Kandidatuppsats i elektronik
Institutionen för systemteknik, Linköpings universitet, 2022

Konstruktion av en autonom taxibil
Design of an Autonomous Taxicab

Henrik Ahlinder, Mikael Josefsson, Sebastian Andersson, Martin Dahl, Charbel Youssef, Edvin Johansson

Projektidentitet

Grupp E-post: TSEA56_2021VT_O2_G10@groups.liu.se

Beställare: Anders Nilsson, Linköpings universitet
Tfn: 013-28 26 35
E-post: anders.p.nilsson@liu.se

Handledare: Peter Johansson
Tfn: +4613281345
E-post: peter.a.johansson@liu.se

Kursansvarig: Mattias Krysander
Tfn: 013-28 21 98
E-post: mattias.krysander@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Henrik Ahlinder	Projektledare (PL)	henah490@student.liu.se
Mikael Josefsson	Kommunikationsmodulansvarig, GUI-ansvarig	mikjo695@student.liu.se
Sebastian Andersson	Testansvarig	seban218@student.liu.se
Martin Dahl	Dokumentansvarig, mjukvaruansvarig	marda545@student.liu.se
Charbel Youssef	Sensormodulansvarig	chayo642@student.liu.se
Edvin Johansson	Styrmodulansvarig	edvjo404@student.liu.se

INNEHÅLL

1 Inledning 1
2 Problemformulering 2
3 Kunskapsbas 3
3.1 Styrmodulens förstudie . . . 3
3.2 Sensormodulens förstudie . . . 3
3.3 Kommunikationsmodulens förstudie . . . 3
3.4 Datablad . . . 3
4 Genomförande 5
5 Teknisk beskrivning 6
5.1 Kommunikation med användardatorn . . . 6
5.2 Kommunikation mellan modulerna . . . 6
5.3 Väglinjedetektion . . . 6
5.4 Destinationsdetektion . . . 7
5.5 Kalibrering av bildbehandling . . . 7
5.6 Reglersystemet . . . 9
5.7 Kartbyggaren . . . 10
5.8 Ruttplanering . . . 10
5.9 Positionering . . . 12
6 Resultat 13
7 Slutsatser 14
Referenser 16

Appendix **17**
A Projektdirektiv 17
B Teknisk dokumentation 19
C Användarhandledning 61
D Kravspecifikation 86
E Förstudie - Regler 105
F Förstudie - Sensor 126
G Förstudie - Kommunikation 144
H Designspecifikation 173
I Projektplan 200
J Systemskiss 220
K Efterstudie 235

DOKUMENTHISTORIK

Version	Datum	Utförda förändringar	Utförda av	Granskad
0.1	2021-03-03	Första utkast	Alla	Alla
1.0	2021-05-18	Slutgiltig version	Alla	Alla

1 INLEDNING

Detta projekt utförs som en del av kursen TSEA56 på Tekniska högskolan vid Linköpings universitet. Projektet går ut på att bygga ett system för att göra en mindre eldriven bil självkörande inom ett väldefinierat vägnät. Vägnätet är i storleksordningen 10x10 meter, byggt på papper med svarttejpade väglinjer. På vägnätet finns raksträckor, 90-graders höger- och vänsterkurvor samt T-korsningar. Det finns även utmarkerade destinationer som bilen ska kunna stanna vid. Bilen kan framföras i tre olika lägen: Manuellt, Semi-Autonomt och Autonomt. I Manuellt läge kan bilen styras från en dator som en vanlig radiostyrd bil. I de autonoma lägena kan bilen egenhändigt köra inom väglinjerna och stanna vid destinationer. Det som skiljer det semi-autonoma läget från det autonoma läget är att i det semi-autonoma läget krävs en given körsekvens från användaren, medan i det autonoma läget kan bilen själv beräkna en optimal rutt mellan en startpunkt och en målpunkt.

Projektet innehåller flera utmaningar. Några av de svårare är linjedetektion med kamera, som används för att hålla bilen inom väglinjerna samt att skapa en snabb och robust kommunikation mellan de processorer som driver bilen. Dessa delar kräver optimering för att uppnå en tillräckligt hög uppdaterinsfrekvens åt reglersystemet. Styrning av bilen är också en utmaning som främst använder sig av PID-regulatorer, tillståndsmaskiner och beräkning av optimal rutt i det autonoma körläget.

Kappan beskriver översiktligt projektets problem, genomförande, resultat och den kunskapsbas som använts under arbetets gång. Appendix innehåller mer ingående dokument såsom Teknisk Dokumentation, Användarhandledning, Designspecifikation och Förstudier.

2 PROBLEMFORMULERING

Projektets mål ställdes enligt kravspecifikationen, vilken formades enligt projektdirektivet från kursen TSEA56 [1]. Bilen ska kunna köras manuellt, det vill säga med piltangenter eller handkontroll. Ett semi-autonomt körläge skulle också införas där användaren kan bestämma om bilen ska hålla till höger eller till vänster i korsningar samt ifall den ska stanna vid nästa destination eller fortsätta framåt. Slutligen eftersöktes ett autonomt läge där bilen själv kan avgöra bästa rutt mellan en startpunkt och en målpunkt, vilket även kräver att bilen kan positionera sig i vägnätet. Projektdirektivet beskriver problemet väl och för detaljer hänvisas läsaren till detta, se Appendix A. Det finns aspekter av problemet som är särskilt intressanta. Problemet är tekniskt intressant från tre synvinklar:

1. **Reglerteknik** används för att reglera bilens framföring i vägnätet. Den använder sig av information från bildbehandlingen och från andra sensorer för att reglera bilens fart och styrutslag så att bilen håller sig på vägen. Reglerteknik är alltså huvudkomponenten som krävs för att skapa en självkörande bil.

2. **Bildbehandling** krävs för att tolka bilens omvärld. Det är ett mycket aktuellt teknikområde och projektet gav goda möjligheter för att utforska detta. I det här projektet räckte det med äldre grundläggande bildbehandlingstekniker för att detektera kanter. Bildbehandling är också ett optimeringsproblem då en hög uppdateringsfrekvens behövs för att bilen sedan ska kunna reglera framgångsrikt efter den information som bildbehandlingen ger.

3. **Hårdvarukommunikation** används för att kommunicera mellan bilens olika processorer: två mikrokontroller av AVR-typ och en Raspberry Pi. Detta utförs med ett I^2C-protokoll, med Raspberry Pi som enskild *master*. Problemet kräver att processer kan exekveras parallellt och med avbrott, samtidigt som deras CPU-inverkan bör minimeras för att tillåta bildbehandling att ta mycket processorutrymme. Det gav viktiga insikter om lågnivåprogrammering i C på processorer av AVR-typ samt praktisk erfaranhet av sammankoppling av parallella processer skrivna i olika programspråk.

Projektet har också erbjudit problem av icke-teknisk karaktär:

1. **Versionshantering** med git och koordinering av flera utvecklare i samma projekt.

2. **Teknisk planering** av ett projekt, det vill säga skrivandet av en designspecifikation samt att förutsäga en rimlig implementation och tidsåtgång för ett större tekniskt problem.

3 KUNSKAPSBAS

I detta kapitel beskrivs de förstudier som har utförts i samband med projektet och i vilken omfattning de har använts vid projektets utförande. Förstudierna beskrivs kortfattat nedan och finns även i sin helhet i Appendix E, F och G. Kapitlet behandlar även kortfattat de viktigaste datablad som använts under projektet.

3.1 Styrmodulens förstudie

I styrmodulens förstudie undersöks bland annat beslutsfattning inom ett vägnät, beräkning av optimal rutt och väglinjeföljning med PID-regulator och/eller regulator av *pure-pursuit*-typ. En stor del av förstudien går utanför vad som visade sig krävas av projektet, till exempel är pure-pursuit en mer avancerad metod som kräver en mer tillförlitlig bildbehandling än den som ges av systemets linjedetektion. PID-regulatorerna visade sig också ge tillräckligt bra prestanda. Förstudien går också in på särskilda metoder för att hitta PID-regulatorernas parametrar, vilka visade sig vara användbara i bilens fall.

3.2 Sensormodulens förstudie

I sensormodulens förstudie undersöks olika delar av ett bildbehandlingssystem för att detektera väglinjer. Det innefattar bildfiltrering vilket inkluderar gråskalekonvertering, HLS-konvertering (*Hue, Lightness, Saturation*), LAB-konvertering (*Lightning*, A-, B-färgkanaler), Gaussisk lågpassfiltrering, korrigering av linsförvrängning, *Canny*-kantdetektion samt fågelperspektivstransformering. Olika linjedetektionsalgoritmer beskrivs, såsom Hough-transformering och *sliding-window*-sökning. Slutligen görs en undersökning om vilka filtreringar och vilken linjedetektionsalgoritm som passar projektets ändamål bäst utifrån de förutsättningar som var satta. Hough-transformeringen visade sig fungera utmärkt för linjer som är relativt raka, men i detta projekt behövde även svängar och korsningar kunna detekteras, något som är en svaghet hos Hough-transformeringen. Sliding-window-sökning visade sig dock vara bättre lämpad för sådana situationer. Förstudien ledde alltså till att linjedetektionen blev robust på alla typer av vägsegment som fanns i projektet och även så att denna kunde köras med en önskvärd hastighet.

3.3 Kommunikationsmodulens förstudie

I kommunikationsmodulens förstudie undersöks metoder för parallellprogrammering i Python och C, vilket kan användas för att effektivisera kommunikation mellan systemets tre processorer. Förstudien nyttjades i stor utsträckning i program förlagda på både mikrodatorn och användardatorn, men den lämpade sig mindre väl för implementation på de enkärniga mikrokontrollerna. Bland annat ledde förstudien till en lyckad separation av bildbehandling och kommunikation till två separata processer. Kommunikationen separerades vidare till ytterligare tre separata processer som vardera ansvarade för serverkommunikation, kommunikation mellan processerna på Raspberry Pi respektive kommunikation över hårdvarubussen. Denna uppdelning gav ett modulärt uppbyggande av programmet som möjliggjorde ett enkelt utbyte av drivrutiner och kommunikationsvägar som inte kräver en fullständig omskrivning av programmet.

3.4 Datablad

Databladet för AtMega 1248p [2] har använts flitigt vid programmering av systemets mikrokontrollers. Självklart har dokumentation för samtliga komponenter och moduler använts, i de viktigaste fallen finns dessa i den tekniska dokumentationens referenser eller i förstudiernas referenser.

Dokumentationen för OpenCV:s färdigskrivna funktioner för matrisbehandling [3] har använts vid skrivande av bilens bildbehandling.

4 GENOMFÖRANDE

Projektet har genomförts enligt projektmodellen LIPS [4]. Först gjordes en kravspecifikation utifrån en projektbeskrivning som var given av kursen. Det inkluderade att specificera bas-, slut- och extrakrav som sedan förhandlades med beställare.

Vid godkännande av kravspecifikationen från beställaren började framtagandet av en projektplan. I projektplanen specificerades hur olika moment i projektet skulle utföras samt vilka roller respektive gruppmedlem skulle ha genom projektets gång. De aktiviteter som behövde utföras för att genomföra projektet beskrevs även i projektplanen. Aktiviteterna byggdes från kraven i kravspecifikationen. För varje aktivitet undersöktes också dess beroende till övriga aktiviteter vilket lade grund för tidplanens utformning.

Med hjälp av alla aktiviteter kunde sedan en tidsplan skapas, där tid allokerades varje aktivitet. Ordningen för aktiviteterna sattes utifrån de beroende som respektive aktivitet hade till de övriga. Aktiviteterna placerades i tidplanen på ett sådant vis att en vattenfallsstruktur uppnåddes.

Därefter skrevs en systemskiss där konturerna till systemet drogs. Vidare definierades viktiga värden, gränssnitt och huvudsaklig funktionalitet för de olika modulerna. I systemskissen sattes det upp blockscheman för hur de olika modulerna skulle samverka med varandra och hur deras inre delsystem skulle fungera. Den innehåller även en skiss på hur hårdvaran på bilen skulle placeras och hur bilens användargränssnitt skulle se ut.

En mer detaljerad designspecifikation skrevs utifrån systemskissen. Varje system fick ett första utkast på hur de skulle fungera, implementeras och kommunicera med varandra. Ett kommunikationsprotokoll etablerades. Med en design specificerad för projektet öppnades möjligheten för att påbörja konstruktionen, vilket var nästa steg i projektet.

I vårterminens andra period påbörjades konstruktionen, vilken utfördes enligt tidplanen som skapats tidigare under projektet. Aktiviteter som tog längre eller kortare tid än förväntat uppdaterades i tidplanen för att stämma överens med verkligheten. Under implementationen av delsystemen utfördes testning kontinuerligt för att säkerställa deras funktionalitet. En renartad testperiod behövde inte genomföras utan integration utfördes kontinuerligt allt eftersom delsystem färdigställdes. Större systemtest utfördes inför beslutspunkter och leveranser.

Vid slutfört projekt skrevs en Teknisk dokumentation, Användarhandledning samt denna Kappa och slutligen redovisades projektet inom ramen av kursen TSEA56.

5 TEKNISK BESKRIVNING

I detta kapitel beskrivs de viktigaste funktionerna hos de olika delsystemen i bilen.

5.1 Kommunikation med användardatorn

Kommunikation mellan bilen och användardatorn implementerades över WIFI via ett TCP-protokoll, även känt som ett *Transmission Control Protocol*. Vid uppstart startar bilen en server och ansluter automatiskt till nätverket med en statisk IP-address. Användardatorn kan, i användargränssnittet, ansluta som en klient till servern. Trådlösa försändelser serialiseras därefter i JSON-format och överförs mellan klienterna via *network sockets*.

5.2 Kommunikation mellan modulerna

Kommunikation mellan modulerna på bilen sker över en hårdvarubuss via ett I^2C-protokoll med enskild *master*. Modulerna ansluts trådlöst till användardatorn via en server på mikrodatorn, samt en integrerad klient i användargränssnittet. Server och klient kommunicerar med varandra via en standarduppsättning av nätverkssockets med TCP-protokoll.

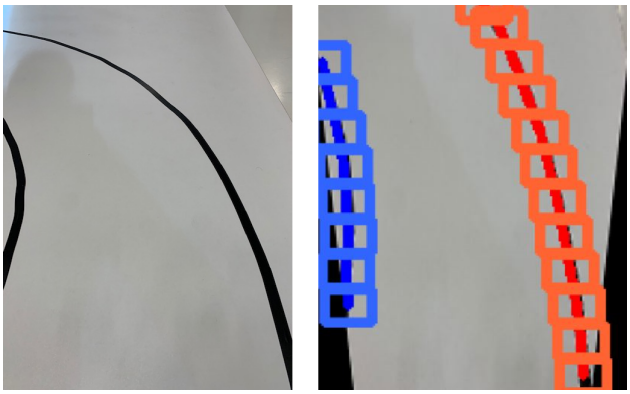
För att maximera busshastigheten på den begränsade utvecklingstid som fanns tillgänglig skrevs ursprungligen kommunkationen i Python, men sågs till att utformas modulerbart. När busshastigheten senare blev den begränsande faktorn för bilens hastighet ändrades kommunikationskoden till en skalkod i Python som synkroniserar försändelser mellan bildbehandling och servern, som i sin tur nyttjar en drivrutin skriven i C++ som exekverar transaktioner över hårdvarabussen. På så vis har en transaktionshastighet av 400 kBaud uppnåtts på hårdvarubussen utan att produktionshastigheten för övriga delar av projektet krävdes stå still vid utvecklande av drivrutinen.

5.3 Väglinjedetektion

Väglinjedetektion utförs med hjälp av bildbehandling. En kamera på bilen ger en videoström till bildbehandlingen. Bildbehandlingen utför först en filtrering av bilden där i stort sätt allt utom vägkanterna filtreras bort. För att få en bättre synvinkel genomförs en perspektivstranformering på bilden vilket ger ett fågelperspektiv av vägen, det vill säga att vägen ses mer ovanifrån. Se Appendix B, Teknisk dokumentation, för detaljerad beskrivning av algoritmen.

Efter utförd filtrering kvarstår linjesökningen, vilken utförs med en sliding-window-sökning. Sliding-window-sökningen tar fram linjerna i form av ett antal punkter, se Figur 1 för ett exempel på linjedetektion i kurva. Rutorna i figuren indikerar hur stort sökområdet är för nästkommande iteration, om nästa ruta inte innehåller en del av linjen kommer sökningen att avslutas. För en detaljerad beskrivning av algoritmen, se Appendix B, Teknisk dokumentation. För en mer ingående förklaring av de olika delstegen i algoritmen hänvisas till Appendix F, Förstudie - Sensor.

När väglinjerna är detekterade utförs beräkningar på punkterna för att få ut relevant data om vägen, till exempel avstånd till respektive vägkant från bilen. Denna data skickas vidare till positioneringen och styrmodulen.



Figur 1: *Exempel på väglinjedetektionen i en kurva. Vänster bild är originalbild och höger bild är transformerad med fågelperspektiv och visar de detekterade kantlinjerna. Rutorna markerar området för sliding-window-sökningen och de heldragna linjerna markerar vart väglinjerna har detekterats.*

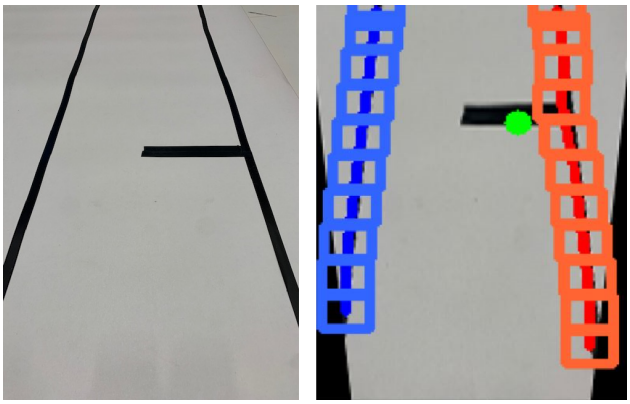
5.4 Destinationsdetektion

Destinationsdetektion genomförs i bildbehandlingen. Den utförs genom att söka av den högra vägkanten, och utförs därmed endast ifall den högra vägkanten är detekterad. Efter detekterad höger vägkant görs en sökning efter destinationer längs med linjen. Sökningen avslutas vid första funna destination. Sökningen i sig är lik den sökning som används vid väglinjedetektionen, skillnaden är att det sökfönster som används placeras precis till vänster om den sökruta som användes vid väglinjedetektionen. På detta sätt minskas risken för falska utslag till följd av återdetektion av vägkant. Detta förbättrades ytterligare med en maskering av vägkanterna. För att undvika detektion av destination på vänstra vägkanten sattes krav på var den detekterad destinationen fick upptäckas. I Figur 2 visas detekterad destination med en grön prick. Det var ett designval att enbart söka efter destinationer på höger sida av vägen eftersom det endast är dessa som bilen får stanna vid, denna selektiva detektion ses i Figur 3. För en mer detaljerad beskrivning av algoritmen, se Appendix B, Teknisk dokumentation. Vid funnen destination beräknas avståndet till destinationen och sedan vidarebefordras detta till styrmodulen och positioneringen.

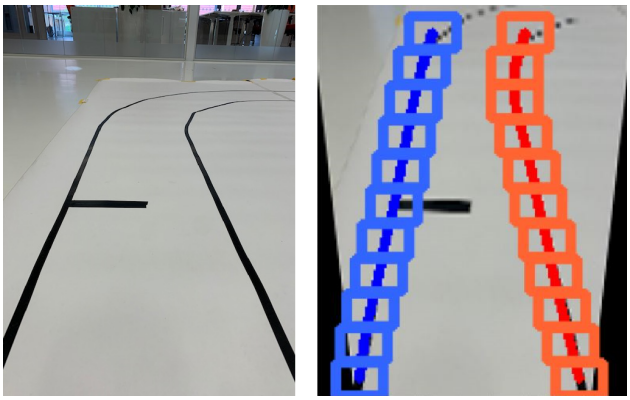
Destinationsdetektionen hade en del problem vid implementationen. En av de stora utmaningarna var att få en tillförlitlig detektion av destinationer. Den implementation som presenteras ovan är en av många olika varianter som testades och det är även den långsammaste varianten. Den valdes eftersom den var den enda variant som uppvisade tillräckligt hög robusthet. Trots att den var den långsammaste varianten påverkades inte prestandan avsevärt eftersom destinationsdetektion fortfarande var en relativt liten del av algoritmen. Ett annat ihärdigt problem var att svängar, av misstag, ofta detekterades som destinationer. Detta problem löstes inte direkt i destinationsdetektionen men istället av sliding-window-sökningen. Problemet löstes där genom att göra tröskelvärdena för detektion hårdare.

5.5 Kalibrering av bildbehandling

Bildbehandlingen behöver kalibreras för olika kameravinklar och körbanor, se Appendix C, Användarhandledning, för mer detaljer om hur den genomförs. Kalibreringen genomförs i ett flertal steg där olika reglage finns som styr de olika parametrarnas värden, se Figur 4 för ett exempel på ett av dessa steg. Parameterförändringens effekt på bildbehandlingens resultat visas i realtid för att ge en god förståelse för vad som justeras. I det steg som ses i Figur 4 är även algoritmen för destinationsdetektering aktiverad så att användaren enkelt kan se vilka parametrar som ger bäst

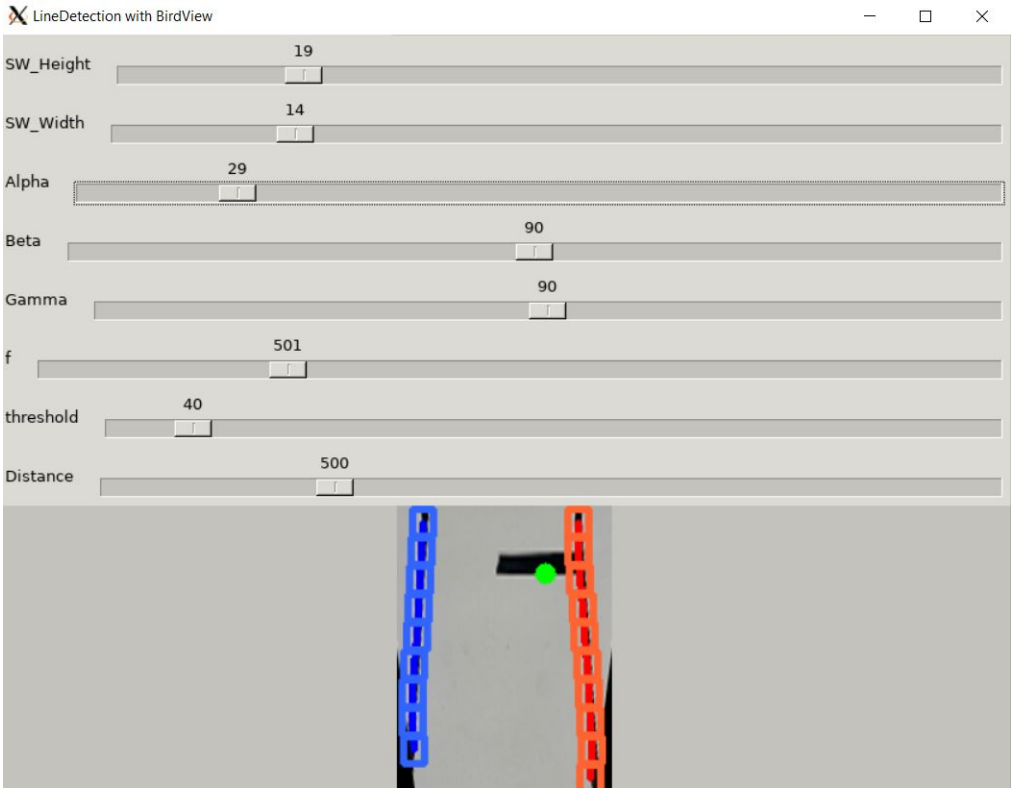


Figur 2: *Exempel på destinationsdetektion då destinationen är sammankopplad med höger vägkant. Detekterad destination markeras med en grön prick. Vänster bild är originalbild och höger bild har processerats av väglinjedetektion och destinationsdetektion.*



Figur 3: *Exempel på destinationsdetektion då destinationen är sammankopplad med vänster vägkant. Etersom destinationen är sammankopplad med fel vägkant innebär det att den ej detekteras. Vänster bild är originalbild och höger bild har processerats av väglinjedetektion och destinationsdetektion.*

detektion. När alla parametrar är satta kommer de värden som valts att sparas och användas vid framtida körningar av bilen.



Figur 4: *Exempel på ett av stegen för kalibrering av bildbehandlingen. I detta steg justeras fågelperspektivtransformens parametrar samt vilka parametrar som sliding-window-sökningen ska ha. Under detta steg är detektionsalgoritmen aktiverad, både för väglinjer och för destinationer. Kalibreringen kan verka komplex, men det är ej fallet. En ovan användare kan använda sig av de förinställda värdena och uppnå god prestanda.*

5.6 Reglersystemet

Bilen använder två regulatorer, en för farthållning och en för väglinjeföljning, som båda utgörs av klassiska PID-regulatorer. Bildbehandlingen beräknar avståndet till höger och vänster vägkant precis framför bilen, och den enkla, men tillräckliga, informationen gör PID-regulatorn till ett säkert och enkelt val. Trots regulatorns enkelhet har mycket goda resultat uppnåtts. Med en uppdateringsfrekvens på 30 FPS (bilder per sekund) från bildbehandlingen har bilen kunnat köra icke-oscillativt i 1.5 m/s. Höga värden på reglerparametern K_D har behövts för detta, för att dämpa oscillationer som annars uppstår till följd av höga hastigheter. Medan bilen kan köra icke-oscillativt i höga hastigheter kan det uppstå problem på grund av artefakter i bildbehandlingen som räcker för att sätta det system som bilen utgör i en divergent självsvängning. Av den anledningen blir bilens maximala hastighet lägre i praktiken. Bilen kan med en god robusthet framföras autonomt i 0.75 m/s.

Det har observerats att farthållaren bör vara en PI-regulator medan väglinjeföljaren bör vara en PD-regulator. Vägen innehåller alldeles för många kurvor i förhållande till raksträckor för att en I-del skulle vara nödvändig, av samma skäl är det också viktigt att D-komponenten finns med. För farthållning har inget skäl uppstått för att använda en D-del då

farthållaren inte har agerat oscillativt vid de hastigheter som bilen kör i. En I-del är dock nödvändig för att eliminera det statiska felet.

Tidigt i projektet existerade problem som begränsade bildbehandlingen och sänkte uppdateringsfrekvensen ner emot 10 FPS. Denna lägre uppdateringsfrekvens märktes tydligt på väglinjeföljarens beteende då den blev mycket oscillativ och i vissa fall instabil. Detta blev en intressant lärdom och en uppvisning på hur tidsfördröjningar påverkar stabiliteten hos ett system.

I styrmodulens förstudie diskuterades att implementera en *pure-pursuit*-regulator för väglinjeföljning. Det märktes dock att en PD-regulator räckte väl. Särskilt beror PD-regulatorns goda prestanda på att vägnätet är väldigt förutsägbart, som av design är anpassat efter bilens kinematik. Vidare kunde inte bildbehandlingen åstadkomma den prestanda som behövs för reglering med pure-pursuit inom projektets tidsram. För pure-pursuit krävs en precis beskrivning av väglinjerna långt framför bilen, exakta mått på avstånd och på vägens krökning. PD-regulatorn krävde enbart avståndet till väglinjer precis invid bilen, vilka med god robusthet och hög uppdateringsfrekvens kunde detekteras av bildbehandlingen.

I styrmodulen hanteras även inbromsning för hinder och destinationer. Det kontrolleras hela tiden hur nära bilen är dessa objekt. Styrmodulen använder en dynamisk anpassning efter bilens hastighet och avstånd relativt objekten för att bromsa in. Bilens bromssträcka är däremot väldigt liten och därför behöver inte mycket hänsyn tas till detta. Hänsyn behöver dock tas till uppdateringsfrekvensen hos bildbehandlingen och hos avståndsmätningen med ultraljud eftersom avstånd till objekten måste hinna uppdateras.

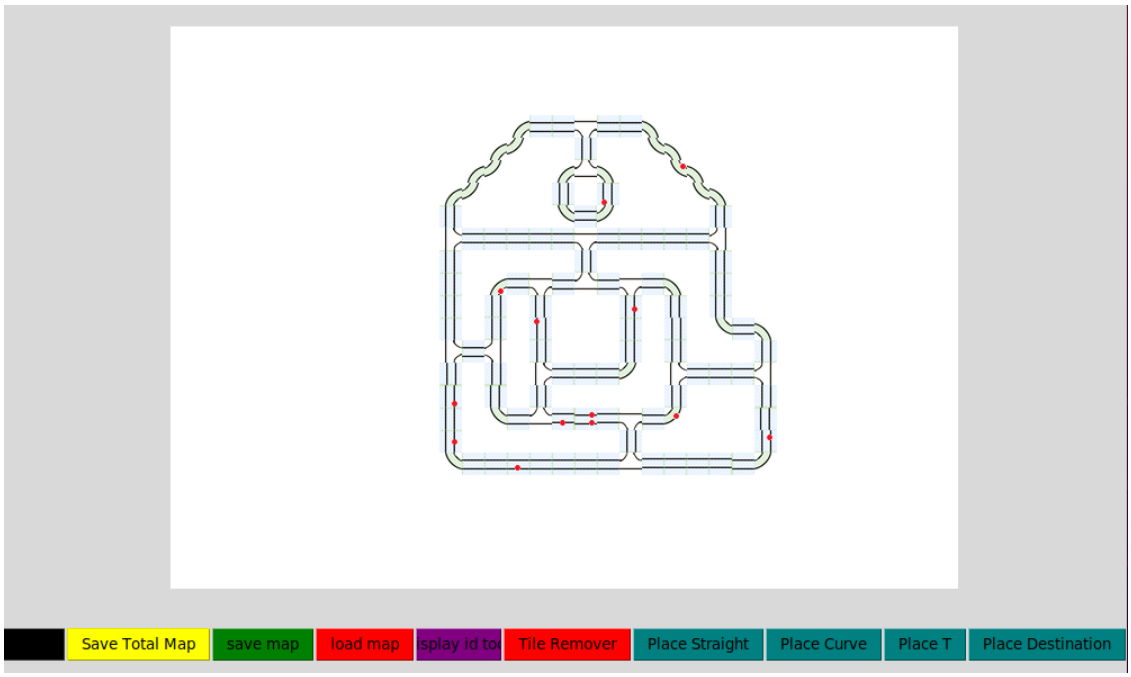
5.7 Kartbyggaren

Kartbyggaren är ett program som används för att enkelt mata in det aktuella vägnät som bilen kör i. Vägnätet används sedan av bilen för positionering och ruttplanering. Kartbyggaren består både utav ett interaktivt användargränssnitt för inmatning och utav en grafgenerator som konstruerar en graf utifrån det som matats in. Dessa grafer hade varit mycket komplicerade och tidskrävande att konstruera för hand, särskilt för ett större vägnät. En karta i kartbyggarens användargränssnitt ses i Figur 5. Kartbyggaren har gjort bilens system väldigt generellt och anpassningsbart till olika vägnät.

Medan kartbyggaren har förenklat stora delar av inmatningen av körbana behöver fortfarande vissa manuella anpassningar av kartorna göras. Kartbyggaren lyckas inte helt skapa tillförlitliga vägkostnader i graferna som den genererar. Detta är mycket viktigt för bilens positionering, som är känslig för dessa vägkostnader. För bilens ruttplanering spelar de också roll men betydligt mindre. Ruttplaneringen kräver förvisso korrekta kostnader på vägsegmenten men dess resultat påverkas inte nämnvärt av att vissa segment har något inexakta vägkostnader.

5.8 Ruttplanering

Ruttplaneringens uppgift är att i det autonoma körläget ta fram den snabbaste vägen mellan två destinationer och generera en sekvens av svängar som bilen ska utföra för att följa denna. För att göra detta används Dijkstras algoritm [5] med mindre modifikationer. För att kunna använda sig av algoritmen omvandlas den visuella representationen i kartbyggaren till en graf med noder och bågar, där varje segment från kartbyggaren utgör en nod. Noder som är ointressanta, i den bemärkelse att inga beslut behöver tas där, optimeras bort. Noder för kurvor behålls dock eftersom det i tidiga iterationer av reglersystemet var intressant att kunna hålla sig närmare inner- eller ytterkanten i en kurva, och algoritmen har tillräckligt hög prestanda även med dessa överflödiga noder.



Figur 5: *Exempel på hur ett vägnät ser ut i kartbyggarens användargränssnitt. De röda prickarna motsvarar destinationer och läggs längs med vägen, på höger eller vänster sida. När användaren har matat in sin karta så genererar kartbyggaren en graf som används vid positionering och ruttplanering.*

Något som skiljer detta problem från ett klassiskt kortaste-väg-problem är att destinationerna endast kan nås ifall bilen har dem på höger sida om sig, vilket kräver att bilen anländer vid en destination från ett visst håll. På grund av bilens och vägnätets fysiska begränsningar kan inte heller U-svängar göras. För att ta hänsyn till dessa begränsningar skapas en utökad graf där varje nod utökas till två eller tre noder, som indikerar vilket håll bilen anlände i noden ifrån. Till exempel skulle noden för en trevägskorsning <A> utökas till de tre noderna <A, vänster>, <A, höger> och <A, nedåt>. Bågar som skulle innebära icke-realiserbara svängar kan då tas bort och det går enkelt att specificera vilken av dessa noder som måste nås för att bilen ska ha en destination på sin högra sida. Med en sådan representation av vägnätet fås 2-3 gånger så många noder men det totala antalet bågar hålls relativt konstant eftersom de flesta dubbelriktade bågar kan bytas ut mot enkelriktade bågar. Totalt sett gör det att beräkningstiden inte ökar avsevärt med denna utökade graf.

Kostnaderna i grafen bör vara tiden det tar att köra mellan två noder, då målet är att köra rutter på så kort tid som möjligt. I grafen används dock sträckan mellan två noder som kostnad då denna är enklare att mäta, och kan användas utbytbart under antagandet att bilen håller en konstant fart. Bilens farthållare gör att detta antagande gäller.

När en optimal väg är funnen genereras en sekvens av instruktioner som säger åt bilens styrmodul att hålla till höger eller till vänster i de korsningar som kommer att passeras, för att bilen ska följa den optimala vägen.

5.9 Positionering

Bilens positionering bygger på odometri och använder sig av den utökade graf som skapats av ruttplaneraren, beskriven i avsnitt 5.8. Positioneringsalgoritmen använder sig av kännedom om bilens startposition, bilens tillryggalagda sträcka och antagandet att bilen följer den väg som ruttplaneraren bestämt. Det är då enkelt att avgöra vilken nod som bilen befinner sig i och därmed var i vägnätet bilen är.

Med tiden ackumuleras fel i mätningen av färdad sträcka som uppstår på grund av noggrannheten hos sensorn, avståndsfel i grafen och för att bilen även färdas en liten sträcka i lateralt led. För att undvika att dessa fel blir betydelsefulla efter längre tid så kalibreras den uppmätta färdsträckan om, med hjälp av bildbehandlingen, under körning. När den uppmätta sträckan indikerar att nästa nod är en destinationsnod så inväntas signal från bildbehandlingen om att bilen befinner sig precis vid destinationen. Då kalibreras bilens position om enligt bildbehandlingens information. Detta sker med olika stora intervall beroende på placeringen av destinationerna som bilen passerar, men avstånden är i storleksordningen 0-5 m. Typiskt ses avvikelser på några enstaka procent när dessa kalibreringar görs.

6 RESULTAT

Målet med projektet var att konstruera hårdvara och mjukvara till en bil för att kunna köra den manuellt, semi-autonomt och autonomt. I det Semi-autonoma läget kan bilen köra inom väglinjer och stanna vid destinationer på kommando från användaren. Kommandona är vilket håll som bilen ska åka ifall den kommer till en korsning samt om bilen ska stanna vid nästa destination eller köra förbi den. I jämförelse är det Autonoma läget mer intelligent då val av svängar görs automatiskt utifrån en satt start- och målpunkt. Rutten mellan startpunkten och målpunkten beräknas optimalt och bilen bestämmer sin egen position i vägnätet under körning för att ta sig till målpunkten.

Dessa mål har uppnåtts med god prestanda och bilen uppfyller de krav som satts i Kravspecifikationen, se Appendix D, Kravspecifikation. Uppdateringsfrekvensen för bildbehandlingen är ca 30 FPS, vilket är tillräckligt för körning av bilen i upp till 1.5 m/s. Notera att den maximala uppdateringsfrekvensen för styrning av bilen är 50 Hz eftersom bilens styrservo maximalt kan uppdateras med denna frekvens. Givet mer tid finns därför möjlighet att förbättra bilens prestation genom att effektivisera bildbehandlingen ytterligare. Bilen har kunnat regleras väl genom nyttjandet av två olika PID-regulatorer: en PI-regulator för farthållning och PD-regulator för väglinjeföljning.

I^2C-kommunikation har använts för att kommunicera mellan bilens olika datorer som består av två mikrokontrollers av AVR-typ och en Raspberry Pi. Kommunikationen har implementerats väl så att den ej begränsar prestandan hos bildbehandlingen märkbart vid parallell körning av de båda.

Vidare har ett användargränssnitt byggts och utvecklats med funktionalitet för diagnos och inställning av bil. I användargränssnittet finns ett fönster för visning av nuvarande vägnät där bilens position och planerade rutt visas. I användargränssnittet finns även visualisering av aktuell information från bilen, uppvisat både i en realtidsgraf och i en informationsruta där valda variablers värden visas. Gränssnittet erbjuder även möjlighet att spara mätserier under körning, samt att analysera dessa mätserier i efterhand via ett separat analysfönster. Se Appendix C, Användarhandledning, avsnitt 2.2 för en mer ingående beskrivning av användargränssnittet. Ett verktyg för inmatning av kartor har skapats, där användaren enkelt kan föra in det vägnät som bilen för tillfället kör på. Kartan kan skapas med stor flexibilitet och har bra verktyg för att hjälpa användaren att placera destinationer korrekt. Det har visat sig att viss manuell modifiering av grafernas kostnader kan behövas för bilens positioneringsalgortimen. Trots denna brist har Kartbyggaren visat sig vara mycket användbar under projektet.

Parametrarna i bildbehandlingen visade sig vara känsliga för olika körbanor och olika vinklar på kameran. Därför finns ett kalibreringsprogram implementerat för bildbehandlingen som låter användaren se effekten av olika värden på parametrarna i realtid, och som sedan sparar dessa inför framtida körningar, se Appendix C, Användarhandledning, kapitel 6 för en mer detaljerad beskrivning om kalibrering av bildbehandling. Detta underlättade testning av bilen då det var enkelt även för projektmedlemmar som ej var djupt insatta i bildbehandlingen att kalibrera kameran. Det sparade även tid eftersom kompilering av bildbehandlingsprogrammet är en tidskrävande process på en Raspberry Pi.

I sin helhet klarar bilen av alla krav och kör robust i vägnätet, både i Semi-autonomt- och Autonomt läge. I Manuellt läge märks ingen latens vid styrning av bilen, och den är lättstyrd och responsiv för alla kommandon. Bilen kan hålla denna goda prestanda vid autonom körning i hastigheter upp till 1.5 m/s. Verktygen, såsom kalibreringen och kartbyggaren, som skapades under projektets gång gynnade gruppen då det effektiviserade arbetet och testandet av bilen.

7 SLUTSATSER

Gruppen har gjort ett bra jobb och medverkat till att få bilen att klara av de satta kraven med marginal. Hela gruppen är nöjd med samtliga delmoduler i systemet som tillsammans har medverkat till en autonom bil med hög prestanda. God bildbehandling, processorkommunikation och styralgoritmer leder till att bilen kan köra på ett mjukt och smidigt sätt i vägnätet. Ytterligare är gruppen nöjd över användargränssnittet, som är både snyggt och enkelt att använda samt visar nödvändig data på ett snyggt sätt.

Vidare utveckling av bilden kan göras och fler funktioner kan tilläggas. Bland annat kan funktionalitet införas för att kartlägga ett okänt vägnät som inte redan finns inmatat i systemet, vilket till viss del hade ersatt kartbyggarens uppgift. Bildbehandlingen kan optimeras med avseende på uppdateringsfrekvens och robusthet, inom de begränsningar som hårdvaran sätter. Trots att den i nuläget har god robusthet och uppdateringsfrekvens finns det alltid rum för förbättringar. Under projektet nedprioriterades dock optimering av bildbehandlingens prestanda till förmån för en att skapa mer robust bildbehandlingen. En annan förbättring som kunde göras är att justera kamerans position på bilen. Kamerapositionen på bilen borde optimeras för att se maximalt mycket av vägen framför bilen, utan att tappa närperspektivet, eftersom det tillåter ett bättre nyttjande av transformeringen (fågelperspektivstransformationen). Det bidrar till att avståndskonverteringarna skulle bli mer representativa av verkligheten, speciellt avståndet till destinationer. Utöver detta finns det möjligheter för expansioner av samtliga gränssnitt, där en av de mest betydelsefulla är att ta bort behovet av manuell kostnadskorrigering i kartbyggaren.

Olika utmaningar och problem har uppstått under projektets gång. Flest problem har uppstått i relation till diffusa beteenden hos mikrokontrollerna. Problem har också uppstått med projektets versionshantering, där Git användes. Versionshanteringen blev studnvis problematisk då många *branches* skapades utan att tas bort, och då många projektmedlemmar arbetade parallellt på samma branch, med samma filer. Inom bildbehandlingen uppstod en del problem med destinationsdetekionen, som nämndes i avsnitt 5.4. Problemen var att svängar ofta felaktigt identifierades som destinationer och att algoritmen inte kunde detektera destinationer med tillräckligt hög tillförlitlighet. Till sist implementerades en detektionsalgoritm som uppfyllde robusthetskraven men var då istället ej optimerad. En annan utmaning var ett försök att implementera en observatör, det vill säga en interpolation av reglerutsignal mellan varje uppdatering av bildbehandlingen. Observatören fungerade dock dåligt i kurvorna eftersom den utgick från att bilen körde på en rak väg. Om säker information om bilen nuvarande segment fanns tillgänglig skulle observatören kunna förbättras och utnyttjas för att kompensera för skillnaden mellan uppdateringsfrekvensen för bildbehandlingen och uppdateringsfrekvensen för styrservot. Problem uppstod även när kod skrevs på Raspberry Pi då indikation av batterinivå saknades, vilket orsakade flera icke-rena nedstängningar av program. Detta skapade flertalet korrupta *git-repositories*, vilket är något som bör undvikas.

Något som borde lagts mer tid på var planering och uppdelning av de olika modulernas ansvar, samt en bättre kommunikation inom projektgruppen om vad som utvecklades mellan veckomötena. Exempelvis var det oklart om det var bildbehandlingens ansvar att göra en rimlighetsbedömning av resultaten av dess algoritmer eller ifall det skulle avgöras av den modul som använde sig av informationen. Denna bristande planering resulterade även i att flera projektmedlemmar ovetande arbetade på var sin lösning av positionshanteringen, vilket ledde till onödigt extra arbete. Det hade även varit bra att kontinuerligt uppdatera vad som förväntas från de olika modulerna samt att i god tid informera varandra ifall några ändringar skulle göras i gränssnitten mellan modulerna.

Några synpunkter inför framtida arbeten kan sammanfattas enligt följande:

- **Styrmodulens mikrokontroller** bör begränsas till att enbart hantera motorer, styrservo, inbromsning vid hinder, PID-regulatorer och heartbeat. All annan intelligens kan med framgång utföras på Raspberry Pi:n. Styrmodulens

PD-regulator kan göras mer avancerad genom att inför en predikteringsalgoritm och den kan modifieras till att även reglera efter hur vägen ser ut längre fram, utan att nödvändigtvis behöva använda en ren pure-pursuit-algoritm.

- **Bildbehandlingen** bör lägga fokus vid algoritmen för destinationsdetektionen. Det krävdes många iterationer innan ett önskevärt resultat nåddes för denna, vilket i sin tur förhindrade andra delar av projektet från att testa sina delar. Om en bra destinationsdetektion hade uppnåtts tidigare hade det öppnat för möjligheten att optimera bildbehandlingens prestanda. Det hade även tillåtit nog med tid för att förbättra detaljer i bildbehandlingen, som t.ex. att förhindra att väglinjer ibland detekteras i dubbelsidiga destinationer vilket, ifall det inte hanteras, kan ge upphov till kortvariga men kraftiga oscillationer i styrutslaget.
- **Versionshanteringsmetodik** bör utvecklas och planeras i detalj innan uppstart av konstruktion. Se till att enas om hur man nyttjar verktyget på ett effektivt och robust vis. Framförallt bör ett system framarbetas som tillåter frekventa uppdateringar av stabil testkod parallellt med instabil utvecklingskod.

Det visade sig vara mycket fördelaktigt att ha en bärbar dator med linuxinstallation avsedd till projektets användargränssnitt. Detta löstes i projektet genom att en sådan dator lånades ut till projektet av en projektmedlem. Det är också mycket fördelaktigt att ansluta sig till bilens Raspberry Pi med SSH och att ha en fast wifi-router som alla enheter ansluter sig till automatiskt. En sådan lånades ut till projektet av studentföreningen *Föreningen för Intelligenta Autonoma System*.

I slutändan fungerar bilen bra utifrån de förutsättningar som var givna. Det finns alltid rum för förbättringar men den slutprodukt som gruppen har framställt presterar över de förväntningarna som var satta från början. Projektet har varit roligt, lärorikt och väldigt intressant att utföra. Något som hade gjort projektet ännu mer intressant för framtida studenter skulle vara att ge en mer konkret beskrivning av vad som menas med Semi-autonomt läge. Med en tydligare definition skulle gruppen ha en klarare bild av vad respektive läge innebar och ej behöva lägga tid på att försöka tolka det. Även kunde det Manuella läget ha lite fler krav på vad det skulle klara av. Exempelvis skulle en slags virtuell växellåda kunna programmeras så att känsligheten på hastigheten eller topphastigheten kan varieras. Ett annat exempel är att ha krav på bra latens i det Manuella läget. Ett sista tillägg som skulle göra projektet mer intressant skulle vara att ha en annan utformning av vägnätet, då den bana som användes under projektet upplevdes för liten.

REFERENSER

[1] C. Krysander. Tsea56: Projektdirektiv för en autonom taxibil. [Online]. Available: http://www.isy.liu.se/edu/kurs/TSEA56/projektdirektiv/Projektdirektiv_for_autonom_taxibil_21.pdf

[2] Microchip. megaavr data sheet. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega164A_PA-324A_PA-644A_PA-1284_P_Data-Sheet-40002070B.pdf

[3] OpenCV. Opencv documentation. [Online]. Available: https://docs.opencv.org/master/

[4] T. Svensson and C. Krysander, *Projektmodellen LIPS*, 1st ed. Studentlitteratur, 2011, iSBN:9789144075259.

[5] Dijkstra's algorithm. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Appendix

A PROJEKTDIREKTIV

TSEA56: Projektdirektiv för en autonom taxibil

Version: 1.0
Beställare: Mattias Krysander
Datum: 2020-01-15

Inledning
Vi vill undersöka möjligheterna att konstruera en autonom bil. Bilen ska autonomt följa ett köruppdrag bestående av en lista av destinationer i vägnätet. Bilen får inte kollidera med hinder eller andra fordon. För att utvärdera hur man kan göra en sådan bil önskar vi beställa ett antal prototyper. Dessa ska delta i en tävling där vi kan utvärdera olika konstruktionsalternativ. För att erhålla olika konstruktioner ombeds ni att göra en kravspecifikation i dialog med beställaren. Nedan ger vi er ett antal grundkrav som ska vara gemensamma i alla kravspecifikationer. Kraven är inte numrerade vilket de dock ska vara i era kravspecifikationer. Ni förväntas lägga till mer text som beskriver roboten i allmänna termer, figurer samt unika krav för just er robot. Kom ihåg att krav kan prioriteras.

Uppdraget
Uppdraget består av autonom körning mellan olika positioner i ett enfiligt vägnät med 3-vägskorsningar. Positionerna utgörs av stopplinjer. Givet en karta över det aktuella vägnätet, en definierad startposition och en lista med destinationer ska bilen i tur och ordning autonomt åka till destinationerna via kortast möjliga väg. Bilen kan bara plocka upp passagerare från bilens högra sida, så bilen kan bara stanna vid en destination genom att komma från rätt håll. Det kan förekomma hinder i vägbanan, då bilen måste stanna tills hindret försvunnit. Bilen ska använda en kamera för positionering. Detaljer i banspecifikationen kan bestämmas mellan de deltagande grupperna och beställaren.

