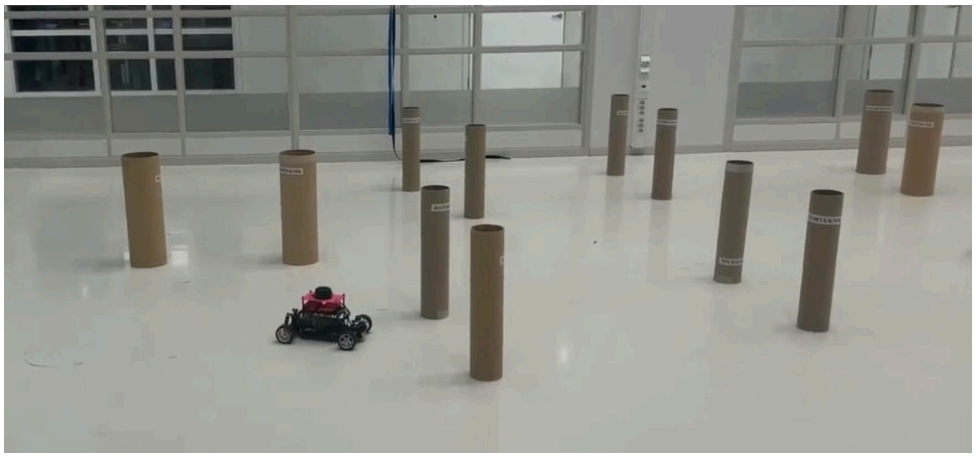
Version 1.1

Version	Datum	Utförda förändringar	Utförd av	Granskad
1.1	2024-12-19	Kompletterad	Hela gruppen	Gustaf Sjögemark

Innehåll

1 Inledning

I detta dokument beskrivs projektgrupp 9:s projekt i kursen TSEA29 (Konstruktion med mikrodatorer), hur bilen ser ut och en exempelbana visas i figur 1. Målet med projektet var att bygga en bil som autonomt kan köra en bana bestående av koner enligt *Banspecifikationen*. Syftet med detta dokument är att möjliggöra vidareutveckling samt möjligheten att bygga bilen utifrån informationen från dokumentet.



Figur 1: Bild på produkten körandes en bana

1.1 Bakgrund och syfte

Kursen bygger vidare på tidigare hårdvarunära kurser som *Datorteknik*, *Digitalteknik* och *Datorkonstruktion*. I likhet med *Datorkonstruktion* är *Konstruktion med mikrodatorer* en projektkurs där deltagarna får möjlighet att tillämpa sina kunskaper i praktiken. I projektet får deltagarna öva på problemlösning i en friare miljö än tidigare labbkurser. Det ger möjlighet till att lösa problemen som uppkommer på flera olika sätt. Projektets omfattning gör också att en större mängd problem kan uppstå där det ibland är svårt att avgöra om problemen beror på hårdvara eller mjukvara. Detta gör ett metodiskt arbetssätt där små ändringar görs åt gången avgörande för att projektet ska röra sig framåt. Bakgrundskunskaper finns i form av assembly, lågnivåprogrammering (C) samt programmering i hårdvarubeskrivande språk (VHDL).

Förkunskaper i mjukvaruområdet inkluderar grundläggande imperativ och funktionell programmering i Python, objektorienterad programmering i Java, datastrukturer och algoritmer i C++, samt operativsystemsprogrammering i C. Projektet tillåter projektdeltagarna att omsätta de tidigare kunskaperna till ett enda projekt, där alla bitar är nödvändiga.

2 Produkten

I figur 2 syns hur produkten ser ut. Utifrån figuren kan man se att produkten kan delas upp i fyra lager. På det understa lagret finns batteriet, hjulen, motorn, hallsensorer och styrservon. Över det befinner sig två AVR-processorer (Atmega1284P) och ett gyroskop. AVR:erna används för att skicka kommandon till motor och styrservo samt hämta data från sensorerna till Raspberry PI:n som finns på det tredje lagret. På det översta lagret finns LiDAR:n som används för se hur produktens omgivning ser ut.

Produkten används för att köra en bana enligt *Banspecifikationen* där den kan köra banan autonomt.

Figur 2: Bild på produkten

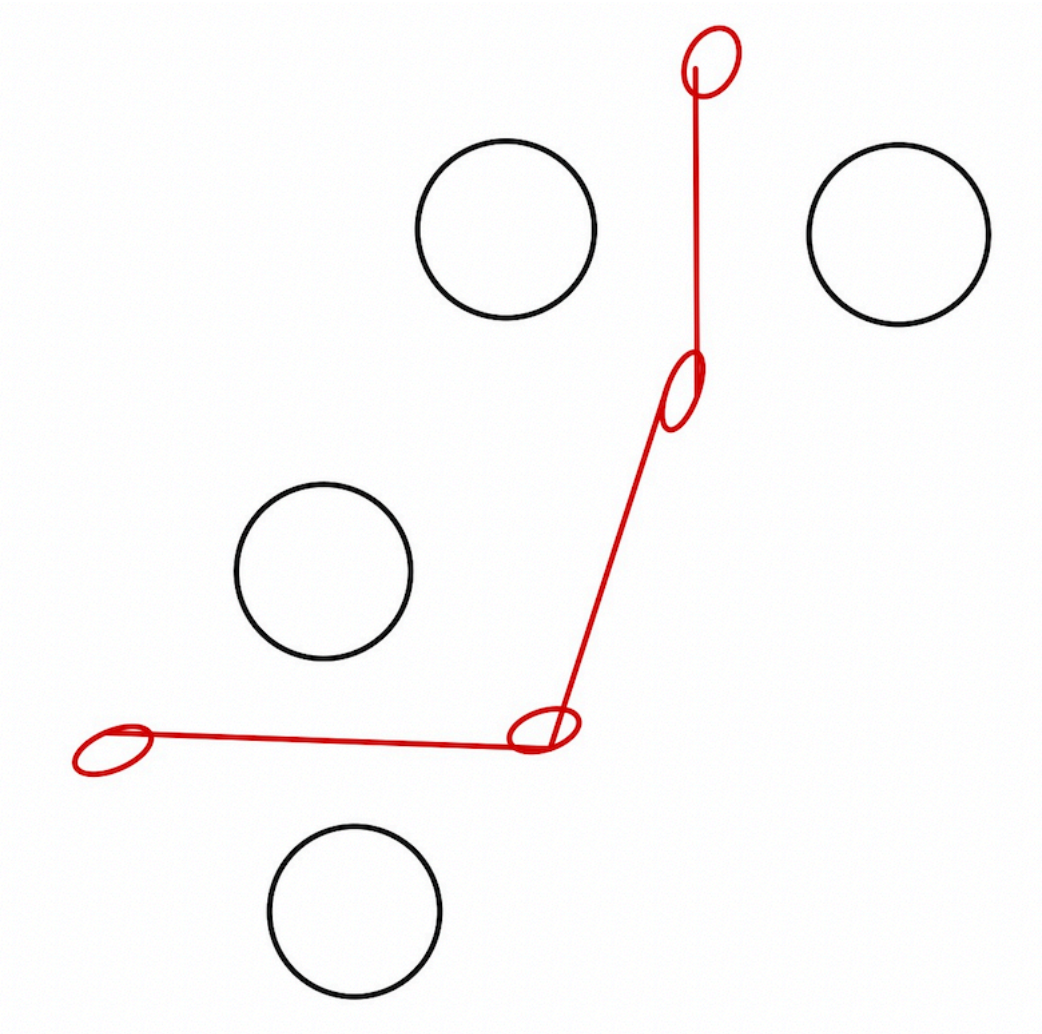
3 Teori

För att kunna köra autonomt har bilen en styralgoritm. Styralgoritmen anpassar styrutslaget baserat på indata från Lidarn. Lidarn identifierar punkter. I kommunikationsmodulen tolkar vi kluster av identifierade punkter som koner. Par av koner betraktas som koner som styralgoritmen försöker köra igenom. Den implementeras i filen `Kommunikationsmodul/car.py` under namnet `autopilot`. I den används en global karta över alla koner som just nu syns till för att hitta den grind bilen ska köra mot. Konerna paras ihop till grindar och sorteras enligt den vinkel bilen behöver svänga för att komma genom den. Dessutom filtreras grindarna så att enbart de med ett avstånd mellan 200 mm och 2000 mm räknas. Den grind med minimal vinkel rakt framifrån bilen väljs som mål. Styrservots vinkel beräknas då enligt listing 1.

```
angle = 127 + self.turn_coeff * math.atan2(next_gate_x, next_gate_y)
```

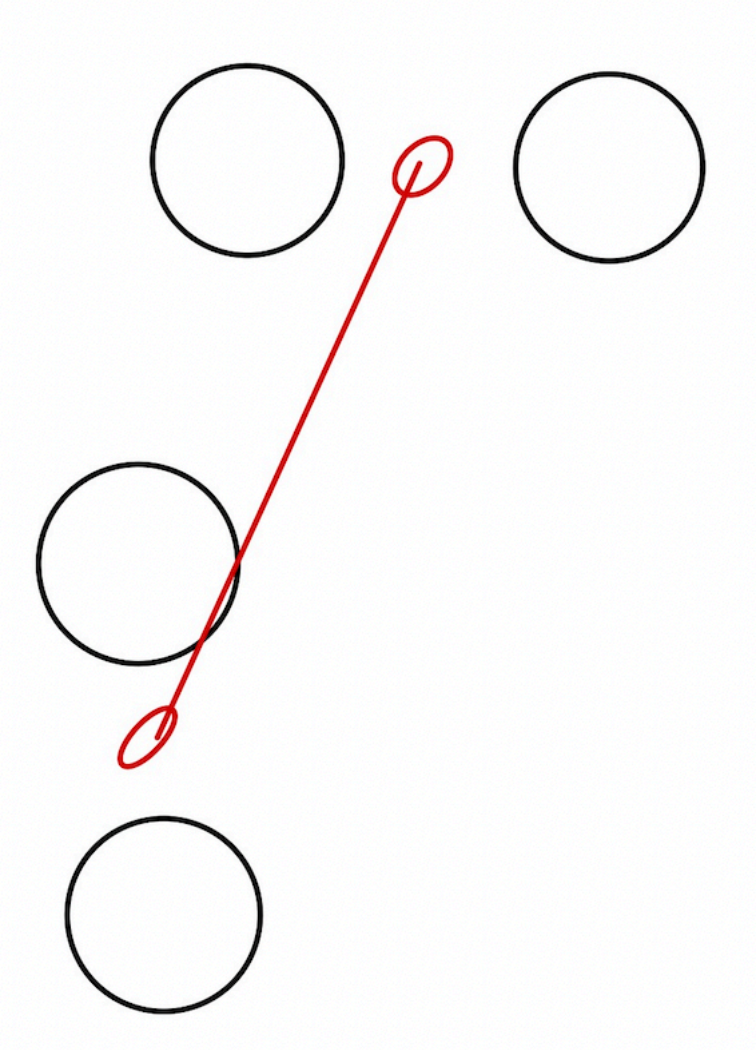
Listing 1: Beräkning av styrutslag

Bilen svänger alltså kraftigare desto större vinkel den har till nästa grind. När algoritmen har lokaliserat en grind som är nästa mål så styr algoritmen in sig på normalen 20 cm framför porten och när den punkten är passerad siktar bilen på normalen 20 cm bakom porten enligt figur 3.