Moduluppbyggnad
För att senare kunna testa alternativa sensorer, fjärrstyrningar och även styralgoritmer, ska bilen vara moduluppbyggd. Gränssnitten mellan modulerna ska vara noggrant specificerade i den tekniska dokumentationen. Man ska enkelt kunna byta ut en modul mot en annan. Varje modul ska innehålla minst en egen processor. Följande tre moduler ska ingå i konstruktionen:
- kommunikationsmodul (med trådlös länk, blåtand eller wifi)
- styrmodul (motorer, eventuell display, styrlogik)
- sensormodul (hit ska alla sensorer vara inkopplade förutom kamera)

Fjärrstyrning och fjärrövervakning
Bilen ska kunna fjärrstyras och fjärrövervakas trådlöst från en bärbar dator i tre moder: manuellt, semi-autonomt och autonomt. Följande kommandon ska bilen reagera på i manuell körning: Gaspådrag inklusive fram/back samt styrutslag. I semi-autonomt läge förutsätts bilen köra i ett vägnät. Föraren ska endast skicka vägvisningskommandon som t ex följ vägen fram till nästa stopplinje, sväng vänster i korsningen, stanna. I de autonoma lägena ska det gå att skicka en karta och en startposition till bilen. I det helautonoma läget ska det dessutom gå att skicka ett köruppdrag till bilen. Under körning ska följande data visas i den bärbara datorn: körtid, körsträcka, hastighet, detektion och avstånd till hinder föröver, gaspådrag och styrutslag. Vid körning i de autonoma moderna ska aktuellt vägsegment och lateral position visas. Under köruppdrag ska planerad rutt till nästa destination visas. Dessa data ska presenteras på datorns skärm på ett användarvänligt sätt och dessutom gå att spara på ett format som gör det möjligt att plotta valda signaler som funktion av tiden för en given körning i t ex Matlab eller Python. Det vore trevligt om bilen skulle kunna skapa en karta över vägnätet med avstånd i semi-autonomt körläge. Under kartläggning ska bilen stanna vid destinationer och invänta ett destinationsnamn från användaren innan kartläggningen fortsätter. När alla vägar är utforskade ska bilen bekräfta att kartan är klar.

Diverse övriga funktionskrav
Det ska finnas någon form av styralgoritm (till exempel PD-reglering), så att bilen kan köra på vägen utan att ”slingra” sig fram (verifieras genom dokumenterade testkörningar). Parametrar till bilens styralgoritm ska kunna initieras trådlöst. Det vore trevligt om bilens position, nästa destination och planerad körväg visas på en karta. Dessutom vore det trevligt om kamerans bild kan visas på den bärbara datorn.

Tävlingsregler:
Vinnare är den bil som på kortast tid klarar att helt autonomt klara ett givet köruppdrag utan att kollidera med eventuella hinder. Repeterbarhet ska dock kunna uppvisas. Reglerna bestäms i samråd med beställaren.

Övriga krav
Projektet ska bedrivas enligt LIPS-modellen och samtliga dokument ska utgå från LIPS-mallar. I förefasen ingår att projektgruppen ska ta fram en kravspecifikation, en systemskiss och en projektplan med tidplan. Samtliga dessa dokument ska godkännas av beställaren. Budget för förefasen finns på beställarens hemsida. Efter godkänd projektplan (BP2) ska projektet ta 230 arbetstimmar/person att slutföra. Projektgruppen ska utföra kontinuerlig tidsredovisning som skickas till beställaren en gång per vecka. Vid verifiering av baskrav (BP5a) ska manuell och semi-autonom körning demonstreras. Detaljer för baskrav utarbetas i samråd med beställare. Vid slutleveransen (BP5b) ska det finnas en fungerande robot samt teknisk dokumentation med användaranvisning. Projektets delleveranser och slutleverans ska senast ske vid de datum som finns specificerade på beställarens hemsida. Även formen för slutleveransen beskrivs på denna hemsida.

B TEKNISK DOKUMENTATION

Teknisk Dokumentation

Henrik Ahlinder
Edvin Johansson
Mikael Josefsson
Sebastian Andersson
Martin Dahl
Charbel Youssef

7 juni 2021

Version 1.0

Status

Granskad	Alla	2021-05-19
Godkänd		

Projektidentitet

Grupp E-post: TSEA56_2021VT_O2_G10@groups.liu.se

Beställare: Anders Nilsson, Linköpings universitet
Tfn: 013-28 26 35
E-post: anders.p.nilsson@liu.se

Handledare: Peter Johansson
Tfn: +4613281345
E-post: peter.a.johansson@liu.se

Kursansvarig: Mattias Krysander
Tfn: 013-28 21 98
E-post: mattias.krysander@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Henrik Ahlinder	Projektledare (PL)	henah490@student.liu.se
Mikael Josefsson	Kommunikationsmodulansvarig, GUI-ansvarig	mikjo695@student.liu.se
Sebastian Andersson	Testansvarig	seban218@student.liu.se
Martin Dahl	Dokumentansvarig, mjukvaruansvarig	marda545@student.liu.se
Charbel Youssef	Sensormodulansvarig	chayo642@student.liu.se
Edvin Johansson	Styrmodulansvarig	edvjo404@student.liu.se

INNEHÅLL

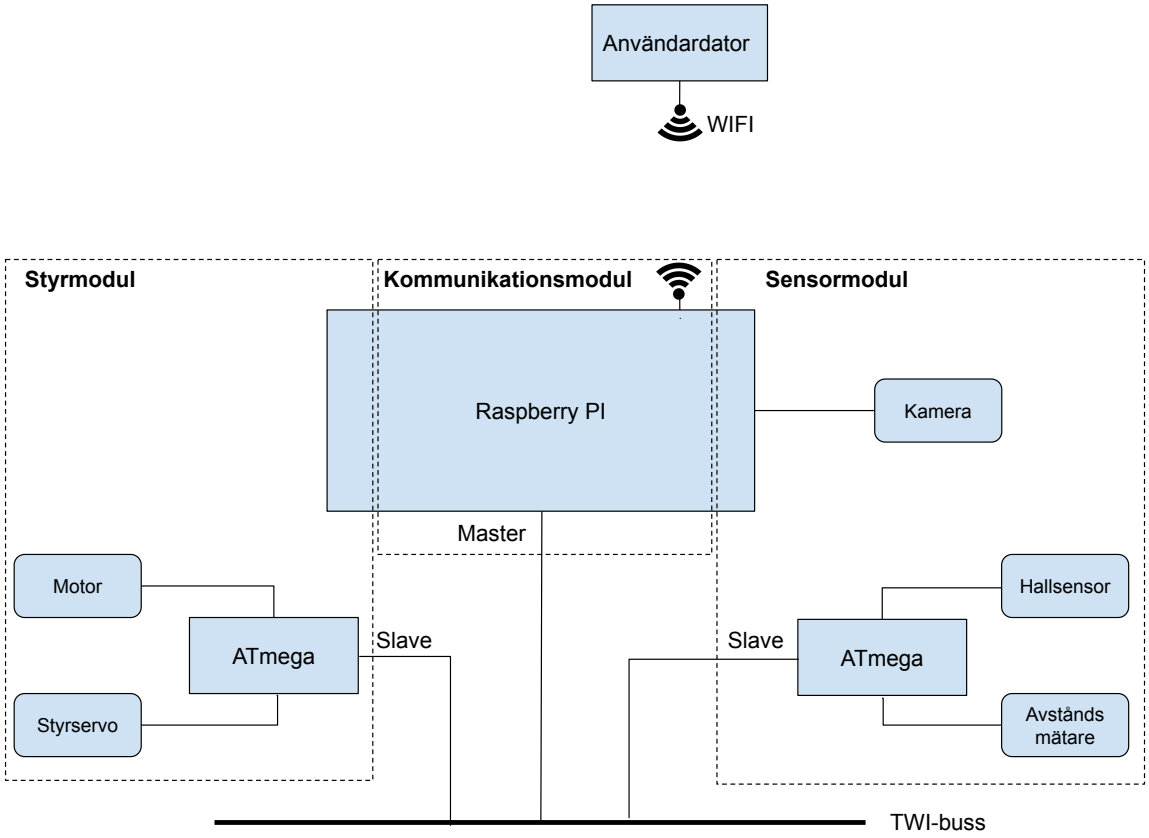
1 Inledning 1
1.1 Förtydligande av begrepp . . . 2
2 Kommunikationsmodulen 3
2.1 Hårdvara . . . 3
2.2 Mjukvara . . . 3
3 Styrmodul 9
3.1 Hårdvara . . . 9
3.2 Mjukvara . . . 11
4 Sensormodul och kamera 20
4.1 Hårdvara . . . 20
4.2 Mjukvara på Raspberry Pi . . . 22
4.3 Styrmodulens mikrokontroller . . . 29
5 Användardator 32
6 Allmänna figurer 35

DOKUMENTHISTORIK

Version	Datum	Utförda förändringar	Utförda av	Granskad
1.0	2021-05-19	Slutgiltig version	Alla	Alla

1 INLEDNING

Det har konstruerats en taxibil som kan köras i tre olika moder, Manuellt, Semi-autonomt och Autonomt. Bilen är uppdelad i tre moduler; sensor-, styr- och kommunikationsmodulen. Bilen kopplas även upp till en användardator som utgör en egen fristående modul. Sensor-, styr, samt kommunikationsmodulen är monterade på bilen medan användarmodulen består av ett gränssnitt som väntas köras från användarens dator. Bilens moduler är implementerade på tre olika processorer: en Rasperry PI och två ATmega mikrokontroller. En översiktsbild av hur modulerna är distribuerade på de olika processorerna ges i Figur 1.



Figur 1: *Översiktbild av systemet. Här ses hur processorer (spetsiga hörn) och övrig hårdvara (rundade hörn) är fördelade över de olika modulerna. Notera speciellt att Raspberry Pi:n används av alla de tre moduler som är monterade på bilen.*

Bilen konstruerades i ett chassi med förmonterat styrservo, motor och halleffektssensor. Utöver dessa komponenter nyttjas en ultraljudsensor och en kamera. Sensormodulen läser av sensorerna och generar information som styrmodulen kan använda för reglering och styrning av bilen. Sensorernas placering och funktion beskrivs i kapitel 4. Reglering och styrning av bilen beskrivs i kapitel 3.

Bilen kontrolleras via användardatorn. När användaren anslutit till bilen kan bland annat körmod, reglerparametrar och körinstruktioner väljas. Information såsom färdad sträcka, hastighet och avstånd till närmaste hinder skickas från kommunikationsmodulen till användardatorn och presenteras antingen i textformat eller grafiskt.

För att möjliggöra kommunikation mellan tre övriga moduler introduceras kommunikationsmodulen i kapitel 2. Kommunikationsmodulen upprätthåller och tillhandahåller kommunikationsmöjligheter mellan de övriga modulerna. Kommunikationsprotokoll och övriga detaljer om kommunikation mellan modulerna beskrivs i avsnitt 2.1.

1.1 Förtydligande av begrepp

- **Benämningar för användardatorn**: I dokumentet syftar ibland ”användargränssnittet” och ”användardatorn” på samma sak, det vill säga det program som användaren styr bilen från.
- **TWI**: TWI, SMBus och I^2C tolkas samtliga som samma kommunikationsstandard i detta dokument.

2 KOMMUNIKATIONSMODULEN

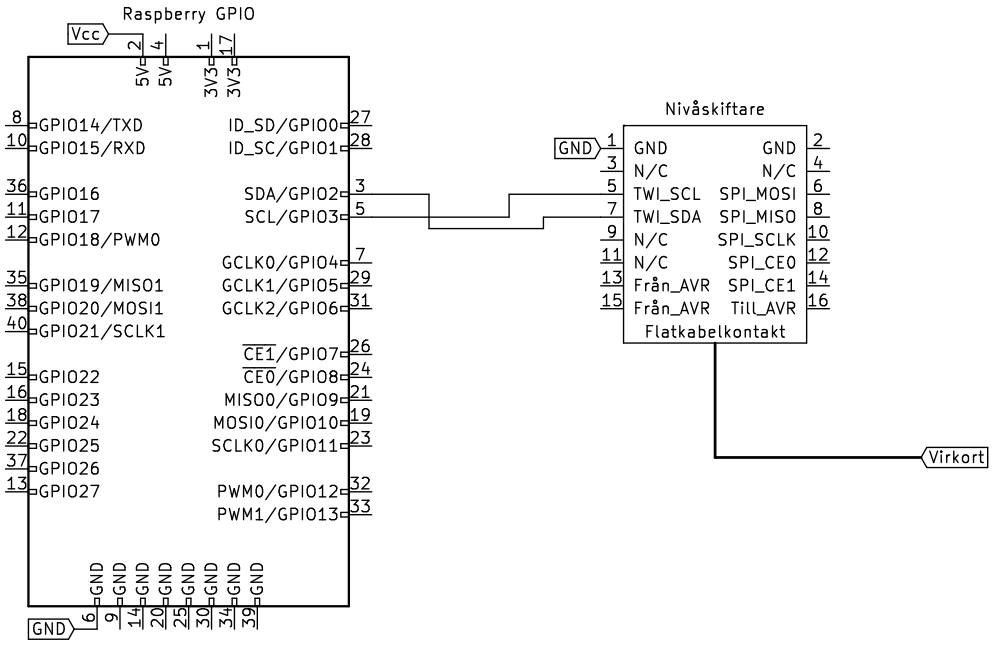
Detta kapitels syfte är att beskriva konstruktionen av kommunikationsmodulen, vars uppgift är att se till att transmissioner sköts tidsenligt och effektivt mellan övriga moduler. Detta sker via tre olika kommunikationsvägar:

1. Trådlös kommunikation mellan användardatorn och bilen.
2. Kommunikation mellan moduler monterade på olika platser på bilen
3. Kommunikation mellan de olika processerna som exekveras på Raspberry Pi.

Kommunikationsmodulen består delviss av en hårdvarukoppling i form av en I^2C-buss som realiserar kommunikation mellan moduler monterade på olika platser på bilen, samt delvis av mjukvara som driver de olika kommunikationsvägarna. Dessa två delar beskrivs i avsnitt 2.1 respektive avsnitt 2.2.

2.1 Hårdvara

I Figur 2 visas hur GPIO-pinnarna i mikrodatorn Raspberry Pi sammankopplas med övriga systemet via en nivåskiftare. En nivåskiftare krävs eftersom GPIO-pinnarna använder 3.3V-logik på mikrodatorn, medan övriga mikrokontroller använder 5V-logik. Notera att JTAG-anslutning inte används på denna modul eftersom mikrodatorn programmeras med hjälp av USB-anslutet tangentbord samt HDMI-ansluten skärm. Mikrodatorn drivs av ett operativsystem som installeras på ett microSD-kort som ansluts till mikrodatorn. Komponentbudgeten visas i Tabell 1.



Figur 2: *Kopplingsschema för GPIO-kontakter på kommunikationsmodulen.*

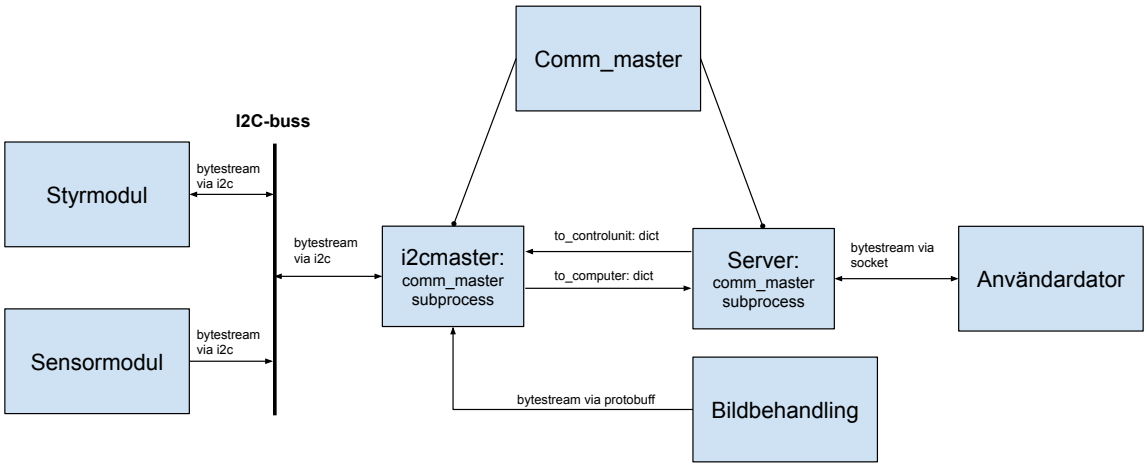
2.2 Mjukvara

I huvudsak delas mjukvaran i kommunikationsmodulen upp i två separata delar; en server och en hårdvarumaster. Dessa två delar implementeras i filerna *server.py* respektive *i2c_master.py*. Dessa båda implementationer knyts samman till ett exekverbart skript i *comm_master.py*, som knyter samman kommunikationen mellan samtliga moduler involverade i projektet, se Figur 3. De två programmen *server* och *i2c_master* är skrivna i Python och är barnprocesser till *comm_master*

Tabell 1: *Komponentbudget för kommunikationsmodulen. Tabellen innehåller både komponenter som krävs för att kommunikationsmodulen ska kunna utföra sin uppgift och komponenter som används vid programmering men som ej är nödvändiga vid exekvering.*

Användningsområde	Antal (st)	Komponent
Exekvering	1	Raspberry Pi 3B+
	1	Strömkälla 5V
	1	microSD-kort (minst 8GB)
	1	Nivåskiftare
	1	16-pin flatkabel
Programmering	1	Datorskärm med HDMI-kontakt
	1	HDMI-kabel
	1	USB-anslutet tangentbord
	1	USB-ansluten datormus

och delar data mellan varandra via Python *dictionaries*, vilket möjliggörs genom nyttjande av paketet *multiprocessing* [1]. Bildbehandlingen är skriven i C++, körs som en egen process och skriver behandlad data till *i2c_master* via en FIFO-buffer. Notera att bildbehandlingen är helt fristående, samt att kommunikationen i FIFO-buffern är enkelriktad mot kommunikationsenheten. Det går alltså ingen dataström från övriga program till bildbehandlingen.

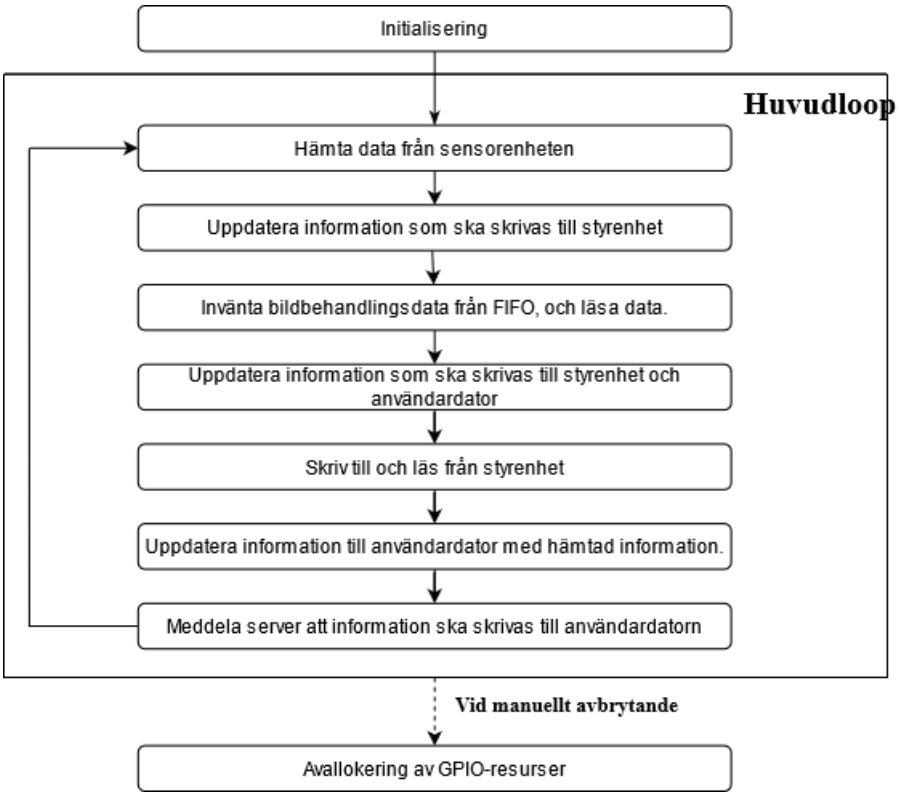


Figur 3: *Skiss över hur information skickas mellan processer.*

2.2.1 I^2C-Kommunikation

Detta avsnitts syfte är att utveckla och förklara hur kommunikationen mellan de tre separata processorerna på bilen sammansluts, och på så vis beskriva informationsflödet mellan modulerna; styr-, sensor- och kommunikationsmodulen; som är monterade på bilen. Informationsflödet mellan dessa moduler och användardatorn sker trådlöst och beskrivs i avsnitt 2.2.2.

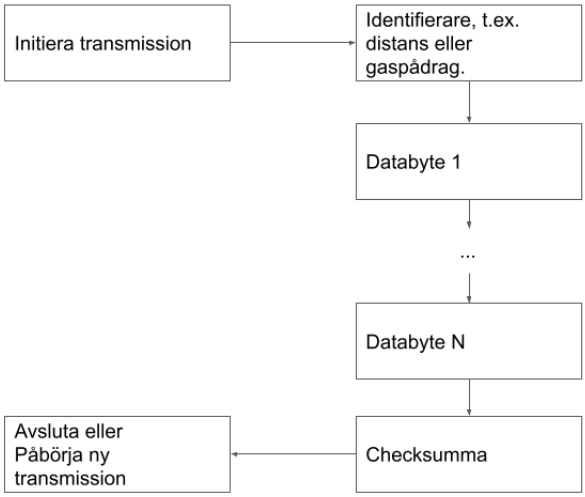
De tre processorerna ansluter till varandra via en I^2C-buss med en enskild *master*. Master i systemet är kommunikationsmodulen och de övriga processorerna agerar som *slaves*. Detta innebär att all datainformation som ska transmitteras mellan de olika delsystem går via kommunikationsmodulen. Master i I^2C-protokollet är Raspberry Pi, och realiseras i mjukvara i programmet *i2c_master*. Programmet gör ett initialt programmuppbyggande vid uppstart och exekverar därefter upprepat en huvudloop tills dess att användaren väljer att avbryta programmet. I varje iteration av programmets huvudloop skickar och mottar alla delsystem relevanta datapaket. Ett flödesschema över huvudloopen visas i Figur 4.



Figur 4: *Flödesschema för i2c_master. Vid uppstart sker en initialisering och sedan körs en huvudloop tills dess att programmet avbryts av användaren.*

Transmissioner över bussen består primärt av tre delement; en inledande datakod som indikerar vilken variabel som transmitteras, data som motsvarar variabelns värde, samt en avslutande checksumma som nyttjas vid kontroll av lyckad transmission. Checksumman generas genom att utföra operationen XOR på datakoden och samtliga databytes som representerar variabelvärdet. Transmissionssekvensen illustreras i Figur 5.

Läsning- och skrivningssekvenser skiljer sig från varandra i vissa detaljer. Vid skrivning från master till slave sänds ytterligare en extra byte som ger slaven möjlighet att svara med NACK om checksumman var felaktig, och master ska återsända variabeln. Vid läsning öppnar master först kommunikationen i skrivläge och skriver datakoden till slaven. Därefter görs en *Repeated-Start*, följt av öppnade av slaven i läsläge som då förväntas transmittera variabelns värde i form av databytes och checksumma. Transmissionens validitet kontrolleras sedan i master, som avgör huruvida checksumman stämmer med mottagna databytes eller ej. Därmed krävs inte den extra byten som används vid skrivning.



Figur 5: *Struktur för hur data skickas enligt kommunikationsprotokollet.*

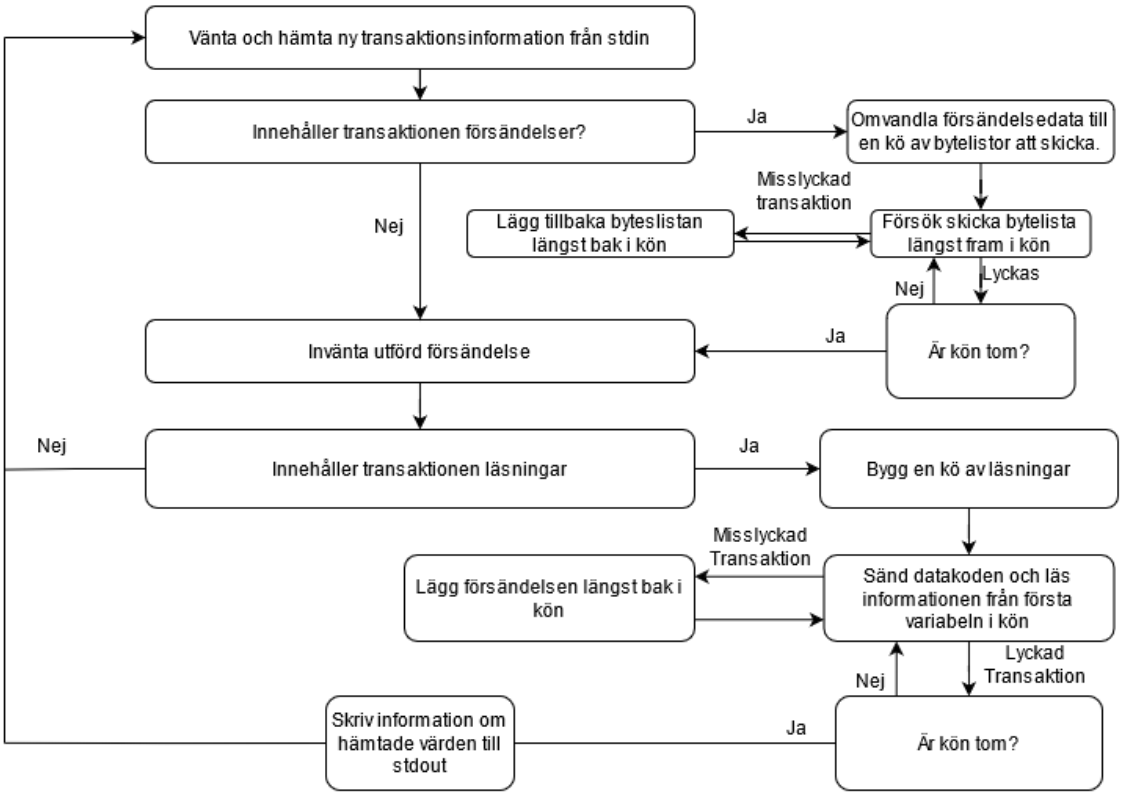
Overhead för transaktionen definieras genom alla transmitterade bytes som ej är datakod eller representerar ett variabelvärde för varje transaktion. Med detta mått ger kanalkodningen definierad ovan ett overhead på två bytes vid skrivtransaktioner och en byte vid lästransaktioner, utöver det overhead som protokollet I^2C medför.

Då ett transmissionsfel har uppstått, oavsett om det är via en mottagen NACK eller en felaktig checksumma, kommer transmissionssekvensen som misslyckades att placeras längst bak i kön av data att transmittera, och därefter återupptas transmissionsförsöken. Detta beteende upprepas tills dess att en *timeout* nås. Då startas *i2c_master* om efter en viss väntetid och transaktionsförsöken återupptas. Syftet med denna felhantering är att uppnå en robust kommunikation som ej kraschar mer än temporärt oavsett vad som sker på bussen, och förhoppningsvis återhämtar sig systemet från eventuella fel vid en omstart. Värden för timeout och väntetiden före omstart kan specificeras vid uppstart av *i2c_master* i filen *Comm_master.py*

Notera att eftersom det är omöjligt att med ovanstående protokoll särskilja av vilken anledning som ett NACK har mottagits under en transmission. Därför behandlar master alla transmissioner som innehåller ett NACK som att ett transmissionsfel har uppstått och reagerar enligt ovanstående beskrivning av protokollet för felaktiga transmissioner.

Ursprungligen skrevs all kod i *i2c_master*, som styr I^2C-kommunikationen, i programspråket Python med hjälp av biblioteket SMBus2 [2]. En begränsning med denna metod var att busshastigheten begränsades till 60 kBaud, vilket mot slutet av projektet innebar att transaktionstiderna överskred vad som ansågs vara en rimlig begränsning. På grund av detta skrevs ett nytt program i programspråket C++ vars syfte var att kontrollera transaktionera över I^2C-bussen, och tillåta en högre busshastighet. Koden i C++ använder paketet PiGPIO för att hantera hårdvaran i transaktionerna [3]. Skalkoden i *i2c_master* som knyter samman bildbehandlingen och servern med I^2C-bussen förblev densamma, skriven i Python, men parallellt med denna skalkod löper den externa drivrutinen vars *stdin* och *stdout* knyts till var sin anonym *pipe*. När transaktioner över I^2C-bussen ska utföras i programmets huvudloop serialiseras transaktionsinformationen via Googles *Protocol Buffers* och skrivs till drivrutinens stdin. Drivrutinen deserialiserar information, utför transaktionerna och returnerar sedan hämtad information till skalkoden via skrivning till stdout, som skalkoden läser från. Drivrutinens programloop beskrivs i Figur 6.

Nedan följer två tabeller vars syfte är att underlätta läsningen av koden genom att markera viktiga variabler som nyttjas frekvent. Viktiga variabler i *i2c_master* ses i Tabell 2. Kommunikationsstandarder, förlagda i filen *communication_standards.py*, ses i Tabell 3.



Figur 6: *Flödesschema för drivern för TWI-bussen.*

Tabell 2: *Viktiga variabler i i2c_master.*

Variablennamn	Typ	Beskrivning
self.from_<enhet>	Dictionary	Definierar vilka variabler som ska hämtas från enheten. Värden som har hämtats uppdateras som värden till respektive nyckel i dictionaryt. Notera att dessa aldrig töms, utan variabler här i hämtas alltid.
self.to_<enhet>	Dictionary	Definierar variabler samt värden som ska skickas till enheten. Variabenamnet sätts som nyckel och variabelvärdet som värde. Vid transmission till C++-drivrutinen konverteras variabelnamnen från strängar till datakoder. Notera att dessa dictionarys töms efter varje transaktion för att inte skicka samma värde flera gånger till samma enhet.
self.timeout	Flyttal	Definierar hur länge enheten försöker kommunicera med en slav före den bestämmer sig för att slaven har hängt sig. Ursprungligt värde är 1 sekund.
self.restart_time	Flyttal	Specifierar hur länge mastern pausar innan den startar om vid en krasch.
self.stop_buffer	Objekt	Då vissa värden ger upphov till artefakter från bildbehandlingen buffras dessa och ett medelvärde ges istället. Detta för att undvika att falskt detekterade stopp- eller kantlinjer ej ska påverka regleringen drastiskt och orsaka urspåring.
self.pipemsg	Objekt	Klass genererad genom Google Protocol Buffer. Nyttjas vid mottagande och avserialisering av meddelande från bildbehandlingsprogram, vari ett liknande objekt nyttjas för att serialisera meddelandet.

Tabell 3: *Innehåll i filen communication_standards.py*

Namn	Typ	Används till
DATACODES	Dictionary	Definierar datakoder (värden) till variabelnamn (nycklar).
GET_DATATYPE	Dictionary	Definierar datatypen (värden) för variabler genom deras variablenamn (nycklar).
VARIABLENAMES	Dictionary	Inversen av DATACODES. Konverterar datakoder (nycklar) till variabelnamn (värden).

2.2.2 *Server*

Den server som används för att kommunicera med användardatorn är skriven i Python och använder modulerna *Socket* och *Select*. Med hjälp av *Socket* tillhandahålls en ändpunkt i nätverket som identifieras av en ip-adress och en port som klienter kan ansluta till. Nätverksprotokollet som används är TCP/IP. När en klient ansluter erhålls en socket som pekar på klienten. Modulen *Select* berättar vilka klient-sockets som är läsbara och skrivbara. I teorin kan flera användardatorer ansluta sig samtidigt men den funktionaliteten har inte undersökts.

Mellan sockets skickas data som en ström av bytes. På mottagarsidan läggs bytesen i en buffer. För att kunna avkoda data och skilja mellan olika datapaket på buffern används ett kommunikationsprotokoll, se Figur 7.

header	data

Figur 7: *Illustration av kommunikationsprotokollet som används tillsammans med Sockets.*

Varje datapaket som skickas inleds med en header som alltid är 10 bytes lång. Headern berättar hur många bytes data som ska läsas ifrån buffern. På så vis är det tydligt vilka bytes som hör till vilket datapaket. Data som skickas mellan server och klient är alltid av typen JSON.

Som nämnt i avsnitt 2.2 skickas data mellan *server* och *i2c_master* via gemensamma dictionaries. När servern mottar data från användardatorn så tolkar den om datat till en dictionary och uppdaterar dictionaryt *to_controlunit*. På samma sätt skickas data till användardatorn genom att *i2c_master* uppdaterar dictionaryt *to_computer*. Servern översätter sedan detta dictionary till JSON och transmitterar bytesströmmen till användardatorn.

2.2.3 *Kommunikation med bildbehandling*

För att sammansluta bildbehandlingsprogrammet med övriga enheter skapas en FIFO-buffer automatiskt vid uppstart av *Comm_master*, se avsnitt 2.2. Bildbehandlingsinformation som ska transmitteras till övriga moduler serialiseras till en bytesström via Googles Protocol Buffers [4] före överföring, genom skrivning till FIFO-buffern som moniteras av *i2c_master*. Genom att nyttja detta protokoll säkerställs lätt utbytbara och expanderbara moduler. Detta eftersom att vid byte av bildbehandlingsprogram krävs endast att i den nya koden inkludera *pipeprotocol.pb.h*, bygga ett meddelande av typ *Imagemessage*, fylla i nyttjade fält, serialisera meddelandeinformationen via en av de inbyggda metoderna samt slutligen att skriva meddelandet i FIFO-buffern. Vid expandering av transmitterad information krävs att filen som definierar meddelandet, *pipeprotocol.proto* uppdateras, återkompileras samt att nödvändiga fält fylls i före transmission samt extraheras i mottagaren *i2c_master*.

Mottagen data från bildbehandling avserialiseras och transmitteras till användardatorn och styrmodulen via uppdateringar av respektive dictionary som beskrivs i avsnitt 2.2.1.

3 STYRMODUL

Styrmodulens uppgift är att givet sensordata styra bilen till dess destination inom ett väldefinierat vägnät. Detta utförs med olika grad av autonomi beroende på vilket körläge som bilen befinner sig i.

- **Manuellt läge**: Bilens motor och styrservo styrs helt av en användare på användardatorn med hjälp av en handkontroll. Styrmodulen hanterar dock motor och styrservo på låg nivå. Användaren behöver enbart önska ett gaspådrag och ett styrutslag med hjälp av spakarna på kontrollen.
- **Semi-Autonomt läge**: Användaren kan önska att bilen ska hålla till höger eller till vänster vid korsningar. Användaren kan också önska att bilen stannar, kör framåt till nästa destination på vägen, eller kör framåt utan hänsyn till destinationer. Dessa kommandon skickas från användardatorn under körning för att ändra bilens beteende i realtid.
- **Autonomt läge**: Användaren kan genom användargränsnittet begära att bilen ska köra till en viss destination i vägnätet, från en given punkt där bilen antas starta. Styrmodulen beräknar då den snabbaste vägen samt de gaspådrag och styrutslag som krävs för att ta sig dit.

Säkerhetsfunktioner som gör att bilen stannar ifall den är på väg att köra in i något eller ifall kommunikation med övriga moduler upphör är aktiverade i samtliga körlägen.

Styrmodulen omfattar alltså system för att hantera motor, autonom styrning längs med väglinjer samt beslutsfattning för navigation. Dessa system inom styrmodulen beskrivs i kommande avsnitt.

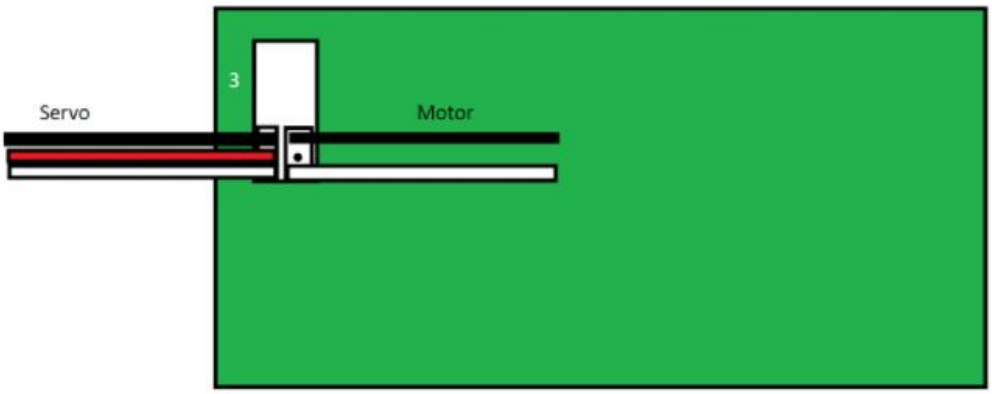
3.1 Hårdvara

I detta avsnitt beskrivs den hårdvara som används för att implementera styrmodulen.

Bilchassi
Styrmodulen utgår från det bilchassi som tillhandages av kursledningen. Styrmodulen måste därför enbart generera insignaler till bilchassits motor och styrservo.

Gaspådrag och styrutslag sätts med pulsbreddsmodulering, *PWM*. Specifik information om period och aktiv tid för PWM hittas i databladet för bilchassit [5].

Koppling av motor och styrservo till virkortet ses i Figur 8. Svart och röd kabel används i vanlig mening för jord respektive 5V, vit kabel används för PWM-signalen. Notera att motorn saknar anslutning för 5V från virkortet då den kräver en högre ström och därför drivs från separat strömkälla. Virkortet är i sin tur kopplat till en mikrokontroller, enligt kopplingsschema i Figur 9, som genererar PWM-signalerna för motorn och styrservot.



Figur 8: *Koppling av motor och styrservo till virkortet. Virkortet är i sin tur kopplat till en mikrokontroller som genererar PWM-signalerna för motorn och styrservot.*

ATmega1284P
Övervägande del av styrmodulen är implementerad på mikrokontrollern ATmega1284P [6]. Styrmodulens mikrokontroller sköter följande: allmän hantering av bilens körlägen med en tillståndsmaskin, regulatorer för farthållare och väglinjeföljning, generering av PWM-signaler för motor och styrservo samt reglering av inbromsning vid hinder och destinationer.

Mikrokontrollern är monterad på ett virkort, se Figur 28 markering 1, och är inkopplad till övrig hårdvara på bilen enligt Figur 9. Mikrokontrollern programmeras i programmet Atmel Studio version 6.2 där mikrokontrollern ansluts till datorn via *JTAG*-USB, som visas i Figur 31.

Raspberry PI
Beräkning av optimal färdväg är implementerat på samma Raspberry Pi som kommunikationsmodulen och sensormodulens bildbehandling använder. Anledningen till det är att detta görs mer smidigt med hjälp av en mer avancerad datastruktur som är enklare att implementera i programspråket Python än i C, som är det programspråk som mikrokontrollern använder. Lagring av en bana kräver även potentiellt mycket minne vilket gav ytterligare en anledning till att implementera dessa delar på Raspberry Pi:n, då mikrokontrollern endast har 16kB SRAM. Se avsnitt 3.2.6 för implementation av ruttplanering. Avsnitt 4.2.6 beskriver hur ruttplaneringen kopplas ihop till bilens positionering.

Lysdioder
Lysdioder används för att ge användaren snabb feedback på styrmodulens nuvarande läge och informera om särskilda händelser. Dioderna kan ge information om bilen befinner sig i Manuellt, Semi-autonomt eller Autonomt läge. Dioderna avslöjar också om bilen står still på grund av att den detekterat ett hinder, ifall den har tappat kontakten med användardatorn eller ifall den står vid en destination. Lysdioderna ses i Figur 28 markering 6. Lysdiodernas koppling till styrmodulens mikrokontroller syns i Figur 9. Exakt hur lysdioderna är definierade beskrivs i avsnitt 3.2.7.

Resetknapp
På virkortet finns en *resetknapp* för hårdvaruåterställning av mikrokontrollern. Se Figur 28 markering 4. Inkoppling av resetknappen syns i Figur 9.

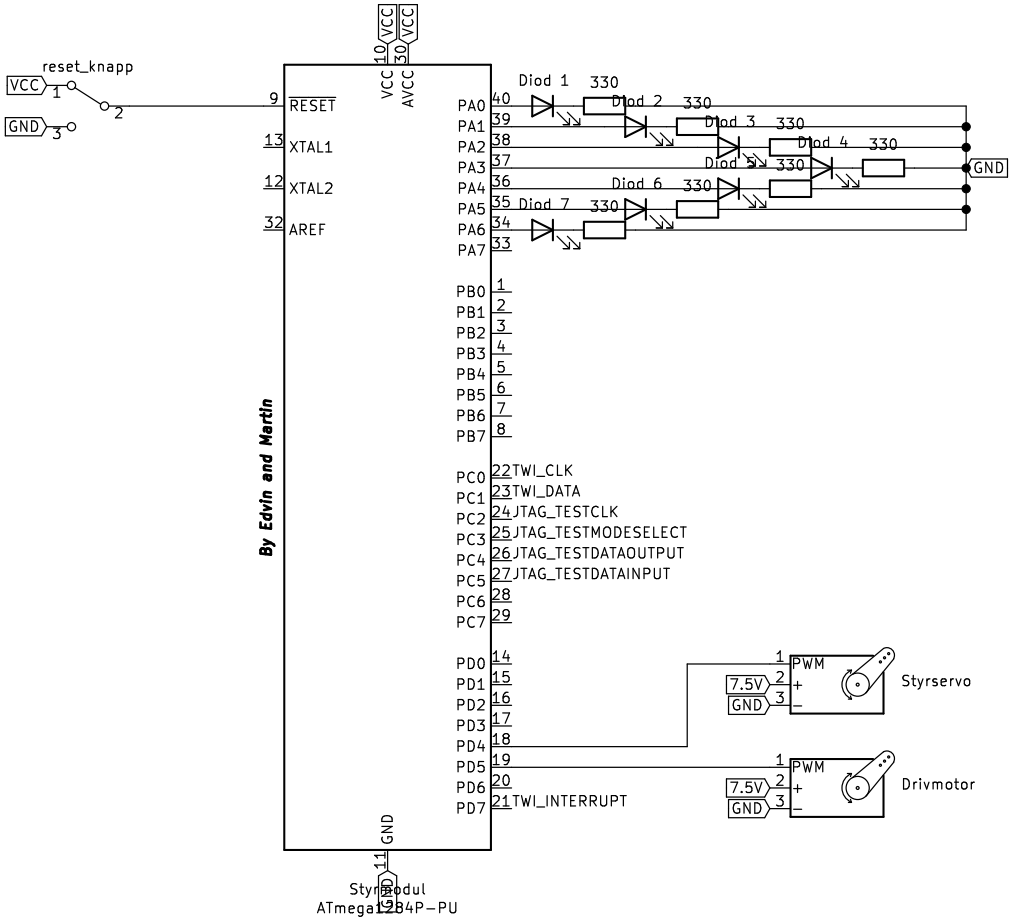
Virkort
Virkortet är det kort som flera utav styrmodulens komponenter är monterade på, se Figur 28.

3.1.1 *Kopplingsschema för mikrokontroller*

Kopplingsschemat för styrmodulens mikrokontroller ses i Figur 9. Funktionen hos de pinnar på ATmega1284P som används ses i Tabell 4.

Tabell 4: *De pinnar som används i styrmodulen och en beskrivning av deras funktion.*

Pin	Funktion
PA0-PA6	Pinnarna är kopplade till lysdioderna som används för att signalera bilens tillstånd.
PC0 och PC1	Pinnarna är kopplade till klockpuls respektive data på I^2C-bussen.
PC2-PC5	Pinnarna är kopplade till JTAG. JTAG används för att programmera mikrokontrollern.
PD4 och PD5	Pinnarna är kopplade till styrservots respektive motorns PWM-ingångar.
RESET	RESET pinnen är kopplad till en knapp för att det ska vara möjligt att manuellt utföra en hårdvaruåterställning av styrmodulen.



Figur 9: *Kopplingsschema för styrmodulen.*

3.2 Mjukvara

I detta avsnitt beskrivs hur styrmodulens mjukvara är konstruerad. Om inte annat nämns beskriver delavsnitten nedan mjukvara som implementerats på styrmodulens mikrokontroller ATmega1284P. I specialfallet då Raspberry Pi:n används kommer detta att nämnas. För att se implementationen i detalj hänvisas läsaren till Appendix; detta avsnitt är avsett för förklaring och motivering av mjukvaran.

3.2.1 *Överblick av programmet*

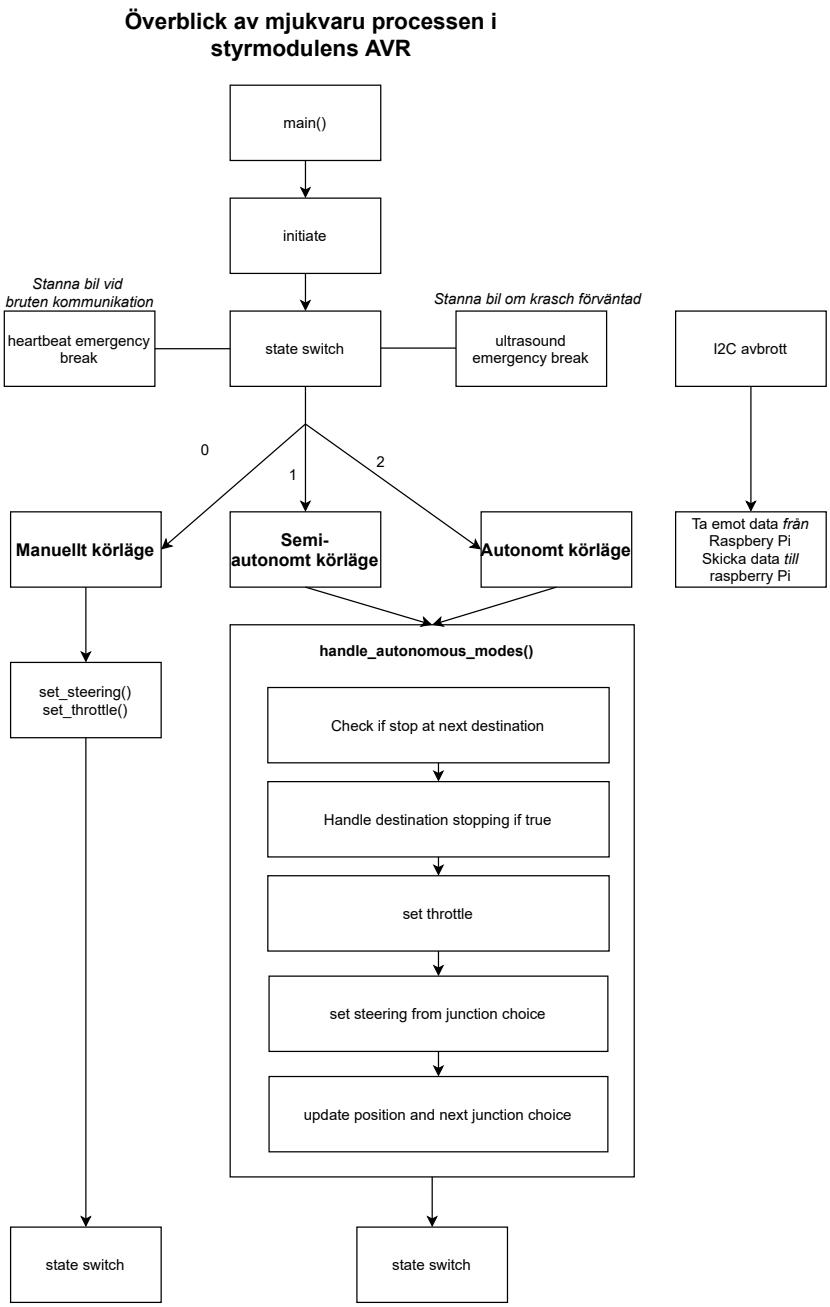
I avsnittet beskrivs överskådligt hur mjukvaran är uppbyggd, i Figur 10 illustreras programmets flöde.

1. **Initiering**: Detta steg körs endast en gång och initierar alla variabler, registervärden, timers och lågnivå-funktionalitet hos mikrokontrollern som behövs för det fortsatta programmet.