Figur 3: Hur bilen nu bygger vägen mellan konerna

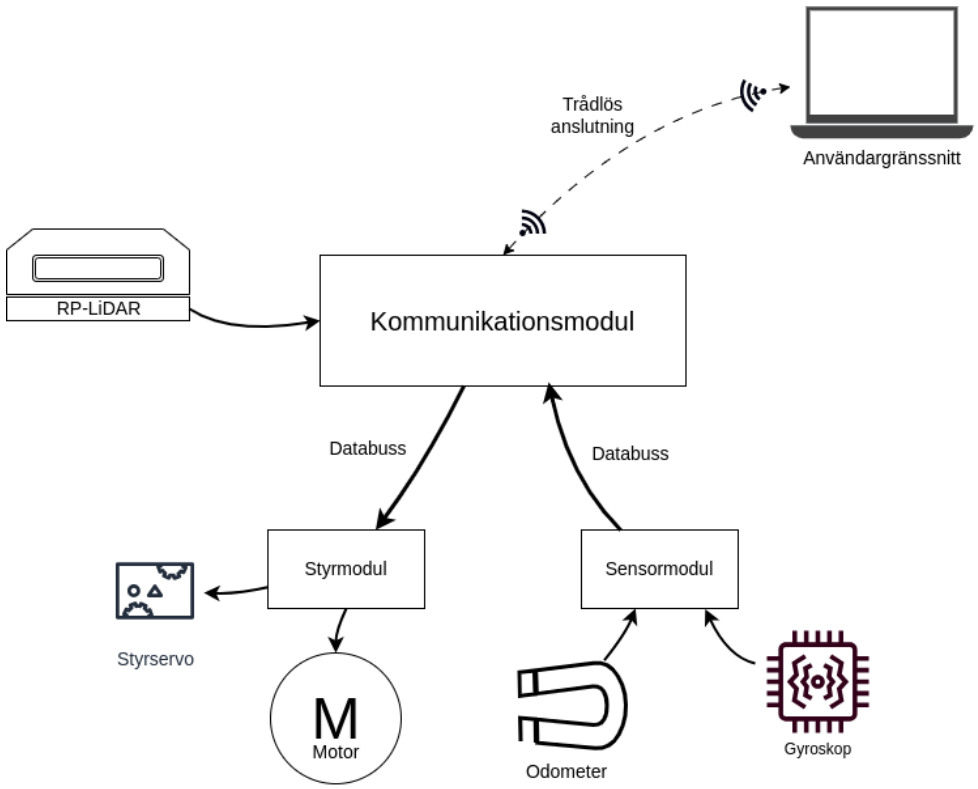
Anledningen till varför detta gjörs kan utläsas ur figur 4. En mer simplistisk algoritm hade riskerat att bilen körde på koner trots att den körde rakt mot sitt mål. Om bilen istället siktar in sig på normalen undviks kollision med koner i om vinkeln är ogynnsam.



Figur 4: Illustration av hur bilen tidigare hittade vägen

4 Systemet

En översikt av hela systemet ges i figur 5. En riktad pil representerar ett flöde av information. Det är evident att all information från alla sensorer propageras upp till Kommunikationsmodulen där den behandlas. På samma sätt propageras alla styrkommandon till Styrmodulen som exekverar dem. GUI:t har en tvåsidig kommunikation med Kommunikationsmodulen. Det kan be Kommunikationsmodulen om information och vänta på ett svar. Den kan dessutom asynkront skicka kommandon till Kommunikationsmodulen för att exempelvis manuellt styra bilen eller sätta på eller stänga av självkörning.



Figur 5: En översiktlig systemskiss med samtliga delsystem och gränssnitt.

5 Modulerna

Nedan beskrivs de olika modulerna som bygger upp bilen.

5.1 Kommunikationsmodulen

Kommunikationsmodulen styr över styrmodulen och tar hand om majoriteten av alla beräkningar. Den består av en Raspberry Pi 3B+ och kommunicerar med Styrmodulen och Sensormodulen via simpla seriella protokoll. I båda fallen är kommunikationen ensidig: Kommunikationsmodulen skickar enbart kommandon till styrmodulen och tar enbart emot datapaket från sensormodulen. Kommandon till styrmodulen hanteras med ett eget CLI-program, `steer`, som formaterar datan korrekt och skickar den via styrmodulens seriella anslutning.

Lidardatan läses in av ett program från tillverkaren som startar upp lidarn och skriver ut data till dess output.

Resterande kod på kommunikationsmodulen består av en websockets-server, skriven i Python, som internt hanterar all bilens tillstånd, behandlar sensordata och lidardata, och styr bilen. Den tar också emot kommandon från användargränssnittet så att denna ska kunna styra bilen. Slutligen förmedlar den sensordata till GUI:t på efterfråga. All kod på bilen har sitt ursprung i klassen `Car`. Denna skapar asynkrona processer som i sin bland annat läser sensordata och styr bilen.

Internt hanteras alla händelser genom att sätta in ett `event` i en global event-kö. Dessa hanteras därefter av en central process som, beroende på meddelandet, gör olika saker. Alla sådana event består av en lista där första elementet är eventets namn, medan det andra elementet är en dictionary av argument. Alla meddelanden redovisas i tabell 1. Dessa meddelanden kan läggas till i kön av alla processer som är medlemmar i `Car`, och eftersom de hanteras i ordning undviks dataraces. I kommunikationsmodulen finns filen `car_settings`. Där sätts inställningar för main-loopen. Konstanter defineras också där.

Tabell 1: Kommunikationsmodulens meddelanden

Struktur	**Beskrivning**
`["auto", {"auto": bool, "drive": bool}]`	Sätt på eller stäng av manuell och autonom körning
`["brake", {"brake": bool}]`	Kontrollera bilens broms.
`["exit", {}]`	Stäng av bilen och koppla bort GUI-klienten.
`["get_cones", {}]`	Skicka tillbaka de koner som nu syns till GUI:t samt punkten som bilen siktar på.
`["get_data", {}]`	Skicka avstånd och vinkel till punken bilen siktar på samt nuvarande styrutslag och styrvinkel.
`["get_dots", {}]`	Skicka alla punkterna lidarn ser.
`["get_sensors", {}]`	Skicka tillbaka aktuell sensordata till GUI:t
`["steer", {"angle": int}]`	Justera styrutslaget på hjulen.
`["throttle", {"reverse": bool, "velocity": int}]`	Justera hastigheten och riktningen som bilen kör med.

Strukturen på eventen gör det möjligt att serialisera dem via json, vilket är hur de förmedlas mellan Kommunikationsmodulen och GUI:t.

Fyra av eventen får bilen att skicka data till GUI:t, formatterat enligt tabell 2. Notera även att alla koordinater förmedlas enligt ett kartesiskt koordinatsystem med bilen i origo. Positivt x går åt höger om bilen och positivt y går framåt.

Tabell 2: Svar på get_*-meddelanden

Meddelande	Svar	Beskrivning

`get_cones`	`["get_cones", {`
	`  "cones": [(int, int, int)], # (x, y, storlek)`
	`  "next_center": (int, int), # (x, y)`
	`}]`
		Returnerar en lista av koner i formatet `(x, y, storlek)` samt en punkt som bilen siktar på just nu.

`get_dots`	`["get_dots", {` `  "dots": [(float, float)]`	Returnerar alla punkterna lidarn ser i polära koordinter.

`get_sensors`	`["get_sensors", {`
	`  "velocity": int,`
	`  "angular_rate": int,`
	`  "delta_time": int,`
	`  "angle": int,`
	`}]`
		Returnerar en dict med sammanfattad sensordata samt tiden under vilken denna var insamlad.

`get_data`	`["get_data", {`
	`  "distance": float,`
	`  "angle_to_gate": float,`
	`  "throttle": int,`
	`  "turn": int,`
	`}]`
		Returnerar en dict med data på vart bilen siktar och med vilken hastighet och styrvinkel den kör.

5.1.1 Steer-kommandot

`steer` är ett kommando som formatterar och skickar ett styrkommando seriellt via en angiven port. Det används av kommunikationsmodulen för att interagera med styrmodulen. Kommandot tar fyra argument:

`--port (-p)` Bestämmer vilken seriell port som signalen ska skickas genom.
`--velocity (-v)` Sätter hastigheten till allt mellan 1-255.
`--angle (-a)` Sätter styrutslaget på hjulen.
`--direction (-d)` Sätter riktningen som bilen kör mot samt broms. `1` ger körning framåt, `2` ger körning bakåt, och `3` ger broms.

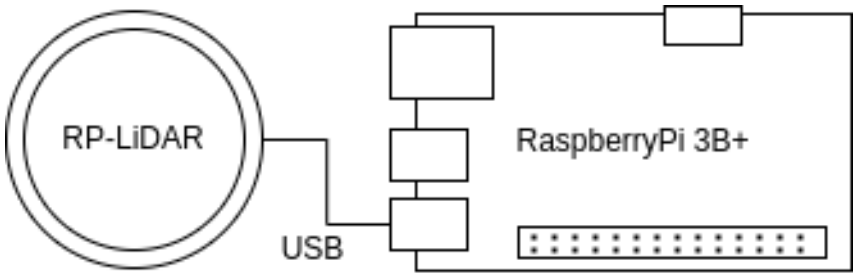
Datan skickas med formatet `[0, velocity, angle, direction]` där `0` agerar som en synkpuls.

5.2 Lidar

För att bilen ska kunna identifiera konerna i sin närhet används en RP-LiDAR från företaget Slamtec[1]. LiDARn snurrar ungefär 10 varv per sekund och mäter upp ungefär 400 punker per varv. Varje varv som LiDARn snurrar behandlas datan med en algoritm som grupperar närliggande punkter och avgör om de fyller kraven för att räknas som en kon. För att en uppsättning punkter ska räknas som en kon får maxavståndet mellan två på varandra följande punkter vara 100mm och avståndet mellan första och sista punkten får maximalt vara 250mm. Gränsen mellan om en kon skulle identifieras som en stor eller liten kon sattes till 120 mm. Dessa avstånd(100mm, 250mm samt 120 mm) är inte baserade på verkligheten utan istället vad som gav rätt identifiering oftast. På avstånd över 2 meter bestämms ingen storlek på konerna då det är så långt bort att det ofta blev fel. Algoritmen för konidentifiering körs löpande i kommunikationsmodulen som vidare beskrivs i

kapitel 5.1. I figur 6 syns hur LiDAR är kopplad med USB till kommunikationsmodulen (RaspberryPi 3B+).