2. **Ändra tillstånd**: Denna del upprepas kontinuerligt och hanterar programmets flöde. Här väljs vad som skall göras beroende på vilket tillstånd som bilen befinner sig i. I Figur 10 är även *heartbeat emergency break* och *ultrasound emergency break* kopplade till state switch:en. Denna koppling innebär att dessa sker oberoende av körläge.

3. **Styrning**: Programmet fortsätter in i någon av de tre körmoderna beroende på vad användaren har satt körmodet till. Vad som görs i de enskilda körmoderna beskrivs i avsnitt 3.2.2.

4. **Kommunikation** Uppdatering av styrmodulens värden från kommunikationsmodulen sker avbrottsstyrt utan hänsyn till programmet i övrigt.



Figur 10: *Överblick av styrmodulens mjukvara. Figuren beskriver endast mjukvaran översiktligt.*

3.2.2 *Implementation av körlägen*

I detta avsnitt beskrivs hur de olika körlägena är implementerade.

Manuellt körläge

Det Manuella körläget sätter gaspådrag och styrutslag enligt *manual_throttle* och *manual_steering*. Dessa värden kommer från användarens handkontroll. Inkomna värden hanteras i $set_throttle()$ och $set_steering()$ som omvandlar dem till motsvarande arbetscykel för PWM-signalerna som driver motorn och styrservot.

I det manuella körläget kontrolleras kontinuerligt om bilens ultraljudsensor avger ett avstånd som kan innebära att bilen kommer att kollidera med ett föremål framöver. Ifall så är fallet stannar bilen och inget gaspådrag framåt sätts, oavsett vad användaren anger. Det är dock tillåtet för bilen att backa så att den kan ta sig bort från hindret.

Semi-autonomt körläge
I det Semi-autonoma läget tas användarens kommandon emot från användardatorn under körning. Dessa kommandon säger om bilen ska hålla till höger eller till vänster i korsningar samt om bilen ska stanna vid nästa destination eller köra förbi den. Användaren kan också välja att stanna bilen där den är oavsett ifall det finns en destination där eller inte.

När bilen befinner sig i det Semi-autonoma läget så håller den sig inom väglinjerna med hjälp av regulatorn, beskriven i avsnitt 3.2.3, och data från sensormodulen. För väglinjeföljning används bilens hastighet samt avstånden till höger och vänster väglinje.

Likväl som i det manuella körläget återkopplar systemet aktivt information från ultraljudsensorn för att kontrollera om bilen är på väg att kollidera med ett objekt framöver. Till skillnad mot det manuella läget så tillåts inte bilen att backa när den kör autonomt, så bilen tvingas istället vänta tills att hindret har avlägsnats.

Autonomt körläge
Det Autonoma körläget är mycket likt det Semi-autonoma körläget. Skillnaden är att bilen i detta läge själv måste beräkna en optimal sekvens av svängar för att köra från en startdestination till en slutdestination. I själva verket exekveras samma kod på mikrokontrollern i både det semi-autonoma och i det autonoma läget. Skillnaden mellan körlägena är att en längre sekvens av körinstruktioner skickas från ruttplaneraren då bilen kör i det autonoma läget, medan dessa instruktioner fås en efter en från användaren då bilen kör i det semi-autonoma läget.

Algoritmen för ruttplaneringen beskrivs i avsnitt 3.2.6. För att köra autonomt behöver bilen även bestämma sin position i vägnätet, hur detta görs beskrivs i avsnitt 4.2.6. Det autonoma läget kräver även en karta över vägnätet. Den fås efter att ha skapats i användargränssnittet *Kartbyggaren*, se avsnitt 3.2.5 för mer information om kartbyggaren.

3.2.3 *Reglering*

Bilen använder sig av två regulatorer, en för att reglera hastigheten och en för att reglera styrutslaget, även kallade farthållaren respektive väglinjeföljaren. Målet med farthållaren är att hålla en jämn fart, så nära en given referenshastighet som möjligt. Väglinjeföljarens uppgift är att se till att bilen kör inom väglinjerna, håller sig mitt på vägen samt kör på ett mjukt sätt utan att svänga fram och tillbaka över körbanan. För att göra detta används två PID-regulatorer. För en ingående beskrivning av en PID-regulator och dess digitala implementation hänvisas läsaren till Appendix Förstudie - Regler.

Reglerparametrar och referensvärden till de båda regulatorerna kan sättas och ändras under körning från användardatorn. För båda regulatorerna finns möjlighet att sätta godtyckliga P-, I- och D-parametrar. Farthållaren uppvisar en god prestanda med endast en P-del och en I-del. Användaren råds därför att inte använda någon D-del men hon har möjlighet att göra det. Väglinjeföljaren uppvisar bäst prestanda med endast en P-del och en D-del, men återigen är det upp till användaren att välja vilka parametrar hon vill använda.

Istället för att försöka hålla bilen mitt emellan de båda väglinjerna så reglerar väglinjeföljaren enbart efter avståndet till den högra- eller avståndet till den vänstra kantlinjen. Eftersom vägens bredd är konstant så har regulatorn som mål att hålla bilens avstånd från ena kantlinjen som halva vägens bredd. Detta gör regleringen mer robust då den lättare kan

hantera fall då en av linjerna inte kan detekteras av bildbehandlingen. Ifall den kantlinje som inte regleras efter inte kan detekteras så påverkas givetvis inte regulatorn av det. Ifall den kantlinje som regleras efter inte kan detekteras så tolkas det som att kantlinjen har försvunnit ut ur bilden, som till exempel den inre linjen i en kurva. Avståndet sätts då högt för att generera ett stort reglerfel och därmed ett stort styrutslag som svänger bilen tillbaka mot den väglinje som försvunnit ur bild. Bildbehandlingens robusthet gör att detta antagande väldigt ofta stämmer. Med en sådan reglering är det också enkelt att få bilen att svänga som önskat. Det räcker nämligen med att ange ifall bilen ska reglera efter höger eller vänster väglinje för att få bilen att svänga åt respektive håll i en korsning. Notera att vägnätet inte innehåller några fyrvägskorsningar vilket tillåter lösningen.

Som bekant kräver en PID-regulators dynamiska element ett minne. För att undvika oförutsedda beteenden så nollställs dessa minnen vid ändring av körläge.

Om I-delen är aktiverad i någon av regulatorerna så integreras (summeras) kontinuerligt det aktuella reglerfelet. Om bilen befinner sig i ett läge då regulatorn av någon anledning inte har möjlighet att påverka motor eller styrservo på ett normalt sätt så kommer denna integral att växa till ett mycket stort värde. Ett exempel på ett sådant läge är när bilen har stannat för ett hinder, farthållarens integrator kan då fortfarande vara aktiverad utan att dess värde används. Då bilen återgår från sådana tillstånd nollställs regulatorernas minnen för att undvika oönskade beteenden.

3.2.4 *Lågnivåhantering av gaspådrag och styrutslag*

För att styra motorn och styrservot behöver en PWM-signal skickas, där arbetscykeln avgör vilket gaspådrag och styrutslag som sätts. 5% arbetscykel ger det mest negativa gaspådraget alternativt styrutslaget, och 10% ger det mest positiva. Periodtiden för PWM-signalen är 20 ms vilket innebär att den aktiva tiden bör vara mellan 1 och 2 ms. I fallet med styrutslaget innebär negativa värden att bilen svänger åt vänster och positiva att den svänger åt höger. Användarens kommandon eller utsignaler från någon av regulatorerna anges mellan -100 och 100, och måste därför omvandlas till motsvarande pulsbredd för PWM-signalen. Detta görs enkelt genom sambandet

$$\text{pulsbredd} = 1,5 + \frac{\text{gaspådrag}}{200}. \tag{1}$$

För att generera dessa PWM-signaler används specialiserad hårdvara i mikrokontrollern. Denna hårdvara arbetar i termer av i hur många klockcykler som signalen ska vara hög, och i hur många cykler som den ska vara låg. En ytterligare konvertering från millisekunder till klockcykler behöver därför göras.

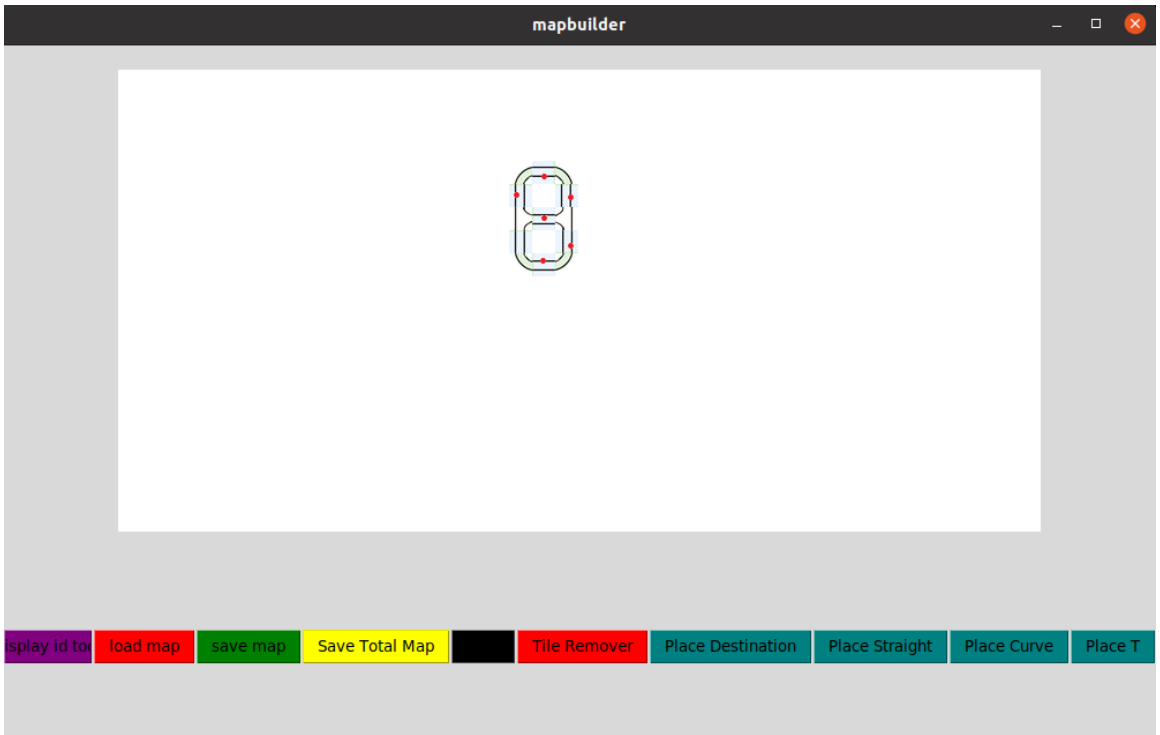
Ett maximalt gaspådrag innebär en väldigt hög hastighet. För att ett sådant inte ska sättas av misstag finns det en spärr implementerad som begränsar hur stort beloppet av gaspådraget tillåts vara. För maximal säkerhet är denna spärr hårdkodad och kräver omkompilering av programmet för att ändras. Det rekommenderas inte att en användare gör detta.

3.2.5 *Inmatning av körbana*

För att kunna bestämma en optimal rutt krävs först kunskap om körbanans utseende. Därför behöver användaren kunna mata in den nuvarande körbanan till bilen. Detta görs med hjälp av programmet *Kartbyggaren*, som är byggt med Python och med paketet *Tkinter*. Kartbyggaren innehåller både ett grafiskt gränssnitt för att användaren ska kunna mata in en karta, och en algoritm som skapar en graf utifrån vad användaren matat in. Se Appendix Användarmanual för en mer ingående beskrivning av hur kartbyggaren används.

I kartbyggaren kan man till exempel bygga en vägbana som ser ut som en åtta, se Figur 11. Det som ses i kartbyggarens fönster är en grafisk beskrivning av banan, som är användbar för en människa eftersom den i hög grad överensstämmer med körbanans utseende i verkligheten. I en dator är det å andra sidan mer användbart med en mer matematisk graf-beskrivning av banan, uppbyggd med noder och bågar. En sådan graf behövs för att sedan kunna utföra positionering och ruttplanering i det autonoma läget. Grafen förs över till bilen som en nästlad Python *dictionary* som innehåller information om noderna. Denna information är vilka bågar som noden har, bågarnas kostnaderna, och om noden repre-

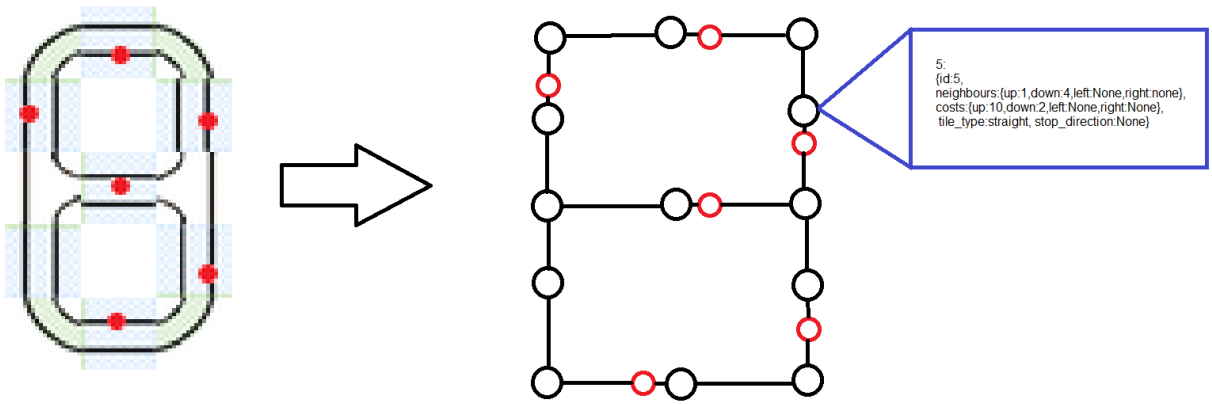
senterar en destination (visas visuellt som röda prickar i Figur 11) eller ett annat vägsegment som exempelvis en korsning. Destinationsnoder innehåller även information om vilken sida av vägen som de befinner sig på.



Figur 11: *Exempel på hur en karta ser ut när den byggs i kartbyggaren. De röda prickarna representerar destinationer och kan placeras längs med de båda vägkanterna.*

I Figur 12 visas hur det visuella vägnätet från kartbyggaren omvandlas till en graf. Det vägnät som bilen kan köra på har vissa begränsningar. Vägnätet kan beskrivas av ett rutnät bestående av tomma segment, raksträckor, 90-graders svängar eller T-korsningar; som alla är roterade i en av fyra möjliga riktningar och ryms inom en ruta om 1,5x1,5 m. Med den vetskapen kan bågarnas kostnad enkelt bestämmas. Beroende på hur destinationerna placeras inom ett segment ges de en viss kostnad till dess närliggande noder. Detta har inte varit en smärtfri metod då ett segment i kartbyggaren endast består av 20x20 pixlar. Så en användare kan endast, även med pixel-perfekt precision, placera en destination med en knapp decimeters noggrannhet. Detta har gjort att viss manuell justering av bågarnas kostnader har behövt göras i efterhand.

Notera hur samtliga (icke-tomma) segment i vägnätet är representerade av en nod i grafen, även i fallet då en destination sitter mitt i segmentet. I fallet då en destination sitter exakt mitt i ett segment kommer kostnaden mellan segmentet och destinationen att bli 0, eftersom segmentets position definieras som den punkt längs med vägen där bilen har tagit sig halvvägs igenom segmentet.



Figur 12: *Den visuella representationen av kartan, till vänster i bilden, omvandlas till en graf, till höger i bilden. Den blå rutan visar exempel på hur nodernas information är formatterad.*

3.2.6 *Ruttplanering*

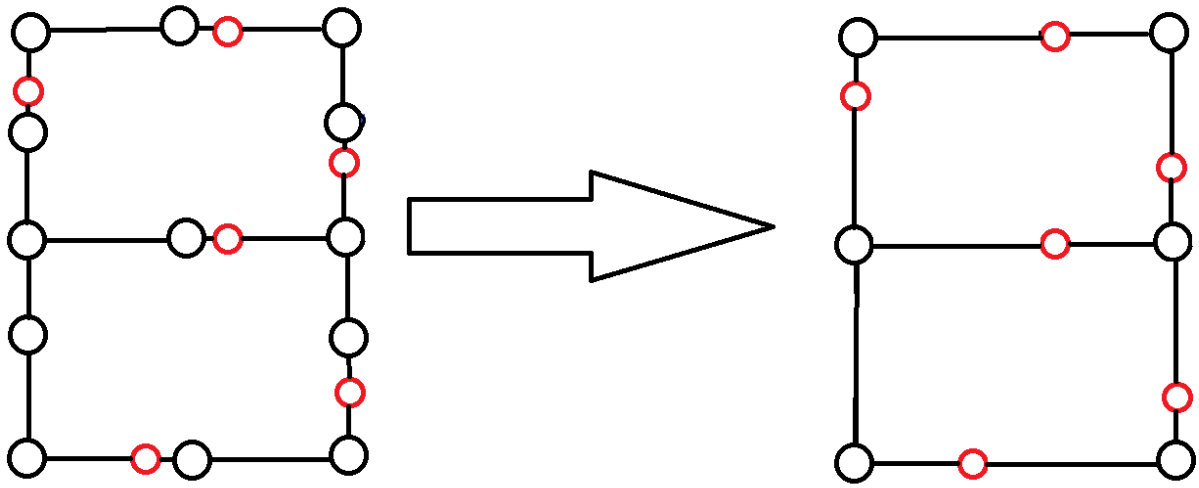
Ruttplaneringens uppgift är att, i det autonoma körläget, ta fram den snabbaste vägen mellan två destinationer och generera en sekvens av svängar som bilen ska utföra för att följa denna. För att göra detta används Dijkstras algoritm [7] med mindre modifikationer. All ruttplanering görs i Python på Raspberry Pi:n.

Något som skiljer detta problem från ett klassiskt kortaste-väg-problem är att destinationerna endast kan nås ifall bilen har dem på höger sida om sig, vilket kräver att bilen anländer vid en destination från ett visst håll. På grund av bilens och vägnätets fysiska begränsningar kan inte heller U-svängar göras, varken på vägar eller i korsningar. För att ta hänsyn till dessa begränsningar skapas en utökad graf, utifrån kartbyggarens graf, där varje nod utökas till två eller tre noder, som indikerar vilket håll bilen anlände i noden ifrån. Till exempel skulle noden för en trevägskorsning <A> utökas till de tre noderna <A, vänster>, <A, höger> och <A, nedåt>. Noder som är ointressanta, i den bemärkelse att inga beslut behöver tas där, optimeras bort, se Figur 13. Noder för kurvor behålls dock eftersom det i tidiga iterationer av reglersystemet var intressant att kunna hålla sig närmare inner- eller ytterkanten i en kurva, och algoritmen har tillräckligt hög prestanda även med dessa överflödiga noder. Bågar som skulle innebära ej realiserbara svängar kan, med den utökade grafen, tas bort och det går enkelt att specificera vilken av dessa riktade noder som måste nås för att bilen ska ha en destination på sin högra sida. Efter att ruttplaneraren har gjort om grafen ser grafen ut som i Figur 14. Med en sådan representation av vägnätet fås 2-3 gånger så många noder men eftersom dubbelriktade bågar ersätts med enkelriktade bågar hålls det totala antalet bågar relativt konstant, vilket gör att beräkningstiden inte ökar avsevärt.

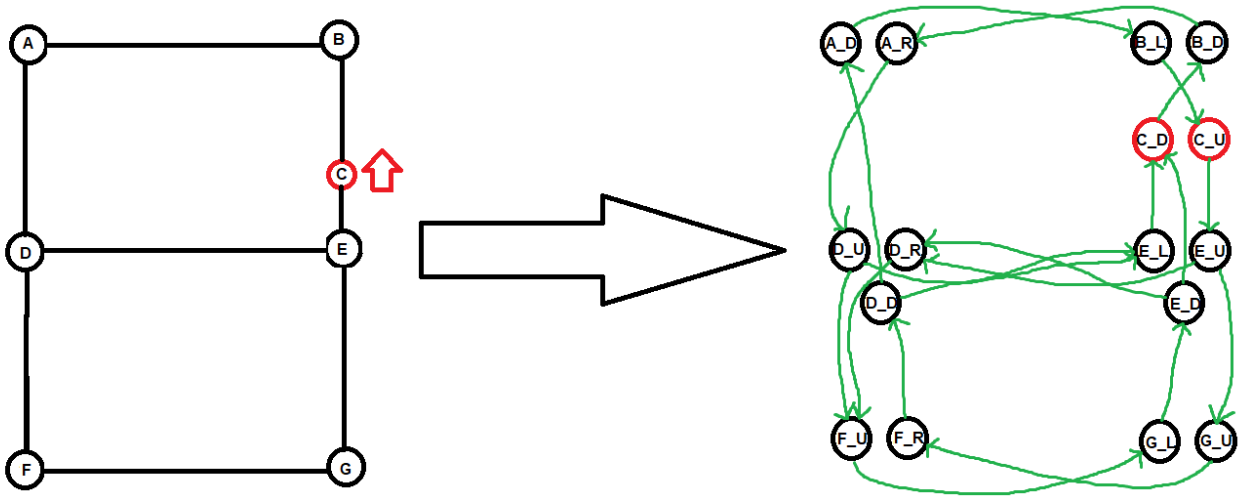
Kostnaderna i grafen bör vara tiden det tar att köra mellan två noder, då målet är att köra rutter på så kort tid som möjligt. I grafen används dock sträckan mellan två noder som kostnad då denna är enklare att mäta, och kan användas utbytbart under antagandet att bilen håller en konstant fart. Bilens farthållare gör att detta antagande gäller.

I implementationen av Dijkstras algoritm har ett tillägg gjorts som gör det möjligt att lägga till extra kostnader ifall bilen svänger i en kurva eller korsning. Detta för att ta hänsyn till att bilen behöver sakta ned när den svänger. Vid testkörning i upp till 1 m/s visade det sig dock att bilen kan hålla samma fart när den svänger som när den kör rakt, så denna kostnad är satt till 0. Dock ifall framtida iterationer av bilen uppnår en högre hastighet på rakstäckor kan den komma att behöva sakta in vid svängar och det kan då vara intressant att göra denna kostnad nollskild för att kompensera för detta.

När en optimal väg är funnen så itererar sig algoritmen baklänges genom vägen och noterar vilka svängar som görs i den bestämda rutten. En binärsträng av nollor och ettor skapas där varje nolla innebär att en vänstersväng ska göras och varje etta innebär att en högersväng ska göras. Sekvensens längd noteras även för att ta hänsyn till fallen då sekvensen börjar på en nolla. Sekvensen och dess längd skickas till styrmodulens mikrokontroller som använder den för att välja vilken väglinje den ska reglera efter, se avsnitt 3.2.3. Positioneringsalgoritmen, beskriven i 4.2.6, säger till när en sväng har utförts och då det är dags för styrmodulen att byta till nästa element i sekvensen.



Figur 13: *Ruttplaneraren tar kartan som genererats av kartbyggaren och börjar med att ta bort alla noder som motsvarar rakstäckor. Övriga noder länkas om och deras kostnader uppdateras.*



Figur 14: *En generell karta omvandlas till en utökad graf. Bokstäverna efter nodernas namn; ”L”, ”R”, ”U” och ”D”; är den riktning som bilen kom in i noden ifrån. Den röda pilen vid nod C, i den vänstra bilden, visar att detta är en destinationsnod där bilen måste vara riktad uppåt för att få stanna vid destinationen. I den utökade grafen, i den högra bilden, motsvaras detta av att bilen ska ta sig till nod ”C_D” för att ta sig till destinationen. I denna figur har övriga destinationsnoder inte ritats ut för att göra figuren mindre rörig.*

3.2.7 *Hjälpfunktionaliteter*

Nedan beskrivs ytterligare hjälpfunktionaliteter som används för att möjliggöra samtliga körlägen.

Heartbeat
I många kommunikationsprotokoll, och speciellt även det som byggts för bilen, så skickas endast data när någonting behöver uppdateras. Det innebär att ifall styrmodulen inte mottar några kommandon så tolkas det som att inget behöver ändras och att den ska fortsätta att styra bilen som innan. Ifall styrmodulen däremot, av någon anledning, förlorar kommunikationen med användardatorn så skulle samma sak detekteras av styrmodulen: att inga kommandon mottas. I fallet då kommunikationen har förlorats är det dock inte sannolikt att användaren faktiskt vill att bilen ska fortsatta köra som innan och det behövs därför ett sätt att skilja på dessa två tillstånd.

För att göra detta används en så kallad *heartbeat*-signal, där namnet kommer från att signalerna skickas med jämna mellanrum, som ett hjärtslag, för att verifiera att kommunikationslänken fungerar. Heartbeatsignalen skickas tio gånger i

sekunder från användardatorn och kontrolleras fem gånger i sekunden av styrmodulen. Ifall styrmodulen inte har mottagit en ny heartbeatsignal sedan då den senast kontrollerade denna tolkas det som att kommunikationslänken har förlorats och styrmodulen ser till att bilen stannar. Vid stopp på detta vis tänds diod 5, se Tabell 5.

Lysdioder
Bilen har sex lysdioder som används för att indikera vilket tillstånd som bilen befinner sig i, se Figur 28, markering 6. Det är styrmodulens mikrokontroller som styr dessa. Dioderna tänds enligt Tabell 5 och släcks då bilen går ur tillståndet. Diod 1 är första diod sett till bilens färdriktning, vilket är dioden längst till höger i Figur 28, markering 6.

Tabell 5: *Beskrivning av vad bilens lysdioder representerar.*

Diodnummer	Aktiverad då
Diod 1	Manuellt körläge aktiverat
Diod 2	Semi-Autonomt körläge aktiverat
Diod 3	Autonomt körläge aktiverat
Diod 4	Bilen har stannat för ett hinder
Diod 5	Bilen har stannat efter att ha förlorat kontakten med användardatorn
Diod 6	Bilen har stannat vid en destination
Diod 7	Används ej
Diod 8	Används ej
Diod 9	Används ej
Diod 10	Används ej

Tidsräkning på mikrokontroller
För att räkna tid på mikrokontrollerna - i både styr- och sensormodulens mikrokontroller - används specialiserad hårdvara inbyggd i mikrokontroller. Specifikt för styrmodulen används *timer1* som är en 16-bitars räknare. Timer1 nytttjas till att generera PWM-signaler till bilens motor och styrservo. Eftersom PWM-frekvensen på bilens motor och servo är 50 Hz och mikrokontrollerns klockfrekvens är 8 MHz är timerns toppvärde satt till 19999. Samtidigt har timern en *prescaler* satt till 8 vilket innebär att det krävs 8 klockcykler för att räkna upp timern ett steg. Således krävs 8*(19999+1) klockcykler innan ett *overflow* sker, detta motsvarar en PWM-period på 20 ms (50 Hz). Notera att overflow-värdet för en 16-bitars timer normalt är $2^{16} - 1 = 65535$ medan detta istället sätter overflow-värdet till 19999.

I datastrukturen *TimerValue* lagras *timer1_num_overflows* samt *timer1_counter_value*. Antal timer1 overflows räknas från programmets början medan counter value är nuvarande värde på timer1 (0 till 19999). Antal timer1 overflows kan räknas upp till $2^{32} - 1 = 4294967295$ eftersom denna lagras i som ett 32-bitars heltal. 4294967295 overflows motsvarar ungefär 1000 dygn och med en batteridriven bil innebär det att bilens batteri kommer att ha tagit slut och orsakat en hårdvaruåterställning långt innan timerns räknare blir full. Denna struktur innebär en tidräkning med precision $8/8000000 = 1\ \mu s$ bortsett från eventuella bortfall på grund av icke-precis klockfrekvens eller andra blockerande avbrott. För mer detaljer om timer1 och dess inställningar se dokumentation för mikrokontrollern [6].

3.2.8 *Sammanfattning av variabler*

Styrmodulens mikrokontroller innehåller ett antal variabler som kan anses vara av extra intresse för den läsare som är mer nyfiken på hur den funktionalitet som beskrivits i kapitlet ovan har implementerats. Tabell 6 sammanfattar dessa. Variabler vars namn består av endast versaler är konstanter vars värden inte ändras under körning. Variabler vars enhet är '-' är enhetslösa.

Tabell 6: *Sammanfattning av de viktigaste variablerna som används på mikrokontrollern i styrmodulen. Variabler vars namn består av endast versaler är konstanter vars värden inte ändras under körning. Variabler vars enhet är '-' är enhetslösa.*

Variabelnamn	Funktion	Datatyp	Enhet
from_right	Avstånd från höger väglinje som används i regulatorn för väglinjeföljning.	float	cm
from_left	Avstånd från vänster väglinje som används i regulatorn för väglinjeföljning.	float	cm
current_speed	Bilens hastighet som används i regulatorn för farthållning.	float	cm/s
manual_throttle	Gaspådrag som används i manuellt körläge. Kommer direkt från användardatorn.	signed char	%
manual_steering	Styrutslag som används i manuellt körläge. Kommer direkt från användardatorn.	signed char	%
from_stop	Avstånd till framförvarande stopplinje. Kommer från sensormodulen i Raspberry Pi:n.	float	cm
throttle_ref_speed	Referenshastighet för regulatorn för farthållning i de autonoma lägena.	float	m/s
ref_from_right	Referensanvstånd till höger väglinje vid väglinjeföljning i de autonoma lägena.	float	cm
ref_from_left	Referensanvstånd till vänster väglinje vid väglinjeföljning i de autonoma lägena.	float	cm
throttle_PID_KP, throttle_PID_ KI och throttle_PID_ KD	PID-parametrar för regulatorn som styr gaspådraget i de autonoma lägena.	float	-
steering_PID_KP, steering_PID_KI, steering_PID_KD	PID-parametrar för regulatorn som styr styrutslaget i de autonoma lägena.	float	-
ultrasound_distance	Avstånd som ultraljudsensorn ger till framförvarande hinder. Beräknas i sensormodulens mikrokontroller.	uint16	cm
throttle	Det gaspådrag som just nu används.	float	%
steering	Det styrutslag som just nu används.	float	%
heartbeat	Används för att kontrollera om bilen är ansluten till användardatorn.	bool	-
state	Bilens tillstånd (Manuellt, Semi-autonomt, Autonomt)	uint8	-
stop_car_now	Flagga från användardatorn i de autonoma lägena som stannar bilen direkt.	uint8	-
stop_car_next_dest	Flagga från positioneringe i de autonoma lägena som stannar bilen vid nästa destination.	uint8	-
timer1_global_num_overflows	Antal *overflows* hos *timer1*. Används för att räkna tid. Se avsnitt 3.2.7.	uint32	-
MAX_DELTA_ENGINE_PWM_MS	Säkerhetsspärr för gaspådrag.	float	ms
HEARTBEAT_PERIOD_MS	Heartbeat kontrollen sker periodiskt med denna tid.	float	ms
ACTIVATE_DEST_STOPPING	Bilen stannar vid nästa detekterade destination om denna är sann.	bool	-
ACTIVATE_UL_STOPPING	Bilen stannar in för hinder om denna är sann.	bool	-
ACTIVATE_HEARTBEAT	Heartbeat-funktionaliten sätts igång om denna är sann.	bool	-
OBSTACLE_CONSTANT	Används för inbromsning vid hinder. En additiv konstant för att sätta minimalt avstånd beroende på hastighet.	float	-
OBSTACLE_FACTOR	Används för inbromsning vid hinder. En multiplikativ konstant för att sätta minimalt avstånd beroende på hastighet.	float	s
STEERING_PID_T	Periodtid för uppdatering av PID-regulatorn för väglinjeföljning	float	ms
THROTTLE_PID_T	Periodtid för uppdatering av PID-regulatorn för farthållning	float	ms
DIST_TO_START_SLOWING_DOWN	Det avstånd från en destination då bilen ska börja sakta in.	float	cm

4 SENSORMODUL OCH KAMERA

Sensormodulens uppgift är att ta in data från de sensorer som är monterade på bilen och omvandla mätdata till information som de andra modulerna kan använda sig av. De sensorer som används är en hallsensor och en ultraljudsensor. Hallsensorn används för att mäta bilens fart och färdad sträcka. Ultraljudsensorn används för att bestämma avstånd till eventuella hinder som kan förekomma framför bilen. Sensordatan uppdateras kontinuerligt och ingen gammal data sparas. Datan kan göras tillgänglig för andra moduler genom en förfrågan från kommunikationsmodulen.

Sensormodulen använder sig också av en kamera som är monterade på Raspberry Pi:en. I Raspberry Pi:en utförs även bildbehandling. Bildbehandlingen utför beräkningar på kamerans bildström, av vägen framför bilen, för att beräkna koordinater för vänster- och höger vägkant samt för att detektera destinationer och avstånd till dessa.

Sensormodulen används också för att bestämma bilens position i vägnätet. Det görs med information från bildbehandlingen tillsammans med information om bilens färdade sträcka, se Figur 17.

4.1 Hårdvara

Följande komponenter ingår i sensormodulen:

- **ATmega1284P:** ATmega1284P är en mikrokontroller som sköter beräkning av hastighet, avstånd till närmsta hinder och beräkning av hur långt bilen har färdats. Detta beskrivs i detalj i avsnitt 4.2. Mikrokontrollern är kopplad till sensorerna enligt Figur 15.
- **A1120:** Hallsensor som ger data för att kunna avgöra bilens fart.
- **SRF04:** Ultraljudsensor för att kunna se eventuella hinder och avståndet till dem. Den har en räckvidd på 3-300 cm.
- **Raspberry Pi Camera Module v2:** Kamera som tar bilder till bildbehandling. Kameran sitter i bilens front och är vinklad nedåt med en justerbar vinkel. Denna vinkel är justerad så att hjulen på bilen precis inte syns i bild, så att de inte ska störa bildbehandlingen. Genom att ha kameran vinklad nedåt så mycket det går ger kameran en bra bild på vägen framför bilen.
- **Raspberry Pi:** Mikrodator för bildbehandling samt positioneringen. Delas med kommunikationsmodulen.

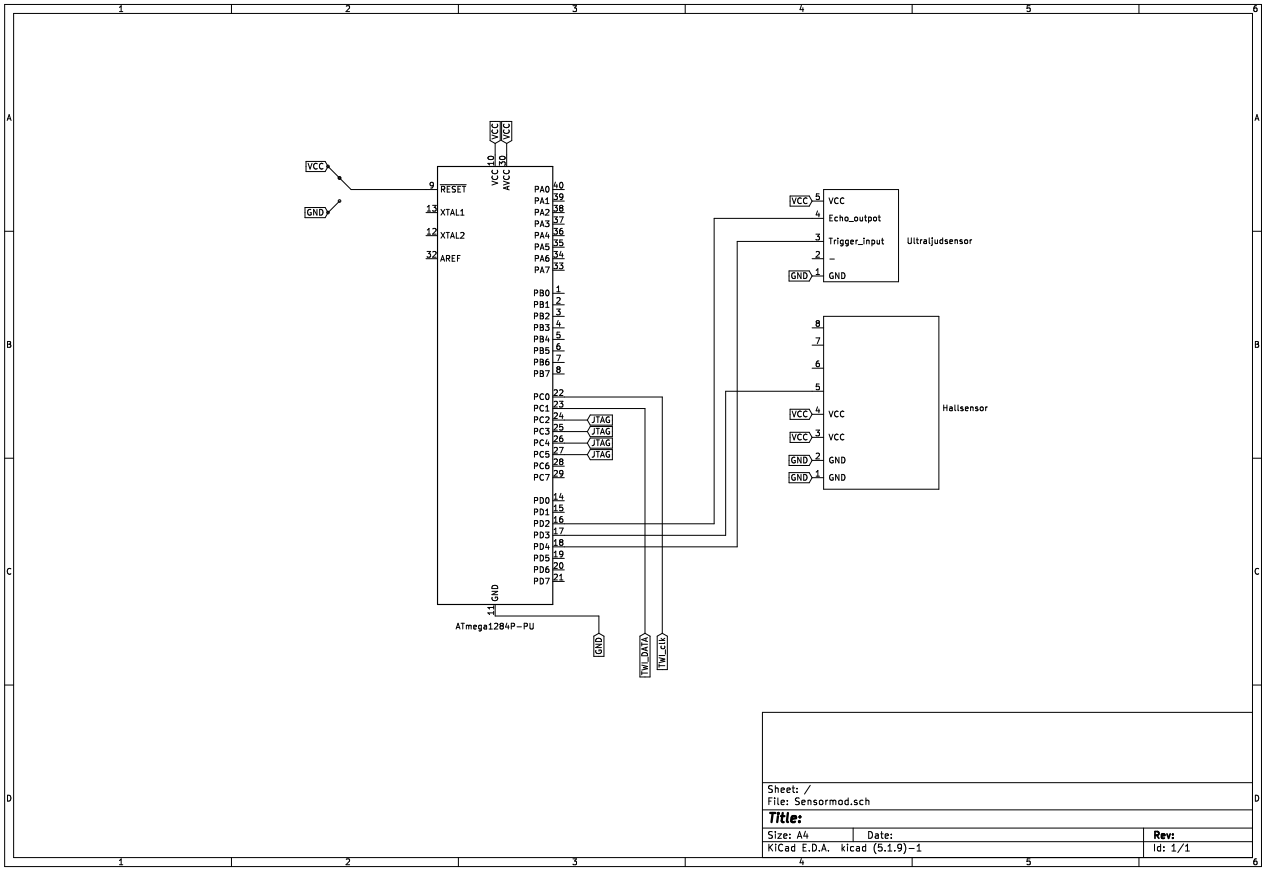
4.1.1 *Kopplingsschema för mikrokontroller*

Kopplingsschemat för sensormodulens mikrokontroller ses i Figur 15. Funktionen hos de pinnar på ATmega1284P som används ses i Tabell 7.

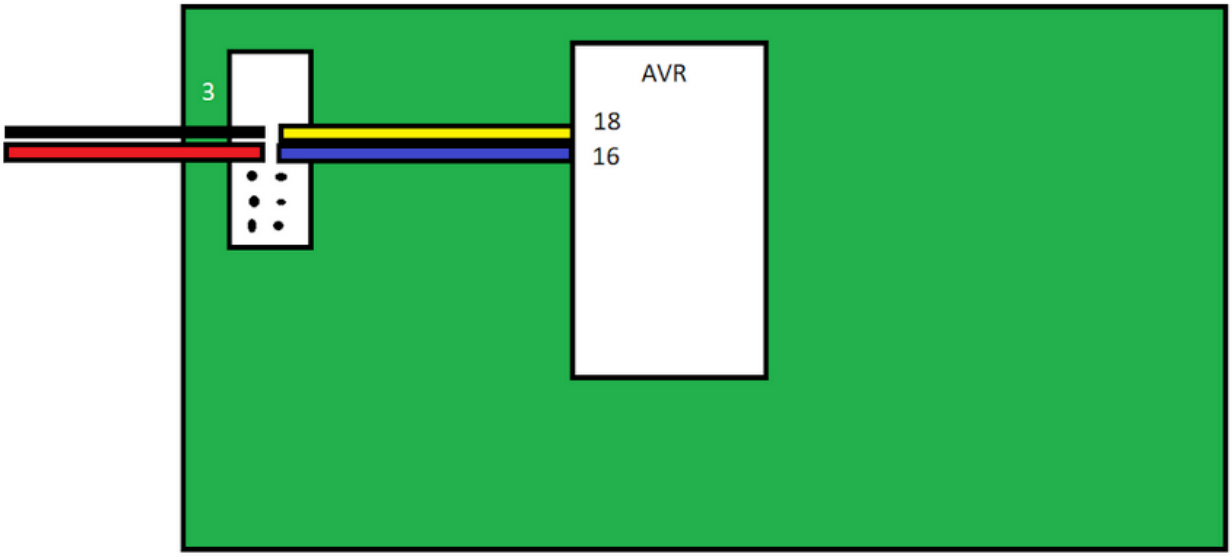
Tabell 7: *De pinnar som används i styrmodulens mikrokontroller och en beskrivning av deras funktion.*

Pin	Funktion
PC0 och PC1	Pinnarna är kopplade till klockpuls respektive data på I^2C-bussen.
PC2-PC5	Pinnarna är kopplade till JTAG. JTAG används för att programmera mikrokontrollern.
PD2	Pinnen är kopplad till ultraljudsensorns utsignal. Denna pinne är en avbrottspinne som aktiveras när utsignalen från ultraljudsensorn mottas.
PD3	Pinnen är kopplad till pinne 5 på hallsensorn. Denna pinne är en avbrottspinne som kommer att starta ett avbrott när hallsensorn skickar en signal (negativ flank).
PD4	Pinnen är kopplad till ultraljudsensorns insignal. På denna pinne skickas pulser ut som startar nya mätningar för ultraljudsensorn.
RESET	RESET pinnen är kopplad till en knapp för att det ska vara möjligt att manuellt utföra en hårdvaruåterställning av styrmodulen.

Koppling av hallsensor och ultraljudsensor till virkortet ses i Figur 16.



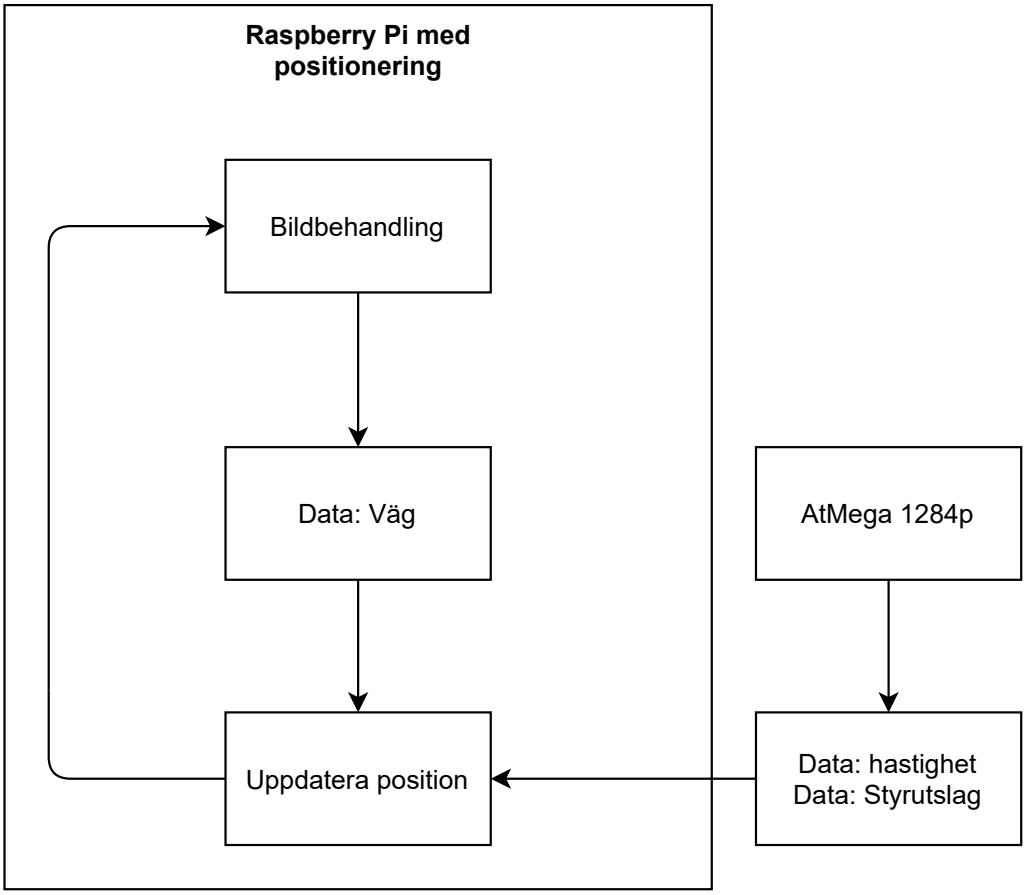
Figur 15: *Kopplingsschema för Sensormodulen.*



Figur 16: *Koppling av ultraljudsensor och hallsensor till sensormodulens mikrokontroller.*

4.2 Mjukvara på Raspberry Pi

På Raspberry Pi:en finns ett program som hanterar bildbehandling och ett program som hanterar positionering. Huvudloopen i detta program går ut på att ta emot en bild från kameran, förprocessera bilden och sedan detektera vägkanter i bilden. När detta är utfört beräknas vinkelfelet, den laterala positionen samt avstånd till eventuella destinationer. Positioneringen använder denna data tillsammans data från sensormodulens mikrokontroller för att uppdatera bilens position i vägnätet. I Figur 17 ses ett flödesschema för detta. I Figur 18 ses ett flödesschema för blocket *bildbehandling*.

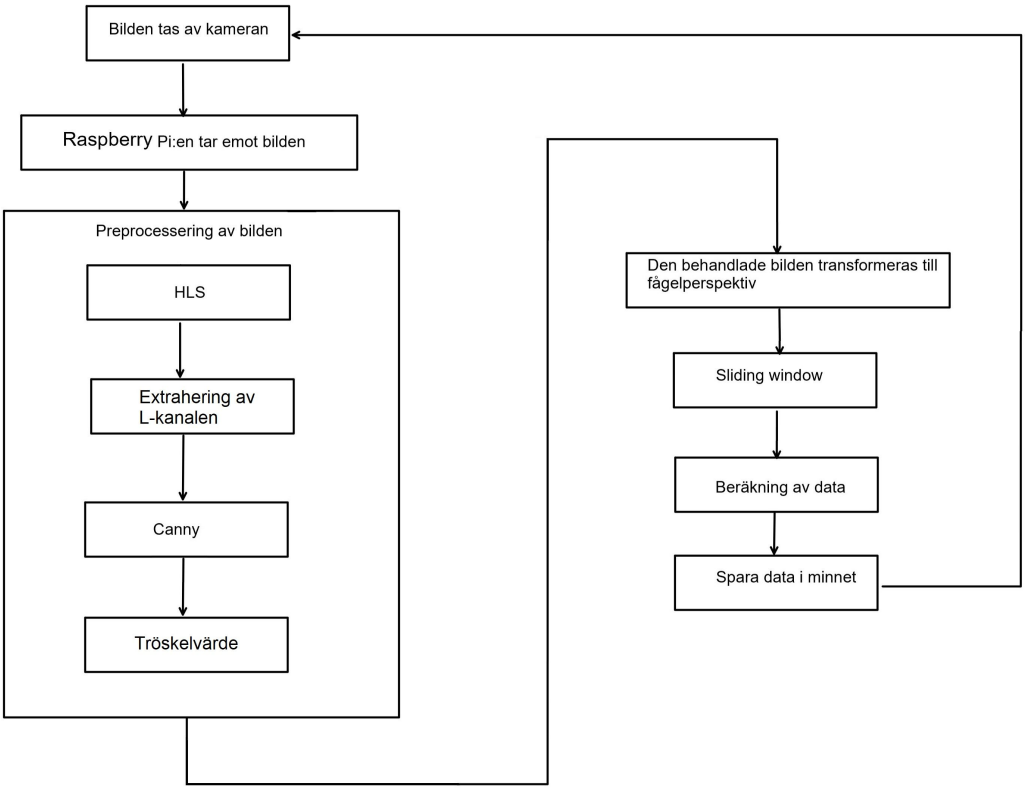


Figur 17: *Övergripande bild av sensormodulens mjukvara.*

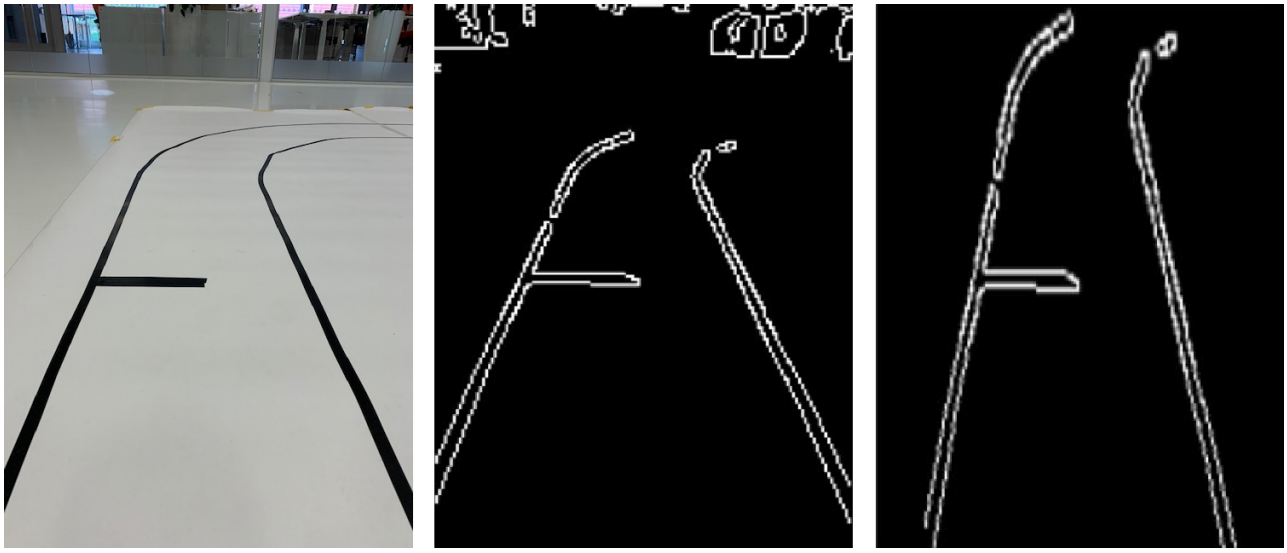
4.2.1 *Linjedetektion*

Linjedetektion är en av de uppgifter som utförs av bildbehandlingen. Bilden som kommer från kameran konverteras först till HLS-färgrummet och L-kanalen extraheras som beskriver hur mycket svart kontra vitt respektive pixel har. För att endast inkludera pixlar med mycket svart i sig används trösklar på bilden vilket resulterar i att de svarta vägkanterna syns bättre. Väglinjernas bakgrund var vit. För att filtrera ut vägkanterna ytterligare utnyttjas en *Canny*-kantdetektion. Med Canny fås en bild som endast visar kanterna i bilden vilket ger en bild där i stort sett endast väglinjer och destinationer syns. Med Canny appliceras även två trösklar som bestämmer om en samling pixlar tillhör en kant eller inte. Nästa steg som utförs är en perspektivstransfomering av bilden, där det nya perspektivet är ett fågelperspektiv. Med transformationen fås en synvinkel på banan sedd mer ovanifrån vilket underlättar för den uppkommande linjesökningen. På grund av det nya perspektivet kommer även brus i kanterna av bilden att klippas bort. Med detta är bilden redo för linjedetektion, Figur 19 visar hur den filtrerade och transformerade bilden ser ut.