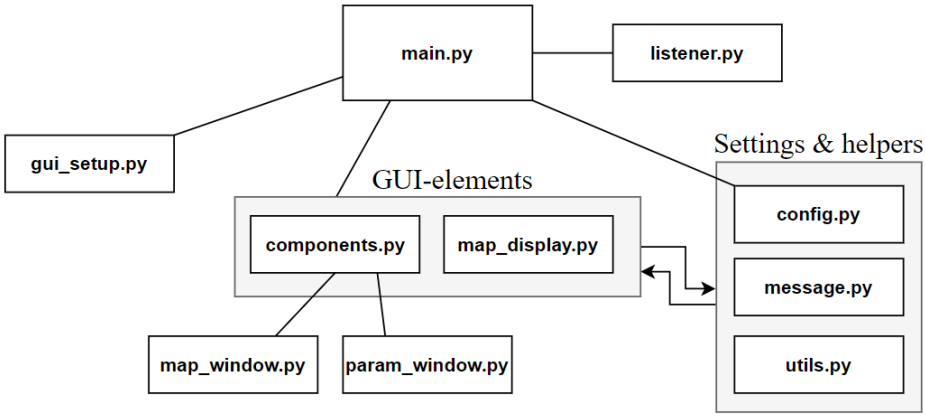
För att köra LiDARn används ett färdigt program, `simple_grabber`, från tillerkaren Slamtec som ingår i open-source-utveklingspaketet `rplidar_sdk`[2] som modifierats till att bara skriva ut vinkel och avstånd för varje mätpunkt. Det programmet är skrivet i C++ och körs från Kommunikationsmodulen.



Figur 6: Ritning för koppling mellan LiDAR och Kommunikationsmodulen.

5.3 Användargränssnittet

För att en användare ska kunna kontrollera bilen och observera dess beteende har ett grafiskt användargränssnitt utvecklats. I detta GUI kan användaren manuellt styra bilen, observera drift-information, samt stänga av och på autonom körning. Användaren kan även modifiera parametrar som påverkar bilen i både manuellt och autonomt läge. Användargränssnittet byggs och hanteras med hjälp av tio Pythonmoduler (se figur 7 nedan). Som namnet antyder är `main.py` huvudmodulen som i sin tur exekverar eller tar hjälp av resterande moduler.



Figur 7: Blockschema för filerna som används för användargränssnittet.

Tack vare användningen av asyncio och websockets-biblioteket behövs inte threading för att exekvera flera saker parallellt. Istället är main-funktionen, och alla funktioner som behöver köras parallellt definierade som asynkrona. Anropen till dessa föregås av ’await’ vilket pausar exekveringen av den aktuella metoden tills dess att den asynkrona metoden är färdig. Fördelen är att andra asynkrona metoder kan exekvera under denna väntetiden vilket eliminerar mycket busy-waiting som annars hade saktat ner GUI:t och kommunikationen med bilen.

`config.py` innehåller diverse parametrar, såsom gasstyrka och vilket styrutslag som räknas som ”mitten”, samt exempelvis nätverksadress.

`main.py` använder nätverksadressen i `config.py` för att via websockets koppla upp sig till Raspberry-Pi:en, varpå `gui_setup` och filerna i lådan 'GUI-elements' i figur 7 exekveras. Genom olika knapptryck i GUI:t kan även `map_window.py` och `param_window.py` exekveras, vilket öppnar separata fönster med en stor karta respektive en parameter-lista.

Filerna i lådan 'Settings & Helpers' innehåller bland annat det ovannämnda i `config.py` men även diverse hjälpfunktioner. `message.py` paketerar meddelanden i json.format (se listing 3 nedan samt tabell 1 och tabell 2 ovan) och *utils.py* som innehåller diverse simulerings-metoder (se listing 3 nedan.)

```
def steer(angle: int) -> str:
    return json.dumps(["steer", {"angle": angle}])
```

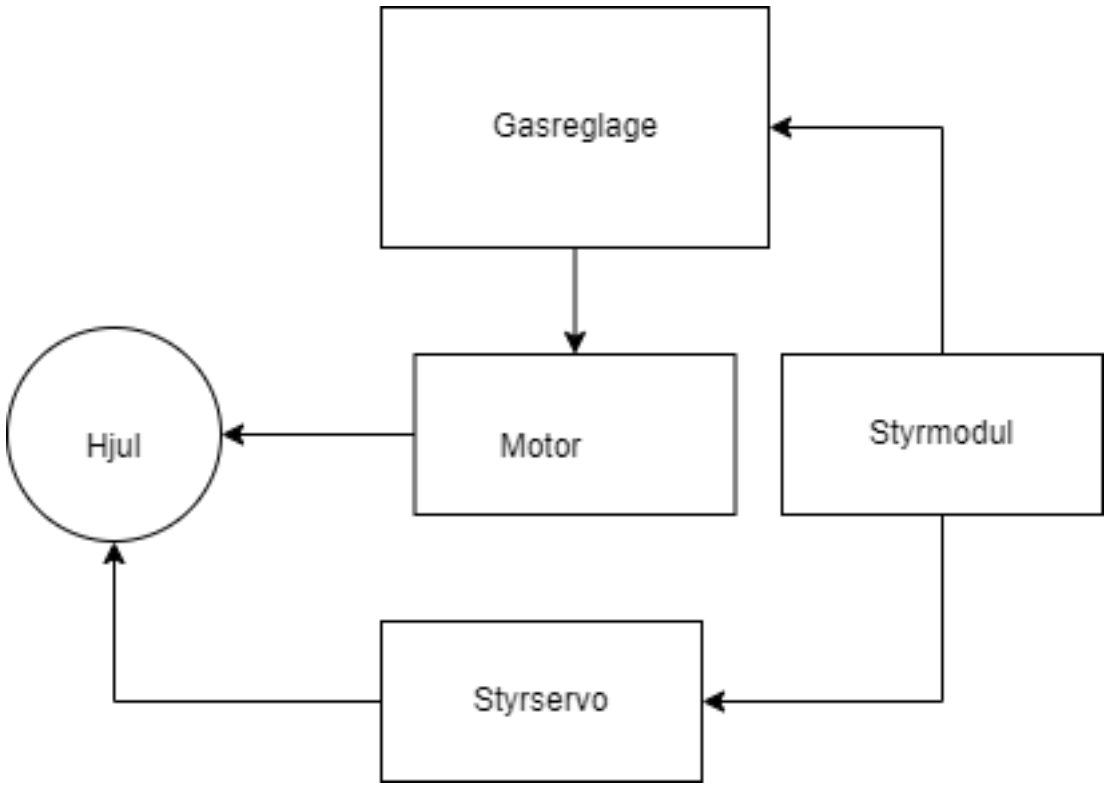
Listing 2: Kod i modulen `message.py`.

```
def simulate_elapsed_time(root: tk.Tk, elapsed_time_var: tk.DoubleVar) -> None:
    """Simulerar passerad tid"""
    elapsed_time_var.set(elapsed_time_var.get() + 1)
    root.after(
        1000, lambda: simulate_elapsed_time(root, elapsed_time_var)
    )  # Schemalägg nästa förändring
```

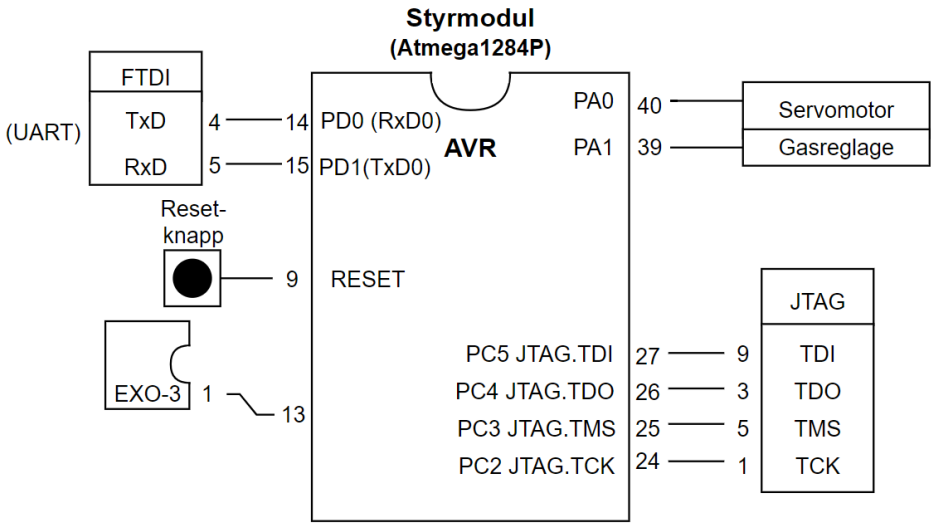
Listing 3: Kod i modulen `utils.py`.

5.4 Styrmodulen

I figur 8 visas översiktligt hur styrmodulen är uppbyggd. För en mer detaljerad vy, se figur 9. Styrmodulen får instruktioner från kommunikationsmodulen om hur den ska styra styrservot och vilket gaspådrag den ska ge motorn. Dessa instruktioner verkställer styrmodulen genom att skicka pulser styrda av timers till styrservot och gasreglaget.



Figur 8: Illustration av styrmodul



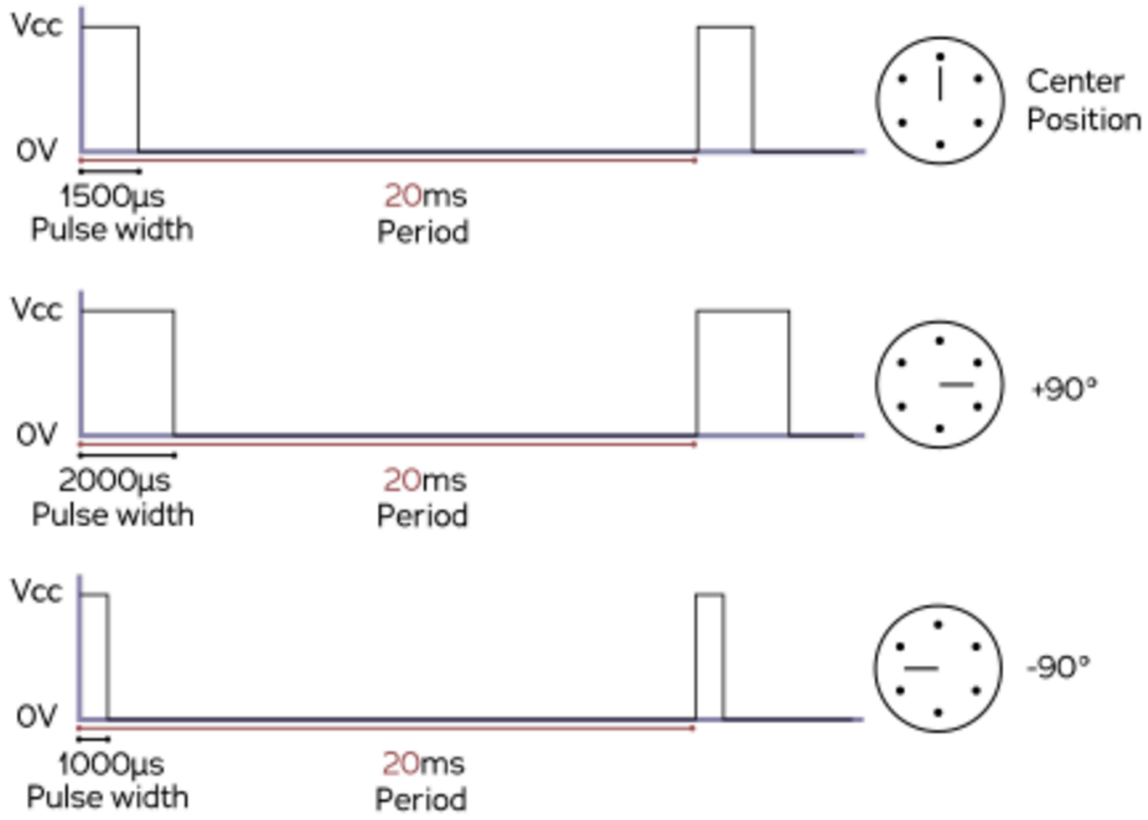
Figur 9: Kopplingsschema för styrmodulen.