Innan linjesökningen görs behöver en startpunkt för höger och vänster vägkant bestämmas. Det görs genom att dela upp bilden i två halvor: vänster respektive höger. På dessa halvor klipps en bit av nedre bilden ut där ett medelvärde i vågrät led beräknas utifrån antalet nollskilda pixlar som finns i den utklippta biten. Eftersom bilden är processerad är det bara pixlarna i vägkanten som är nollskilda. Detta medelvärde anger starten för sökningen där det högra medelvärdet ges för



Figur 18: *Övergripande flödesschema för bildbehandlingsprocessen.*

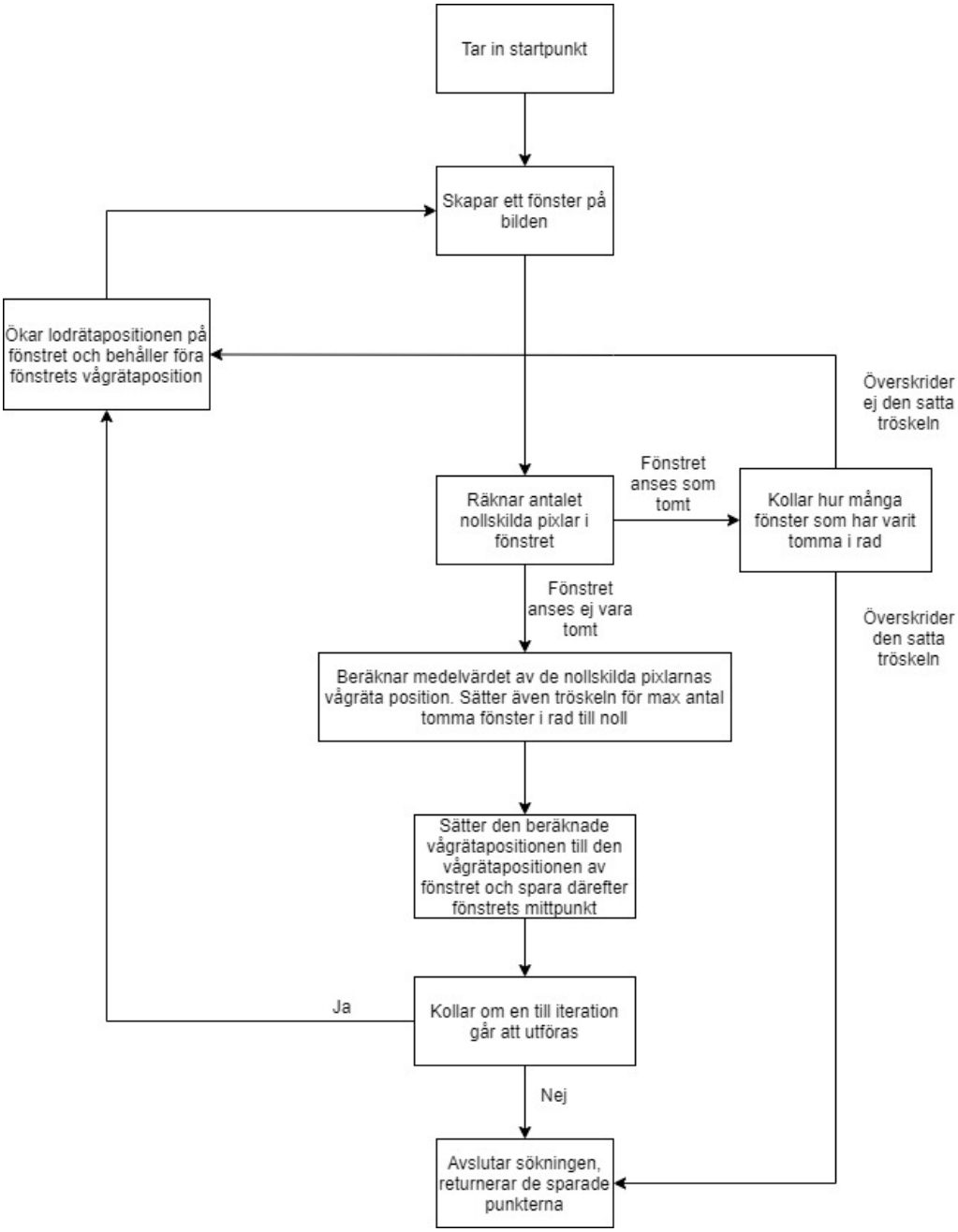


Figur 19: *Bilden till vänster visar bilden från kamerna. Bilden i mitten visar hur bilden ser ut efter filtrering. Bilden till höger visar hur bilden ser ut efter filtrerring och transformation.*

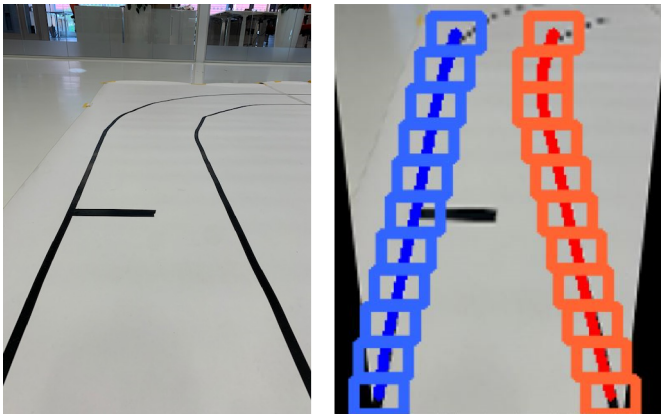
högra vägkanten och vänstra medelvärdet för den vänstra vägkanten. Sökningen utförs med en *sliding-window*-metod som kommer att ge en samling punkter som representerar respektive vägkant.

Ett flödesschema för sliding-window-metoden ses i Figur 20. Den börjar i de bestämda startpunkterna där ett fönster sätts på bilden. Antalet nollskilda pixlar beräknas i fönstret. Om antalet pixlar inte överskrider den satta tröskeln kommer den vågräta positionen att vara den samma och rutan anses då som tom, vilket innebär att fönstret ignoreras. I det andra fallet, då antalet pixlar överskrider tröskeln kommer fönstret få en ny vågrät position. Nästa iteration upprepar processen med den vågräta position som beräknades från förra iterationen samt ökar den lodräta positionen med sökfönstrets höjd. Ifall endast tomma fönster detekteras från start kommer sökningen att avslutas efter att en satt tröskel överskridits (tre i

nuvarande implementation). Är fönstren inte tomma sätts tröskeln till 0. Detta innebär att om ett fönster blir tomt efteråt avslutas sökningen. Med detta minskar risken att sökningen får problem på grund av på brus i omgivningen. Om inga av de ovan nämnda fallen stoppar sökningen kommer den att avslutas när sökfönstret når toppen av bilden. När sökningen avslutas returneras de sparade punkterna i form av en vektor som då representerar den detekterad väglinjen. Sliding-window-sökningen utförs på både den högra och vänstra startpunkten och anger vilken väglinje det är utifrån dem. I Figur 21 ses resultatet från linjedetektionen.



Figur 20: *Flödesschema för sliding-window-sökningen.*



Figur 21: *Bilden till vänster visar bilden från kameran. Bilden till höger visar de detekterade väglinjerna där de blå är den vänstra- och de röda är den högra vägkanten. Notera att den högra bilden även har transformerats till att ha ett fågelperspektiv. Rutorna representerar de sökfönster som användes under sliding-window-sökningen.*

Från de detekterade väglinjerna extraheras avståndet till högra respektive vänstra vägkanten. Avståndet i vågrätt led mäts från mitten av bilden till linjen i de detekterade rutorna. De första fem rutorna används i nuvarande implementation. Om linjen inte har det satta antalet punkter kommer de punkter som finns att tas. Ett medelvärde beräknas av dessa avstånd vilket ger antalet pixlar till respektive vägkant. Avståndet skalas med en faktor som anger avståndet mellan två pixlar i centimeter. Eftersom bilden är transformerad med ett fågelperspektiv så stämmer detta avståndet väl överens med verkligheten. Om en vägkant ej har detekterats sätts dess avstånd till 100 cm, vilket var ett designval som underlättade för styrmodulen.

Funktionalitet finns implementerad i koden för att applicera ett intresseområde (eng. *Region of interest*) som tar bort det som ej är önskevärt i bilden. Ett intresseområde är effektivt för att undvika störande element i omgivningen. Under testning visade det sig att detta processeringssteg tog bort mycket av vägen vilket berodde på att svängarna var relativt skarpa tillsammans med vidvinkeln på kameran. För att undvika förlust av nödvändig information var intresseområdet tvunget att vara relativt stort, vilket gjorde denna funktionalitet onödig. Dessutom orsakade fortfarande ett stort intresseområde att delar av vägen klipptes bort. Ett intresseområde orsakade alltså mer problem än det löste, vilket ledde till att den exkluderades från slutprodukten. Den finns dock kvar eftersom den kan användas för vidareutveckling av produkten.

Förklaring av process:

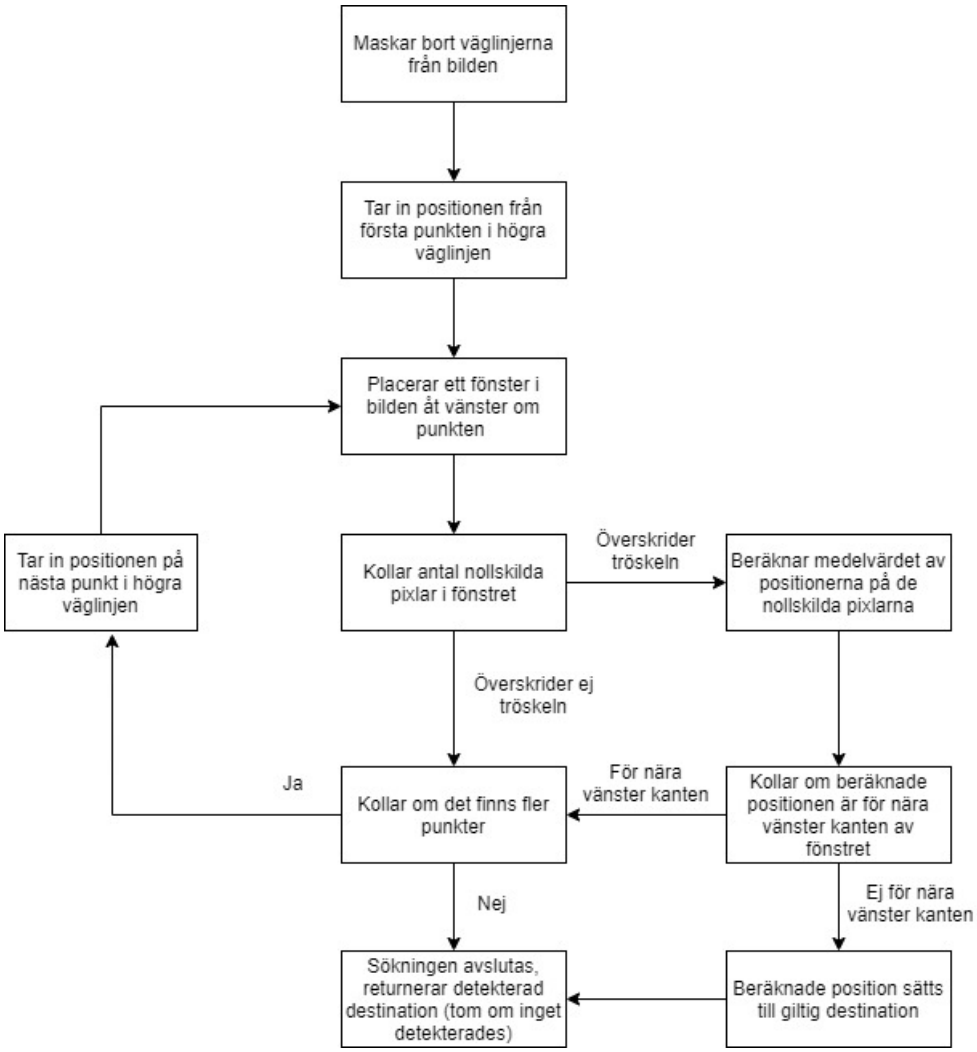
- **HLS**: (Nyans, ljusstyrka och mättnad) Konverterar bilden till HLS-färgrummet (bilden är från början i RGB-färgrummet). I detta färgrum är det lättare att få specifika färger att isoleras genom applicering av trösklar och filter.
- **Extrahering av L-kanalen**: L-kanalen extraheras då den innehåller information om hur mycket svart kontra vitt respektive pixel har.
- **Canny**: Hittar kanter i bilden, där en kant definieras som stora förändringar i pixlarnas intensitet.
- **Tröskelvärde** (*Thresholding* på engelska): Ett filter som undersöker om värdet på varje pixel är mindre än ett tröskelvärde (*threshold* på engelska) sätts pixelvärdet då till noll, annars sätts pixelvärdet till det maximala. Bilden ska vara i en gråskalad variant. Denna filtrering ger en mer stabil vägdetektion för de efterföljande faserna.
- **Fågelperspektiv**: En transformering av bilden till en vy som representerar vägen ovanifrån.
- **Sliding-window**: Hittar vägkanterna i bilden genom att sätta ut en startpunkt på ett fönster och beroende på antalet pixlar med en viss intensitet bestäms nästa position för fönstret. Fönstret flyttas uppåt tills hela bilden sökts av eller tills dess att linjen inte längre detekteras.

I huvudsak är ovan beskriven funktionalitet implementerat med biblioteket *OpenCV* i C++. Bildbehandlingen sker kontinuerligt utan hänsyn till avbrott eller övriga processer som exekveras på mikrodatorn. Slutresultatet från denna del är

två vektorer som innehåller koordinaterna för de pixlar som motsvarar vänster respektive höger vägkant. Koordinaterna är relativt bild-koordinatsystemet.

4.2.2 *Detektion av destinationer*

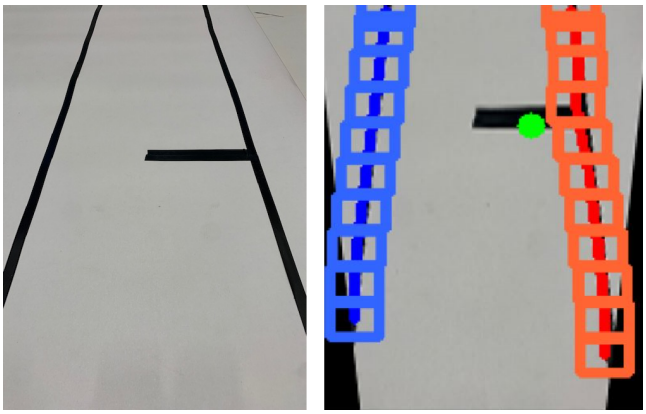
Detektion av destinationer är en till uppgift som utförs av bildbehandlingen. Ett flödesschema för denna process ses i Figur 22. Detektion av destinationer görs genom att utföra en sliding-window-sökning över den högra vägkanten. Anledningen till varför det endast görs på högra vägkanten är för att bilen endast får stanna vid destinationer som är sammankopplade med höger vägkant.



Figur 22: *Flödesschema för destinationsdetektionen.*

Varje punkt på den högra väglinjen undersöks med ett fönster. Fönstret sätts brevid det fönster som används under silding-window-sökningen för att få fram just den punkten. Genom att sätta fönstret där minskas risken att väglinjen stör sökningen. I fönstret räknas antal nollskilda punkter och jämförs med ett tröskelvärde. Om antalet nollskilda pixlar underskrider tröskeln fortsätter sökningen på samma sätt över resterande punkter. Om tröskeln istället överskrids kommer ett medelvärde från de nollskilda pixlarnas position att beräknas. Det är endast destinationer som är sammankopplade med högra vägkanten som är av intresse, vilket innebär att en destination som är förbunden med vänstra vägkanten ska ignoreras. För att undvika detta kontrolleras det beräknade medelvärdet relativt fönstret. Om medelvärdet ligger åt den vänstra kanten av fönstret, med en viss marginal, anses detta som en destination från vänster, vilket leder till att punkten bortses ifrån och sökningen fortgår. Om allting tyder på att det är en giltig destination kommer det beräknade medelvärdet att sättas

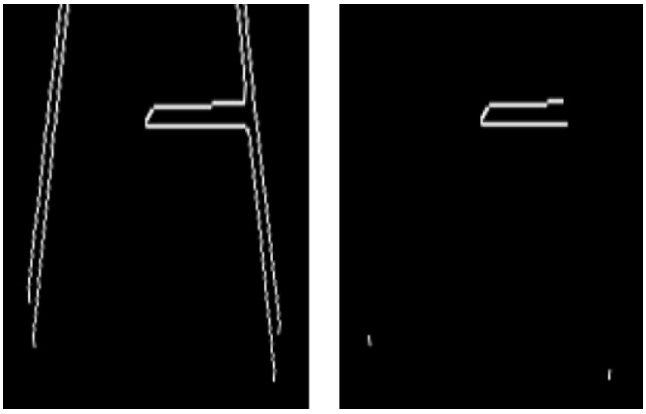
som den detekterade destinationen och sökningen avslutas. Det innebär att den giltiga destination som är närmast är den som kommer att detekteras. I Figur 23 visas resultatet från destinationdetektionen.



Figur 23: *Exempel på destinationsdetektion. Bilden till vänster är originalbilden. Bilden till höger visar var en destination har detekterats, markerat med en grön prick. Destinationen är giltig, eftersom den ligger på höger sidan av vägen, vilket innebär att den ska detekteras. Notera att bilden till höger är transformerad med fågelperspektiv.*

När en destination är detekterad kommer avståndet till destinationen att beräknas och skickas vidare till resterande moduler. Om ingen destination är hittad skickas 10000 cm som avstånd till destination. Att ha ett stort värde var ett designval då det är så orimligt stort att styrmodulen (den som sköter inbromsning vid destination) inte kommer att reagera på det. Från testning har det visats att 70 cm är det maximala avståndet som en destination kan detekteras med säkerhet på, då bilen kör på en raksträcka. I svängar minskar avståndet men inte tillräckligt mycket för att styrmodulen inte ska kunna hinna med att stanna bilen. Det innebär att denna detektionsalgoritm fungerar med hög träffsäkerhet.

För att ytterligare minska risken för falskt detekterade destinationer maskeras de detekterade väglinjerna innan destinationsökningen börjar, se Figur 24. Maskeringen utförs på både vänstra och högra vägkanten. Endast de linjer som detekterats vid väglinjedetektionen är det som maskeras. Det som återstår är därför de linjer som markerar destinationer.



Figur 24: *Figuren visar hur väglinjerna maskeras inför destinationsdetektionen. Bilden till vänster visar en väg med en destination, på höger sida av vägen, som har blivit filtrerad och transformerad. Bilden till höger visar väglinjerna efter att de har maskeras för att minska falska utslag.*

En känd bugg med metoden är att falska destinationer ibland kan detekteras i svängar. Detta händer dock bara i någon enstaka bild, medan om en faktiskt destination är i bild detekteras den i varje bild den är synlig. Detta kan hanteras av bilens regulatorer.

4.2.3 *Detektion av T-korsningar*

Algoritmen för att hitta T-korsningar går ut på att undersöka ökningen av vägbredden. Detta görs genom att jämföra den aktuella vägbredden med den som detekterats i föregående iteration av bildbehandlingen. Om den aktuella bredden är större än den föregående med en viss marginal indikerar det en ökning. Det undersöks sedan ifall det sker en kontinuerlig ökning av vägbredden under flera iterationer av bildbehandlingen. Om vägbredden blir större än en tröskel indikerar detta att bilen befinner sig i en korsning. För att kontrollera om bilen har passerat korsningen undersöks bredden på vägen igen. Om den sjunkit ner till den normala vägbredden för en raksträcka, med en viss marginal, indikerar detta att korsningen har passerats.

Detektion av T-korsningar används inte för tillfället men den kan utnyttjas i en framtida utveckling av positioneringen. Anledningen till att den inte användes beror på att den var i ett fungerande tillstånd relativt sent i projektet.

4.2.4 *Kalibrering av bildbehandling*

Bildbehandlingen har många parametrar som behöver ändras ifall kameran byter position eller vinkel. I kalibreringen väljs hur mycket bilden ska samplas ner, hur mycket bilden ska processeras, hur stark kantdetektionen ska vara, parameterar för fågelperspektiv samt sliding-window sökningen och till sist bredden på vägens utifrån kamerans synvinkel (från transformerat perspektiv). Effekten som parametrarna har på bilden visas i realtid under kalibreringen.

4.2.5 *Beräkning av vinkel relativt väg*

Beräkning av bilens körvinkel relativt vägen sker genom att först räkna en vektor som motsvarar mittlinjen i vägen. Detta görs genom att ta mittpunkten i horisontell led för varje par punkter på respektive vägkant. Sedan beräknas vinkeln mellan varje segment (två intilliggande punkter) av mittlinjen relativt en vertikallinje med hjälp av arctan-funktionen. Medelvärdet av alla beräknade vinklar motsvarar en vinkel som beskriver hur mycket vägen kröker sig åt höger eller åt vänster.

4.2.6 *Positionering*

Positioneringen bygger på odometri och använder sig av den utökade graf som skapats av ruttplaneraren, beskriven i avsnitt 3.2.6. Positioneringsalgoritmen använder sig av kännedom om bilens startposition, bilens tillryggalagda sträcka och antagandet att bilen följer den väg som ruttplaneraren bestämt. Det är då enkelt att avgöra vilken nod som bilen befinner sig i och därmed var i vägnätet bilen är.

Positioneringen realiseras med hjälp av klassen *Positioning_simple* som hittas i filen *Dijkstras.py*. Klassen *Positioning_simple* instansieras efter att *i2c_master.py* har mottagit grafen från användardatorn och den utökade grafen skapats av funktionen *find_optimal_path*. Under körning så mottar postioneringsalgoritmen färdad sträcka från sensormodulen och information från bildbehandlingen om avstånd till nästa destination, ifall en sådan detekterats. Från användaren erhålls startposition och startriktning. Med hjälp av kostnaderna i den utökade grafen och information från hallsensorerna om hur långt bilen har färdats så kan positioneringen beräkna vilken nod som bilen befinner sig i. Positionering uppdateras i varje iteration av *i2c_master*, vilken arbetar i samma hastighet som bildbehandlingen, ca 30 FPS. När bilen tar sig till en ny nod skickas den informationen till styrmodulens mikrokontroller. Där används bilens position för att avgöra vilken instruktion från ruttplaneraren som ska användas för tillfället, se avsnitt 3.2.6 för mer detaljer.

Positioneringen tar även beslut om ifall bilen ska stanna för en destination. Bilen antas vara framme vid en destination när kommande nod är en destination och kameran signalerar ett avstånd till destinationen som är mindre än tröskelvärdet *stop_dist*, denna är satt till 10 cm. Bilen kommer att stanna vid destinationen under en specificerad tid som anges av *destination_wait_time*, denna är satt till 1 sekund. Ifall ett uppdrag körs som innefattar flera destinationer efter varandra så kommer bilen att stanna i *destination_wait_time* sekunder vid varje destination, förutom vid den sista där bilen kommer att stanna permanent och invänta nya instruktioner.

Med tiden ackumuleras fel vid mätningen av färdad sträcka som uppstår på grund av noggrannheten hos sensorn, avståndsfel i grafen och för att bilen även färdas en liten sträcka i lateralt led. För att undvika att dessa fel blir betydelsefulla efter längre tid så kalibreras den uppmätta färdsträckan när bilen befinner sig vid en destination. Då kalibreras bilens position om enligt bildbehandlingens information. Detta sker med olika stora intervall beroende på placeringen av destinationerna som bilen passerar, men avstånden mellan destinationerna är i storleksordningen 0-5 m. Typiskt ses avvikelser på några enstaka procent när dessa kalibreringar görs.

4.2.7 *Sammanfattning av variabler*

Bildbehandlings C++ på Raspberry Pi:en

Tabell 8: *Tabellen sammanfattar viktiga variabler som används i bildbehandlingens algoritm på Raspberry Pi:en.*

Variabelnamn	Klass	Funktion	Typ
transformationMat	BirdView	En transformationsmatris som används för att omvandla bilden till fågelperspektivtransform	cv::Mat
left_line	BirdView	En vector som innehåller de punkter som representerar vänster vägkant	std::vector<cv::Point2i>
right_line	BirdView	En vector som innehåller de punkter som representerar höger vägkant	std::vector<cv::Point2i>
right_line	BirdView	En vector som innehåller de punkter som representerar stopplinje som motsvarar destination	std::vector<cv::Point2i>
image	Camera	En matris som innehåller de pixlar som representerar den tagna bilden	cv::Mat
cam	Camera	Kamerans utström - Den intagna inspelningen	cv::VideoCapture
left_line	RoadIdentifier	Innehåller information om vänster vägkant	std::vector<cv::Point2i>
right_line	RoadIdentifier	Innehåller information om höger vägkant	std::vector<cv::Point2i>
stop_line	RoadIdentifier	Innehåller information om stopplinjedetektion	std::vector<cv::Point2i>
previous_distance	RoadIdentifier	Den passerade bredden på vägen	int
current_distance	RoadIdentifier	Den aktuella bredden på vägen	int
TcrossCounter	RoadIdentifier	Antalet frames som kontrollera ökningen på vägbredden och som indikerar en korsningdetektion	int
is_T	RoadIdentifier	Visar om en korsning är upptäckt eller passerad	bool
check_previous_state	RoadIdentifier	Kontrollerar en start på en ökning i vägbredden	bool

4.3 Styrmodulens mikrokontroller

Förutom bildbehandlingen hanterar även sensormodulen bilens ultraljudsensor och bilens hallsensor. Dessa sensorer hanteras på mikrokontrollern AtMega1284P [6]. I detta avsnitt beskrivs den del av sensormodulen som är implementerad på sensormodulens mikrokontroller. Denna del står för en betydligt mindre del av sensormodulen jämfört med hur stor del av styrmodulens som är implementerad på styrmodulens mikrokontroller. Detta eftersom merparten av sensormodulen behandlar datorseende på Raspberry Pi:n.

4.3.1 *Överblick av programmet*

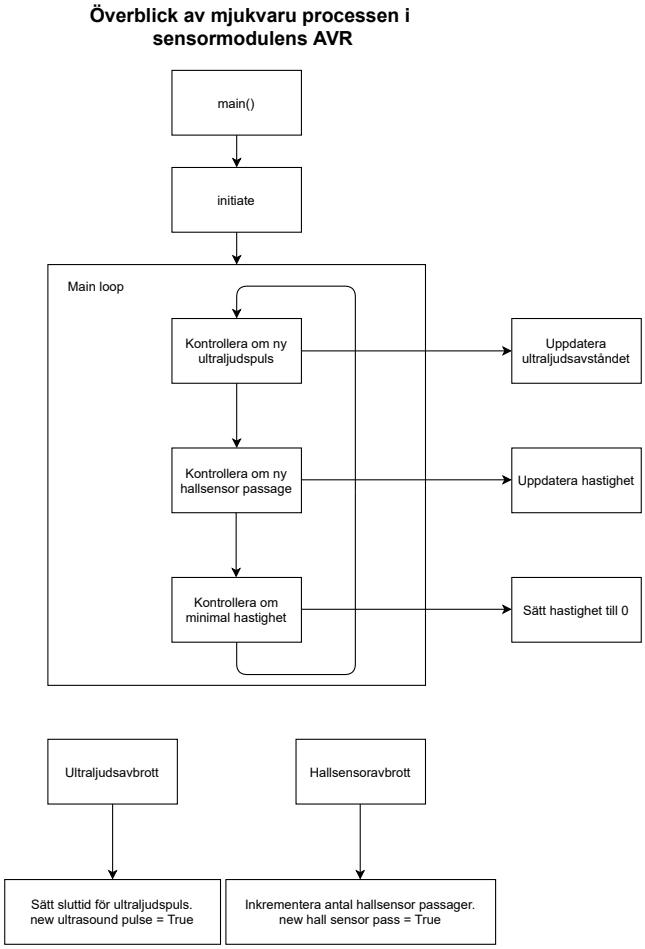
1. **Initiering:** Innan huvudloopen körs initieras alla variabler och lågnivåfunktionalitet som exempelvis timers på mikrokontrollern.

2. **Huvud-loop:** Detta steg körs upprepat under programmets livstid och hanterar programmets flöde. Hela sensormodulens huvudprogram är avbrottsstyrt. Det enda som sker i huvud-loopen är att kontrollera ifall någon av avbrotten har aktiverats sen senaste iterationen. I så fall görs nödvändiga beräkningar.

3. **Se ifall en ny ultraljudspuls har mottagits:** Om ultraljudsensor har tagit emot ekot från en tidigare sänd signal så ska denna hanteras för beräkning av avstånd.

4. **Se ifall en ny passage av hallsensorn har skett:** Om bilens hallsensor har detekterat passage av ett hallelement ska detta hanteras för beräkning av färdad sträcka och hastighet.

5. **Avgör ifall bilen står stilla:** Om det var länge sedan hallsensorn detekterade passage av ett hallelement sätts hastigheten till 0. Detta för att signalera att bilen har stannat.



Figur 25: *En överblick av mjukvaran på sensormodulens mikrokontroller. Hela sensormodulens huvudprogram är avbrottsstyrt. Det enda som sker i huvud-loopen är att kontrollera ifall någon av avbrotten har aktiverats sedan den senaste iterationen. I så fall görs nödvändiga beräkningar.*

4.3.2 *Beräkning av sträcka till framförvarande hinder med ultraljud*

På bilen sitter en SRF04 ultraljudsensor [8], vilken ses i Figur 32. Sensorn har en ultraljudsändare och en ultraljudsmottagare. Sändaren aktiveras med en puls på en pinne 3 och mottaget eko genererar en puls på pinne 4, se Figur 15. Sensormodulen skickar periodiskt aktiveringssignaler till ultraljudsensorn, med en period på 40 ms. Detta görs med hjälp av specialiserad hårdvara i mikrokontrollern som genererar en PWM-signal på pinne PD4, se Figur 15. Mottagen puls är kopplad till pinne PD2 som orsakar ett avbrott, varefter en flagga för att beräkna fart och färdad sträcka sätts.

Beräkning av sträcka, i enhet cm, görs enligt Ekvation 2.

$$s = \frac{t_{\text{mottagen}} - t_{\text{transmitterad}}}{2} v_l, \tag{2}$$

där t_{mottagen} är tidpunkten då ultraljudsensorn detekterade den återvända ultraljudspulsen och $t_{\text{transmitterad}}$ är tidpunkten då ultraljudsensorn transmitterade ultraljudspulsen, enhet sekund. Vidare är v_l ljudets hastighet i centimeter per sekund. Tiden delas på 2 eftersom det är tiden för ljudet att färdas till föremålet och tillbaka. Se avsnitt 3.2.7, delavsnitt Tidsräkning på mikrokontroller, för en beskrivning av hur tidtagning görs i mikrokontrollern.

Sträckan som ges av ultraljudsensorn måste korrigeras då den kan ha ett statiskt fel. Felet korrigerades vid konstruktionen av bilen genom att med testning hitta det statiska felet. I detta fallet visade sig det statiska felet ligga på 10 cm.

Ultraljudsensorn räckvidd är 300 cm, ifall en puls mottages som indikerar en längre sträcka än det tolkas den som ogiltig och avståndet sätts till 300 cm som standardvärde. Ett detekterat avstånd på 300 cm innebär således att avståndet till hindret är större än, eller lika med, 300 cm.

4.3.3 *Beräkning av hastighet och färdad sträcka med hallsensorer*

I ett av bilens bakre hjul sitter tio hallelement, se [5]. När hjulet snurrar och ett hallelement passerar hallsensorn, som är monterad i hjulhuset, genereras en puls. Denna puls är kopplad till pinne PD3 på mikrokontrollern, se Figur 15, som aktiverar ett avbrott vid övergång från låg till hög puls. Avbrottsrutinen räknar upp det totala antalet hallelement som har passerat hallsensorn.

Beräkning av hastighet
För att beräkna bilens hastighet med hallsensorn så räknas tiden τ mellan två passager av hallelement. Tiden räknas med samma typ av tidräknare beskrivna i avsnitt 3.2.7, delavsnitt Tidsräkning på mikrokontroller. Tio hallelement ger ω radianer mellan varje hallelement och med radien r kan den färdade sträckan mellan varje hallelementpassage beräknas. Med parametrar enligt ovan beräknas hastigheten v som

$$v = \frac{\omega r}{\tau}. \tag{3}$$

När bilen stannar slutar avbrotten för hallelementpassager att genereras eftersom hjulen inte längre snurrar. Detta behöver hanteras för att på ett korrekt sätt ange bilens hastighet när den är på väg att, eller helt har, stannat. I varje iteration av huvud-loopen beräknas därför den hastighet som bilen skulle ha haft ifall en hallelementpassage hade inträffat då. Ifall denna fiktiva hastighet sjunker under en tröskelhastighet, som är satt till 0,1 m/s, sätts hastigheten istället till 0 m/s.

Beräkning av färdad sträcka
För att beräkna bilens färdade sträcka summeras antal passager av hallelement och multipliceras med samma ωr som vid beräkning av hastighet. Tester visar att denna metod, trots sin enkelhet, ger en avvikelse på mindre är 10 cm efter 5 meters körning.

Parametrar
Bilens specifika parametrar för Ekvation 3 är följande: $r = 4.25$ cm, $\omega = 0.6283$ rad.

4.3.4 *Sammanfattning av variabler*

Sensormodulens mikrokontroller innehåller ett antal variabler som kan anses vara av extra intresse för den läsare som är mer nyfiken på hur den funktionalitet som beskrivits i kapitlet ovan har implementerats. Tabell 9 sammanfattar dessa. Variabler vars enhet är '-' är enhetslösa.

Tabell 9: *Sammanfattning av de viktigaste variablerna som används på mikrokontrollern i sensormodulen. Variabler vars enhet är '-' är enhetslösa.*

Variabelnamn	Funktion	Typ	Enhet
timer1_global_num_overflows	Antal overflows hos timer 1. Används för att räkna tid i tidräkningsstructen. Se förklaring under avsnitt **??**.	uint32	-
ultrasound_start and ultrasound_end	Den tid då ultraljudsmätning startade respektive slutade.	uint16	s
flag_new_ultrasound	Flaggar för att avstånd ska beräknas med hjälp av ny mätning från ultraljudsensorn.	bool	-
ultrasound_distance_float	Det senaste avstånd som har beräknats med ultraljudsensorn.	float	cm
ultrasound_time_diff	Tiden mellan transmitterade och mottagen ultraljudspuls.	float	s
num_hall_sensor_passes	Antalet passager av hallelement förbi hallsensor sedan programmets början.	uint16	konstant
time_since_last_magnet	Tid sedan det senaste hallelementet passerade hallsensorn.	float	s
velocity_16bit	Bilens hastighet	uint16	cm/s
distance_travelled_hall_float_cm	Bilens färdade sträcka sedan programmets start.	float	cm

5 ANVÄNDARDATOR

Användardatorn interagerar med taxibilen via ett grafiskt användargränsnitt som byggts med hjälp av Python och modulen Tkinter. En bild av gränssnittet kan ses i Figur 26 och dess olika funktioner beskrivs nedan. En stor del av gränssnittets uppgift är att skicka instruktioner till bilen som sedan utför dem, och är därför tämligen ointressanta att diskutera här eftersom kommunikationstekniken redan har beskrivits i kapitel 2. För läsbarhetens skull listas översiktligt vilka delar som ingår i gränssnittet nedan. För visuell referens hänvisas läsaren till Figur 26

1. Informationsruta som visar värdet på de mest intressanta variabler som skickas från bilen.

2. Vid försök till anslutning läses IP och port ut ur respektive textruta och en anslutning upprättas till servern med denna adress. Om anslutningsförsöket misslyckas visas en textruta.

3. Graf för realtidsvisualisering av data inhämtad från bilen givet en anslutning till bilen. Först filtreras datafilen på antalet rader som motsvarar valt tidsfönster, vilket sätts genom 'time_scale', därefter filtreras kolumn som motsvarar vald variabel och återstående data ritas ut i grafen. Maximalt värde i y-led sätts i fältet 'ylim'.

4. Realtidsvisualisering av karta samt bilens position på kartan. Kartfil väljs och laddas in via knappen 'Öppna fil'. På kartan är destinationer markerade med en röd prick och bilen med en grön prick. För att se vilka namn samtliga hållplatser har så bockas rutan 'Visa namn' i. Ett autonomt uppdrag skapas genom att lägga till en sekvens av destinationer - detta görs med hjälp av dropdown-menyn och knappen 'Lägg till'. Uppdrag kan tas bort genom att klicka på knappen 'Töm lista'. När användaren klickar 'Skicka' sänds kartinformationen över till bilen i JSON-format, tillsammans med det autonoma uppdraget, som sedan tolkas i ruttplaneringsprogrammet, se kapitel 3.2.6. När karta och autonomt uppdrag skickats så kan bilens planerad rutt ritas ut på kartan genom att klicka på 'Visa rutt'. Under körning så uppdateras bilens position och planerad rutt på kartan.

5. Här väljs hur bilen ska bete sig då den körs i semi-autonomt läge. Knappen 'Tillsvidare' betyder att bilen ska ignorera stopplinjer och knappen 'Till dest' betyder att den ska stanna vid nästa stopplinje. Knapp 'Vänster' och 'Höger' väljer vilket håll som bilen ska svänga då den kommer till en korsning. Bilen kör då 'Kör' är valt, och står stilla då 'Stanna' är valt. Notera att användardatorn endast sänder instruktioner om hur bilen ska regleras, själva reglering sker i styrmodulen, se kapitel 3.

6. Ändrar läge på bilen, mellan de tre olika valbara lägena. I manuellt läge kontrolleras bilen via en ansluten handkontroll, i semi-autonomt via knapparna beskrivna ovan och slutligen i autonomt läge enligt instruktionerna som byggs genom beskrivningen ovan. Notera att en ansluten handkontroll krävs vid uppstart av användargränssnittet för körning i manuellt läge.

7. Reglerparametrar för gaspådrag och styrutslag samt referensvärden för de avstånd bilen ska hålla till vänster- respektive höger vägkant. Dessa parametrar och referensvärden används i både semi-autonomt och autonomt läge.

8. Sparande av mätserier påbörjas genom att trycka på knappen 'Starta mätning'. Vid en påbörjad mätning skapas en ny fil med inmatat namn och inkommande data från bilen loggas i datafilen. Om en fil med samma namn redan existerar när en mätning påbörjas så skrivs filen över. Sparade datafiler kan sedan visualiseras och datan analyseras i analysfönstret som öppnas med hjälp av den sista knappen, 'Starta analys'.

Figur 26: *Användardatorns grafiska gränssnitt.*

De delar av gränssnittet som utför mer än simpla försändelser är beskrivna i detalj nedan.

- **Sparande av data:** Alla dataformat och tillhörigheter specificeras i filen *communication_standards*. Därifrån nyttjas listan HEADERS för att skapa kolumnerna i en csv-fil. Inkommande data från servern filtreras och sparas i motsvarande kolumn i datafilen. Notera att den första kolumnen alltid är en tidstämpel som markerar när datat anlände. Inkommande data sparas alltid i filen *.temp.csv*, som töms på tidigare data vid uppstart av gränssnittet och därefter var 60:e sekund. Vid val av mätseriestart töms först vald fil på data varefter inkommande data läses till vald fil. Denna fil töms endast vid start av ny mätserie, ej vid omstart av gränssnittet.

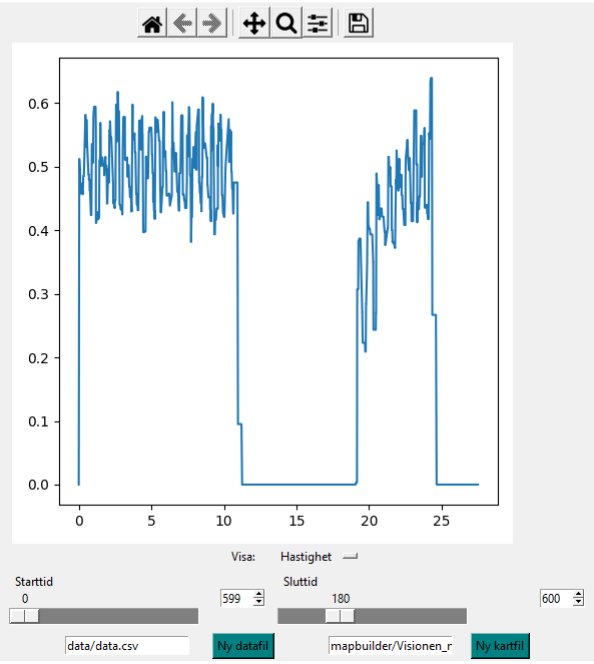
- **Realtidsuppdatering av applikationer i användargränssnittet:** Alla delar av gränssnittet som uppdateras kontinuerligt har en funktion *refresh*. Denna funktion kallas från huvudprogrammet *GUI_main.py* kontinuerligt med jämna tidsintervall som sätts i metoden *_set_refresh_rates*.

- **Visualisering av data:** Användargränsnittet erbjuder två verktyg för datavisualisering; realtidsvisualisering och ett analysfönster. Realtidsvisualisering utgår alltid från filen *.temp.csv.*, ur vilken rader som motsvarar valt tidsfönster samt kolumn som motsvarar variabeln i fråga filtreras ut. I grafen visas sedan samtlig kvarvarande information, medan informationsrutan och positioneringsfönstret endast visar det senast inkomna värdet.

- **Hantering av handkontroll:** För att kunna nyttja en handkontroll för styrning av bilen användes paketet *inputs* tillsammans med en kontrollklass som hittas i filen *xboxcontroll.py*. Vid uppstart letar programmet efter en handkontroll och startar en *daemon*-tråd vars ansvar är att hålla informationen om kontrollen aktuell. Vid varje uppdatering av applikationen sänds nuvarande hastighet och styrutslag till bilen. Notera att eftersom kontrollen är analog är värdet aldrig exakt noll, utan flyter mellan 0 och 0.07 när spaken lämnas fri. För att detta ska motsvara noll gas är toleransnivån för att ge gas 0.1; alla värden mindre än så sätts i programmet till noll före försändelse. Kontrollen kan ge ett värde mellan 0 och 20, där 20 representerar full gas. Maxvärdet är teoretiskt så högt som 100, men av säkertsskäl rekommenderas att inte modifiera skalfaktorn högre än 60, för att undvika kollisioner.

- **Analysfönstret:** Filtrerar vald csv-fil för rader med tidsstämpel mellan de två valda värdena på tidsaxlarna, samt för variablen motsvarande kolumnen. Visar samtliga kvarvarande värden i grafens fönster. Notera att utfiltrerade rader

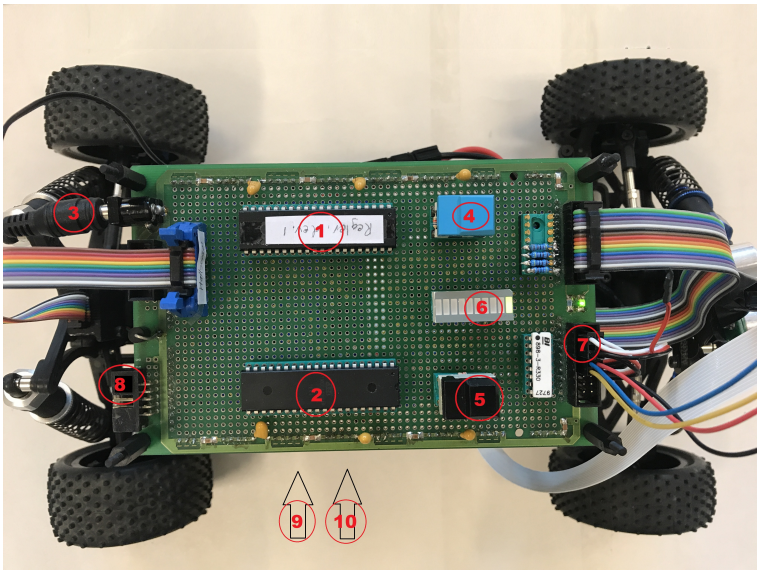
endast uppdateras vid förändring av någon av parametrarna i fönstret; det är alltså säkert att använda analysfönstret samtidigt som en mätserie samlas in. En bild av analysfönstret visas i Figur 27.



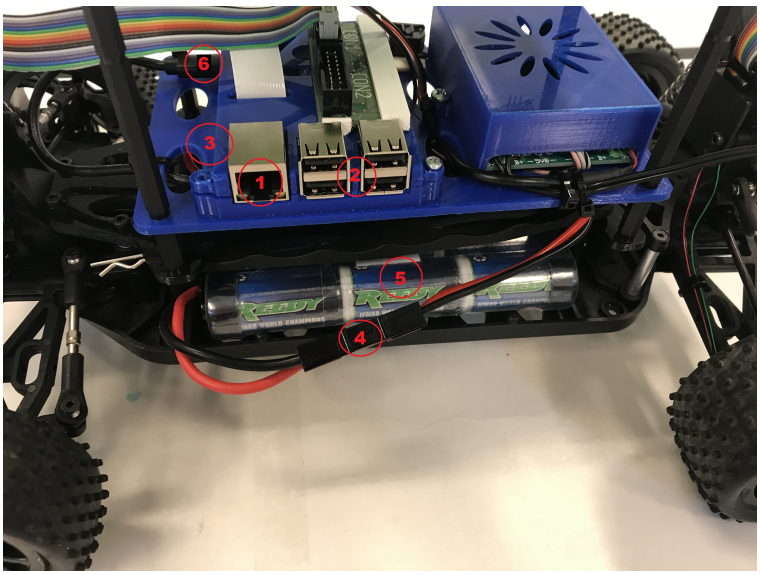
Figur 27: *Användargränssnittets analysfönster. Här kan sparad data från körningar visas upp i efterhand för analys.*

För datafiltrering användes genomgående Pythonpaketet *Pandas*. Samtlig information angående datakoder, läsbara variabelnamn, vad som ritas ut i grafer och vad som sparas på fil finns specificerat i filen *communication_standards.py*.