5.4.1 Virning

Styrmodulen består av en ATmega1284P som är fäst på ett virkort tillsammans med sensormodulen. På virkortet finns en extern klocka på 16MHz som processorerna drivs med. Strömförsörjning och jordning ges via ett IDC-don på virkortet. Där finns även kontakt till motorn (`PIN5`) och styrservot (`PIN8`). Dessa två drivs via två separata PWM-signaler som båda genereras med interna timers. Kommunikationen mellan Styrmodulen och Kommunikationsmodulen sker via USART (se kapitel 5.6). Utöver detta har virkortet JTAG-uttag för att kunna flasha AVR:erna samt FTDI-uttag för den seriella UART-kommunikationen. Alla virningar som går från styrmodulens AVR går till ett IDC-don (på AVR:en kontakterna 5-10). `PB3` på AVR:en är virat till `PIN5` på IDC-donet (gasreglage). `PD2` på AVR:en är virat till `PIN6` på IDC-donet som styr över rotationsriktning på hjulen. `PC1` på AVR:en är virat till `PIN7` på IDC-donet (Broms). `PA4` på AVR:en är virat till `PIN8` på IDC-donet (styrservo).

5.4.2 Styrservot

Styrservot styrs via ett 8-bitars tal som skickas från Raspberry Pi:n till styrmodulen. Storleken på 8-bitarstalet modulerar pulsbredden på PWM-signalen som AVR:en skickar till styrservot. (Se figur 10.) 127 tolkas som ”inget utslag”, dvs en pulsbredd på 1500μs. Ett tal mellan 1-126 motsvarar en vänsterutslag (där 1 är maximalt utslag, med minsta pulsbredd 1000 us). På analogt sätt fungerar det för högerutslaget (men då är gränserna 128-255, där 255 ger pulsbredden 2000μs). Anledningen till att 0 inte finns med är för att Den är reserverad till en synkpuls. För att generera avbrott för styrservot används Timer 1. Timer 1 är i CTC-läge (clear timer on compare match) där räknaren nollställs vid en jämförelse med ett förinställt värde. Vi sätter en prescaler på clk/8, detta initeras i den följande raden `TCCR1B = (1 << WGM12) | (1 << CS11)` (se tabell 14-6 i [3])



Figur 10: Hur pulsbredden på PWM-signalen påverkar styrutslaget, figuren är hämtad från [4]

5.4.3 Gasreglaget

Gasreglaget styrs också av en PWM-signal, men på ett annat sätt. Motorn använder en signalens duty-cykel för att reglera sitt gaspådrag. Som exempel om dutycykeln är 75% av sitt maxvärde $0.75 \times 255 \approx 191$ motsvarar det ett gaspådrag av 75% av max. Värdet skickas till styrmodulens ATmega1284P från Raspberry Pi:n. Värdet på cykeln sätts vid avbrott genererade av TIMER0. TIMER0 räknar upp med en prescaler clk/8 och är inställd på Snabb PWM. När TIMER0 räknat upp till 255 uppstår ett avbrott. I avbrottsrutinen sätts `OCR0A` till det nya värdet.

5.4.4 Brake & direction

I C-koden som är flashad till styrmodulens AVR (`Styrmodul/styrservo/styrservo/main.c`) finns funktionen `set_direction` som hanterar riktning och bromsning för styrmodulen baserat på värdet av den globala variablen `directionState`. Funktionen inaktiverar eller aktiverar broms och riktningspinnar på `PORTC` och `PORTD` enligt följande lägen:

Framåt (1) Stänger av bromsen (genom att sätta `PC1` låg) och sätter riktning framåt.

Bakåt (2) Stänger av bromsen och sätter riktning bakåt (genom att sätta `PD2` hög.)

Bromsa (3 & 4) Stoppar gasen (genom att sätta dutycykeln till 0) och aktiverar bromsen (genom att sätta `PC1` hög).

5.5 Sensormodulen

Sensormodulen består av en ATmega1284P. Sensormodulen skickar data som den får från sina sensorer till kommunikationsmodulen. Modulen delar virkort med styrmodulen. På virkortet finns tillgång till ström, jordning, en 16MHz extern klocka uttag till UART, JTAG och IDC-don. I Sensormodulen ingår två olika typer av sensorer, ett gyro och två hall-sensorer.

5.5.1 Virning

PA0 på AVR:en är virat till gyrot och mottar gyrots utspänning. AREF (PIN 32) på AVR:en är virad till 5V på kretskortet för referensspänning. PD1 (lyssnar på vänster odometer) och PD2 (lyssnar på höger odemeter) på AVR:en är virat till IDC-don uttag 9 och 10 på IDC-donet som ger avbrott då en magnet på höger respektive vänster odometer sätts låg.

5.5.2 Gyro

Gyrot används för att beräkna vinkelhastigheten. Då vinkelhastighet är noll skickar den ut 2,5 volt på OUTAR pinnen. Gyrot har känsligheten 0.00667V/deg/s. Vid variation av vinkelhastighet ändras alltså spänningen på OUTAR i proportion till känsligheten. Vid rotation åt vänster minskar spänningen på OUTAR, vid rotation åt höger ökar volten på OUTAR. Gyrots OUTAR-pin är virad till AVR:ens PA0. Datan från gyrot behandlas i sensormodulen till en vinkelhastighet som skickas till sensormodulen.