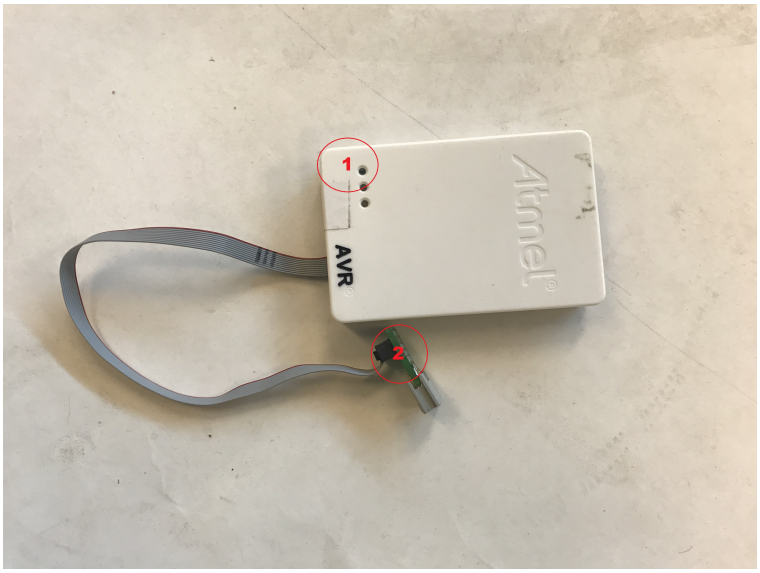
6 ALLMÄNNA FIGURER



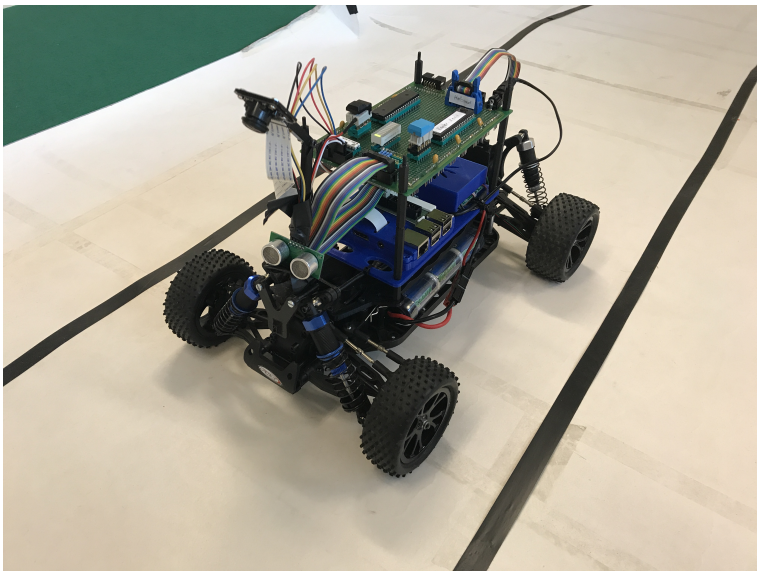
Figur 28: *1: Mikrokontrollern som hanterar styrning av bil. 2: Mikrokontrollern som hanterar bilens hastighetsmätning och ultraljud. 3: Virkortets strömsladd. 4, 5: Resetknappar för 1 respektive 2. 6: Lysdioder, i figuren lyser diod 1. 7: Anslutningar för hallsensor, ultraljudsensor, styrservo och motor. 8: JTAG-kontakt för programmering av mikrokontrollerna. 9: Syns inte på bilden men här sitter en silver-aktig spak som agerar strömbrytare för hela bilen. 10: Syns inte heller på bilden men här sitter en svart spak som stänger av och på bilens styr- och motorfunktionalitet.*



Figur 29: *1, 2: USB-uttag för inkoppling till Raspberry Pi:n. 3: HDMI-anslutning för inkoppling till Raspberry Pi:n. 4: Kontakt för batteri. 5: Batteri. 6: Anslutning för strömkabel till Raspberry Pi:n.*



Figur 31: *JTAG-enhet för programmering av bilens mikrokontroller.*



Figur 32: *Helhetsbild av bilen.*

REFERENSER

[1] P. S. Foundation, "python multiprocessing library," acessed 2021-05-13. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html

[2] K.-P. Lindegaard, "smbus2," acessed 2021-05-13. [Online]. Available: https://pypi.org/project/smbus2/

[3] Community-Driven, "Pigpio," accessed 2021-05-13. [Online]. Available: http://abyz.me.uk/rpi/pigpio/index.html

[4] Google, "Protocol buffers." [Online]. Available: https://developers.google.com/protocol-buffers

[5] Dacar taxi och tävlingsbil. [Online]. Available: http://www.da.isy.liu.se/vanheden/pdf/DACar.pdf

[6] Microchip. (2020) megaavr data sheet. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega164A_PA-324A_PA-644A_PA-1284_P_Data-Sheet-40002070B.pdf

[7] Dijkstra's algorithm. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[8] robot electronics. (2003) Srf04 - ultra-sonic ranger technical specification. [Online]. Available: https://www.robot-electronics.co.uk/htm/srf04tech.htm

C ANVÄNDARHANDLEDNING

Användarhandledning

Martin Dahl
Edvin Johansson
Mikael Josefsson
Charbel Youssef
Henrik Ahlinder
Sebastian Andersson

7 juni 2021

Version 1.0

Status

Granskad	Alla	2021-05-19
Godkänd		

Projektidentitet

Grupp E-post: TSEA56_2021VT_O2_G10@groups.liu.se

Hemsida:

Beställare: Anders Nilsson, Linköpings universitet
Tfn: 013-28 26 35
E-post: anders.p.nilsson@liu.se

Kund:
Tfn:
E-post:

Handledare: Peter Johansson
Tfn: +4613281345
E-post: peter.a.johansson@liu.se

Kursansvarig: Mattias Krysander
Tfn: 013-28 21 98
E-post: mattias.krysander@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Henrik Ahlinder	Projektledare	henah490@student.liu.se
Mikael Josefsson	Kommunikationsmodulansvarig, GUI-ansvarig	mikjo695@student.liu.se
Sebastian Andersson	Testansvarig	seban218@student.liu.se
Martin Dahl	Dokumentansvarig, mjukvaruansvarig	marda545@student.liu.se
Charbel Youssef	Sensormodulansvarig	chayo642@student.liu.se
Edvin Johansson	Styrmodulansvarig	edvjo404@student.liu.se

INNEHÅLL

1 Inledning 1
2 Genomgång av systemets olika delar 2
2.1 Bilen . . . 2
2.2 Användardatorn och användning av dess GUI . . . 3
2.3 Vägnät . . . 7
2.4 Bilens säkerhetsspärrar . . . 8
3 Använda manuellt läge 8
4 Använda semi-autonomt läge 8
5 Använda autonomt läge 9
6 Kalibrering av bildbehandlingen 10
7 Övrigt 13
7.1 Felsökning . . . 13
7.2 Viktiga noter . . . 13
7.3 Egen programmering av systemet . . . 14
8 Förklarande figurer 15

DOKUMENTHISTORIK

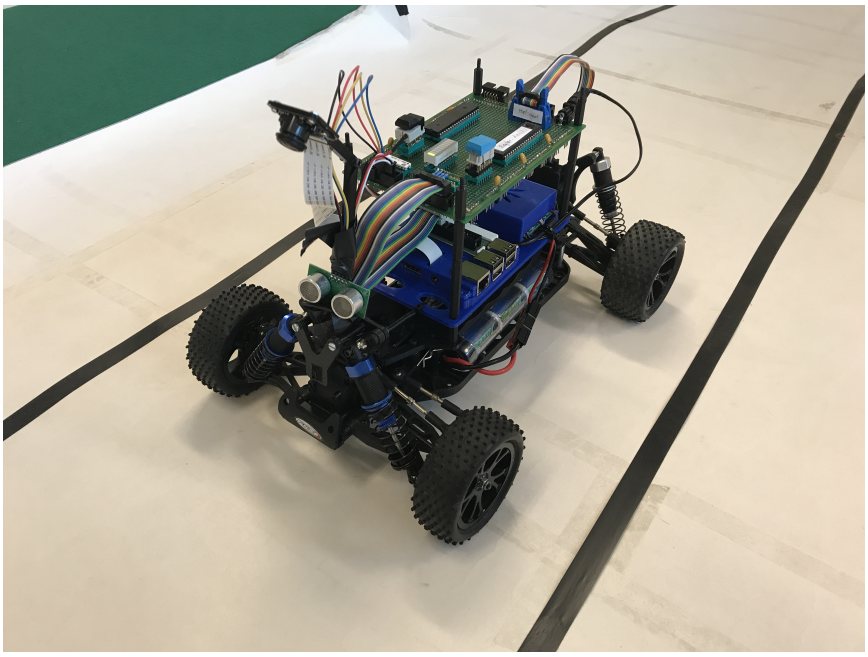
Version	Datum	Utförda förändringar	Utförda av	Granskad
1.0	2021-05-19	Första version	Alla	Alla

1 INLEDNING

Detta dokument beskriver hur taxibilen i dess olika lägen ska användas. Den går också in på metoder för felsökning och vad man bör ha i åtanke vid användning. Taxibilen visas upp i sin helhet i Figur 1.

Användarmanualen bör läsas som en genomgång sida till sida då avsnitten beror på varandra.

Använd bilen med försiktighet. Systemet har implementerat säkerhetsspärrar for att undvika kraschar men dessa kan misslyckas med sin kraschdetektion. Var därför försiktig med vilken omgivning bilen framförs i samt vilken hastighet den kör i. Läs mer nedan om vilka typer av säkerhetsspärrar som är implementerade.



Figur 1: Bilen i sin helhet på den vägtyp som den är konstruerad för att köra på.

2 GENOMGÅNG AV SYSTEMETS OLIKA DELAR

I detta kaptiel behandlas de delar av systemet som användaren måste interagera med för att kunna framföra bilen.

2.1 Bilen

De delar av bilen som är viktigast att ha i åtanke är :

- Bilens strömbrytare: Se Figur 13.
- Styrservo och motorers skjutomkopplare. Då bilen startas och strömbrytaren är hög, kan det vara bra att först sätta skjutomkopplaren till låg och sedan till hög för att kalibrera styrdatorns motorhantering. Vid körning måste denna omkopplare vara hög. Se Figur 13.
- Bilens 3 datorer : Två mikroprocessorer av typen AVR finns monterade på det gröna virkortet, se Figur 13 markering 1 och 2. Slutligen finns en Raspberry Pi inkapslad i ett blått skal under virkortet, se Figur 12.
- Batteri och batteriladdaren, Se Figur 14. Notera att batteriladdarens statuslampa (markering 1 i Figur 14) lyser grönt vid färdigladdat batteri, annars rött. Det är även viktigt att notera hur batteriet ska placeras och anslutas till bilen, se Figur 12 markering 4 och 5.
- Bilens debug-anslutningar, se Figur 12.
- Bilens testställning, se Figur 15.
- Bilens resetknappar, se Figur 13.
- Bilens lysdioder som sitter på virkortet, se Figur 13 samt avsnitt 2.1.1.
- Strömanslutningen till bilens virkort, se Figur 13.

2.1.1 *Bilens lysdioder*

Nedan beskrivs hur lysdioderna är definierade. Se även Figur 13 markering 6. I figuren är det diod 1 som lyser vilket innebär att bilen stod i manuellt läge när bilden togs.

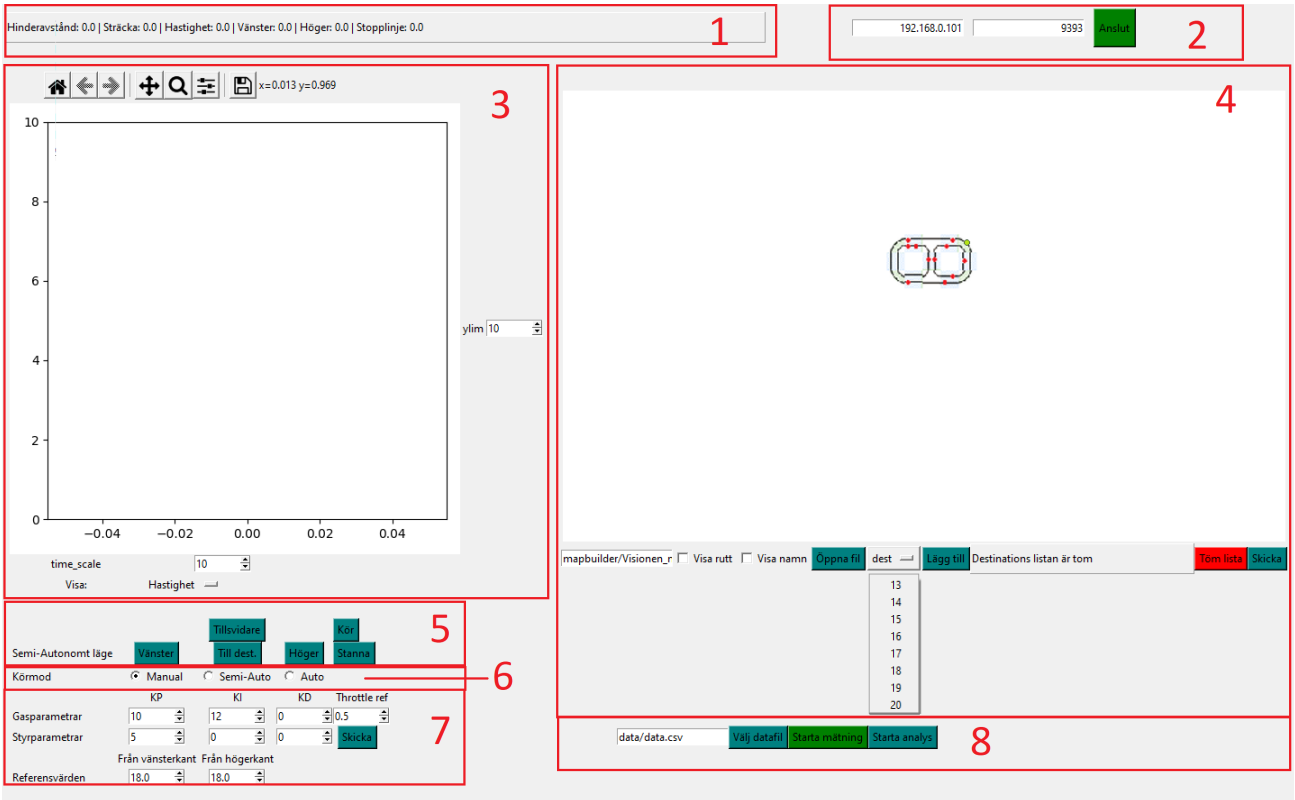
1. Körläge = Manuellt
2. Körläge = Semi-Autonomt
3. Körläge = Autonomt
4. Bilen har stannat för ett hinder
5. Bilen har stannat p.g.a heartbeat. Se avsnitt 2.4
6. Bilen har stannat vid en destination

Observera att lysdioder 7 till 9 ej används.

2.2 Användardatorn och användning av dess GUI

I detta kapitel beskrivs de olika funktionerna som finns i det grafiska användargränsnittet.

2.2.1 *Huvudsakliga grafiska gränssnittet*



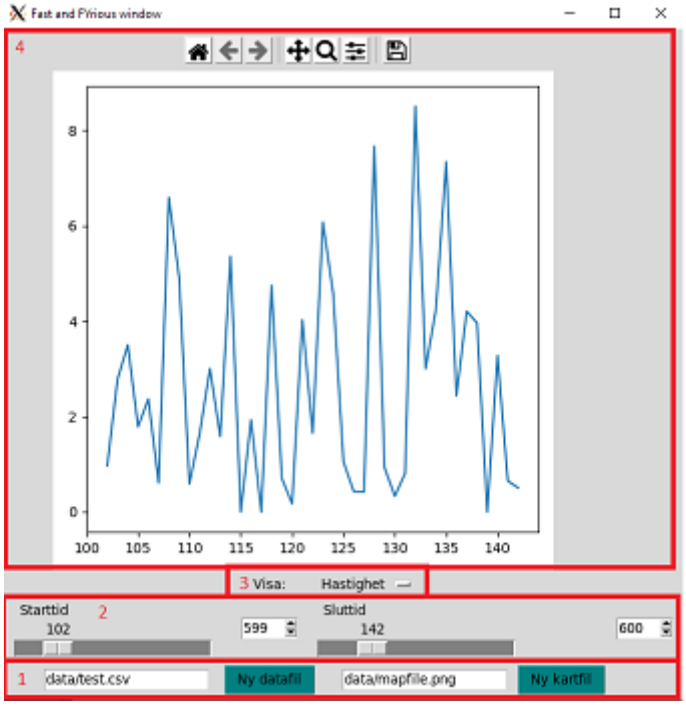
Figur 2: Grafiska gränssnittet. Observera att GUI:t kan verka komplext men så är inte fallet.

Observera att du måste ansluta en handkontroll till användardatorn innan du startar GUI:t om du vill använda det manuella körläget.

1. Ruta med värden av de mest intressanta variablerna som skickas ifrån bilen.
2. Ruta för att ansluta till bilen. Till vänster en ruta för serverns ip-adress, till höger en ruta för dess port (denna bör vara konstant 9393). Använd den gröna *Anslut* knappen för att ansluta till bilen med given ip-adress och port. Knappen blir sedan en röd *Koppla ifrån* knapp för att koppla ifrån bilen.
3. Realtids visualisering av värden på variabler som skickas ifrån bilen.
4. Ruta för ladda in bilens vägnät och ange köruppdrag. När köruppdraget skickats uppdateras bilens position och planerad rutt kontinuerligt på kartan.
5. Kommandon för det semi-autonoma läget.

6. Val av körmod
7. Inställning av reglerparameterar för gaspådrag och styrutslag samt referensvärden för de avstånd bilen ska hålla till vänster respektive höger vägkant. Dessa parametrar skickas till bilen när man trycker på knappen *Skicka*.
8. Val av fil för dataanalys, knapp för att starta datainsamling samt uppstart av analysfönstret, se Figur 3.

2.2.2 *Analysfönstret*

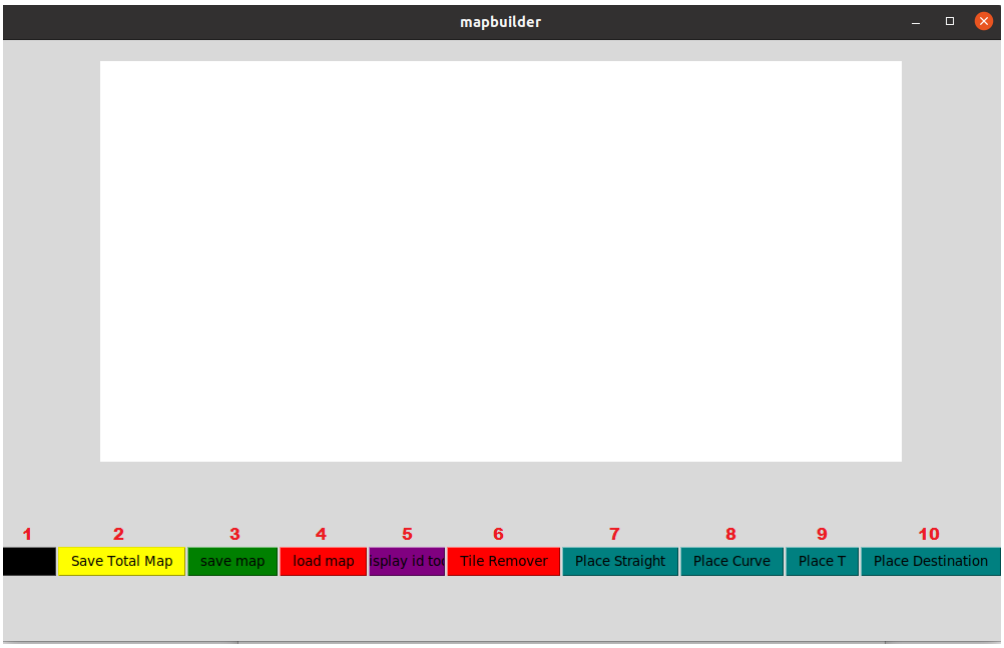


Figur 3: Analysfönster

Analysfönstret används för att analysera mätserier som har sparats under körningar. För att utföra en mätning används knappen ”starta mätning” i GUIts huvudfönster, se rubrik 2.2.

Analysfönstret används på följande vis:

1. Välj datafil samt kartfil där data som önskas analyseras placeras.
2. Välj vilket tidsspann som ska visas. Känsligheten av skalorna kan ökas genom att sätta ett egetdefinierat maxvärde i respektive talruta till höger om skalan.
3. Välj vilken variabel som ska visas i fönstret.
4. Information visas nu i fönstret. Via knapparna längst upp i fönstret går det att zooma in, zooma ut, flytta sig längs axlarna, justera grafens position i fönstret samt spara visad graf till datorn.



Figur 4: Kartbyggare

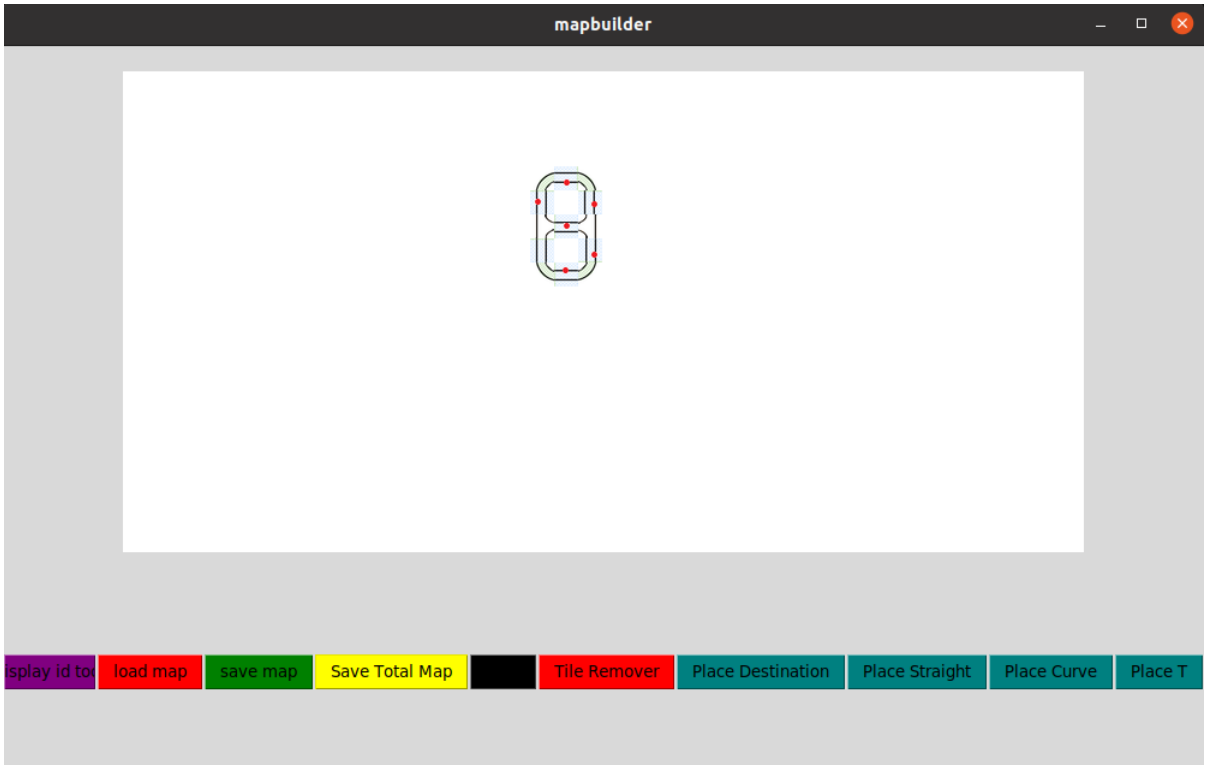
2.2.3 *Kartbyggaren*

En inledande not om kartbyggaren är att det alltid går att klicka *escape* på tangentbordet för att avbryta nuvarande selektion. Detta inkluderar dock inte knapp 2, 3 och 5 av självklara skäl. Varning, knapp 2 rensar helt nuvarande karta oavsett om du väljer att ladda en karta eller inte, det är en medveten funktionalitet som får levas med. I allmänhet bör kartbyggaren användas varsamt.

1. *exit*: Stänger av programmet utan att spara.
2. *Save Total Map* : Sparar banans grafiska specifikation samt banans representiva graf. Dessutom sparas en .png bild som föreställer banan. Notera att det är viktigt att kartan är fullständig innan du försöker skapa grafen, det fungerar till exempel inte att ha en väg som inte leder någon vart.
3. *save map*: Spara kartan till en json-fil som hamnar i kartbyggarens nuvarande sökväg. Denna rekommenderas inte att användas.
4. *load map*: Ladda in en sparad karta, det ska vara en särskild json fil som fås genom att spara en karta med alternativ 2. Varning! Om *load map* trycks rensas hela den nuvarande kartan från programmet. En bra vana är därför att spara karta frekvent vilket är en säker operation.
5. *display id tool*: Gör att en destinations namn dyker upp när muspekaren hålls nära destinationen. Avbryt med *escape*.
6. *Tile Remover*: Verktyg för att radera de rutor som klicks på. Avbryt med *escape*, radera med vänster musknapp.

7. *Place Straight*: Verktyg för att placera ett rakt vägsegment. Avbryt med *escape*, placera med vänster musknapp, rotera med höger musknapp.
8. *Place Curve*: Verktyg för att placera kurva. Avbryt med *escape*, placera med vänster musknapp, rotera med höger musknapp.
9. *Place T*: Verktyg för att placera T-korsning. Avbryt med *escape*, placera med vänster musknapp, rotera med höger musknapp.
10. *Place Destination*: Verktyg för att placera destinationer på vägsegment. Avbryt med *escape*, placera med vänster musknapp. Observera att vägsegment måste vara utplacerade innan detta verktyg används, annars kan programmet krascha. Destinationerna kommer följa vägkanterna vid utplacering för att underlätta placeringen av dem.

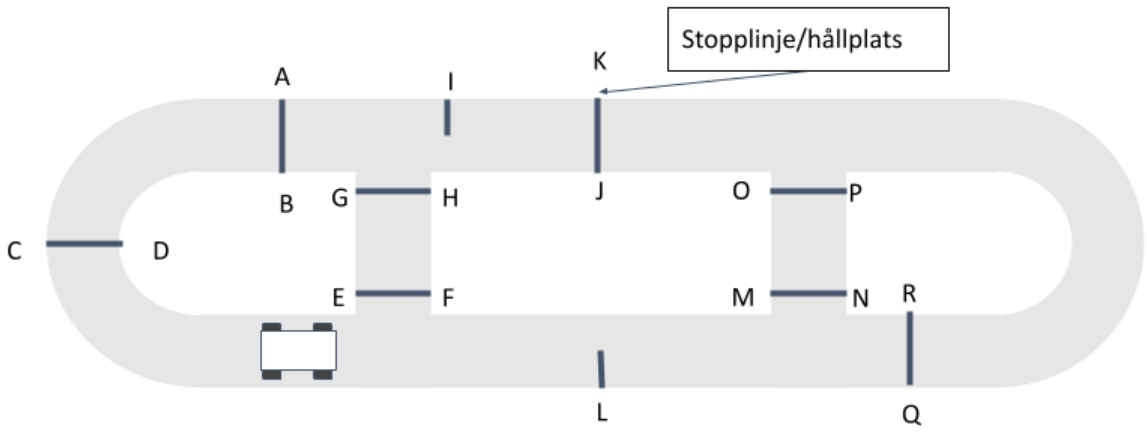
I Figur 5 syns hur en karta kan byggas. En betydligt mer avancerade kartan kan byggas om behovet finns. Det finns ingen storleksbegränsning förutom att ytan på skärmen är begränsad. De röda prickarna är destinationer.



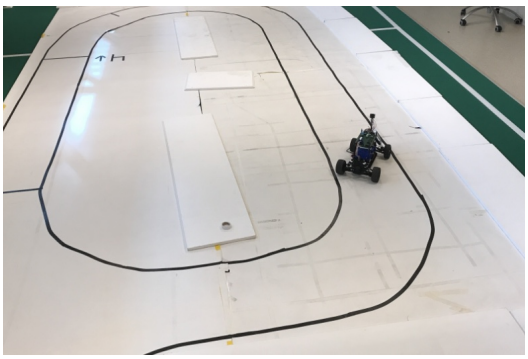
Figur 5: En konstruerad karta i kartbyggaren. Vägsegmentens olika färger betyder inget annat än att vi inte har en grafisk designer, de gör dock att man lättare kan känna igen segmenten.

2.3 Vägnät

Den typ av väg som bilens är gjord för att köra på är strikt definierad: Vägbredden är 40 cm och kurvorna har en svängradie på 90 cm. Vägens kanter är svarta linjer samt grunden är vit. I Figur 6 syns en skiss av vägnätet som lade grunden för bilens design under dess utveckling. I Figur 7 visas istället ett fotografi av en verklig version av ett (annat) vägnät.



Figur 6: Skiss av det grundläggande vägnätet. I figuren syns både heldragna och halvdragna stopplinjer, det vill säga dubbelsidiga och enkelsidiga destinationer. I verkligheten är samtliga svängar avrundade jämfört med de hårda svängarna som syns i figuren.



Figur 7: Fotografi av ett faktiskt vägnät. Just vägnätet i bilden användes under konstruktionens testfas, det rekommenderas därför att inte ha en lika dålig bana vid användning. En konsekvens av detta är däremot att bilen är robust i sin körning.

2.4 Bilens säkerhetsspärrar

Bilen har två olika system för att undvika krasch. Det första systemet är ultraljudssensorn som mäter avstånd till framförvarande hinder eller vägg. Bilen beräknar hur och när den måste stanna för att undvika krasch med hindret eller väggen.

Det andra systemet används för att stanna bilen ifall kommunikationen mellan användardatorn och Raspberryn Pi:n avbryts. Detta är för att undvika situationer vid manuell körning där ett gaspådrag är givet och kommunikationen bryts så att inga kommandon för att stanna eller svänga kan skickas. Detta system utnyttjar en så kallad *heartbeat* för att hela tiden kontrollera att koppling finns mellan Raspberryn Pi:n och användardatorn. Denna kontroll sker ungefär 4 gånger per sekund. När ett heartbeat-stopp aktiveras kommer den översta lysdioden (se Figur 13 markering 1) slås på vilket kan användas för att felsöka bilen.

3 ANVÄNDA MANUELLT LÄGE

1. Innan du gör något med bilen, kontrollera att alla sladdar är inkopplade. Kontrollera även att inga sladdar är i närheten av något av bilens 4 hjul.
2. Starta användardatorn.
3. Koppla in handkontrollen.
4. Starta användarprogrammet på användardatorn.
5. Välj manuellt läge i ruta 1, se Figur 2.
6. Anslut till bilen med den gröna anslutningsknappen i övre högra hörnet. Observera att bilens ip-adress måste först skrivas in till vänster om anslutningsknappen.
7. Kör bilen med handkontrollen, se Figur 16.

4 ANVÄNDA SEMI-AUTONOMT LÄGE

1. Innan du gör något med bilen, kontrollera att alla sladdar är inkopplade. Kontrollera även att inga sladdar är i närheten av något av bilens 4 hjul.
2. Starta användardatorn.
3. Starta användarprogrammet på användardatorn
4. Anslut till bilen med den gröna anslutningsknappen i övre högra hörnet. Observera att bilens ip-adress måste först skrivas in till vänster om anslutningsknappen.
5. Ange styrutslag- och gaspådragsparamterar i ruta 2, se Figur 2 och klicka på *skicka*.
6. Sätt körlaget till semi-autonomt läge i ruta 2, se Figur 2.
7. Välj nu den den väglinje du önskar att bilen ska köra efter.

8. Klicka på kör för att påbörja körningen.
9. Bilen bör nu köra efter den önskade väglinjen. Vilken linje som följs går att ändra under körning. Notera att valet av detta kommer avgöra vilket håll som bilen kommer ta i korsningar.
10. Det går att få bilen att stanna vid destinationer genom att ange att den ska stanna vid nästa destination. Då kommer den stanna vid nästa giltiga destination som bilen ser.

5 ANVÄNDA AUTONOMT LÄGE

1. Innan du gör något med bilen, kontrollera att alla sladdar är inkopplade. Kontrollera även att inga sladdar är i närheten av något av bilens 4 hjul.
2. Starta användardatorn.
3. Starta användarprogrammet på användardatorn.
4. Kontrollera att läget står i manuellt läge, detta syns under hastighetsplotten i vänstra halvan av programmet.
5. Anslut till bilen med den gröna anslutningsknappen i övre högra hörnet. Observera att bilens ip-adress måste först skrivas in till vänster om anslutningsknappen.
6. Ange styrutslag- och gaspådragsparamterar längst ner, samt sätt körlaget till autonomt läge.
7. Klicka på skicka. När du har klickat på skicka har bilen tagit emot de ändrade körläge och styrparametrarna.
8. Ladda in den karta som har skapats i kartbyggaren, denna måste motsvara det vägnät som bilen just nu står i, se avsnitt 2.2.3.
9. Lägg till de destinationer som bilen ska besöka, se till att börja med destinationen som bilen startar vid. Om något blir fel vid inmatning, klicka på töm lista för att börja om.
10. Klicka på skicka, nu bör bilen ha laddat in det autonoma uppdraget.
11. Klicka på kör för att påbörja körningen
12. Bilen bör nu köra till din önskade sekvens av destinationer.

6 KALIBRERING AV BILDBEHANDLINGEN

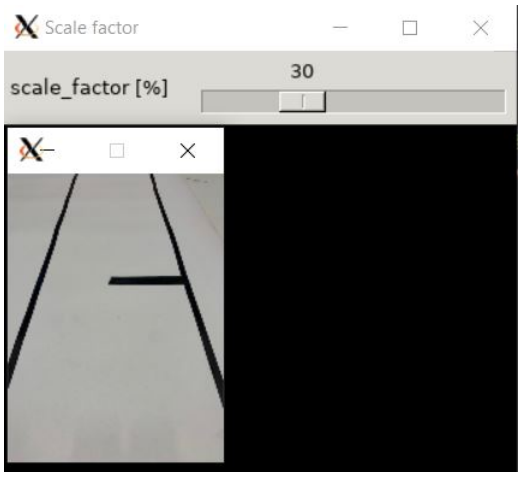
Bildbehandlnings algoritmen har parametrar som behövs uppdateras vid flyttning/byte av kamera. För att starta kalibreringen behöver du vara inne på Raspberry Pi:n och navigera dig till *Machinevision* mappen som ligger i *Sensor* mappen. I den mappen ska du skriva i terminalen *./calibration* för att starta kalibreringen. Några fönster kommer öppnas där skjutreglage finns samt en direktsänd bild från kameran på bilen. När skjutreglagen ändras ändras även den visade bilden för att ge respons i realtid på parametersändringen. När du är nöjd med parametrarna för den delen, klicka *enter* på tangentbordet för att gå vidare till nästa steg. När alla steg är genomförda kommer parametrarna sparas och användas i nästa körning av bilen.

Det första som bestäms är hur mycket bliden ska nedsamplas se figur 8. Reglaget anger procentmässigt hur mycket bliden ska samplas ner, exempel 100 ger original storlek och 50 ger hälften av original storleken.

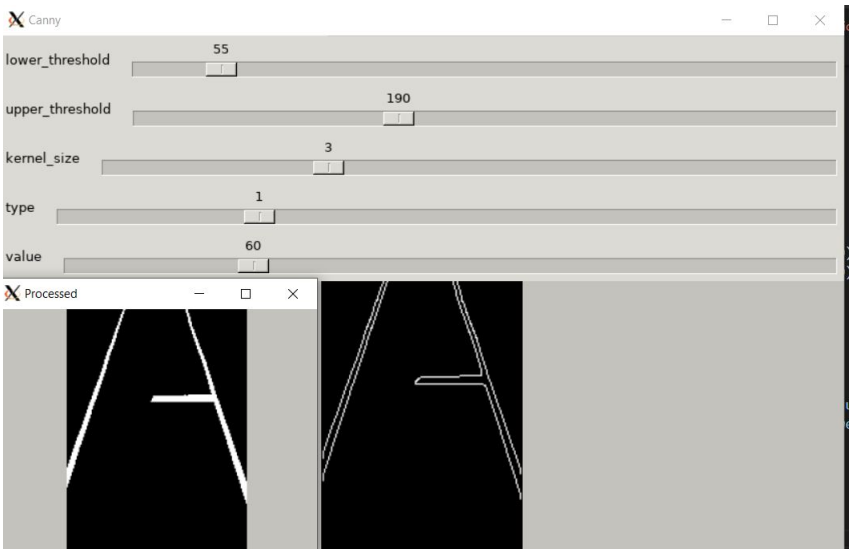
Nästa steg är att sätta trösklar för kantdetektionen samt välja processeringen av bilden, se figur 9. Reglagen *upper_threshold* och *lower_threshold* anger över och undre tröskel för kantdetektionen (se förstudien om sensormodulen under rubriken *Canny kantdetetkion* för förklaring på dessa trösklars innebördan). Reglaget *kernel_size* anger storleken på matrisen i den gaussiska lågpass-filtreringen som utföras i kantdetektionen. Reglaget *type* anger vilket format bliden ska processeras till och *value* anger vilket värde en pixel måste minst ha för att inte ignoreras.

Härnäst är att sätta upp fågelperspektivet och sliding-window sökninegn, se figur 10. Här är väglinjedetektionen igång samt stopplinje detektionen för att man ska kunna ställa in så det blir en bra detektion. Högra vägkanten markeras med rött, vänstra med blått och stopplinjen med en grön prick. Reglagen *SW_height* och *SW_width* anger höjd och bredd på sök fönstret (i search-window metoden), vilket går att se på de utritade rutorna på bliden. Reglagen *alpha*, *beta*, *gamma* ändrar på motsvarande värden i fågelperspektivs transformen, *f* ändrar fokallängden och *distance* zoomar in eller ut bilden. Sista reglaget, *threshold*, anger gränsen för vad som räknas som ett tomt fönster i search-window sökningen. Notera, det är inte rekommenderat att ändra *beta* och *gamma* reglagen, de ge bra transformering vid 90.

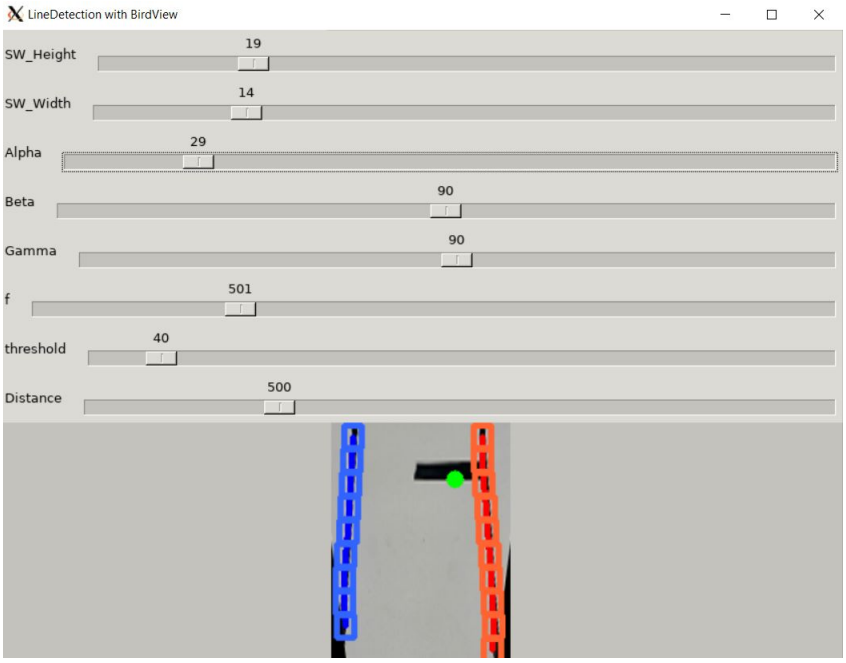
Till sist ska antal pixlar per centimeter anges utifrån kamerans transformerat perspektiv, se figur 11. Reglagen *x_left* och *y_left* anger linjens vänstra ändas position, *x_right* och *y_right* anger linjens högra ändes position. Linjens ska gå från vänstra till högra vägkanten och ligga så nära botten som möjligt för att ge en så säker konvertering som möjligt



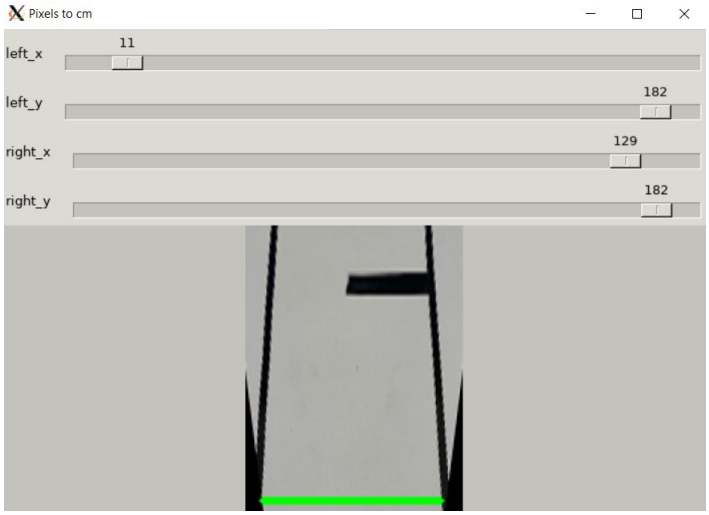
Figur 8: Steget i kalibreringen där nedsamplingen av bilden bestäms.



Figur 9: Steget i kalibreringen där parametrar för kantdetektionen och processeringen sätts.



Figur 10: Steget i kalibreringen där parametrar för fågelprespektivet och sliding-window metoden sätts.



Figur 11: Steget i kalibreringen där pixlar per cm sätts.

7 ÖVRIGT

7.1 Felsökning

Nedan listas olika anledningar till att bilen inte rör sig:

- Batteriet är urladdat. Kontrollera batteriets status genom att koppla in det i laddaren. Om lampan (efter att ha blinkat grönt) lyser grönt konstant så är det full-laddat. Annars är det svårt att veta. För att vara säker rekommenderas du använda ett fulladdat batteri. Se figur 14.
- Bilens körmod är satt fel. Om du vill köra bilen i manuellt läge måste bilens läge vara satt till manuellt läge. Testa att klicka på radioknappen för manuellt läge även ifall indikatorn redan visar manuellt läge. Se avsnitt 2.2.
- Bilen är inte ansluten till användardatorn. Kontrollera att du har angivit korrekt ip-adress (se avsnitt 2.2). Det kan även vara bra att starta om hela bilen och användardatorn. Tryck även på bilens reset-knappar, se Figur 13. Om problemet inte löser sig kan du behöva koppla in dig till bilens Raspberry pi, se avsnitt 7.3.
- Handkontrollern kopplades inte in före du startade användarprogrammet. Detta gäller enbart om du vill framföra bilen i manuell körmod.
- Heartbeat-stoppet är felaktigt aktiverat. Det kan också vara korrekt aktiverat, detta betyder att bilen inte har någon anslutning till användardatorn. Se avsnitt 2.4. Se även punkt ovan om att bilen kan tappa anslutning.
- Ultraljuds-stoppet är felaktigt aktiverat. Kanske finns det också något framför bilen och i sådant fall fungerar ultraljuds-stoppet vilket vore lyckligt. Se avsnitt 2.4.

Nedan listas olika anledningar till att bilen rör sig dåligt, kör av vägen eller stannar inte vid destinationer.

- Kalibreringen är dålig. Kalibrera om bilen och kontrollera att den detekterar vägen bra samt destinationer. Kolla i även i olika vägsegment genom att flyttat bilen till olika platser och se till att detektionen fortfarande är bra, se avsnitt 6.
- Reglerparameterarna är dåliga. Testa ändra de under körning och se om det blir bättre resultat på bilen körning.

7.2 Viktiga noter

Nedan listas några viktiga saker du måste tänka på vid användning av bilen. De har redan nämnts i dokumentet men tåls att upprepas.

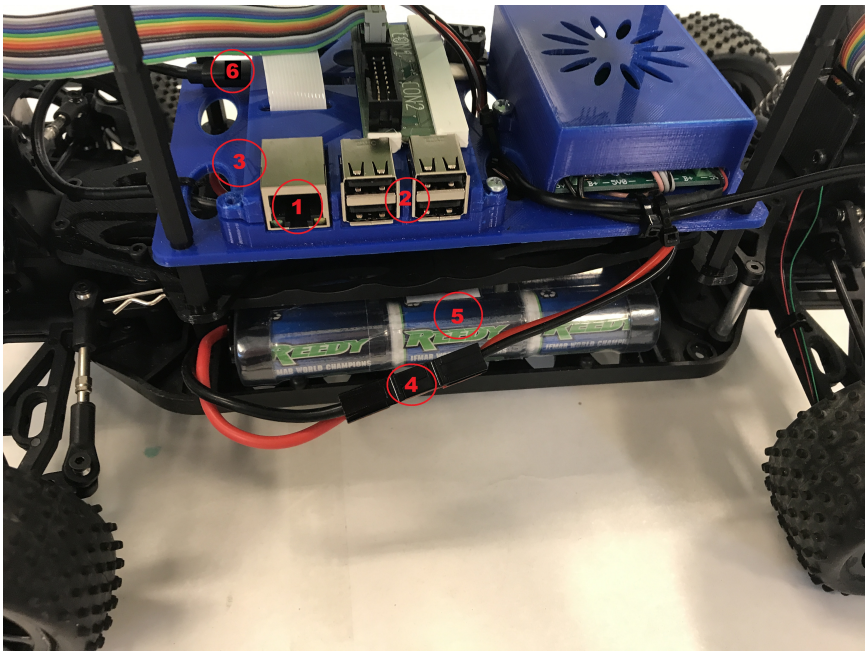
- Innan du gör något med bilen, kontrollera att alla sladdar är inkopplade. Kontrollera även att inga sladdar är i närheten av något av bilens 4 hjul.
- Du måste ansluta handkontrollern till användardatorn innan du startar GUI:t om du vill använda det manuella körläget.
- Använd bilen med försiktighet. Systemet har implementerat säkerhetsspärrar för att undvika krascher men dessa kan misslyckas med sin kraschdetektion. Var därför försiktig med vilken omgivning bilen framförs i samt vilken hastighet den körs i. Läs mer i avsnitt 2.4.

7.3 Egen programmering av systemet

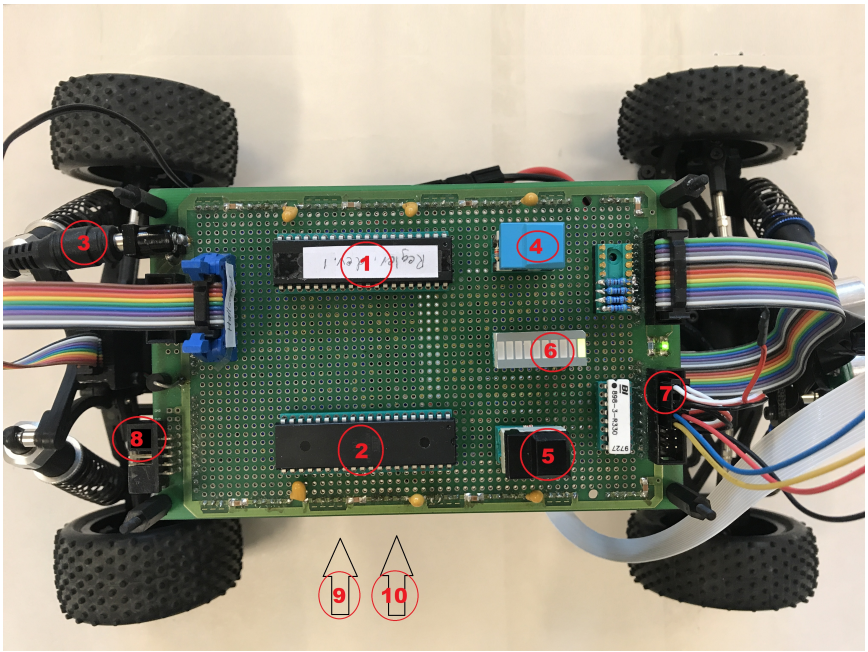
Nedan beskrivs dels hur du programmerar bilens mikrodatorer och dels hur du programmerar och/eller konfigurerar bilens Raspberry pi. Detta kan behövas om bilen tappar anslutning med ett nätverk eller om du vill modifiera bilens funktionalitet. Det senare rekommenderas inte eftersom det kan förstöra systemet.

1. **Programmering av mikrodatorer** : I detta avsnitt är du mogen att veta att mikrodatorerna faktiskt heter AtMega 1248p. De programmeras med Atmel Studio 6.2 och en JTAG-enhet. En JTAG-enhet syns i Figur 17. I figuren syns markering 1 som är själva JTAG-enheten, den skall kopplas in med kontakten markerad med 2 in i JTAG-anslutningen på virkortet. JTAG-anslutningen på virkortet syns i Figur13, markering 8. Du rekommenderas starkt använda dokumentationen för AtMega 1248p, se [1]. Lycka till och använd google flitigt.
2. **Programmering av Raspberry pi**: En Raspberry pi är som en vanlig Linux dator. Koppla in skärm med HDMI, mus, tangentbord och du är redo att programmera!

8 FÖRKLARANDE FIGURER

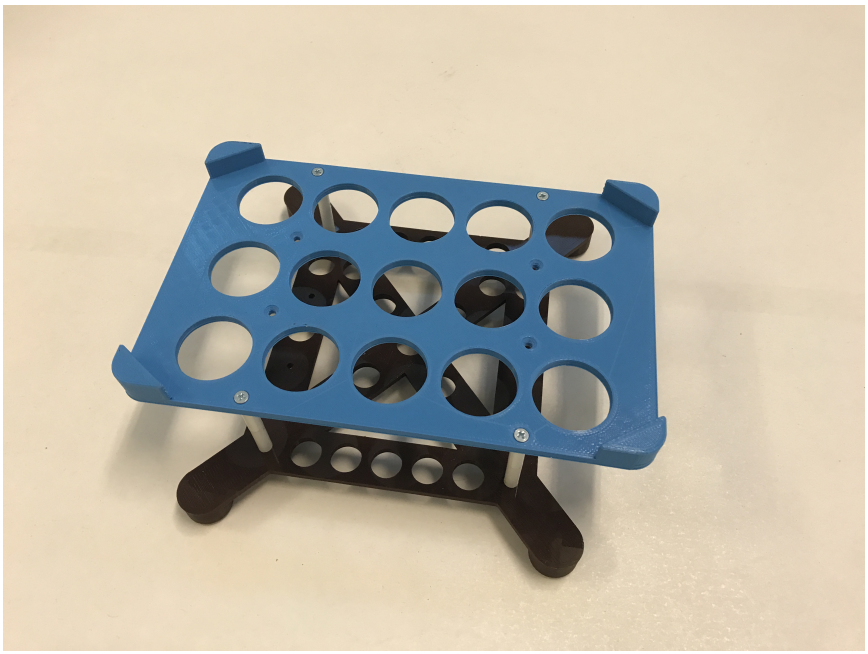


Figur 12: 1, 2: Standardanslutningar för användning av Raspberry Pi:n, 3 är en HDMI-anslutning för denna. 4, 5: Kontakt för batteri och batteriet självt, se till så att batteriets sladd inte hänger löst så som den gör i bilden. 6: Strömkabel till Raspberry Pi:n.



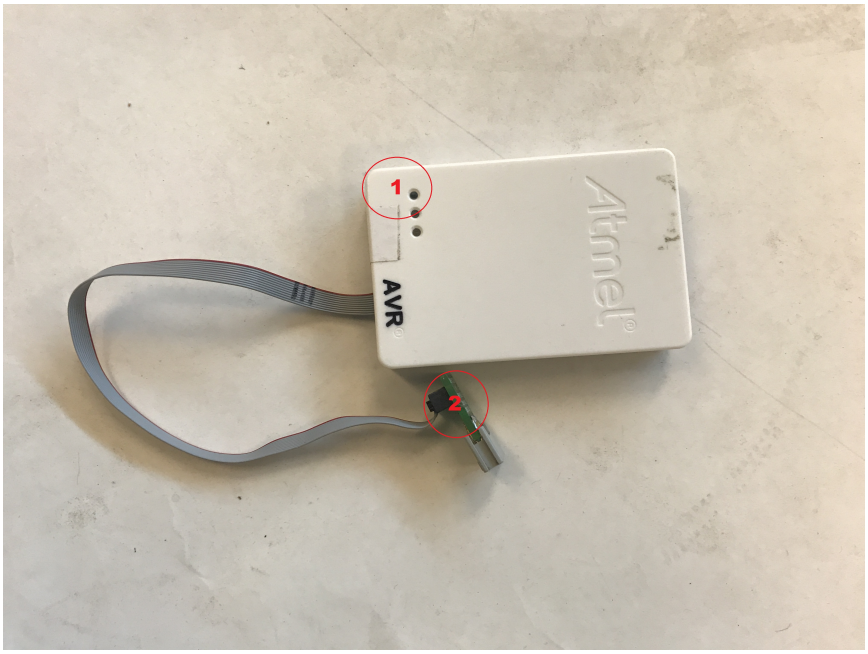
Figur 13: 1: Mikroprocessorn som hanterar styrning av bil. 2: Mikroprocessorn som hanterar bilens hastighetsmätning och ultraljud. 3: Virkortets strömsladd. 4, 5: Resetknappar för 1 respektive 2. 6: Lysdioder, i bilden lyser diod 1. 7: Anslutningar för hastighetsmätning, styrservon och motorer. 8: JTAG-kontakt för programmering av mikroprocessorerna. 9: Syns inte på bilden men här sitter en silver-aktig switch som agerar strömbrytare för hela bilen. 10: Syns inte heller men stänger av bilens styr och motorfunktionalitet.

Figur 14: 1: Statuslampa för laddning av batteri. Grönt ljus betyder färdigladdat batteri. Rött ljus betyder icke-färdigladdat batteri. Blinkande rött ljus betyder att någonting är fel. 2: Anslutning mellan batteri och batteriladdare. 3: Batteri. 4: Strömanslutning till batteriladdare.



Figur 15: Testställning där bilen säkert kan stå och vila utan fara för krasch, såvida användaren inte gör något riktigt oförutsägbart och dumt.

Figur 16: Handkontrollern som kan användas för att styra bilen. 1: Gasspak, 2: Styrspak.



Figur 17: JTAG-enhet för att programmera mikroprocessorerna AtMega. 1: JTAG-enheten, 2: JTAG Anslutning.

REFERENSER

[1] Microchip. (2020) megaavr data sheet. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega164A_PA-324A_PA-644A_PA-1284_P_Data-Sheet-40002070B.pdf

D KRAVSPECIFIKATION

Kravspecifikation

Grupp 10

7 juni 2021

Version 1.1

Status

Granskad	Henrik Ahlinder	2020-04-28
Godkänd	Anders Nilsson	2020-04-28

Projektidentitet

Grupp E-post: TSEA56_2021VT_O2_G10@groups.liu.se

Beställare: Anders Nilsson, Linköpings universitet
Tfn: 013-28 26 35
E-post: anders.p.nilsson@liu.se

Handledare: Peter Johansson
Tfn: +4613281345
E-post: peter.a.johansson@liu.se

Kursansvarig: Mattias Krysander
Tfn: 013-28 21 98
E-post: mattias.krysander@liu.se

Projektdeltagare

Namn	Ansvar	E-post
Edvin Johansson	Utvecklare	edvjo404@student.liu.se
Mikael Josefsson	Utvecklare	mikjo695@student.liu.se
Sebastian Andersson	Utvecklare	seban218@student.liu.se
Martin Dahl	Grafisk Design (GD)	marda545@student.liu.se
Charbel Youssef	Utvecklare	chayo642@student.liu.se
Henrik Ahlinder	Projektledare (PL)	henah490@student.liu.se

INNEHÅLL

1 Inledning 1
- 1.1 Parter 1
- 1.2 Syfte och mål 1
- 1.3 Användning 1
- 1.4 Bakgrundsinformation 1
- 1.5 Definitioner 1

2 Översikt av systemet 2
- 2.1 Grov beskrivning av produkten 2
- 2.2 Produktkomponenter 2
- 2.3 Beroende till andra system 2
- 2.4 Ingående delsystem 2
- 2.5 Avgränsningar 2
- 2.6 Designfilosofi 2

3 Generella krav 3

4 Användardator 4
- 4.1 Gränssnitt 4
- 4.2 Designkrav 4
- 4.3 Funktionella krav 4

5 Kommunikationsmodul 6
- 5.1 Externa gränssnitt 6
- 5.2 Designkrav 6
- 5.3 Funktionella krav 6

6 Styrmodul 8
- 6.1 Externa gränssnitt 8
- 6.2 Designkrav 8
- 6.3 Funktionella krav 9
- 6.4 Användargränssnitt 9

7 Sensormodul 10
- 7.1 Externa gränssnitt 10
- 7.2 Designkrav 10
- 7.3 Funktionella krav 10
- 7.4 Användargränssnitt 11

8 Prestandakrav 11

9 Tillförlitlighet 11

10 Ekonomi 11

11 Leveranskrav 12
- 11.1 Tidrapporter 12

12 Dokumentation 13

DOKUMENTHISTORIK

Version	Datum	Utförda ändringar	Utförd av	Granskad
0.1	2021-01-26	Första utkast	Alla	Alla
0.2	2021-01-29	Korrigeringar efter kommentarer från beställare	Alla	Alla
1.0	2021-01-29	Preciserade krav och kravprioritet	Martin Dahl	PL
1.1	2021-04-29	Omprioritering och förtydligande av krav inför BP5a	PL	PL

1 INLEDNING

I detta dokument beskrivs vilka krav som ställs på systemet som ska utvecklas av kandidatprojektgrupp 10 (”gruppen”).

Krav	Version	Beskrivning	Prioritet
1	Original	Alla dokument skall uppfylla instruktioner beskrivna i [1]	Utgått
1A	2021-04-28	Alla dokument skall uppfylla instruktioner beskrivna i [1]	Slut

1.1 Parter

Projektets parter är beställaren från ISY, handledaren och deltagarna i projektgruppen.

1.2 Syfte och mål

Målet med kravspecifikationen är att definiera systemkraven för taxibilen som utvecklas.

1.3 Användning

Kravspecifikationen utgör ett kontrakt mellan beställare och projektgrupp för funktionalitet av taxibilen. Dokumentet ska utnyttjas för att vägleda gruppen i utvecklingen av taxibilen samt definiera krav för en färdigutvecklad produkt.

1.4 Bakgrundsinformation

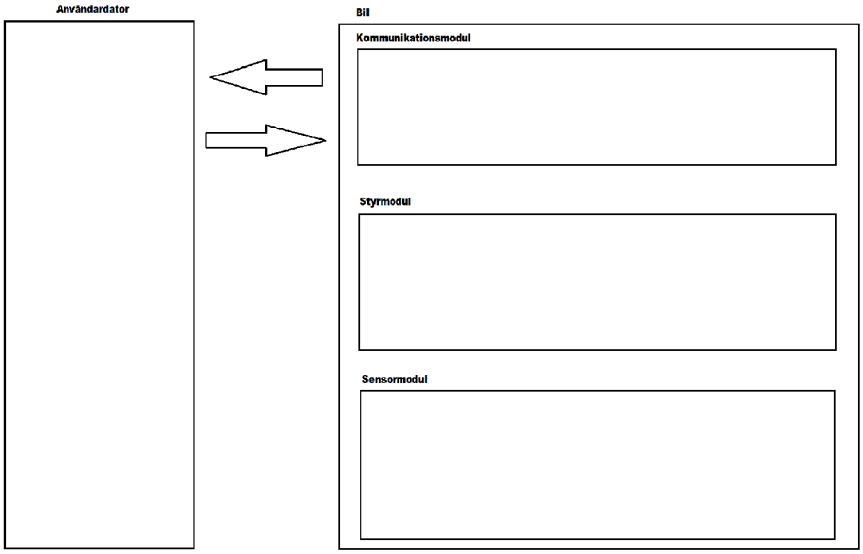
I enlighet med projektdirektivet och ett samtal med beställaren bestäms de krav som ska uppfyllas för att konstruera taxibilen.

1.5 Definitioner

- ”Taxibil” benämns ibland som ”taxibilen” eller ”bilen”
- ”Användarprogrammet” benämns ibland som ”programmet”, ”användardatorn” eller ”datorn”.
- ”De autonoma lägena” är ett samlingsbegrepp för det helautonoma läget och det semi-autonoma läget.
- Krav med prioritet ”Bas” är de krav som ska vara uppfyllda till BP5a.
- Krav med prioritet ”Slut” är de krav som ska vara uppfyllda till BP5b.
- Krav med prioritet ”Extra” ska uppfyllas om tid finns och då samtliga krav med prioritet ”Bas” och ”Slut är uppfyllda.
- Krav med prioritet ”Leverans” är krav som ska vara uppfyllda vid särskilda datum. Datumet då kravet ska vara uppfyllt specificeras i respektive krav.

2 ÖVERSIKT AV SYSTEMET

I Figur 1 ses en översiktlig beskrivning av systemet.



Figur 1: Översikt av systemet

2.1 Grov beskrivning av produkten

Produkten är en taxibil med funktionalitet för delvis och helt autonom styrning. Bilen kan användas i tre lägen; manuellt, semi-autonomt och helautonomt läge. I manuellt läge accepterar bilen direkta styrinstruktioner från användardatorn. I de autonoma lägena följer taxibilen ett väldefinierat vägnät där den accepterar och tolkar navigationsinstruktioner från användardatorn baserat på vägnätet samt autonomt finner en väg som uppfyller instruktionerna.

2.2 Produktkomponenter

I produkten ingår en användardator för att ge kommandon till bilen. Bilen är uppdelad i tre moduler: en kommunikationsmodul, en styrmodul och en sensormodul, se avsnitt 5, 6 respektive 7. Utöver dessa moduler består bilen även av ett chassi med motorer, däck och styrservon.

2.3 Beroende till andra system

Bilen beror inte på några andra system förutom de som har beskrivits ovan.

2.4 Ingående delsystem

De ingående delsystemen är användargränssnittet på användardatorn samt de tre modulerna i bilen.

2.5 Avgränsningar

När bilen kör helautonomt eller semiautonomt så görs det endast i det väldefinierade vägnätet.

2.6 Designfilosofi

Bilens design är med hållbarhet i åtanke och tar hänsyn till och respekterar alla människors upplevelser och erfarenheter.

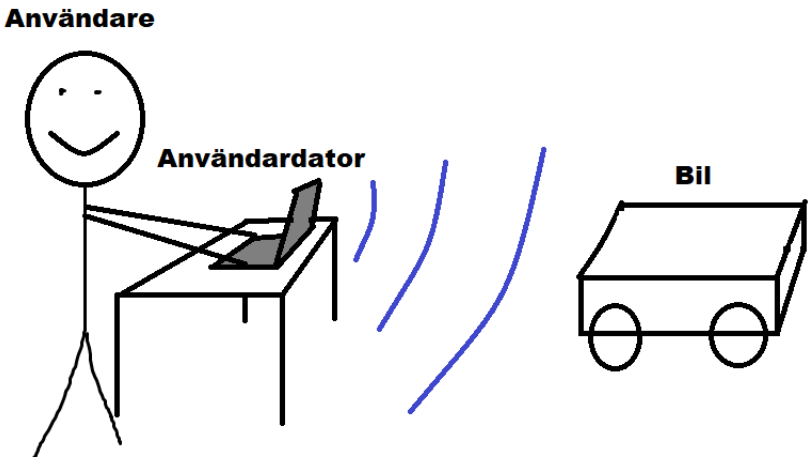
3 GENERELLA KRAV

Listar generella krav på bilen.

Krav	Version	Beskrivning	Prioritet
2	Original,	Bilen ska kunna köra i tre olika lägen: manuellt, semi-autonomt och helautonomt,	Utgått
2A	2021-04-28	Bilen ska kunna köra i tre olika lägen: manuellt, semi-autonomt och helautonomt	Slut
3	Original	I manuellt läge ska användaren kunna styra följande hos bilen: gaspådrag, fram/back körning samt styrutslag.	Bas
4	Original	I semi-autonomt läge ska användaren kunna ge bilen enklare kommandon som avgör vad bilen gör härnäst. Det inkluderar, men är inte begränsat till, kommandon som ”följ vägen fram till nästa stopplinje”, ”sväng vänster i korsningen” och ”stanna”.	Utgått
4A	2021-04-28	I semi-autonomt läge ska användaren kunna ge bilen enklare kommandon som avgör vad bilen gör härnäst. Kommandon består av två kontinuerliga instruktioner: 1. följ vänster eller höger väglinjesamt 2. "Kör framåt eller stå still".	Bas
5	Original	I det helautonoma läget ska det gå att skicka ett avancerat köruppdrag som bilen ska utföra utan ytterligare styrning från användare.	Bas
6	Original	I de autonoma lägena ska bilen följa ett avgränsat väldefinierat vägnät.	Bas
7	Original	I de autonoma lägena ska det gå att skicka en karta och startposition till bilen.	Bas
8	Original	Bilen ska kunna skapa en karta över vägnätet med avstånd, detta kallas för ”kartläggning”.	Extra
9	Original	Vid kartläggning ska bilen stanna vid destinationer och invänta ett destinationsnamn från användaren innan kartläggningen fortsätter.	Extra
10	Original	Gränssnitten mellan modulerna ska vara noggrant specificerade i den tekniska dokumentationen.	Utgått
10A	2021-04-28	Gränssnitten mellan modulerna ska vara noggrant specificerade i den tekniska dokumentationen.	Slut
11	Original	Det ska gå enkelt att byta ut en modul mot en annan.	Bas

4 ANVÄNDARDATOR

Delsystemet ”Användardator” har ett program, härefter refererat till som ”Användarprogrammet”, från vilket användaren styr och observerar bilen, illustrerat i Figur 2. Från Användarprogrammet kan användaren få information om bilens nuvarande tillstånd, plats och ändra bilens mål och styrparametrar.



Figur 2: Användardator

4.1 Gränssnitt

Detta avsnitt beskriver krav på användardatorns externa gränssnitt.

Krav	Version	Beskrivning	Prioritet
12	Original	Användarprogrammet ska utgöra bilens användargränssnitt	Bas
13	Original	Användargränssnittet ska vara användarvänligt	Slut

4.2 Designkrav

Detta avsnitt beskriver krav på användarprogrammets design.

Krav	Version	Beskrivning	Prioritet
14	Original	Användarprogrammet ska kommunicera med kommunikationsmodulen över trådlös länk som wifi eller bluetooth.	Bas

4.3 Funktionella krav

Detta avsnitt beskriver krav på användardatorns funktionalitet.

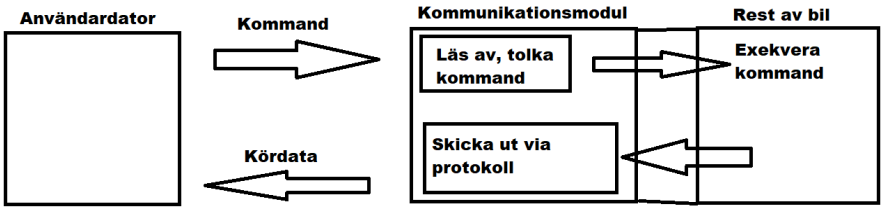
Krav	Version	Beskrivning	Prioritet
15	Original	I Användarprogrammet ska användaren kunna byta till manuellt läge.	Bas
16	Original	I Användarprogrammet ska användaren kunna byta till semi-autonomt läge.	Bas

forts. på nästa sida

forts. från föregående sida			
Krav	Version	Beskrivning	Prioritet
17	Original	I Användarprogrammet ska användaren kunna byta till helautonomt läge.	Bas
18	Original	Användarprogrammet ska spara data från taxibilen.	Slut
19	Original	Den sparade datan ska kunna plottas.	Slut
20	Original	Under körning ska följande data visas i Användarprogrammet: körtid, körsträcka, hastighet, detektion och avstånd till hinder föröver, gaspådrag och styrutslag.	Utgått
20A	2021-04-28	Under körning ska följande data kunna visas i Användarprogrammet: körtid, körsträcka, hastighet, detektion och avstånd till hinder föröver, gaspådrag och styrutslag.	Slut
21	Original	Vid körning i de autonoma moderna ska aktuellt vägsegment och lateral position visas.	Slut
22	Original	Under köruppdrag ska planerad rutt till nästa destination visas.	Slut

5 KOMMUNIKATIONSMODUL

Syftet med kommunikationsmodulen är att ta in data och sköta kommunikationen mellan sensormodulen, styrmodulen och användardatorn. Detta är skisserat i Figur 3.



Figur 3: Kommunikationsmodul

5.1 Externa gränssnitt

Komunikationsenheten tar hand om den trådlösa kommunikationen mellan taxibilen och användardatorn. Den ska bestå av en processor. Information om vägbanan och eventuella hinder tas emot från sensormodulen och kameran, och skickas sedan till datorn för vidare hantering. Kommunikationsmodulen tar också emot styrsignaler från användardatorn som skickas vidare till styrmodulen.

Krav	Version	Beskrivning	Prioritet
23	Original	Kommunikationsenheten ska kommunicera trådlöst med användardatorn.	Bas
24	Original	Kommunikationsenheten ska kunna förmedla information till styrmodulen	Bas
25	Original	Kommunikationsenheten ska kunna ta emot och föra vidare data från sensorenheten och kameran.	Bas

5.2 Designkrav

Detta avsnitt beskriver krav på designmodulens design.

Krav	Version	Beskrivning	Prioritet
26	Original	Kommunikationsenheten ska innehålla en (1) processor.	Bas
27	Original	Under pågående köruppdrag ska följande data om bilen skickas kontinuerligt till användardatorn: körsträcka, hastighet, detektion och avstånd till hinder föröver, gaspådrag och styrutslag.	Bas

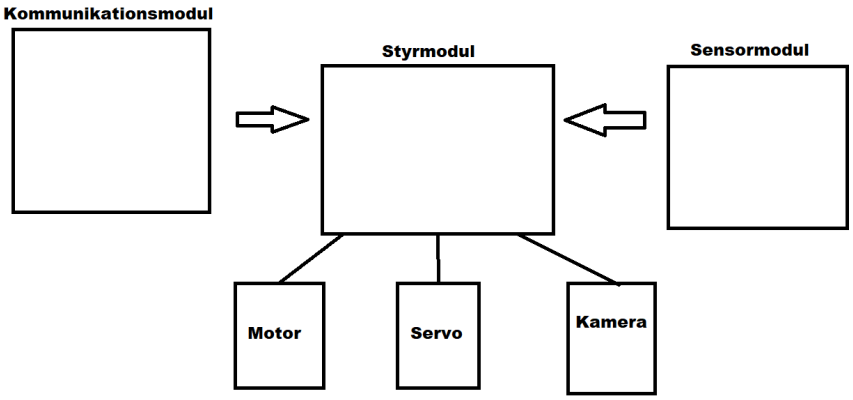
5.3 Funktionella krav

Detta avsnitt beskriver krav på kommunikationsmodulens funktionalitet.

Krav	Version	Beskrivning	Prioritet
28	Original	Kommunikationsenheten ska kunna ta emot data från sensorenheten och kameran	Bas
29	Original	Kommunikationsenheten ska skicka ut data till styrenheten	Bas
30	Original	Kommunikationsenheten ska omvandla utgående och inkommande data till relevant format för överföring.	Bas

6 STYRMODUL

Modulens syfte är att tolka användarens och sensorernas signaler för att styra bilens motorer och därmed kontrollera färdriktning och hastighet. Detta kan tolkas ur Figur 4



Figur 4: Styrmodul

6.1 Externa gränssnitt

Detta avsnitt beskriver krav på styrmodulens externa gränssnitt.

Krav	Version	Beskrivning	Prioritet
31	Original	Styrmodulen ska acceptera bilddata från en kamera.	Bas
32	Original	Styrmodulen ska ta in sensorinformation från sensormodulen.	Bas
33	Original	Styrmodulen ska ta emot kommandon från användaren då bilen är i manuellt eller semi-autonomt läge.	Bas

6.2 Designkrav

Detta avsnitt beskriver krav på styrmodulens design.

Krav	Version	Beskrivning	Prioritet
34	Original	Styrmodulen ska använda en (1) processor.	Utgått
34A	2021-04-28	Styrmodulen ska använda åtminstone en (1) processor.	Bas
35	Original	Styrmodulen ska kontrollera ett antal motorer.	Bas
36	Original	Styrmodulen ska kontrollera ett styrservo.	Bas

6.3 Funktionella krav

Detta avsnitt beskriver krav på styrmodulens funktionalitet.

Krav	Version	Beskrivning	Prioritet
37	Original	Styrmodulen ska använda en styralgoritm som gör att bilen följer banan i de autonoma lägena.	Bas
38	Original	Parametrar till styrmodulens styralgoritm ska kunna initieras trådlöst.	Slut
39	Original	Parametrar till styrmodulens styralgoritm ska kunna ändras under körning.	Bas

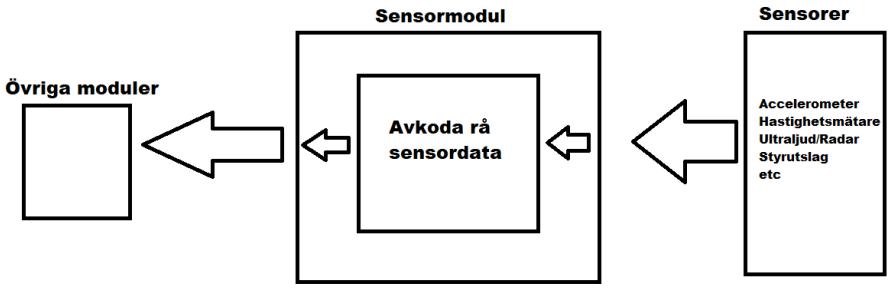
6.4 Användargränssnitt

Detta avsnitt beskriver krav på styrmodulens användargränsnitt.

Krav	Version	Beskrivning	Prioritet
40	Original	Det ska finnas tre lysdioder som visar vilket av styrlägena manuellt, semi-autonomt och helautonomt som är aktivt	Slut

7 SENSORMODUL

Sensormodulens syfte är att läsa av sensorer, konvertera sensorvärdena till data som övriga moduler kan läsa och skicka vidare informationen till övriga moduler. Detta illustreras i Figur 5.



Figur 5: Sensormodul

7.1 Externa gränssnitt

Detta avsnitt beskriver krav på sensormodulens externa gränssnitt.

Krav	Version	Beskrivning	Prioritet
41	Original	Sensormodulen ska kommunicera sensordata till övriga moduler.	Bas

7.2 Designkrav

Detta avsnitt beskriver krav på sensormodulens design.

Krav	Version	Beskrivning	Prioritet
42	Original	Sensormodulen ska använda en (1) processor.	Utgått
42A	2021-04-28	Sensormodulen ska använda åtminstone en (1) processor.	Bas

7.3 Funktionella krav

Detta avsnitt beskriver krav på sensormodulens funktionalitet.

Krav	Version	Beskrivning	Prioritet
43	Original	Sensormodulen ska läsa av sensorer så att följande data om bilen kan bestämmas: Körsträcka, hastighet, detektion och avstånd till hinder förröver, gaspådrag, styrutslag och position på banan.	Bas
44	Original	Sensormodulen ska förse data till övriga moduler när den efterfrågas.	Bas
45	Original	Sensormodulen ska konvertera mätdata från sensorerna till data som de andra enheterna kan läsa och förstå.	Bas

7.4 Användargränssnitt

Detta avsnitt beskriver krav på sensormodulens användargränsnitt.

Krav	Version	Beskrivning	Prioritet
46	Original	Sensormodulen ska innehålla ett debug-gränssnitt som användaren kan ansluta sig till via dator. Där ska det gå att se rå sensordata och vilken data som skickas ut från sensormodulen till övriga moduler.	Extra

8 PRESTANDAKRAV

Detta avsnitt beskriver prestandakrav på systemet som helhet.

Krav	Version	Beskrivning	Prioritet
47	Original	Utvärde från styralgoritm ska inte oscillera märkbart. Detta inkluderar gaspådrag, riktning, lateral position.	Bas

9 TILLFÖRLITLIGHET

Detta avsnitt beskriver krav på tillförlitlighet hos systemet.

Krav	Version	Beskrivning	Prioritet
48	Original	Ska klara av tävlingsuppdraget 75 % av gångerna.	Slut

10 EKONOMI

Detta avsnitt beskriver projektetsmedlemmarnas förväntade tidsinvestering.

Krav	Version	Beskrivning	Prioritet
49	Original	Varje projektmedlem ska lägga åtminstone 230 h på konstruktionsarbete efter beslutspunkt 2.	Slut

11 LEVERANSKRAV

Detta avsnitt beskriver de leveranser som kommer att ske under projektets gång.

Krav	Version	Beskrivning	Prioritet
50	Original	Kravspecifikationen ska vara godkänd av beställare 3/2-2021	Leverans
51	Original	Projektplan, tidplan och systemskiss ska vara godkända av beställare senast 18/2-2021	Leverans
52	Original	Designspecifikationen ska vara godkänd av handledare 12/3-2021	Leverans
53	Original	Design av taxibil ska vara godkänd av handledare senast 16/4-2021	Leverans
54	Original	Krav med prioritet ”Bas” ska verifieras vara uppfyllda av beställare senast 5/5-2021	Leverans
55	Original	Version 1.0 av kappan ska levereras till beställare senast 18/5-2021	Leverans
56	Original	Version 1.0 av Teknisk dokumentation ska levereras till beställare senast 19/5-2021	Leverans
57	Original	Version 1.0 av Användarhandledning ska levereras till beställare senast 19/5-2021	Leverans
58	Original	Krav med prioritet ”Slut” ska verifieras vara uppfyllda av beställare senast 19/5-2021	Leverans
59	Original	Efterstudien ska vara inlämnad till beställaren senast 7/6-2021	Leverans

11.1 Tidrapporter

Detta avsnitt beskriver hur nedlagd tid rapporteras till beställare.

Krav	Version	Beskrivning	Prioritet
60	Original	Summering av nedlagd tid för framställning av kravspecifikation ska levereras till beställare senast 3/2-2021	Leverans
61	Original	Summering av nedlagd tid för framställning av projektplan, tidplan och systemskiss ska levereras till beställare senast 22/2-2021	Leverans
62	Original	Tidrapport, uppdaterad tidplan samt kort sammanfattning av framsteg som gjorts, problem som stötts på samt kommande aktiviteter ska levereras till beställare följande datum: 29/3, 6/4, 12/4, 19/4, 26/4, 3/5, 10/5, 17/5, 24/5, 31/5, 7/6	Leverans

12 DOKUMENTATION

Tabell 22 listar de dokument som skall produceras

Tabell 22: Dokument som skall produceras

Dokument	Språk	Syfte	Målgrupp	Format
Projektplan	Svenska	Beskriver en tolkning av hur krav och andra behov ur projektdirektivet ska uppfyllas.	Beställare och projektmedlemmar	pdf
Kravspecifikation	Svenska	Entydig beskrivning av vad som ska produceras. Utgör underlaget för systemskiss, konstruktionsdokument och projektplan.	Beställare och projektgrupp	pdf
Designspecifikation	Svenska	Beskriver hur systemet ska byggas och hur taxibilen ska konstrueras.	Handledare och projektmedlemmar	pdf
Teknisk dokumentation	Svenska	Konstruktionsunderlag och dokumentation för underhåll, felsökning och beskrivning av olika lösningar som berör både hård- och mjukvara.	Handledare och projektmedlemmar	pdf
Användarhandledning	Svenska	Beskriver hur produkten ska användas	Handledare och projektmedlemmar	pdf
Kappa	Svenska	Ger en komplett överblick av det genomförda arbetet i projektet.	Beställare	pdf
Efterstudie	Svenska	Sammanfattning av erfarenheterna och utvärderingar från projektarbetet. Utgör ett underlag för utveckling av genomförande av framtida projekt.	Beställaren och projektmedlemmar	pdf

REFERENSER

[1] S. Språknämnden, *Svenska skrivregler*, 2nd ed. Liber AB, Stockholm, 2000, iSBN47-04974-X.

E FÖRSTUDIE - REGLER

Förstudie - Regler

Edvin Johansson
Martin Dahl

7 juni 2021

Version 1.01

Status

Granskad		
Godkänd		

INNEHÅLL

1 Introduktion 1
- 1.1 Frågeställning . . . 2
- 1.2 Metod . . . 2
- 1.3 Förkortningar, begrepp och storheter . . . 2

2 Kunskapsbas 3
- 2.1 Beslutfattningshierarki . . . 3
- 2.2 The Kinematic single-track model . . . 4
- 2.3 Rörelseplanering . . . 6
- 2.4 PID-reglering . . . 6
- 2.5 Pure-pursuit . . . 9
- 2.6 Viktad styrning med pure-pursuit och PID-reglering . . . 13
- 2.7 Rear-/front wheel position based feedback . . . 13

3 Sammanfattning, Diskussion och Slutsatser 16
- 3.1 Sammanfattning . . . 16
- 3.2 Diskussion . . . 16
- 3.3 Slutsats . . . 16

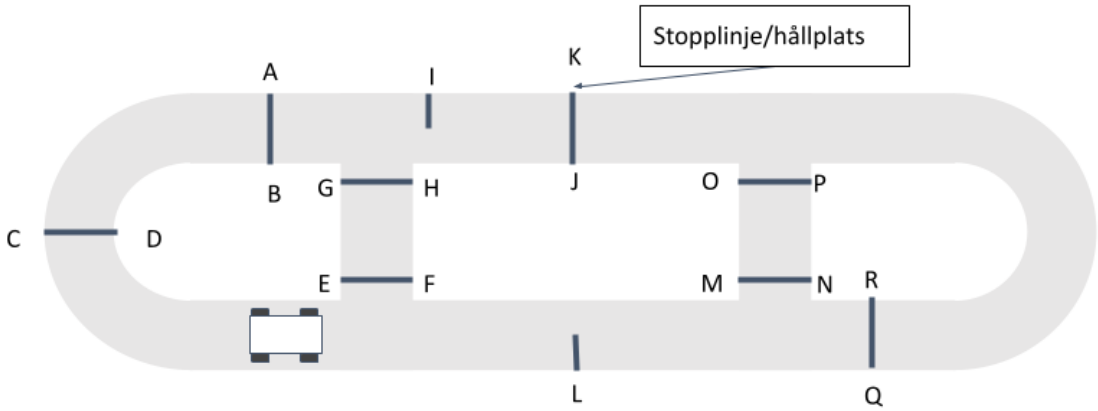
DOKUMENTHISTORIK

Version	Datum	Utförda förändringar	Utförda av	Granskad
0.1	2021-02-26	Första utkast	EJ och MD	EJ
1.0	2021-04-08	Slutgiltig version	EJ och MD	EJ
1.01	2021-04-27	Åtgärdad kommentar	MD	MD

1 INTRODUKTION

Denna förstudie avser att undersöka vilka typer av reglertekniska system som behövs hos en självkörande bil, vilken byggs under kandidatprojektet i kursen TSEA56.

Studiens omfattning avgränsas till en mindre bil (likt en RC-bil) som ska navigera på ett mindre väldefinierat vägnät, likt det illustrerat i figur 1, med tydliga väglinjer och stopplinjer/hållplatser. Det ska noteras att det verkliga vägnätet har avrundade kanter i samtliga svängar och korsningar. Exakta detaljer om vägnätet och dess utformning ges i Appendix A “Banspecifikation och tävlingsregler".



Figur 1: *Skiss över vägnät som bilen ska navigera i, tagen från projektets systemskiss. Bokstäverna representerar stopplinjer som kan vara bilens destination. Det riktiga vägnätet kan ha ett betydligt mer avancerat utseende men det förekommer endast raksträckor, svängar och T-korsningar.*

I vägnätet ska bilen utföra uppdrag givna från en användare. Exempelvis är ett uppdrag att köra från hållplats A till hållplats L på kortast möjliga tid och stanna. Bilen måste hålla alla sina fyra hjul innanför väglinjerna. Bilen måste även undvika hinder som kan komma att placeras mitt på vägen: Bilen får inte köra på eller förbi dessa utan måste invänta att hindret blir borttaget av användaren.

Studien undersöker inte sensorhantering utan bilen antas ha en perfekt uppfattning om sin omgivning. Det vill säga dess spatiala förhållande till väglinjer, hållplatser och hinder. Bilen behöver inte heller kartlägga vägnätet då en karta av denna ges av bilens användare. Även bilens startposition ingår i kartan. Sammanfattningsvis utgår studien från att bilen vet var den befinner sig i förhållande till vägnätet och dess vägmärken.

Syftet med studien är därmed att undersöka metoder för att skapa ett system som kan framföra bilen på ett önskvärt sätt i vägnätet.

1.1 Frågeställning

Studiens frågeställning sammanfattas med nedanstående tre punkter:

1. Vilka reglertekniker bör användas för att få bilen att köra inom väglinjerna?
2. Hur hanteras olika typer av vägsegment i ett vägnät, såsom korsningar och svängar?
3. Vilken information krävs för att navigera i vägsegment?

1.2 Metod

Frågeställningen ska besvaras med hjälp av litteraturstudier, med källor som forskningsartiklar, liknande projekt på internet och webbartiklar.

1.3 Förkortningar, begrepp och storheter

- **LABV**: *Lateralt avstånd mellan bil och väglinje*, ett mätvärde för att mäta hur fel bilen kör i förhållande till vägens mittlinje vid väglinjeföljning.
- **VVB**: *Vinkel Väg-Bil*, vinkeln mellan bilen och vägen, beskriver hur parallell bilens riktning är med vägens riktning.
- **RMS**: *Root Mean Square Error*, taget från den engelska benämningen för *kvadratiskt medelvärde*. Kvantiteten används för att mäta ett absolut fel hos bilens körning.
- **T**: Integrationsintervall för I-delen hos PID-regulatorn.
- **Väglinje**: De visuella väglinjerna i vägnätet.
- **Referenslinje**: Den bana som bilen önskas följa, ofta en förskjutning i sidled av väglinjerna.

2 KUNSKAPSBAS

Att få en bil att köra efter väglinjer är ett väl studerat problem. I detta kapitel presenteras olika reglertekniker för att uppnå detta mål, styrkor och svagheter med de olika teknikerna samt de algoritmer eller formler som används.

2.1 Beslutfattningshierarki

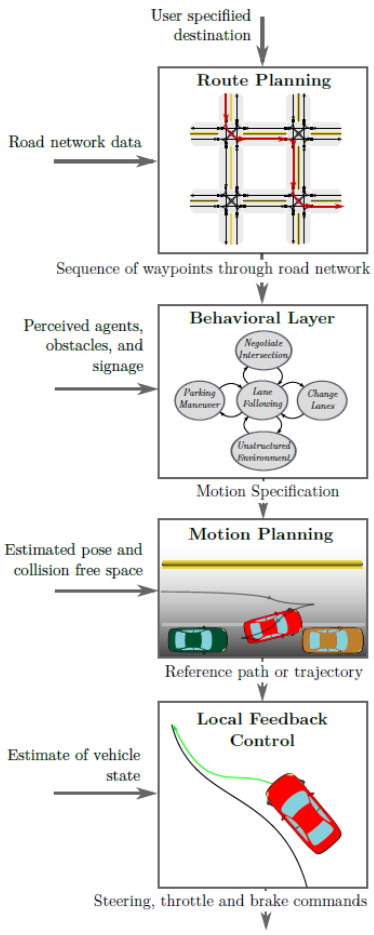
Enligt [1] kan beslutfattningshierarkin hos autonoma fordon delas upp i fyra nivåer, där beslut tagna i den översta nivån används som insignaler till den nästkommande nivån, illustrerade i figur 2. De första två nivåerna är utanför omfattningen av denna studie men beskrivs kortfattat eftersom besluten från dessa påverkar besluten för de två sista nivåerna, vilka är de två som är av intresse i denna studie.

Den översta nivån kallas *ruttplanering* och dess uppgift är att, givet ett vägnät, bestämma den snabbaste vägen från en startdestination till en slutdestination. För små vägnät kan denna väg snabbt beräknas med klassiska kortaste-väg-algoritmer som A* [2] eller Dijkstras algoritm [3]. Denna nivå genererar en sekvens av vägpunkter, exempelvis de korsningar som ska passeras, som skickas till nästa nivå som kallas för *beteendenivån*.

Beteendenivån kombinerar vägpunkterna från ruttplaneringen med information om omgivningen såsom hinder, väglinjer, och vägens utseende, för att bestämma ett lämpligt körbeteende. Detta innebär vanligtvis att olika beteenden modelleras som tillstånd i en tillståndsmaskin med villkor som specificerar övergångar mellan dessa. [1] Olika tillstånd innebär generellt sett olika önskade körbeteenden. Det kan t.ex. vara önskat att bilen snabbt tar sig till referenslinjen ifall den är långt ifrån den, men då bilen är nära referenslinjen kan det vara önskvärt att använda en annan regulator som exempelvis prioriterar stabilitet eller ett lågt statiskt fel. Det är sedan dessa önskade körbeteenden som skickas vidare till den tredje nivån.

Den tredje nivån kallas *rörelseplanering* och dess uppgift är att utifrån det givna önskade körbeteendet generera en önskad körbana som uppfyller detta beteende. Flera olika metoder föreslås i [1] och en grafsökningsmetod, som anses mest lovande i vårt fall, undersöks noggrannare i avsnitt 2.3.

Körbanan från rörelseplaneringen skickas till sist till den fjärde nivån, den *lokala återkopplingsloopen* eller *regulatorn*, vars uppgift det är att följa den genererade banan. Den gör det genom att generera styrsignaler för hastighetsutslag och styrutslag. Även här föreslår [1] flera olika metoder vilka undersöks noggrannare i avsnitt 2.4, 2.5, 2.6 och 2.7.



Figur 2: *Beskrivande figur av beslutfattningshierarkin hos autonoma fordon. Det är främst de två sista stegen, rörelseplanering och den lokala återkopplingsloopen, som undersöks i denna studie. Tagen från [1].*

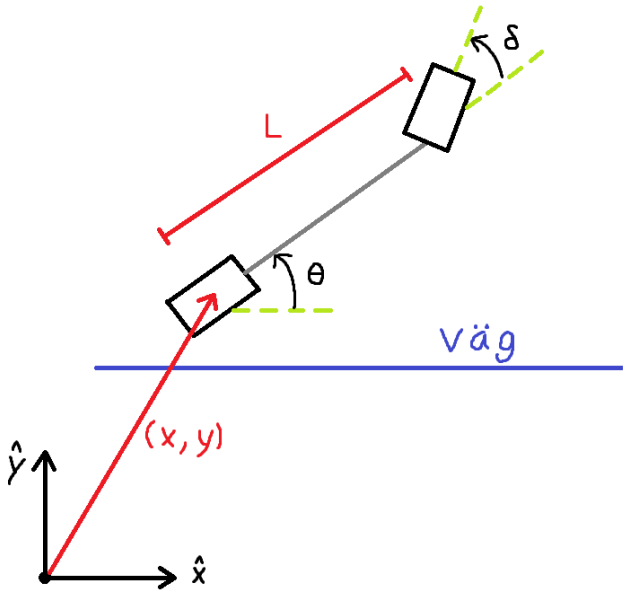
2.2 The Kinematic single-track model

Studien undersöker främst regulatorer för att behandla den fjärde nivån av beslutsfattning. Flera av studiens undersökta regulatorer använder sig av en kinematisk modell av en bil som kallas för *The kinematic single-track model*. Här modelleras bilen som ett fram- och ett bakhjul som är sammankopplade med ett fixt avstånd. Det främre hjulet kan även rotera kring en vertikal axel för att tillåta styrning. Bilen antas endast kunna röra sig i planet och dess position kan därför beskrivas med två koordinater. Här finns ett val i vilken punkt på bilen som ska användas, där typiska val är bakhjulet, framhjulet eller masscentrum. Valet av punkt ger aningen olika uttryck för tillståndsvariablerna men är främst en smaksak, i fortsättningen kommer bakhjulets position vara det som refereras till som bilens position (x, y) ifall inget annat nämnts. Värt att nämna är att denna modell använder ett koordinatsystem fixt i marken, till skillnad från vissa andra modeller som kan använda ett koordinatsystem fixt i bilen. [1]

De tillståndsvariabler som behövs för att beskriva bilen i The kinematic single-track model är

- x och y: bilens position i x- och y-led. I figuren är koordinatsystemet fixerat i vägen, alltså tangerar x-axeln vägens riktning.
- θ: bilens vinkel i förhållande till x-axeln, där positiva värden på θ innebär att bilen är riktad åt vänster medan negativa värden innebär att bilen är riktad åt höger.
- δ: framhjulets vinkel i förhållande till bilens rörelseriktning. Framhjulets vinkel i förhållande till vägen ges alltså av $\theta + \delta$.

Dessa illustreras i figur 3.



Figur 3: *Illustration av tillståndsvariablerna x, y, θ och δ i the kinematic single track model. The kinematic single track model är en generell kinematisk modell av en bil som kan användas med små modifikationer i många olika regulatorer.*

Utifrån detta kan en tillståndsbeskrivning för bilen fås som

$$\begin{aligned} \dot{x} &= v cos(\theta) \\ \dot{y} &= v sin(\theta) \\ \dot{\theta} &= \frac{v}{L} tan(\delta) \end{aligned} \tag{1}$$

där v är bilens fart (skalär) och L är avståndet mellan fram- och bakhjul. [1] Vanligtvis används även krav på hastigheten v och styrvinkeln δ sådana att $v \in [v_{min}, v_{max}]$ och $\delta \in [\delta_{min}, \delta_{max}]$ för att ta hänsyn till bilens fysikaliska begränsningar.

I denna modell finns det inget som kräver att hastighet och styrvinkel är kontinuerliga storheter, vilket dock är något som ofta är önskvärt ifall dessa storheter ska appliceras i en verklig bil. För att kräva detta kan modellen i 1 utökas med

$$\begin{aligned} \dot{\delta} &= v_\delta \\ \dot{v} &= a \end{aligned} \tag{2}$$

där a är bilens acceleration i rörelseriktningen och v_δ är hastigheten som framhjulet svänger med. Den utökade modellen har dock en högre dimension än den beskriven i 1 vilket kan göra det svårare att beräkna styrsignalerna. Som ovan begränsas ofta dessa storheter till inom ett intervall enligt $\dot{v} \in [\dot{v}_{min}, \dot{v}_{max}]$ och $v_\delta \in [\dot{\delta}_{min}, \dot{\delta}_{max}]$ för att garantera styrsignaler som kan realiseras av den hårdvara som finns.

För att använda en regulator som bygger på en kinematisk modell som denna behövs mätdata för att ta fram dessa tillstånd. Information som behöver fås från sensorer är alltså:

- Bilens position i x- och y-led.
- Bilens hastighet.
- Bilens vinkel relativt körbanan.
- Vinkeln för bilens framhjul relativt bilen.

Ifall den utökade modellen används behövs även sensorer som mäter accelerationen i bilens rörelseriktning och svänghastigheten för framhjulet. Det behöver även göras tester för att avgöra värden på v_{min}, v_{max}, $\dot{v}_{min}$, $\dot{v}_{max}$, δ_{min}, δ_{max}, $\dot{\delta}_{min}$ och $\dot{\delta}_{max}$. Dessa är dock konstanter och behöver bara mätas en gång för en specifik uppsättning hårdvara.

Utöver detta kan ytterligare information behövas som är specifik för olika regulatorer. Detta kommer att behandlas i avsnitten för respektive regulator.

2.3 Rörelseplanering

Som nämnts i avsnitt 2.1 beskrivs flera metoder i [1] för att generera en körbana i ett generellt vägnät. Detta problem blir dock kraftigt förenklat i fallet som studeras i denna studie eftersom hela vägnätets utseende är känt på förhand och då vägnätet är litet. Det gör att grafsökningsmetoder är väl lämpade, och speciellt en variant som använder handgjorda grafer över vägfiler. Till skillnad från många andra metoder kan grafsökningsmetoder hitta globala minima vilket gör dem till ett bra val då de går att använda. Denna metod innebär att användaren på förhand har specificerat den körbana som ska följas då bilen kör på ett visst vägsegment. På raka segment är detta oftast vägens mitt och i kurvor kan olika skarpa banor skapas som tar hänsyn till bilens svängradie och förväntade hastighet. Observera att körbanorna är skapade på förhand och inte kan ändras under körning för att anpassas till bilens faktiska hastighet. Med denna metod blir problemet med rörelseplanering likt problemet med ruttplanering, och precis som i det fallet kan rörelseplaneringen lösas med hjälp av A* eller Dijkstras algoritm. Om beräkningstiden orsakar problem kan metoder som viktad A* [2] användas för att göra en avvägning mellan optimalitet och beräkningstid.