Gyrot initieras enligt koden i listing 4, där ADLAR väljer ut de åtta höga bitarna från konversionen. MUX0 definierar ADC0 som används som input (vilket korresponderar till pinne 40). REFS0 sätter referensspänningen. I bilens fall är detta 5V.

Då ADEN sätts till 1 möjliggörs ADC-konvertering och ADIE tillåter ADC-avbrott. Anledningen till att ADPS1 sätt är för att den ska ha rätt frekvens.

För att kunna skicka information mellan AVR:en och Raspberry PI:n används avbrott. När en konvertering blir klar genereras ett avbrott och datan skickas till Pi:n.

```
ADMUX = (1 << ADLAR) | (0 << MUX0) | (1 << REFS0);
ADCSRA = (1 << ADEN) | (1 << ADIE) | (1 << ADPS1) | (1 << ADPS2);
```

Listing 4: Initiering av ADC till gyrot.

5.5.3 Hallsensorer

Hallsensorerna initieras enligt listing 5. Hallsensorerna ger flytande output, därav används pull-up resistor för att sätta signalen till aktivt hög. När en magnet passerar jordas signalen och det är därför resultatet läses av vid fallande flank. Då sensorerna studsar en hel del måste de läsas av ett kort moment efter den initiala flanken, när signalen är stabil.

```
//Enable interrupt 1 & 0.
EIMSK = (1 << INT1) | (1 << INT0);
// Set interrupt 1 & 0 to trigger for a falling-edge.
EICRA = (1 << ISC11) | (1 << ISC01);
// Set PORTD[3:2] to inputs.
DDRD = (0 << PORTD3) | (0 << PORTD2);
// Enable the pull-up resistors for these ports.
4 PORTD = (1 << PORTD3) | (1 << PORTD2);
```

Listing 5: Initiering av hall-sensorer.

5.6 USART

Kommunikationen mellan Atmega1284P och Raspberry Pi sker via USART (Universal synchronous and asynchronous receiver-transmitter). På bägge AVR:erna används USART0.

5.6.1 Initiering

Vid initieringen av USART0 sätts UBRR0 registret (Usart Baud Rate Register). Värdet på UBRR0 bestäms av vilken frekvens AVR:ens klocka har och vilken Baude rate som önskas. AVR:erna som ingår i Styr-, resp Sensormodul har en externklocka med frekvens på 16MHz. Enligt tabell Table 17-12 i [3] motsvarar Baude rate = 9600 att UBRR0 = 103. I initieringen sätts UCSR0B = (1 << RXEN0) | (1 << TXEN0) vilket initierar mottagning respektive sändning av data via USART. USART har asynkront läge

som utgångsläge. I (`USART` Controll and Status REGISTER 0C) sätts `(1 << USBS0)` för att få 2 stoppbitar. `(3 << UCSZ00)` ger 8 databitar.

5.6.2 Sändning av Data

Data skickas via funktionen `USART_Transmit`. I Transmit funktionen skickar vi data tills data register empty flaggan är satt till 0 då vet vi att all data är skickad. Vi skickar data genom att lägga bytes på UDR0 registret.

5.6.3 Mottagning av Data

Data tas emot via funktionen `USART_Receive`. Mottagarfunktionen börjar att ta emot data när den har fått en startbit den fortsätter tills den har mottagit den första stoppbiten. Då sätts `RXC0`-flaggan till 0 och vi går ut ur funktionen.

6 Slutsatser

Vid projektets start var planen att först köra ett kalibreringsvarv med bilen. Detta var för att bygga upp en karta över banan med absoluta koordinater. Med datan från kalibreringsvarvet var tanken att skapa en mer optimal körväg som skulle möjliggöra en högre hastighet för bilen. I slutändan gjordes inte detta. Istället styrs bilen helt av LiDAR-data. Anledningen till detta designbeslut var att vi fick för dålig data från våra hallsensorer och vårt gyro. Hallsensorerna skickade hastigheter som varierade kraftigt trots att bilen körde med ett konstant gaspådrag, en källa till felen kan bero på att hallsensorn på det ena hjulet gav data som såg fel ut och varierade kraftigt. Å andra sidan gav gyrot konsekvent högre vinkelhastighet åt höger än åt vänster, något vi försökte kompensera för med begränsad framgång.

Det var med hjälp av de sensorerna som det var tänkt att kartlägga var bilen befann sig i förhållande till dess startposition. En vidareutveckling av projektet kunde därmed vara att med hjälp av fungerande sensorer/bättre implementation av dessa få sensordata att läsas och tolkas korrekt.

Istället för att använda det färdiga programmet till LiDAR som nämnts i kapitel 5.2 hade man kunnat göra en egen implementation av kommunikationen med LiDAR:n. Ett ordentligt försök gjordes med detta och vi lyckades få LiDAR:n att ge svar. Tyvärr lyckades vi aldrig att få motorn att börja snurra. Vi övergav försöken efter många timmar då vi redan hade lösningen med färdiga programmet från Slamtec. Fördelen en egen implementation hade varit att det hade krävts mindre kod därför givit en mer elegant lösning, men tidsbristen fick oss att prioritera annat.

Bibliografi

[1] L. Shanghai Slamtec.Co., ”RPLIDAR A2, Low Cost 360 Degree Laser Range Scanner, Introduction and Datasheet”. 15 maj 2017.

[2] L. Shanghai Slamtec.Co., ”rplidar_sdk”. Åtkomstdatum: 12 december 2024. [Online]. Tillgänglig vid: https://github.com/Slamtec/rplidar_sdk

[3] A. Corporation, ”8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash”. 09 november 2009.

[4] I. för Systemteknik, ”Servostyrning”. Åtkomstdatum: 05 december 2024. [Online]. Tillgänglig vid: https://da-proj.gitlab-pages.liu.se/vanheden/page/handledningar/servostyrning/