2.4 PID-reglering

PID-reglering är en mycket vanlig reglerteknik som genererar en styrsignal $u(t)$ baserat på tre parametrar: K_P *proportionell*, K_I *integrerande*, K_D *deriverande*, samt en felterm $e(t)$. Där $e(t) = r(t) - y(t)$ är skillnaden mellan en önskad *referenssignal* $r(t)$ och det faktiska mätvärdet eller *utsignalen* $y(t)$. [4] Styrsignalen beräknas enligt ekvation 3.

$$u(t) = K_P e(t) + K_D \frac{\partial e}{\partial t} + K_I \int_T e(t)dt, \tag{3}$$

där T är ett valbart integrationsintervall som beskrivs senare.

PID-reglering är en allmän regulatorstruktur som kan anpassas för att användas för automatisk väglinjeföljning med en bil i ett vägnät. Då skulle $r(t)$ exempelvisvara mitten av körbanan, $y(t)$ vara det laterala avståndet mellan bil och väglinjer, LABV, och $e(t)$ vara bilens avvikelse från mittlinjen. Utsignalen $u(t)$ är då vinkeln för framhjulen.

Utmaningen med PID-reglering är att de tre PID-parametrarna måste ställas in för optimal reglering, det vill säga en konsekvent minimering av feltermen. Ett mått på detta är *root mean square error RMS* mätt N gånger under körning [4] enligt ekvation 4.

$$\text{minimera}\,\{RMS\} = \min \frac{1}{N} \sum_{i=0}^{i=N} e(t)_i^2. \tag{4}$$

Parametrarna kan ställas in med kända metoder beskrivna i avsnitt 2.4.1.

2.4.1 *Ställa in PID parametrar*

Det finns många olika algoritmer för att ställa in PID-parametrarna. Några populära sådana är: *WAF-Tune*, *Twiddle* och *Ziegler-Nichols*-algoritmen [4]. Då WAF-tune, enligt [4], ger bättre resultat för en bil än Ziegler-Nichols och Twiddle-algoritmen beskrivs enbart WAF-tune-algoritmen nedan. Bland annat var Twiddle-algoritmen svår att använda och Ziegler-Nichols-algoritmen gav ett oscillativt styrbeteende [4]. Vid inställning av PID-parametrarna i artikeln användes främst LABV som mätsignal, det vill säga målet var att minimera bilens laterala avstånd till en mittlinje. I artikeln körs dessa tester i en simulator och ingen hänsyn tas till om bilen är i en kurva eller raksträcka.

WAF-Tune [4]
Metoden går ut på att testa och ändra parametrar för PID-reglering av styrutslag iterativt tills bra parametervärden hittas. Antalet mätvärden, N, bör vara högt (ca 10,000).

1. Starta med att sätta gaspådraget lågt (t.ex 10%), samt $K_p = 0.1, K_I = 0, K_D = 0$.
2. Kör med satta parametervärden minst två gånger och beräkna RMS.
3. Öka K_P lite utan att ändra övriga parametrar och kör igen.
4. Om resultatet förbättras, fortsätt öka K_P.
5. Om resultatet inte förbättras, gå då tillbaka till föregående K_P och börja istället öka K_I på samma sätt som K_P ökades tidigare.
6. När även K_I nått det värde där det bästa resultatet observerades, börja då istället öka K_D på samma sätt.
7. Fortsätt tills även K_D har nått maximal förbättring på RMS.
8. Parametrarnas värde sägs nu tillhöra nuvarande gaspådrag.

9. Med samma parametervärden som tidigare gaspådrag, öka gaspådraget lite (5 procentenheter) och börja om från steg 2, fortsätt tills PID-regulatorn inte längre klarar av att hålla bilen på vägen utan att den kör av.

Detta ger en uppsättning parametrar för respektive gaspådrag som ska användas vid körning på det gaspådraget.

En fördel med PID-reglering är teknikens effektivitet trots dess enkelhet. En nackdel med tekniken är att den kräver att användaren hittar bra PID-parametrar, vilket kan vara svårt då de inte följer någon regel eller kriterium för optimalitet. [4].

2.4.2 *Diskret algoritm*

PID-regulatorn implementeras traditionellt sett i en elektrisk krets där storheter beter sig kontinuerligt. Idag är det dock vanligt att regulatorn implementeras digitalt vilket kräver en diskret algoritm som beskrivs i algoritm 1 enligt [5].

Algorithm 1: Diskret implementation av PID-regulatorn [5]

$previous_error \leftarrow 0$
$integral \leftarrow 0$
while *True* **do**
  $error \leftarrow reference - actual_position$
  $integral \leftarrow integral + error * dt$
  $derivative \leftarrow \frac{error - previous_error}{dt}$
  $output \leftarrow K_p * error + K_I * integral + K_d * derivative$
  $previous_error \leftarrow error$
  $wait(dt)$
end

I koden är *error* $e(t)$, *output* $u(t)$, *dt* tiden T (nämnd tidigare som integrationsintervall) mellan varje iteration av loopen. Enligt [5] bör T vara större än uppdateringsperioden för systemet som PID-regulatorn reglerar. I taxibilens fall syftar det på uppdateringsperioden för de sensorer som mäter till exempel taxibilens LABV eller VVB men även PWM-perioden hos bilens styrservo. Sätts T lägre än dessa begränsande perioder fås ingen ytterligare effekt.

2.5 Pure-pursuit

Ett alternativ eller komplement till PID-reglering är väglinjeföljning med *pure-pursuit-tekniken* som är en kinematisk styrteknik för fordon. Denna metod använder en *single-track model* för att beskriva bilen enligt avsnitt 2.2.

Metoden går enligt [6] ut på att mäta krökningen av en cirkelbåge som går från bilens bakre axel till en *målpunkt* (g_x, g_y) längre fram någonstans på bilens önskade färdväg. Med *look-ahead-avståndet* l_d från bilens bakre axel till målpunkten, krökningsradien R, vinkeln α mellan bilens önskade riktning och bilens nuvarande riktning kan bilens styrutslag δ enligt [6] beräknas med ekvationer 5 till 9. Ovan nämnda storheter illustreras i figur 4.

$$\frac{l_d}{\sin 2\alpha} = \frac{R}{\sin \frac{\pi}{2} - \alpha} \tag{5}$$

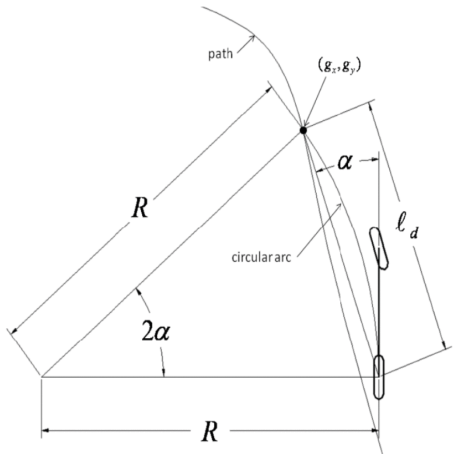
$$\frac{l_d}{2\sin(\alpha)\cos(\alpha)} = \frac{R}{\cos(\alpha)} \tag{6}$$

$$\frac{l_d}{\sin(\alpha)} = 2R \tag{7}$$

$$\kappa = \frac{2\sin(\alpha)}{l_d} \tag{8}$$

$$\delta = \tan^{-1}(\kappa L) \tag{9}$$

Där κ beskriver kurvans krökning och L är avståndet mellan hjulaxlarna.



Figur 4: *Skiss över de storheter som används i ekvationerna för pure-pursuit ovan. Här visas även ett användningsfall av kinematic single-track model vilken beskrevs i 2.2. Tagen från* [7].

2.5.1 *Val av målpunkt*

Vid användning av pure-pursuit-tekniken är val av målpunkt centralt. Enligt [8] finns det tre huvudsakliga problem som kan uppstå vid detta val:

1. Målpunkten väljs för nära bilen vilket ger ett oscillativt styrbeteende med stora väg-bil-vinklar. Pure-pursuit tekniken tar endast hänsyn till bilens position.
2. Målpunkter väljs för långt bort från bilen vilket ger en långsam reglering.
3. Författarnas *cutting corner problem*: då en målpunkt väljs tvärs över en kurva och bilen kör raka vägen till målpunkten istället för att följa vägens krökning, och alltså då kör av vägen.

Problem 1 och 2 uppstår då bilen ska ansluta till en närliggande referenslinje, medan 3 uppstår då bilen befinner sig i en kurva. Ett fjärde problem beskrivs av [1] som menar att hänsyn måste tas till bilens hastighet, då en hög hastighet med ett litet l_d ger ett oscillativt körbeteende. [1] föreslår därför att l_d skalas med hastigheten.

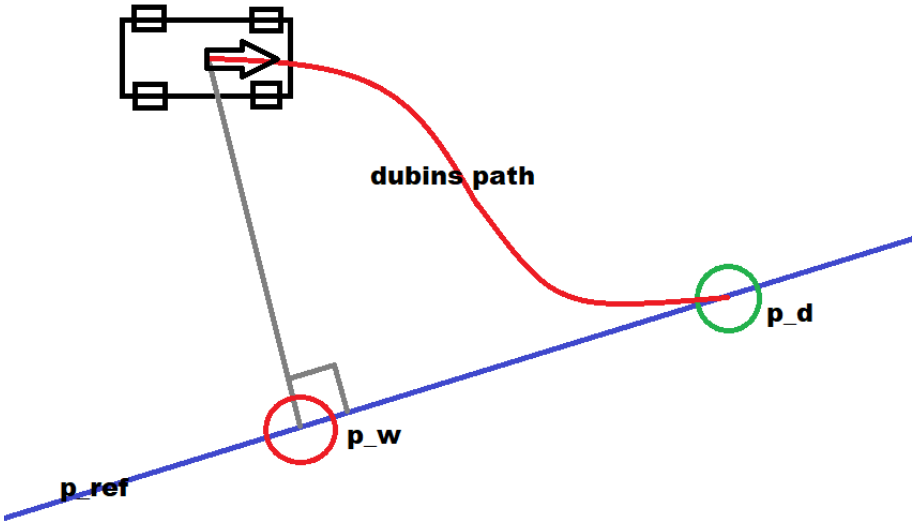
Problem 1, 2 och 3 kan minskas med en tvådelad metod enligt nedan [8]:

1. VAL AV MÅLPUNKT LÄNGS RAKSTRÄCKA Om bilen befinner sig nära referenslinjen och om väglinjen inte är en kurva kan denna metod användas. Metoden utnyttjar *Dubins path* [9] som är den kortaste färdvägen mellan två punkter och två orienteringar för ett fordon som har en minsta svängradie. Dubins path beräknas mellan bilen och väglinjen. I denna metod ska bilen inte följa Dubins path, utan slutet på Dubins path används som målpunkt för pure-pursuit, vilket ger ett mjukt styrbeteende enligt [8]. Målpunkten för Dubins path väljs alltså på referenslinjen, så nära bilen som möjligt, men begränsat av bilens svängradie. Algoritm 2 och figur 5 beskriver hur målpunkter väljs utifrån Dubins path.

Algorithm 2: Selection of Look-ahead Point p_d [8]

Input: $p_{ref}, p_v, r_{min}(v)$
Output: p_d
for $i = 0$ *to* $p_{ref}(i)$ **do**
  $p_w \leftarrow argmin_{p_{ref}(i)} ||p_{ref}(i) - p_v||$
  $i \leftarrow i + 1$
end
i_{pw} = index of p_w
for $i = i_{pw}$ *to* $p_{ref}(i)$ **do**
  if $r_{min}(v_{cmd}) \leq ||p_w - p_{ref}(i)||$ **then**
    $len_{dubin} \leftarrow DubinsPathLength(p_v, r_{min}(v_{cmd}), p_{ref}(i))$
    $p_d \leftarrow argmin_{p_{ref}(i)}(len_{dubin})$
  end
  $i \leftarrow i + 1$
end

Algoritm 2 hittar först den punkt p_w på väglinjen (beskriven av p_{ref}) som är närmst bilen. Därefter itererar den över alla punkter $p_{ref}(i)$ med index högre än i_{pw} och beräknar Dubins path från bilen till varje $p_{ref}(i)$. Den $p_{ref}(i)$ punkt som ger kortast längd på Dubins path blir p_d, det vill säga den sökta målpunkten.



Figur 5: *Skiss av hur Dubins path ger målpunkten p_d baserad på [8]. p_d kan inte väljas närmare bilen på grund av kravet på minimal svängradie.*

2. VAL AV MÅLPUNKT I KURVA Som nämnts ovan har pure-pursuit tekniken problem med att navigera i kurvor. Enligt [8] kan detta förbättras genom att välja en målpunkt bortanför referenslinjen, istället för på den, när bilen kör i en kurva. Det utgås från att bilen redan ligger på väglinjen och är riktad parallellt med den.

Låt p_l vara den sökta målpunkten och p_d en punkt längre fram på väglinjen. Observera att det inte framgår tydligt av [8] hur p_d egentligen väljs i detta fall. Enligt [8] beräknas p_l enligt:

$$x_{p_l} = x_{p_d} + \tau|\vec{p_{dl}}|cos(\theta_{\vec{p_{dl}}}), \quad y_{p_l} = y_{p_d} + \tau|\vec{p_{dl}}|sin(\theta_{\vec{p_{dl}}}) \tag{10}$$

där

$$|\vec{p_{dl}}| = |\vec{p_{wd}}|tan(|\theta|), \tag{11}$$

$$\theta_{\vec{p_{dl}}} = \theta_{\vec{p_{wd}}} - sign(\theta)\frac{\pi}{2}, \quad \theta = \theta_{\vec{p_{wl}}} - \theta_{\vec{p_{wd}}}, \tag{12}$$

samt

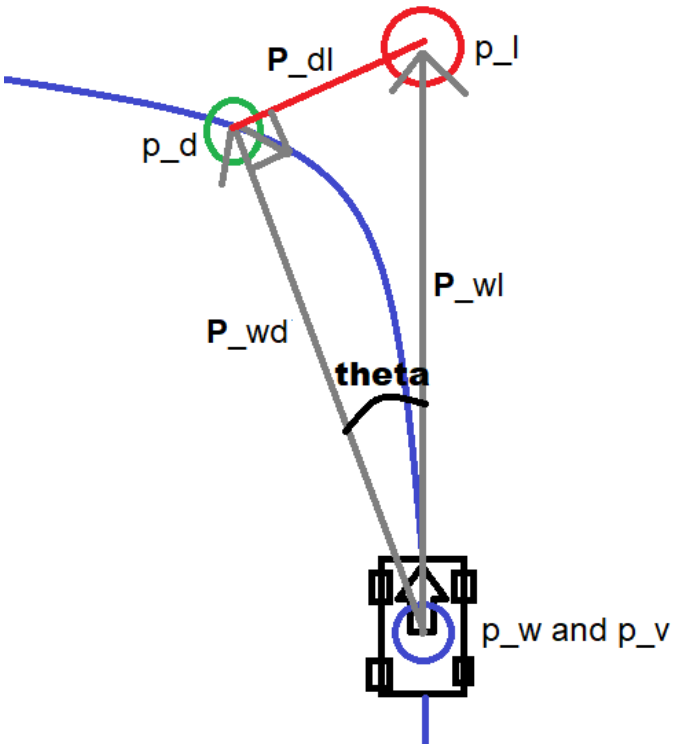
$$\tau = (1-\alpha)\beta. \tag{13}$$

$\alpha, \beta \in [0, 1]$ fås enligt,

$$\alpha = \begin{cases} \frac{||p_v - p_w||}{\alpha_{max}} & ||p_v - p_w|| < \alpha_{max} \\ 1 & ||p_v - p_w|| \geq \alpha_{max} \end{cases}, \tag{14}$$

$$\beta = \begin{cases} 0 & \text{if } cur_{p_w} \geq cur_{p_d} \\ \frac{cur_{p_d} - cur_{p_w}}{\beta_{max}} & \text{else if } cur_{p_d} - cur_{p_w} < \beta_{max} \\ 1 & \text{else if } cur_{p_d} - cur_{p_w} \geq \beta_{max} \end{cases}, \tag{15}$$

där *cur* syftar på krökningen i respektive punkter. α_{max} och β_{max} är parametrar som relaterar till avståndet till p_d och krökningen på kurvan. I [8] fås optimal styrning för $\alpha_{max} = 5$ och $\beta_{max} = 0.06$. Se figur 6 för en skiss över metodens delar.



Figur 6: *Skiss av hur en målpunkt väljs utanför väglinjen baserad på [8]. I detta fall ligger p_w och p_v i samma punkt. Om p_d väljs som målpunkt framgår det av figuren att cutting-corner problem kan uppstå, varför man istället väljer p_l som målpunkt.*

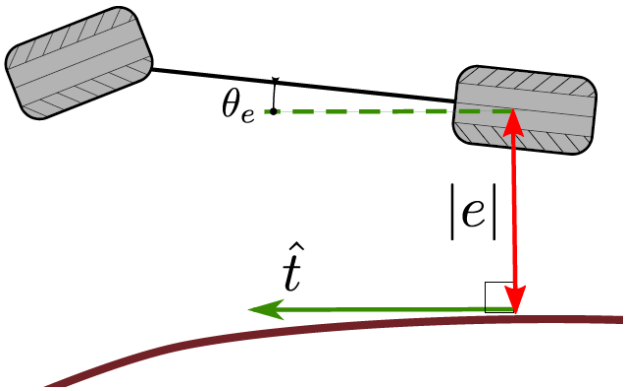
2.6 Viktad styrning med pure-pursuit och PID-reglering

Enligt [6] kan en kombinerad styrteknik byggas med hjälp av viktning av pure-pursuit och PID-reglering. I artikeln används en PID-regulator för att reglera bilens vinkel relativ vägen, motsvarande θ i figur 3, och pure-pursuit för att reglera LABV. Artikeln visar på en tydlig förbättring med en kombinerad styrteknik jämfört med då endast en PID-regulator används.

2.7 Rear-/front wheel position based feedback

Rear wheel position based feedback och *front wheel position based feedback*, härefter benämnda *RW* respektive *FW*, är två kinematiska reglermetoder för väglinjeföljning som fungerar på liknande sätt. För enkelhetens skull används resonemang och formler för RW, och i slutet av avsnittet förs en diskussion om de huvudsakliga skillnaderna mellan RW och FW.

Idén med RW är att styra bilen så att den hastighetskomponent hos bakhjulet som är vinkelrät mot referenslinjen blir proportionell mot avståndet till referenslinjen. Enligt [1] ger RW bra linjeföljning även i kurvor och ett exponentiellt avtagande fel. För att göra detta används tillståndsvariablerna s, e och θ_e, där $s(t)$ är bakhjulets position, $e(t)$ är den laterala avvikelsen från referenslinjen (mittlinjen) och $\theta_e(t)$ är skillnaden mellan framhjulens vinkel relativt bilen och referenslinjens tangentvektor. Dessa är illustrerade i figur 7.



Figur 7: *Illustration av tillståndsvariablerna vid rear wheel position based feedback. $e > 0$ innebär att bilen befinner sig t.h. om referenslinjen och $e < 0$ innebär att bilen befinner sig t.v. om referenslinjen. Tagen från [1].*

Från [1] fås att tillståndsvariablerna definieras enligt ekvation 16 - 20. Den första tillståndsvariabeln ges av

$$s(t) = \arg\min_{\gamma} \|(x_r(t),\ y_r(t)) - (x_{ref}(\gamma),\ y_{ref}(\gamma))\|. \tag{16}$$

Tangentvektorn till kurvan i $s(t)$ ges av

$$\hat{t} = \frac{\left(\left.\frac{\partial x_{ref}}{\partial s}\right|_{s(t)}, \left.\frac{\partial y_{ref}}{\partial s}\right|_{s(t)} \right)}{\left\| \left(\frac{\partial x_{ref}(s(t))}{\partial s}, \frac{\partial y_{ref}(s(t))}{\partial s} \right) \right\|} \tag{17}$$

och följningsfelvektorn ges av

$$d(t) = (x_r(t),\ y_r(t)) - (x_{ref}(s(t)),\ y_{ref}(s(t))). \tag{18}$$

Det laterala felet, som är den andra av tillståndsvariablena, ges av kryssprodukten av $\hat{t}$ och $d(t)$. Alltså fås

$$e(t) = d_x(t)\hat{t}_y(t) - d_y(t)\hat{t}_x(t). \tag{19}$$

Slutligen ges den sista tillståndsvariabeln av

$$\theta_e(t) = \theta - arctan2\left(\frac{\partial y_{ref}(s(t))}{\partial s},\ \frac{\partial x_{ref}(s(t))}{\partial s}\right) \tag{20}$$

där θ, som i avsnitt 2.2, är bilens vinkel i förhållande till vägen.

Med detta ges bilen på tillståndsform av

$$\begin{aligned} \dot{s} &= \frac{vcos(\theta_e)}{1-\kappa(s)e} \\ \dot{e} &= vsin(\theta_e) \\ \dot{\theta}_e &= \dot{\theta} - \frac{v\kappa(s)cos(\theta_e)}{1-\kappa(s)e} \end{aligned} \tag{21}$$

där $\kappa(s)$ är referenslinjens krökning i s.

Bilens styrutslag δ fås, från avsnitt 2.2, som

$$\delta = arctan\left(\frac{L\dot{\theta}}{v}\right). \tag{22}$$

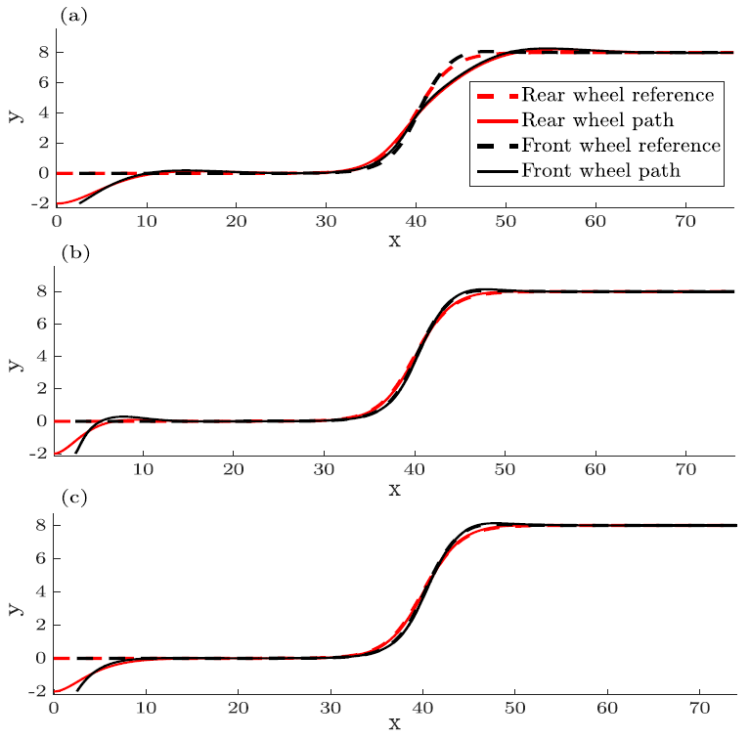
Resultatet i [1] är att följande styrlag bör användas:

$$\dot{\theta} = \frac{v\kappa(s)cos(\theta_e}{1-\kappa(s)e} - (k_\theta|v|)\,\theta_e - \left(k_e v\frac{sin(\theta_e)}{\theta_e}\right)e \tag{23}$$

där k_θ och k_e är två parametrar. Den ger ett exponentiallt avtagande fel oberoende av bilens hastighet, sålänge bilen inte står stilla.

Skillnaden mellan RW och FW är att RW använder sig av bakhjulens position medan FW använder sig av framhjulens position. Ekvationerna för tillståndsvariablerna blir därmed något annorlunda och metoderna har även något olika stabilitetskrav. FW har fler och striktare krav än RW men klarar av att följa referenslinjer av klass $\mathcal{C}^1$, medan RW kräver referenslinjer av klass $\mathcal{C}^2$. Som beskrivet ovan behöver RW två parametrar i sin styrlag men i FW kan förenklingar göras så att endast en parameter behövs.

I [1] görs en simulering där RW, FW och pure-pursuit (utan tillägget att välja målpunkt bortom referenslinjen) jämförs med varandra då en bil utför ett filbyte, resultatet kan ses i figur 8. Simuleringen med pure-pursuit uppvisar cutting corner problem men både simuleringen med RW och simuleringen med FW uppvisar god linjeföljning. RW ger små översläng för framhjulen, som inte ses för FW. RW konvergerar mot referenslinjen något snabbare än vad FW gör.



Figur 8: *Simulering av ett filbyte med hjälp av tre olika regulatorer. (a) Pure-pursuit med look-ahead-avstånd* $l_d = 5$. *(b) RW med* $k_e = 0.25$ *och* $k_\delta = 0.75$. *(c) FW med* $k = 0.5$. *I samtliga fall är hastigheten* $v = 1$ *konstant.*

3 SAMMANFATTNING, DISKUSSION OCH SLUTSATSER

3.1 Sammanfattning

Studien har undersökt tre olika tekniker för att få en bil att köra i ett vägnät: PID-reglering, pure-pursuit och rear-/front wheel position base feedback. Som framgår i studien tyder mycket på att en kombination av flera tekniker är optimalt beroende på vilket vägsegment bilen befinner sig i.

Studien har även kort undersökt beslutstagande och ruttplanering för en autonom bil.

3.2 Diskussion

De tre undersökta teknikerna visar stor potential för att kunna ge en effektiv väglinjeföljning. De är alla relativt lätta att implementera och enligt [1] är en fördel med både pure-pursuit och RW/FW att de kräver lite beräkningskraft. Från algoritm 1 inses att även PID kräver lite beräkningskraft. Metoderna kräver även ungefär samma information om bilen. Det är ännu inte klart hur bra information bilens sensormodul kommer att kunna ge och metodernas effektivitet beror också på bilens begränsningar. Dessa begränsningar är till exempel bilens vägfäste, acceleration, svängradie och styrservohastighet vilket måste testas innan något kan sägas om detta. Vad som ges direkt från bilchassits datablad [10] är att styrservots utsignal kan ändras med en frekvens på maximalt 50 Hz eftersom periodtiden mellan varje PWM-puls är 20 ms. Det föreslår att en informationsfrekvens (till exempel frekvens på bildbehandlingen) över 50Hz kan vara överflödig. Periodtiden kan också hjälpa till vid beslutet om ett lämpligt integrationsintervall T vid PID-regleringen.

I avsnitt 2.4 uppmärksammades det att WAF-Tune algoritmen i [4] inte tog hänsyn till om ifall bilen är i en kurva eller raksträcka. Detta föreslår att WAF-Tune eventuellt kan användas för att hitta separata uppsättningar med parametrar beroende på om bilen är i en kurva eller på en raksträcka.

3.3 Slutsats

De tre problem som presenterades i studiens frågeställning, i kapitel 1.1, besvaras här:

1. För att få bilen att köra inom väglinjerna bör en PID-regulator kombinerad med varianter av pure-pursuit ge ett bra resultat och en enkel implementation.
2. För att hantera olika typer av vägsegment kan olika varianter av pure-pursuit-metoden användas när bilen befinner sig i olika vägsegment. Alternativt kan rear-/front wheel position based feedback användas för att förbättra regulatorns prestanda i svängar.
3. För att använda den föreslagna lösningen behövs kunskap om var vägens mittlinje är och om var bilen befinner sig i förhållande till mittlinjen. Referenslinjen måste också vara detaljerad nog för att kunna beräkna vägens krökning längre fram.

REFERENSER

[1] B. e. a. Paden, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, 2016.

[2] A* search algorithm. [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm

[3] Dijkstra's algorithm. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[4] W. A. Farag, "Complex trajectory tracking using pid control for autonomous driving," *International Journal of Intelligent Transportation Systems Research*, vol. 18, pp. 356–366, 2020.

[5] codeproject.com Icady. (2009) Pid process control, a "cruise control" example. [Online]. Available: https://www.codeproject.com/Articles/36459/PID-process-control-a-Cruise-Control-example

[6] M. K. Diab, H. H. Ammar, and R. E. Shalaby, "Self-driving car lane-keeping assist using pid and pure pursuit control," in *2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT)*, 2020, pp. 1–6.

[7] J. M.Snider, "Automatic steering methods for autonomous automobile path tracking," 2009.

[8] J. Ahn, S. Shin, M. Kim, and J. Park, "Accurate path tracking by adjusting look-ahead point in pure pursuit method," *International Journal of Automotive Technology*, vol. 22, pp. 119–129, 2021.

[9] D. Yang, D. Li, and H. Sun, "2d dubins path in environments with obstacle," *Mathematical Problems in Engineering*, 2013.

[10] Dacar taxi och tävlingsbil. [Online]. Available: http://www.da.isy.liu.se/vanheden/pdf/DACar.pdf

F FÖRSTUDIE - SENSOR

Förstudie - Sensor

Sebastian Andersson
Charbel Youssef

7 juni 2021

Version 1.1

Status

Granskad		
Godkänd		

INNEHÅLL

1 Inledning 1
1.1 *Bakgrund* . . . 1
1.2 *Syfte* . . . 1
2 Frågeställning 1
2.1 *Avgränsningar* . . . 1
3 Metod 2
4 Kunskapsbas 3
4.1 *Bildprocessering* . . . 3
4.2 *Detektion av vägkanter* . . . 5
5 Algoritmer 9
5.1 *Houghtransformbaserade algoritm* . . . 10
5.2 *Fågelperspektivsbaserade algoritm* . . . 10
6 Färdiga paket för implementation av algoritmer 11
7 Diskussion och slutsatser 11

DOKUMENTHISTORIK

Version	Datum	Utförda förändringar	Utförda av	Granskad
0.1	2021-02-26	Första utkast	CY och SA	-
0.2	2021-03-28	Andra utkast	CY och SA	-
1.0	2021-04-07	Färdigställt	CY och SA	-
1.1	2021-04-18	Förtydligat sådant som var otydligt	CY och SA	-

1 INLEDNING

Detta kapitel ger en översiktlig bild av projektet och beskriver syftet med denna studie.

1.1 Bakgrund

Projektet går ut på att skapa en autonomt körande taxibil som kan navigera i en godtycklig bana. Taxibilen är kopplad trådlöst med en användardator och ska kunna köras i tre lägen: manuellt, semi-autonomt och helautonomt. Manuellt körläge innebär att bilen kan ta emot direkta styrinstruktioner från användardatorn. I de autonoma lägena är vägbanan väldefinierad och bilen ska kunna ta emot navigationsinstruktioner från användardatorn baserat på vägnätet. Navigationsinstruktionerna kan vara en sekvens av förprogrammerade delar av köruppdraget i det semi-autonoma läget eller ett fullständigt köruppdrag i det helautonoma läget. Under körning ska eventuella hinder upptäckas samt destinationer, som är markerade med stopplinjer, ska detekteras. Kraven för projektet ska uppfyllas enligt kravspecifikationen. Taxibilens teknik, för att kunna upptäcka omgivningen i de autonoma lägena, är baserad på datorseende. Datorseendet bygger på att använda en kamera som sitter i bilens front. Kameran ska fånga vägnätet och systemet ska tolka och omvandla bilderna till data som kan hanteras via de ingående modulerna.

1.2 Syfte

Syftet med denna studie är att studera de olika tekniker som används i maskinseendet för att navigera i ett vägnät. Detta utförs genom att jämföra de olika algoritmer och programpaket som finns och vilka beräkningskrafter som behövs. Dessa jämförelser ska ligga till grund för slutsatser om vilken teknik som passar en autonomt körande taxibil bäst för att snabbt och noggrant navigera och utföra ett köruppdrag.

2 FRÅGESTÄLLNING

Studiens frågeställning kan sammanfattas i följande punkter:

- Vilka algoritmer finns för att möjliggöra navigeringen genom ett vägnät med hjälp av en kamera?
- Hur tunga är dessa algoritmer när det kommer till beräkningskraft?
- Finns det färdiga paket som kan underlätta implementationen av dessa algoritmer?
- Vilken upplösning krävs för att fordonet med säkerhet ska kunna navigera sig genom vägnätet?

2.1 Avgränsningar

Avgränsningar i systemet är att vägbanan är förutbestämd och den kommer att innefatta raka sträckor, cirkulära svängar med en viss radie och T-korsningar. Vägkanterna är svartmarkerade och själva vägen och omgivningen är vit. Allt detta lägger en grund till vilken algoritm som används i bildbehandling samt vägdetektion.

Avgränsningar i studien är att inga tester utförs. Det innebär att det endast kommer information om olika metoder som används inom bildbehandlig och olika algoritmer för att detektera kanter.

3 METOD

Metoden går främst ut på att studera relevanta källor som handlar om bildbehandlingen och de olika algoritmer som används så att ett autonomt fordon kan detektera en väg med hjälp av en kamera. Olika algoritmer undersöks och utifrån resultatet och jämförelserna dras slutsatser om vilken teknik som passar projektet bäst under de tillgängliga förutsättningarna.

4 KUNSKAPSBAS

Detta kapitel behandlar den teoretiska delen som ligger bakom diskussionen och de slutsatser som dras i kapitel 7. I *Bildprocessering* 4.1 och *Detektion av vägkanter* 4.2 presenteras de olika begrepp, aspekter och metoder som ingår i de algoritmer som beskrivs i *algoritmer* 5.

4.1 Bildprocessering

Vad som behövs för att kunna hitta vägkanterna är intensitetsskillnader i bilden. Intensitetskillnaden mellan vägen och vägkanten är ganska hög, eftersom det ska vara enkelt att kunna urskilja dem. För att inte annat runt omkring ska ge intensitetsutslag behöver bilden processas för att filtrera bort sådan oönskad information som i sin tur kommer att öka prestandan och pricksäkerheten på algoritmen. Detta kan underlättas genom att ändra vilken färgrymd bilden representeras i. Nedan nämns olika färgrymder och metoder för att processa bilden (minska bruset) och få ut intensitetskillnader.

4.1.1 *Gråskala konvertering*

En bild i gråskala är en bild med endast en färgkanal. Det enda en sådan bild visar är intensiteten på vardera pixel [1]. Det innebär att en konvertering av bilden till en gråskala kommer att resultera i en binärbild med färgerna svart och vitt. För att få fram dessa pixelvärden tas värdena för den röda (R), gröna (G) och blåa (B) pixeln och summeras till ett värde genom,

$$p = R \cdot 0.3 + G \cdot 0.59 + B \cdot 0.11, \tag{1}$$

som då ger det nya pixelvärdet, p [2].

4.1.2 *HLS-konvertering*

HLS, som står för hue, lightness and saturation (nyans, ljusstyrka och mättnad på svenska), är ett alternativt sätt att representera RGB-färgrummet (röd, grön och blå) . Med denna representation kan specifika färger enklare isoleras genom att applicera olika trösklar på varje färgkanal (H, L, S) [1]. L-kanalen anger även hur mycket svart respektive vitt som finns i pixeln. Konvertering från RGB- till HLS-rummet utförs genom följande,

$$L = \frac{V_{max} + V_{min}}{2}, \tag{2}$$

$$S = \begin{cases} \frac{V_{max}-V_{min}}{V_{max}+V_{min}} & \text{om } L < 0.5 \\ \frac{V_{max}-V_{min}}{2-(V_{max}+V_{min})} & \text{om } L \geq 0.5 \end{cases}, \tag{3}$$

$$H = \begin{cases} \frac{G-B}{V_{max}-V_{min}} & \text{om } V_{max} = R \\ 120 + 60(\frac{B-R}{V_{max}-V_{min}}) & \text{om } V_{max} = G \\ 240 + 60(\frac{R-G}{V_{max}-V_{min}}) & \text{om } V_{max} = B \end{cases}, \tag{4}$$

där,

$$V_{max} = max(R, G, B), \quad V_{min} = min(R, G, B), \tag{5}$$

vilket ger största respektive minsta värdet från RGB-värdena. Med detta ges det nya värdet på respektive pixel [1].

4.1.3 *LAB-konvertering*

LAB, där L står för lightness (ljusstyrka) och A och B är färgkanaler, är ett alternativt färgrum. Med detta färgrum ökar färgernas kontrast vilket gör att det kan bli enklare att filtrera bort sådant som inte är av intresse [1]. Kanalerna beräknas genom följande,

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ B \\ G \end{bmatrix}, \tag{6}$$

$$L = \begin{cases} 116Y^{1/3} - 16 & \text{om } Y > 0.008856, \\ 903.3Y & \text{om } Y \leq 0.008856 \end{cases}, \tag{7}$$

$$A = 500(f(\frac{X}{0.950456}) - f(Y)) + \delta, \tag{8}$$

$$B = 200(f(Y) - f(\frac{Z}{1.088754})) + \delta, \tag{9}$$

där

$$f(t) = \begin{cases} t^{1/3} & \text{om} \quad t > 0.008856, \\ 7.787t + \frac{16}{116} & \text{om} \quad t \leq 0.008856 \end{cases}, \quad \delta = \begin{cases} 128 & \text{för åtta-bitars bilder,} \\ 0 & \text{för flytals bilder} \end{cases} \tag{10}$$

vilket då ger värdena på respektive kanal (R, G och B är RGB-värdena) [1].

4.1.4 *Gaussisk lågpass-filtrering*

Gaussisk lågpass-filtrering (Gaussian blur på engelska) är en brusreducerande metod som jämnar till bilden [3]. Detta utförs genom att skapa oskärpa i den bilden som genereras genom att falta orginalbilden med en matris som har pixelvärden utifrån Gaussiska funktionen,

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{11}$$

där origo är i mitten av matrisen och σ är standardavvikelsen (avgör hur snabbt pixelvärdena faller med avståndet från origo).

4.1.5 *Korrigering av kameraförvrängning*

Linsförvrängningar som uppstår vid användning av en kamera är fel som kan göra stor skada för bildbehandling, även om det inte är synligt för det mänskliga ögat. Den vanligaste typen av förvrängningen är *barrel distortion* som kännetecknas av att att raka linjer (i verkligheten) blir böjda utifrån kamerans perspektiv. Dessa förvrängningar kan korrigeras genom att ta matcha 2D-bilder (från kameran) till ett känt 3D-mönster (exempel ett schackbräde) [4]. När en bra uppsättning av parametrar har funnits kan denna utnyttjas på resterande bilder för bildbehandlingen.

4.1.6 *Canny-kantdetektion*

Canny-kantdetektion (Canny edge detection på engelska) används för att hitta stora förändringar i pixlars intensitet. Denna metod att hitta kanter har bra prestanda samt producerar en bild med bra distinktion på kanter [1]. Den funkar genom att först applicera ett gaussiskt lågpassfilter på bilden. Sedan beräknas gradienten på bilden med hjälp av en *Sobel-operator* som är en diskret differential metod som approximerar bildens gradient baserat på intensiteten [5]. Efter det jämförs gradienten på vardera pixel utifrån två trösklar som har angivits; är den under den lägre tröskelen anses den som för låg och tas bort. De som är över den högre tröskeln anses vara starka och tas med i avbildningen. Resten (de som ligger mellan de två trösklarna) anses vara svaga. De svaga pixlarna tas endast med om de är kopplade till en stark pixel, vilket innebär att de måste vara i närheten av en sådan samt vara i samma riktning (så de bildar en kedja av pixlar), annars tas de bort [6]. Detta producerar höga utslag där det är en drastisk ändring i orginalbilden. Med detta kommer det att fås skarpa linjer där kanter har hittats.

4.2 Detektion av vägkanter

I denna del behandlas de matematiskt baserade metoder som används för att ge numeriska värden som motsvarar kantpixlarna i den förbehandlade bilden. Dessa värden representerar den detekterade vägbanan.

4.2.1 *Houghtransformering*

Syftet med Houghtransformation är att hitta en ekvation som representerar en rät linje i bilden. Bildrummet och Houghrummet defineras enligt följande:

- *Bildrummet*: Ett koordinatsystem som representeras med (x, y)-planet.
- *Houghrummet*: Ett koordinatsystem med (ρ, θ)-planet.

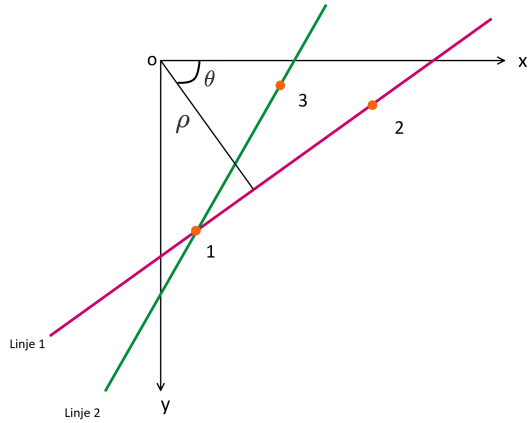
En linje i (x, y)-planet kan representeras med följande samband $y = kx + m$ där k är lutningen med avseende på $x-$axeln och m är skärningen med $y-$axeln. För att anpassa sambandet så att det kan inkludera vertikala linjer används polära representationen av linjeekvationen:

$$x \cos\theta + y \sin\theta = \rho, \tag{12}$$

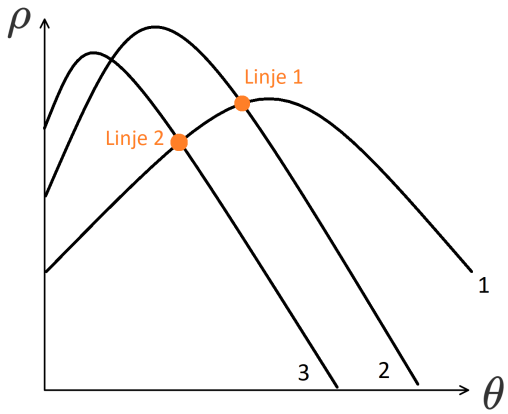
där ρ är avståndet mellan linjen och origo, θ vinkeln som bildas mellan linjen och x-axeln och är begränsad inom intervallet $[0, \pi)$. Oändligt många raka linjer går genom varje punkt (x_i, y_i) i (x, y)-planet, där i är en index till en godtycklig punkt. En punkt (x_0, y_0) i (x, y)-planet avbildas i (ρ, θ)-planet som en sinusoidalkurva enligt:

$$x_0 \cos\theta + y_0 \sin\theta = \rho. \tag{13}$$

Varje punkt (ρ, θ) som tillhör kurvan (13) motsvarar en linje genom (x_o, y_o) i bildrummet. Detta betyder att alla punkter som ligger på samma linje i (x, y)-planet har avbildade sinusoidalkurvor i Houghrummet som skärs i en gemensam punkt. Detta illustreras i figur (1) och (2) nedan:



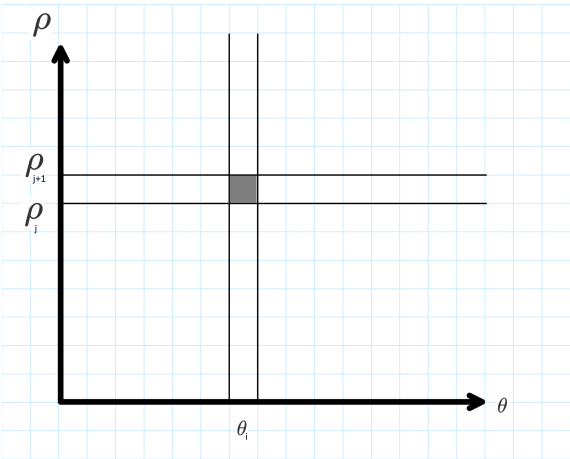
Figur 1: *Skiss över bildrummet. Varje linje kan skrivas i polär form med avseende på ρ och θ. Alla punkter som tillhör samma linje avbildas i Houghrummet av sinusoidalkurvor som skärs i en gemensam punkt.*



Figur 2: *Skiss över Houghrummet. Punkterna Linje1 och Linje 2 motsvarar två olika linjer i bildrummet. Kurvurna 1, 2 och 3 motsvarar däremot tre olika punkter i bildrummet.*

Baserat på denna avbildning är det möjligt att definiera *kollinearitet*, d.v.s. en mängd punkter som ligger på samma linje. Detta utgörs genom *ackumulationen* vilket innebär att sampla värdena (ρ, θ) för de motsvarande detekterade kantpunkterna i (x,y)-planet. I denna process är θ samplad som värde och ρ som ett intervall, vilket resulterar i en

matris av ackumulationer, se figur (3).



Figur 3: *Skiss över ackumulationer. Det visar de samplade värdena (ρ, θ) för de motsvarande detekterade kantpunkterna i (x,y)-planet. θ är samplad som värde och ρ som ett intervall.*

Varje cell i denna matris har ett nollställt värde från början. Varje gång den sinusoidala kurvan i (ρ, θ)-planet skär en punkt som är representerad med en cell i denna matris ökas cellens värde med ett. Som resultat fås en matris med alla ackumulationer och dess värden är lika med antalet punkterna som tillhör motsvarande kantlinje. På så sätt kan man hitta alla punkter som tillhör en rak linje [7].

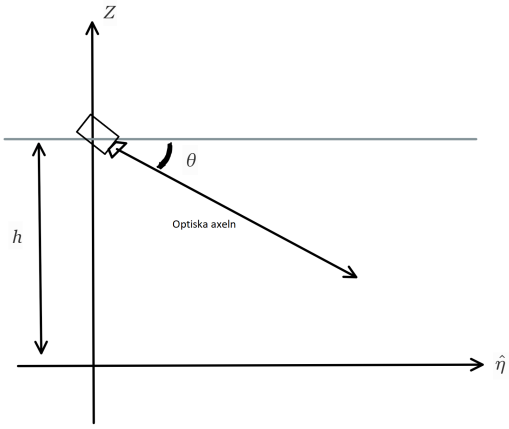
4.2.2 *Fågelperspektiv-transformering*

Fågelperspektivet [8] är en vy över objekten ur en synvinkel som är riktad nedåt. Detta kan användas i vägkantdetektion så att vägkanterna blir relativt parallella i förhållande till varandra. Perspektivet ändras så att linjerna blir vertikala. Detta gör det enklare att anpassa ett polynom för vägkanterna och mäta vägböjning. Fågelperspektivet använder *inverterad perspektivavbildning* (IPM: inverse perspective mapping) baserad på bilder med en färgkanal (exempelvis gråskala).

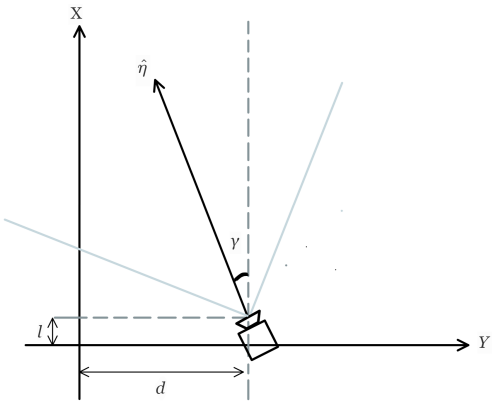
Omgivningens koordinatsystem är framställt beroende på den verkliga vägsituationen medan kamerans koordinatsystem framställs med origo placerad i kameran och ett tvådimensionellt bildkoordinatsystem. Bilden som fångas av kameran är projektionen som avbildas genom att projicera tredimensionella Euklidiska rummet till tvådimensionella Euklidiska rummet engligt följande:

- $W = \{X, Y, Z\} \in E^3$ representerar det tredimensionella rummet för omgivningen. X är rörelseriktningen fram längs vägen och parallellt med vägplanet. Y-riktningen är ortogonal mot X-axeln åt höger och parallell med vägplanet. Z-riktningen är ortogonal mot vägplanet.
- $I = \{u, v\} \in E^2$ representerar det tvådimensionella rummet för bilden, där det tredimensionella perspektivet är projicerat. u-riktningen är horisontell åt höger, och v-riktningen är vertikal neråt. Den avbildade bilden, d.v.s. vägytan i W, definieras som XY-planet i W. Avbildningen S defineras som $\{(X, Y, 0) \in W\}$.

Vidare består IPM-transformation huvudsakligen av två steg, $I \longrightarrow S$ avbildning och $S \longrightarrow I$ avbildning. Denna process kräver kännedom om kamerans inre och yttre parametrar, exempelvis optiska centrumkoordinater, lutnings- eller stigningvinkel, sväng- eller girvinkel och kamerans höjd över marken.



Figur 4: *Skiss över kamerainställning som visar Lutningsvinkel. Lutningsvinkeln θ är avvikelsevinkeln av den optiska axeln med avseende på (Y=0)-planet.*



Figur 5: *Skiss över kamerainställning som visar Girvinkeln. Girvinkel γ är vinkeln mellan projektionslinjen $\hat{\eta}$ av den optiska axeln i XY-planet och X-axeln.*

Koordinaterna för kamerans monteringsposition i 3D-planet som det framgår i figur 4 och figur 5 kan antas vara $(d, l, h) \in W$. Girvinkel γ är vinkeln mellan projektionslinjen $\hat{\eta}$ av den optiska axeln i XY-planet och X-axeln. Lutningsvinkeln θ är avvikelsevinkeln av den optiska axeln med avseende på (Y=0)-planet. $2\alpha_u$ och $2\alpha_v$ representerar

kamerans synfält i horisontell respektive vertikal riktning.

Avbildningen sker med kamerans intrinsiska och extrinsiska parametrar enligt följande:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = k \begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{14}$$

där R och T är kamerans extrinsiska paramterar och matrisen K innehåller kamerans intrinsiska paramterar

$$K = \begin{bmatrix} \alpha_a & \gamma & u_o & 0 \\ 0 & \alpha_b & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{15}$$

u_0 och v_0 motsvarar mitten av bilden.

Första steget är avbildningen $I \longrightarrow S$. En pixel (u, v) väljs i $m \times n$-bilden och följande transformationsekvation gäller:

$$\begin{cases} x(u,v) = h \cot\left(\frac{2v\alpha_v}{n-1} - \alpha_v + \gamma\right) \cos\left(\frac{2u\alpha_u}{m-1} - \alpha_u + \theta\right) + l \\ y(u,v) = h \cot\left(\frac{2v\alpha_v}{n-1} - \alpha_v + \gamma\right) \sin\left(\frac{2u\alpha_u}{m-1} - \alpha_u + \theta\right) + d \\ z(u,v) = 0 \end{cases} \tag{16}$$

Ekvation 16 ger den transformerade punkten $(x, y, 0) \in S$ som motsvarar $(u, v) \in I$. Andra steget är avbildningen $S \longrightarrow I$ enligt ekvationen:

$$\begin{cases} u(x,y,0) = \frac{(m-1)\times\left(\arctan\frac{y-d}{x-l} + \alpha_u - \theta\right)}{2\alpha_u} \\ v(x,y,0) == \frac{(n-1)\times\left(\arctan\frac{h}{\sqrt{(x-l)^2+(y-d)^2}} + \alpha_v - \gamma\right)}{2\alpha_v} \end{cases} \tag{17}$$

Ekvation 17 är transformationen av projektionen. Pixelvektorn av koordinaterna $(x, y, 0) \in W$ genomsöks och varje pixel tilldelas värdet för den motsvarande pixeln av koordinaterna $(u(x, y, 0), v(x, y, 0))$ [8]. Man kan också uppskatta en homprgrafi genom att man markerar en kvadrat på marken.

5 ALGORITMER

De flesta vägkantsdetektionsalgoritmerna följer samma mönster. Detta innebär att bilden först processeras, sedan skapas ett område av intresse (region of intreset, ROI) där vägen med stor sannolikhet befinner sig och till sist någon sorts vägkantsdetektion tillämpas. De algoritmer som har undersökts har varit Houghtransformations- eller fågelperspektivtransformationsbaserade.

5.1 Houghtransformbaserade algoritm

En variant av algoritm baserad på Houghtransformering nämns i [1]. Den genomförs genom att först utföra en HLS-konvertierng på bilden (RGB till HLS) för att enklare kunna färgsegmentera (i [1] fall är det vitt och gult som ska isoleras, men vilken färg som helst kan lyftas). När önskade färger är isolerade är det dags att undersöka intelnsitets-skillnaden på pixlarna i förhållande till sin omgivning. Genom att hitta dessa skillnader kommer vägkanterna att synas ännu bättre. I detta stadie har bilden fortfarande tre färgkanaler (Canny-kantdetektion kräver en kanal), så det första steget är att omvandla bilden till gråskala. Gråskalade bilder består endast av en färgkanal som visar intensiteten på respektive pixel. Bilden är nu i ett stadium där intensiteterna är centrala, så genom att använda Canny-kantdetektion (Canny edge detection) kan dessa intensitetsskillnader framhävas. För att få bort oönskat brus appliceras ett område av intresse (ROI) där vägen med störst sannolikhet befinner sig (ofta en triangel). De skarpa kanterna som är skapade av kantdetektionen kan ge problem, så för att undvika det lågpass-filtreras bilden (exempelvis genom att tillämpa ett gaussiskt lågpassfilter på bilden [3]).

Nästa steg är att utföra Hougtransform på den processerade bilden för att få ut linjer där vägkanterna befinner sig. För att få bort falska positiva värden kan lutningen på de linjer som skapas undersökas. Om det inte uppfyller kriteriet för lutningen på vägen kan de ignoreras.

Nästa steg är att hitta den dominanta linjen (från Houghtransformeringen) i respektive väglinje. Längden på linjen ger en god indikation på hur säker den är. Detta betyder att den längsta linjen är den som sätts som den dominanta. Det sista steget är att tillämpa en linjär regression så vägkanterna kan representeras på formen $y = kx + m$, vilket då resulterar i att väglinjerna är identifierade.

Det som oftast särskiljer olika algoritmer som är baserade på Houghtransformering är hur de hanterar bildbehandling-en. Algoritmen som [3] beskriver (som också är en Houghtransformeringsbaserad algoritm) är snarlik den som anges enligt [1]. I denna algoritm utförs ingen lågpass-filtrering av bilden efter att ha applicerat Canny-kantdetektionen utan utför en gaussisk lågpass-filtrering innan Canny-kantdetektionen används (utöver den som kantdetekionen kommer applicera själv). Det utförs för att ytligare reducera brus så endast de mest framträdande kanterna blir kvar. När Canny-kantdetektionen har utförts på bilden kan även trösklar sättas ut för att endast få ut de maximala utslagen, som med stor sannolikhet är orsakade av vägkanterna [3].

Enligt algoritmen i [3] kan vänstra respektive högra vägkanten identifieras efter utförandet av Houghtransformen och detta genom att undersöka gradienten på pixlarnas koordinater på de linjer som har hittats.

- För den vänstra vägkanten: då kolumnindexet (bredden) ökar ska radindexet (höjden) minska.
- För den högra vägkanten: om kolumnindexet ökar ska radindexet öka.

Efter detta tas medelvärdet på alla linjer i högra respektive vänstra vägkanten för att få två linjer. De linjerna extrapoleras sedan för att täcka hela vägkanten. Det är värt att observera att Houghtransformen är bra på att hitta raka linjer men ger dålig information när det gäller sneda linjer. Detta kan orsaka problem vid sväng eller korsning.

5.2 Fågelperspektivsbaserade algoritm

En algoritm som utnyttjar fågelperspektiv för att hitta vägkanterna ges i [1]. För att denna algoritm ska funka måste en kameraförvrängningskorrektion utföras. Utan den kommer raka linjer att uppfattas som böjda ju närmare kanten

den befinner sig. Bilden konverteras sedan från RGB- till LAB-rummet. A- och B-kanalen detekterar vitt och gult (vanliga vägkantsfärger) extra väl vilket gör det lättare att filtrerar ut sådant som inte är del av vägen. För att hitta intensitetshoppen används Sobel-operatorn för att upptäcka svagare kanter (Sobel-operatorn använder ej trösklar som Canny gör, vilket gör att svagare intensitets hopp också tas med) [1] samt en lågpass-filtrering av bilden (till exempel gaussisk lågpass-filtrering) för att bara få fram de största intensitetshoppen. För att ta bort omgivande brus appliceras ett område av intresse (ROI) där vägen befinner sig med högst sannolikhet.

För att kunna detektera vägkanterna utförs en fågelperskeptivstransform för att ge en fågelvy på vägen. För att nu hitta vägkanterna uttnyttjas en så kallad *Sliding-Window-metod*. Metoden funkar genom att sätta ut en startpunkt på ett fönster. Nästa steg avgörs beroende på hur många pixlar med önskad intensitet som hittas inom rutan. Om Sliding-Window-metoden detekterar tillräckligt många pixlar (anges av en tröskel) kommer den laterala postionen av medelvärdet av pixlarna att bli det nya centrumet för fönstret, om inte behåller den samma laterala postion. Vid varje iteration går fönstret uppåt tills det når andra sidan [4]. För att hitta startpunkten anpassas ett histogram för antalet pixlar och sedan tas de två topparna (som är vägkanternas startposition) för att avgöra starten. Med fönster som representerat vägen kan en tredje-ordningens spline tillämpas för att hitta vägkanterna.

En annan liknande algoritm är beskriven i [4]. Bildbehandlingen är snarlik men den stora skillnaden är att den inte använder LAB-, utan HSL-färgrummet istället. Canny-kantdetektion appliceras på S- och L-kanalen då för att få fram kanterna.

6 FÄRDIGA PAKET FÖR IMPLEMENTATION AV ALGORITMER

För att underlätta och ge bra flyt i implementationen av algoritmer används en del färdiga paket som är förprogrammerade och som innehåller färdigskrivna metoder. Ett sådant paket som används för bildanalys och i datorseende är OpenCV [9]. OpenCV är ett programvarubibliotek som kan användas med flera plattformar, till exempel Linux och Windows. Biblioteket är utvecklat så att det stödjer flera programmeringsspråk som python, C++ och java. OpenCV tillhandahåller metoder för bildbehandling och har färdigskrivna funktioner för att förstärka önskade detaljer eller objekt i bilden. Exempel på dessa metoder är att hitta kanter och identifera cirklar och linjer. Det finns också möjlighet för att filtrera bort oönskad information i bilden.

7 DISKUSSION OCH SLUTSATSER

Enligt teoridelen som behandlats i kapitel 4 och 5 består alla algoritmer av två huvudsteg, nämligen bildbehandling för att processera bilden och reducera brus och onödig information i bilden, och vägkantsdetektion för att hitta en matematisk representation av vägkanter. Slutsatsen som dras utifrån de undersökta algoritmerna är att de har liknande struktur och använder i många fall samma metoder när det gäller bildbehandling. Från dessa metoder går det att observera att användning av fågelperspektivs-transformen och Sliding-window-metoden gör det möjligt att hitta vägkanter som har en mer cirkulär karaktär, medan Houghtransformen endast hittar raka linjer, alltså raka vägkanter.

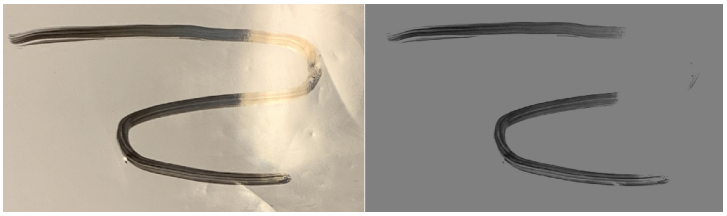
Information om komplexiteten är inte given för respektive algoritm. Det som dock tar lång tid i dessa algoritmer är brusreduceringen (bildprocesseringen) [1]. Då går det att dra slutsatsen att algoritmen med fågelperspektivs-transformeringen

är mer beräkningstung då den kräver mer processeringen (exempel med förvrängningskorrektionen).

Även information om upplösningens inverkan var svår att hitta. Däremot kommer upplösningen att spela en stor roll i hur snabbt algoritmen kommer att köras. Genom att minska upplösningen kommer algoritmen att köra snabbare, men då är risken att information går förlorad. Med andra ord är detta en balansgång, men även ett bra sätt att vinna prestanda på.

För det projektet som denna artikel är kopplat till ska vägen vara enkelfilig, vara inomhus, ha svarta streck som vägkanter samt vita runtomkring (golvet är vitt). Kameran kommer att vara vinklad nedåt för att minimera bruset runt omkring banan. Det primära bruset som kommer att uppstå är reflektioner i golvet, som till exempel kan komma från lampor. Utifrån dessa parametrar och de algoritmer som har undersökts kan en passande algoritm hittas utifrån projektets ändamål.

Bildprocesseringen är det första steget som ska utföras. Först ska en förvrängningskorrigering göras för att bli av med förvrängningar från kameran. I detta projekt kommer dock en kamera med 60° synvinkel att användas, så dessa förvrängningar kommer med stor sannolikhet att vara hanterbara. förvrängningskorrigering utelämnas alltså i detta fall. Går man vidare till själva brusreduceringen är vägkanterna svarta och omgivningen är vit vilket innebär att det i huvudsak är de två färgerna som ska plockas ut. Att byta färgrummen visar sig vara ett effektivt sätt för att bli av med brus utifrån färger. Undersöks specifikt HLS-färgrummets L-kanal visar den hur mycket svart respektive vitt som en viss pixel har. Genom att göra detta fås en binär representation av bilden som endast visar information om färgerna svart och vitt, som då kommer att reducera bruset från reflektioner, se figur 6. För att hitta vägkanterna används Canny-kantdetektion. För att inte få utslag på brus eller dylikt måste ett gaussikt lågpass-filter appliceras, men Canny-kantdetektion gör det automatiskt så det behövs ej tänkas på. Till sist kommer ett område av intresse (ROI) att tillämpas för att reducera eventuellt brus som uppkommit av kantdetektionen. Härmed är bildprocessering klar.



Figur 6: *Skillnaden mellan RGB-rummet (till vänster) och L-kanalen (till höger) från HSL-rummet för ett svart streck med vit bakgrund samt reflektion från ljuskälla.*

Nästa steg är att identifiera vägkanterna. Det som sedan behöver detekteras från vägen är, raksträcka, sväng, stopplinjer och T-korsningar. Det innebär att vägidentifieringen måste möjliggöra extraktion av sådan information. Det som direkt går att notera är att ett houghtransformering endast hittar raka linjer i bilden, vilket inte funkar här då även cirkulära kanter förekommer. Fågelperspektivstransformen med den så kallade Sliding-Window-metoden är dock bra för att hitta cirkulära kanter. Startpunkterna till Sliding-Window-metoden bestäms genom att kolla på var största samlingen vita pixlar finns (fås genom att applicera ett histogram vågrät mot bilden). Problemet med den metoden är att den inte letar efter vågräta linjer, något som stopplinjerna med stor sannolikhet kommer att vara. Det innebär att denna metod kommer inte att ge den informationen som önskas. Detta går att lösa genom att även applicera ett histogram lodrät mot bilden, eftersom stopplinjerna kommer att vara vågräta (när de är nära) kommer det att vara en större ansamling

med vita pixlar där, så då kan även Sliding-Window-metoden användas där för att hitta stopplinjerna. Därmed används fågelperspektiv med Sliding-Window-metoden både lodrät och vågrät mot bilden.

För att identifiera olika vägsegment med de linjer som fås av Sliding-Window-metoden kan följande plan följas:

- **Raksträcka**: Undersök om linjerna är relativt raka. Om båda är det antyder det på en raksträcka.
- **Svängar**: Undersök hur mycket linjerna böjer sig, om båda linjerna böjs åt samma håll indikerar det på en sväng åt det böjda hållet.
- **T-korsning**: Om en av linjerna böjs medan den andra är relativt rak indikerar det till en T-korsning med sväng åt det böjda hållet. Om båda linjerna böjs fast åt olika håll antyder det på en T-korsning som har sväng åt både höger och vänster.

Sammanställning av denna bildbehandlings algoritm:

- HLS-konvertering av bild.
- Extrahera L-kanalen från HLS-bilden.
- Utför Canny-kantdetektion.
- Tillämpa ett område av intresse (ROI).
- Transformera bilden med fågelperspektiv-transformation.
- Använda histogram för att bestämma startpunkterna på respektive linje.
- Tillämpa Sliding-Window-metoden.

För att implementera denna algoritm kommer OpenCV att tillämpas då den har många av dessa metoder som används inom bildbehandling redan färdigskrivna.

Möjliga problem som kan uppstå i denna algoritm är följande:

- Att använda HLS-rummet och bara undersöka L-kanalen kan leda till att information om vägen försvinner, se figur 6. Möjlig lösning är att undersöka HLS-rummet mer detaljerat, exempel genom färgtrösklar som används av algoritmer i 5.
- Algorimen är för långsam. Lösning på detta problem är att nedsampla bilden, då blir det färre pixlar att undersöka. Problemet med det är att information om bilden försvinner, så för stor nedsampling kan ej göras. Annan lösning skulle vara att skriva om bildprocesseringen då det är den delen som tar mest beräkningskraft.

REFERENSER

[1] Raja Muthalagu, Anudeepsekhar Bolimera, and V. Kalaichelvi, "Lane detection technique based on perspective transformation and histogram analysis for self-driving cars," vol. 85, 2020.

[2] Abdulhakam.AM.Assidiq, Othman O. Khalifa, Md. Rafiqul Islam, and Sheroz Khan, "Real time lane detection for autonomous vehicles," 2008.

[3] N. Tanksale, "Finding lane lines — simple pipeline for lane detection." 2019.

[4] hackster.io kemfic. (2018) Curved lane detection. [Online]. Available: https://www.hackster.io/kemfic/curved-lane-detection-34f771?fbclid=IwAR31oyoMDO4HAGhT8XJ5M99QRMCxoLN3uToeA5oNP2ONyqvodQ4tMflgSsI

[5] B. K. opencv.org. (2016) Sobel derivatives. [Online]. Available: https://docs.opencv.org/3.2.0/d2/d2c/tutorial_sobel_derivatives.html

[6] infoaryan, "Implement canny edge detector in python using opencv," *Geeks for Geeks*, 2020. [Online]. Available: https://www.geeksforgeeks.org/implement-canny-edge-detector-in-python-using-opencv/

[7] Q. Zhou, L. Zeng, and S. Zhou, "Statistical detection of defect patterns using hough transform," *IEEE Transactions on Semiconductor Manufacturing*, vol. 23, no. 3, pp. 370–380, 2010.

[8] Ding Yong, Xu Zheng, and Z. amd Sun Ke, "Fast lane detection based on bird's eye view and improved random sample consensus algorithm," *Multimedia Tools and Applications*, vol. 76, 2017. [Online]. Available: https://doi.org/10.1007/s11042-016-4184-6

[9] About opencv. [Online]. Available: https://opencv.org/about/

G FÖRSTUDIE - KOMMUNIKATION

Förstudie - Kommunikation

Henrik Ahlinder
Mikael Josefsson

7 juni 2021

Version 0.5

Status

Granskad	Henrik Ahlinder, Mikael Josefsson	2021-04-07
Godkänd		

INNEHÅLL

1 Introduktion 1
- 1.1 Frågeställning 1
- 1.2 Avgränsningar 1
- 1.3 Metod och struktur 1
- 1.4 Diskussion kring källor 2
- 1.5 Terminologi 2

2 Bakgrundsteori 3
- 2.1 Parallellprogrammering 3
- 2.2 Parallellprogrammering i Python 6
- 2.3 Parallellprogrammering i C 12

3 Resultat 13
- 3.1 Test av parallell programmering i Python 13
- 3.2 Test av parallell programmering i C 19
- 3.3 Testning av processorbundna program i C 21

4 Diskussion och slutsatser 24
- 4.1 Python 24
- 4.2 C 24

Referenser 25

DOKUMENTHISTORIK

Version	Datum	Utförda förändringar	Utförda av	Granskad
0.1	2021-02-24	Första utkast	MJ och HA	MJ, HA
0.2	2021-03-08	Korrigering av kapitelnumrering	HA	HA
0.3	2021-04-06	Korrigering av språk från feedback	MJ	MJ
0.4	2021-04-07	Andra utkast	HA och MJ	MJ, HA
0.5	2021-04-23	Korrigering av referenser	HA	MJ, HA

1 INTRODUKTION

I den moderna världen där allt större och mer komplexa uppgifter utförs av datorer behövs nyanserade metoder för att effektivisera beräkningarna. Därför finns det i de flesta moderna datorer mer än en processorkärna, och med det möjlighet att exekvera flera instruktioner simultant. Detta område kallas parallellprogrammering och denna förstudie syftar till att analysera de två huvudparadigmerna multitrådning och multiprogrammering, samt utvärdera deras användbarhet i olika programtyper.

1.1 Frågeställning

Frågeställningen kan sammanfattas i följande punkter:

- Hur utförs parallell programmering i Python och C?
- Vilka program kan med fördel parallelliseras?

1.2 Avgränsningar

Förstudien kommer endast att behandla programspråken Python (Cpython 3.x) och C. Programmen förutsätts skrivas och exekveras i LINUX-baserade operativsystem. Eftersom multitrådning och multiprogrammering av den typ som undersöks ej stöds på de mikrokontroller som används i projektet kommer dessa paradigmers inverkan inte att utvärderas på uppgifter som väntas utföras på dessa kort. I experiment där exekveringstid jämförs med och utan parallellism kommer endast de övergripande tendenserna att analyseras. Detta beror på att det är mycket svårt att få fram exakta tidsrepresentationer av exekveringar utan särskild testningshårdvara.

1.3 Metod och struktur

Frågeställningen ämnas besvaras i en följd av fyra övergripande kapitel. I kapitel 2 presenteras en litteraturstudie inom parallell programmering samt paradigmens implementationer i språken C och Python förklaras. Därefter implementeras exempelprogram i kapitel 3 genom vilka exekveringstider av parallellt såväl som sekvensiellt exekverade program undersöks. I det avslutande kapitlet, kapitel 4, diskuteras fördelar och nackdelar med olika paradigmer inom parallell programmering uppdelat i programspråk, och förslag på parallellisering av uppgifter via multitrådning och multiprogrammering i projektet presenteras.

1.4 Diskussion kring källor

Förstudiens källor är till största del baserade på programspråkens dokumentation. Som manual till språket C används dokumentation från projektet GNU, av Free Software Foundation. Detta antas vara en god källa då operativsystemet är skrivet i C, och projektet har varit stabilt i många år.

1.5 Terminologi

Nedan förklaras viss terminologi som används i studien.

Tabell 1: Begrepp som används i förstudien

Begrepp	Betydelse
Program	Innehåller alla instruktioner som behövs för att exekvera och utföra en uppgift.
Instruktionskejda	Kedja av datorinstruktioner.
Tråd	Instruktionskedja med nödvändigt stöd för att exekveras. Se kapitel 2.1
Process	Synonymt med program, men används i den här studien för en instans av ett program som körs.
IO	Input-Ouput, sv. Indata-Utdata. Används som benämning på datatransaktioner.
Processorbunden	Uppgift begränsad primärt av tillgänglig processortid.

2 BAKGRUNDSTEORI

I detta kapitel presenteras bakomliggande teori från litteraturstudien. Denna kunskap appliceras sedan i kapitel 3, Resultat.

2.1 Parallellprogrammering

Parallellprogrammering är ett område som handlar om att effektivisera en lösning av ett komplext problem genom att dela upp det i mindre delar som kan lösas parallellt. Sker denna uppdelning på ett effektivt vis kan vilotid i hårdvaran minskas, och därmed även exekveringstiden av programmet. Eftersom schemaläggning i processorn är en uppgift som operativsystemet sköter är parallell programmering hårt sammanknuten till operativssystemet som används. I denna studie behandlas de två vanligaste paradigmerna inom parallellprogrammering applicerade i ett LINUX-baserat operativsystem: multitrådning och multiprogrammering. Dessa två paradigmer beskrivs i detalj i avsnitten 2.1.3 och 2.1.4 nedan.

2.1.1 *I enkelkärniga processorer*

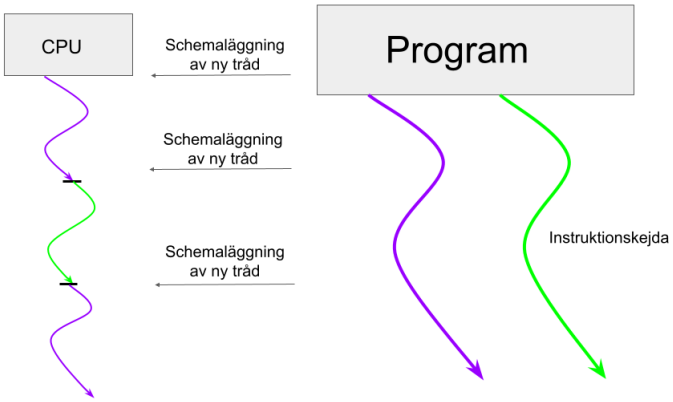
En processorkärna kan endast hantera en instruktionskedja åt gången. Därför saknas det möjlighet till simultan exekvering av instruktioner i en enkelkärnig processor som är slutmålet i parallellprogrammering. Trots det kan man ändå nyttja parallell programmeringsmetodik i enkelkärniga processorer eftersom instruktionskedjor ofta kan kräva minnesaccesser eller andra operationer som är långsamma jämfört med beräkningar. Man utnyttjar då en struktur som exekverar en tråd i kärnan fram tills den når en instruktion som orsakar vilotid i kärnan. Vanligtvis hade processorkärnan vilat medan instruktionen utfördes, men istället kan vi utnyttja uppdelning av programmet för att exekvera en annan instruktionskedja under vilotiden. Processen att på så vis byta mellan exekvering av olika instruktionskedjor visas visuellt i 1. Att skifta instruktionskedjor i kärnan går så fort att i mänskliga ögon uppfattas det som om trådarna exekveras simultant, trots att så inte är fallet.

2.1.2 *I flerkärniga processorer*

Metodiken för att utnyttja parallell programmering i flerkärniga processorer grundar sig i samma schemaläggningsmetoder som används i enkelkärniga processorer. Men till skillnad från i enkelkärniga processorer kan man i system med flera kärnor exekvera instruktionskedjor simultant genom att exekvera instruktionskedjor på olika kärnor simultant.

2.1.3 *Multitrådning*

I paradigmen multitrådning delas en process upp i små byggblock som kallas trådar. Trådar innehåller utöver en instruktionskedja bara det som krävs för att koden ska kunna exekveras. Detta betyder att kostnaden för att skapa trådar är relativt låg i förhållande till att skapa nya processer, men också att en tråd är hårt knuten till sin moderprocess. Kon-



Figur 1: Parallell programmering på enkelkärnig processor

ceptuellt strävar programmeraren i ett multitrådat program efter att minska exekveringstiden av programmet genom åtminstone en av programmets trådar alltid kan exekveras i processorn [1].

Eftersom trådar är knutna till sin moderprocess kan konflikter uppstå när flera trådar försöker modifiera gemensamma resurser. Ett vanligt exempel är två trådar som modifierar samma minnesutrymme simultant, vilket kan orsaka oväntade och oönskade fel i programmet. För att förhindra dessa typer av konflikter finns ett vanligt protokoll för kommunikation mellan trådar som kallas för mutex (Mutually Exclusive, sv. ömsesidigit uteslutande). Mutex fungerar som ett hänglås till sårbara resurser som trådar kan utnyttja för att förhindra konflikter. Metoden är mycket elementär och går ut på att en tråd som behöver ha åtkomst till en viss resurs markerar den som ”låst” via ett mutex. När tråden sedan har använt resursen färdigt ”låser den upp” utrymmet igen. Om en annan tråd vill utnyttja samma resurs kontrollerar den först om det är ”låst” via mutexen. Notera att detta protokoll endast ger en möjlighet att förhindra konflikter av gemensamma resurser. Det finns ingenting som hindrar ett program från att försöka använda den gemensamma resursen trots att en annan tråd har markerat den som ”låst”, t.ex. om tråden inte kontrollerar mutexen alls. Det är därmed programmerarens ansvar att alla aktiviteter som utnyttjar samma resurs använder sig av samma mutex, och att de har en lämplig rutin om resursen skulle vara låst [2].

Ibland är det även nödvändigt att synkronisera olika trådar och kontrollera att de exekveras i korrekt hastighet. För detta ändamål introduceras semaphores, som fungerar som en signalmetod mellan olika trådar. I enkla fall kan en semaphore beskrivas som en global resurs som utnyttjas av alla trådar. Det är då möjligt för de olika trådarna att samverka genom att läsa av informationen i semaphoren, och även att meddela sin status till de övriga trådarna. Notera att man därmed kan se mutex som en binär semaphore, med antingen ”låst” eller ”olåst” som möjliga tillstånd [2].

2.1.4 *Multiprogrammering*

I paradigmen multiprogrammering delas ett program upp i flera olika processer. Varje process kan ses som ett eget program, med eget minnesutrymme. Notera dock att vissa resurser delas mellan moderprogrammet och barnprocesser. Detta betyder att kostnaden att skapa en ny process är ungefär lika stor som att skapa ett nytt program, men också att nya processer som skapas inte är lika hårt knutna till moderprocessen som trådar (se kapitel 2.1.3). En skarpögd läsare

märker nu att multitrådning och multiprocessning alltså kan kombineras eftersom de olika processerna som skapas i multiprogrammering mycket väl kan innehålla flera trådar vardera.

Eftersom processer existerar likt egna program sker färre konflikter angående gemensamma resurser och minnesutrymme mellan processer som exekveras parallellt jämfört med trådar. Men denna separation av resurser innebär också att det är svårare att kommunicera mellan olika processer, och tekniker som Mutex och Semaphores som beskrivs i 2.1.3 fungerar dåligt för detta ändamål. För att skicka nödvändig information mellan olika processer används istället andra meddelandekanaler, såsom pipes. Det finns två varianter av pipes, anonyma och namngivna. För att skilja dessa åt kallar vi namngivna pipes för FIFO (First In First Out, sv. Först In Först Ut). En pipe kan liknas med en datatank mellan två processer. En process kan skriva, ”hälla”, data in i pipen från ena änden och en annan process kan sedan läsa, ”tömma”, data ur samma pipe. Traditionellt sett skapar moderprocessen en pipe som sedan skickas till barnprocesserna när de skapas. Pipen försvinner sedan i samband med att processen som skapat den avslutas. Det är möjligt för processerna att utnyttja samma pipe till både skriv- och läsoperationer. Det är dock ovanligt eftersom det är lätt att missbedöma tidpunkten för operationen, vilket kan resultera i att processen skriver information ovanpå information som ska läsas. Istället används oftast två (eller fler) pipes om tvåvägskommunikation mellan processer är nödvändigt [3]. Den andra typen av pipe, FIFO, kan istället liknas med en temporär fil då livslängen för en FIFO är lika långt som livslängden av operativsystemet som skapar den. Efter skapandet av en FIFO kan alla processer nyttja den för kommunikation. Dock är det inte möjligt att skriva till en FIFO före en process som läser av den existerar [4].

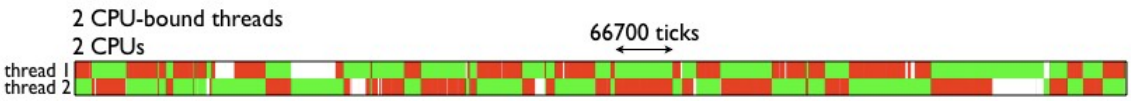
2.2 Parallellprogrammering i Python

Detta kapitel inleds med avsnitt 2.2.1 som behandlar vilka fundamentala begränsningar Python har när det kommer till parallellprogrammering. Det visar sig att Python på grund av dess *"Global Interpreter Lock"* endast kan erhålla prestandavinster med trådar då programmet är IO-bundet, medan multiprogrammering kan hjälpa till att förbättra program som är IO- eller processorbundna. Avsnitt 2.2.2 och 2.2.3 beskriver de vanligaste modulerna som används vid hantering av trådar respektive processer, med hjälp av en kort förklaring följt av ett kodexempel. Modulerna som behandlas är *threading*, *multiprocessing* och *concurrent.futures*. Det inses lätt att de två förstnämnda modulerna används för att hantera trådar och processer. Den senare modulen innehåller klasser med hög abstraktionsnivå som i många fall kan underlätta hantering av både trådar och processer. Det visas att användning av modulerna som hanterar trådar och processer är nästintill identisk. För att undvika upprepning görs därför en djupare genomgång av multitrådning medan multiprogrammering endast behandlas ytligt.

2.2.1 *Begränsningar*

Python är ett språk som använder en tolk för att exekvera kod. I praktiken innebär det att Python aldrig kompileras direkt till maskinkod, utan istället kompileras källkoden till en fil med bytekod. En bytekod motsvarar flera maskinkodsinstruktioner. Tolken kan ses som att en virtuell maskin som läser bytekoden och kör motsvarande maskinkod i realtid [5].

I Python styrs trådar av en mekanism som i tolken som kallas för *"Global Interperter Lock"*, nedan benämnt GIL. För att en tråd ska få exekvera kod i tolken måste den inneha kontrollen över GIL. Endast en tråd kan inneha GIL och följdaktligen kan endast en tråd inom en process exekveras åt gången. En visualisering av denna process visas i figur 2, där trådar som exekveras är gröna, trådar som försöker få tillgång till GIL markeras i rött samt trådar som är inaktiva visas i vitt. Utan GIL skulle även de enklaste operationer kunna orsaka problem, exempelvis om två trådar försöker läsa och skriva till en resurs samtidigt.



Figur 2: Visualisering av två trådar som byter GIL mellan sig[6]

Tolken frigör trådar som vilar i processorn. För att ge intrycket av simultan exekvering växlar tolken också periodiskt mellan trådar utan hänsyn till om tråden är i vila eller aktiv. En tråd förlorar aldrig GIL under exekvering av en bytekod, men kan med andra ord alltid förlora kontrollen över GIL när den befinner sig mellan två bytekoder [7]. För vissa typer av operationer kan det bli stora konsekvenser om en tråd avbryts mellan två bytekoder. På grund av detta är vissa typer av operationer atomära, vilket betyder att de endast består av en bytekod och kan därmed inte avbrytas. Ett exempel på en atomär operation är funktionen sort() som sorterar en lista. Att det är önskvärt att sort() är atomär inses lätt. Många operationer är från början atomära men inte alla. Vid användning av icke-atomära operationer på data som delas mellan trådar bör tråden låsas tills att den är klar för att förhindra problem. Ett exempel på en icke-atomär operation och hur en tråd kan låsas ses i listning 1 [8].

```
n = 0
lock = threading.Lock()

def foo():
    global n
    with lock:
        n += 1
```

Listing 1: Exempel på hur en tråd kan hindra GIL från att växla tråd automatiskt [8].

Operationen ”n += 1” består av 3 rader bytekod och riskerar därför att utan lås avbrytas mellan bytekoder. Om tolken skulle växla till en annan tråd som manipulerar samma variabel riskerar information att gå förlorad. Men om operationen, som i exemplet, sker i blocket ”with lock:” kommer all kod i blocket att exekveras utan risk att tappa GIL [8].

En nackdel med GIL är att multitrådade lösningar till problem som är processorbundna inte är effektiva. Sådana problem får istället lösas genom att skapa nya processer. Varje process erhåller då en egen tolk och GIL, därmed kan de exekvera kod oberoende av varandra [9].

2.2.2 *Trådhantering i Python*

I detta kapitel beskrivs hur trådar skapas och hanteras i Python både via modulen *threading* och modulen *concurrent.futures*. Som beskrivet i avsnitt 2.2.1 är användningen av trådar i Python främst fördelaktiga vid användning av program som är IO-begränsade.

Modulen *threading* innehåller klassen *Thread* som representerar en aktivitet som körs i en annan tråd. Konstruktorn till *Thread* har ett flertal parametrar, se listning 2.

```
threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None))
```

Listing 2: Parametrar som konstruktorn till klassen *Thread* har

De viktigaste parametrarna är *target* och *args* samt *kwargs*. Dessa anger vilken *callable* som ska exekveras samt argumenten till denna *callable* [10].

Klassen *Thread* har även ett flertal metoder för att underlätta multitrådad programmering. Två av de viktigaste och användbara är *start* och *join*. Det är först när metoden *start* anropas som den *callable* som angavs som *target* kommer att börja exekvera i en egen tråd. Metoden *join* gör att programmet väntar till tråden har avslutats innan fortsatt exekvering [10]. Ett enkelt exempel på ovanstående koncept illustreras i listning 3.

```
from threading import Thread
import time

def some_func(param1, param2):
    # do something with param1 and param2
    time.sleep(1)
    print("Sum: ", param1 + param2)

thread = Thread(target=some_func, args=(13, 37))
thread.start()
```

```
11 thread.join()
12 print("Program finished")
```

Listing 3: Ett enkelt exempel på hur en funktion kan exekveras i en egen tråd *Thread*

Funktionen *some_func* tar två parametrar som indata. Den börjar med att vänta 1 sekund och därefter adderas parametrarna och resultatet skrivs ut. På rad nio skapas en instans av klassen *Thread*. Vi anger att funktionen vi vill exekvera i tråden ska vara *some_func* och skickar även i talen 13 och 37 som argument. På rad tio påbörjas exekveringen av tråden och på rad elva anger vi att huvudprogrammet ska vänta på att tråden ska exekvera klart innan slutligen *"Program finished"* ska skrivas ut på rad tolv.

Två uppenbara nackdelar med att skapa trådar med hjälp av klassen *Thread* är att den inte erbjuder något smidigt sätt att få tillbaka returvärden från en tråd som exekverat klart. Det är även osmidigt att hantera ett stort antal trådar. Om dessa problem ska lösas kan det istället vara värt att använda klassen *ThreadPoolExecutor* som är en del modulen *concurrent.futures*. Den ger en möjlighet att på ett smidigt sätt hantera ett stort antal trådar samt att få tillbaka returvärden från avslutade trådar. Klassen *ThreadPoolExecutor* har fyra stycken parametrar i sin konstruktor, se listning 4.

```
1 ThreadPoolExecutor(max_workers=None, thread_name_prefix='', initializer=None, initargs=())
```

Listing 4: Parametrar som konstruktorn till klassen *ThreadPoolExecutor* har

Parametern *max_workers* specificerar det maximala antalet trådar som klassen tillåts skapa. Det nyttjas då vi vill utföra en metod många gånger, men samtidigt vill begränsa antalet trådar [11].

ThreadPoolExecutor ärver två användbara metoder från klassen *Executor*: *submit* och *map*. Metoden *submit* tar en *callable* som skall exekveras samt ett godtyckligt antal parameterar i form av *args* och *kwargs* och returnerar ett objekt av typen Future som ska representera exekveringen av *callable*. För erhålla resultatet anropas metoden *result* på *Future*-objektet, vilket då först eventuellt inväntar trådens avslutade exekvering om tråden ej är redan avslutad och sedan returnerar resultatet. I listning 5 visas ett enkelt exempel av ovanstående koncept. Funktionen some_activity är nästan identiskt med some_func i listning 3, men returnerar summan istället för att skriva ut den.

```
1  from concurrent.futures import ThreadPoolExecutor
2  import time
3
4  def some_activity(param1, param2):
5      # do something with param1 and param2
6      time.sleep(1)
7      return param1 + param2
8
9  executor = ThreadPoolExecutor(max_workers=1)
10 future = executor.submit(some_activity, 13, 37)
11 result = future.result()
12 print("Sum: ", result)
13 print("Program finished")
```

Listing 5: Exempel på hur en funktion kan exekveras i en egen tråd med hjälp av klassen *ThreadPoolExecutor*

Programmet är snarlikt det i listning 3, men nu returneras summan från tråden när den exekverat färdigt och skrivs ut i huvudprogrammet istället.

Modulen *concurrent.futures* innehåller två funktioner som är till hjälp när vi vill behandla en *iterable* med data i flera trådar: *as_completed* och *wait*. Funktionen *as_completed* tar in ett set med *Future*-objekt och returnerar en iterator

över alla dem. Objekten återfås i den ordning som de färdigställs (de exekveras asynkront) och möjliggör därmed behandling av resultat från individuella trådar utan att behöva vänta på att övriga trådar ska avslutas. Om det istället är önskvärt att alla *Future*-objekt ska färdigställas innan någon data behandlas görs detta via funktionen *wait* istället för *as_completed*. Funktionen tar in ett set med *Future*-objekt och returnerar sedan en tupel med två element: avslutade trådar och oavslutade trådar [11]. Ett exempel som där funktionen *as_completed* används kan ses i listning 6.

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import time

def some_activity(param1, param2):
    # do something with param1 and param2
    time.sleep(1)
    return param1 + param2

executor = ThreadPoolExecutor(max_workers=4)
data = ((13,37), (24,12), (19,18), (12,13))

futures = {executor.submit(some_activity, num_pair[0], num_pair[1]) : num_pair for num_pair in
    data}

for future_finsihed in as_completed(futures):
    print(future_finsihed.result())
print("Program finished")
```

Listing 6: Ett enkelt exempel på hur en uppsättning data kan behandlas med *ThreadPoolExecutor* och *as_completed*

I programmet i listning 6 utförs samma uppgift som i programmen i listningarna 3 och 5. Skillnaden är att vi nu har en uppsättning data som vi vill beräkna. På rad nio i listningen skapas en *ThreadPoolExecutor* där *max_workers* är satt till fyra vilket kommer göra det möjligt att påbörja en ny exekvering av funktionen *some_activity* parallellt under tiden som *time.sleep* exekveras. På rad tolv skapas ett set med *Futures*-objekt som behandlar och påbörjar exekvering av vår uppsättning av data som är definerad på rad tio. På rad fjorton återfås *Future*-objekten i den ordning som de färdigställs och resultatet kan skrivas ut.

Som tidigare nämnt har klassen *ThreadPoolExecutor* en metod som heter *map*. Metoden behandlar data på samma sätt som Pythons inbyggda funktion *map*. Att använda *map* är ett alternativ till arbeta med *Future*-objekten då vi har en stor mängda data i *iterables* och endast är intresserade av resultatet från varje exekvering. Funktionen ger tillbaka en lista som innehåller resultaten från varje tråd [11]. Ett exempel ges nedan i listning 7 som visar användningen av funktionen *map*.

```
from concurrent.futures import ThreadPoolExecutor, as_completed
import time

def some_activity(param1, param2):
    # do something with param1 and param2
    time.sleep(1)
    return param1 + param2

executor = ThreadPoolExecutor(max_workers=4)
data = (13, 24, 19 ,13)
data2= (37, 12, 18, 13)

results = executor.map(some_activity, data, data2)

```

```
for result in results:
    print(result)
print("Program finished")
```

Listing 7: Ett enkelt exempel på hur en uppsättning data kan behandlas med *ThreadPoolExecutor* och dess funktion *map*

Programmet i listing 7 utför samma sak som i 6 med skillnaden att vi använder oss aldrig av *Future*-objekt. Istället använder vi oss a funktionen *map* som direkt retunerar samtliga resultat. Observera att talparen nu har delats upp i två olika *iterables* som korresponderar mot de två olika parametrarna i funktionen some_activity.

2.2.3 *Processhantering i Python*

Som tidigare nämnt i inledning till detta kapitel så är hantering av trådar och processer nästintill identiskt i Python, därför kommer endast en översiktlig behandling av processer ske. Allt som behandlas i avsnitt 2.2.2 för trådar gäller även för detta avsnitt [9].

För processhantering i Python finns modulen *multiprocessing*. Modulen innehåller en klass som heter *Process* som innehar metoderna *start* och *join* med samma funktionalitet som klassen *Threads* metoder. Parametrarna till konstruktorn och metodern accepterar samma argument som i *Thread* [9].

I modulen *concurrent.futures* finns en *ProcessPoolExecutor* som precis som *ThreadPoolExecutor* ärver ifrån den abstraka klassen *Executor*. Därmed har även *ProcessPoolExecutor* metodern *submit* och *map* som används på motsvarande sätt som för trådar. Funktionerna *as_completed* och *wait* används även på samma sätt [9].

Det finns två viktiga skillnader som måste tas hänsyn till när multiprogrammering används. Den första är att en startmetod bör väljas via funktionen *set_start_method*. Det finns finns tre olika startmetoder att välja mellan, *spawn*, *fork* och *forkserver*. Alternativet *forkserver* behandlas inte i studien. Beroende på om *fork* eller *spawn* väljs som startmetod kommer huvudprocessen och den skapade processen att få olika relation till varandra. Om *spawn* väljs ärvs inga onödiga resurser. En helt ny instans av tolken startas och hela programmet laddas in på nytt. Om *fork* används uppstår beteendet som beskrivs i avsnitt 2.1.4. Det är då viktigt att vara uppmärksam på problem som kan uppstå vid delade resurser.

Om en metoden *spawn* används kommer hela programmet att laddas in igen och det kan få oönskade effekter som t.ex. att ett nytt huvudprogram startar. Huvudprogrammet måste därför endast startas efter att en kontroll som undersöker om nuvarande process är huvudprocessen genomförts. Kontrollen görs genom att undersöka om processen har namnet ”__main__”. I listning 8 och 9 visas exempel hämtade ifrån Python manualen som visar hur en process inte bör startas respektive hur den bör startas [9].

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Listing 8: Ett enkelt exempel på hur en ny process inte bör startas

```python
from multiprocessing import Process, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

Listing 9: Ett enkelt exempel på hur en ny process bör startas

2.3 Parallellprogrammering i C

C använder inte en tolk som Python utan istället en kompilator som översätter programkod direkt till maskinkod som sedan kan exekveras. Detta innebär att låsningen som GIL innebär för multitrådad kod i Python inte har någon motsvarighet för C, utan multitrådad kod kan exekveras simultant på flera olika processorkärnor.

2.3.1 *Trådhantering i C*

I studien skapas och hanteras trådar i C via biblioteket *pthreads.h*, som är standardimplementationen av trådar i C. *Pthreads* är en exekveringsmodell utiven av IEEE. Själva namnet *Pthreads* är en förkortning för *POSIX Threads* och är den trådmodell som definierias i *POSIX*, och samma modell används även i LINUX. Trådhanteringsmodellen i operativsystemet Raspberry Pi OS, som används i projektet, är nästan identiskt med denna trådmodell [12]. I resterande avsnitt av studien används därför begreppet tråd synonymt med *pthread*.

2.3.2 *Processhantering i C*

Processhantering i C sker genom funktion *fork*, som skapar en kopia av programmet som en ny process. Man kan särskilja moderprocessen från barnprocessen i exekverande kod genom att *fork* returnerar värdet 0 i barnprocessen, och *processID* för barnprocessen i moderfunktionen [13]. Notera att det är viktigt att kontrollera vilken processinstans som koden exekveras i, eftersom det annars är enkelt att skapa för många processer. Exempelvis ser det ut som om den korta programkoden i listning 10 skapar fyra trådar, medan den i verkligheten skapar totalt åtta separata processer.

```c
#include <unistd.h>

int main()
{
    fork();
    fork();
    fork();
}
```

Listing 10: Exempel på *fork*

3 RESULTAT

I detta kapitel undersöks vilka prestandaeffekter parallell programmering kan ha på program som är IO-bundna eller processorbundna. I avsnitt 3.1 behandlas programspråket Python och därefter i avsnitt 3.2 behandlas C. För varje programspråk utförs först tester av ett IO-bundet program och därefter ett processorbundet program. Ett test inleds med en sekventiell exekvering av programmet och därefter en multitrådad exekvering och slutligen en multiprogrammerad exekvering. Testerna i Python visar att multitrådad exekvering kan ge stora prestanda vinster i IO-bundna program men blir en flaskhals i processorbundna program. Multiprogrammering ger prestandavinster i båda fallen. I C blir skillnaderna i prestanda mellan multitrådning och multiprogammering inte lika tydliga som i Python.

3.1 Test av parallell programmering i Python

I detta avsnitt undersöks vilka prestandaeffekter parallell programmering i Python kan ha på program som är IO-bundna respektive processorbundna. Testerna visar att multitrådad exekvering kan ge stora prestandavinster i IO-bundna program men blir en flaskhals i processorbundna program. Multiprogrammering ger prestandavinster i båda fallen.

3.1.1 *Testning av IO-bundet program*

För att testa hur exekveringstiden av program påverkas när det utför en IO-bunden uppgift skapades ett enkelt program, se listning 11. Funktionen IO_dependent_task ska härma beteendet hos en funktion som kräver vilotid mellan förfrågan om data och mottagning av data. För att uppnå en visuell vilotid användes funktionen time.sleep, via vilken vilotiden sattes till en sekund.

```
import time

def IO_dependent_task():
    #Simulates a task that needs two inputs to complete
    print('Sent request for new package')
    time.sleep(1)
    print('Recevied data package')

if __name__ == "__main__":
    start_time = time.perf_counter()
    IO_dependent_task()
    IO_dependent_task()
    IO_dependent_task()
    finish_time = time.perf_counter()
    print(f'Finished in {round(finish_time-start_time,2)} seconds')
```

Listing 11: I/O-bunden funktion körd på en tråd

Programmet i listning 11 generade utskriften i listning 12. Utskriften består av skickat och mottagit i tre omgångar och avslutas sedan med tidsresultatet. Tiden det tog för programmet att exekvera sekvensiellt var 3.05 sekunder, d.v.s. största delen av programmet spenderades i vilotid.

```
>>>Sent request for new package
>>>Recevied data package
>>>Sent request for new package
>>>Recevied data package
>>>Sent request for new package
>>>Recevied data package
>>>Finished in 3.05 seconds
```

Listing 12: Utskrift genererad av programmet i listning 11

För att testa hur ett IO-bundet program påverkas när det exekveras i flera trådar skapades programmet i listning 13. Funktionen threading.Thread användes för att skapa en ny tråd som exekverade funktionen IO_dependent_task. Tre trådar skapades, x, y och z. Funktionen join() användes för att huvudprogammet skulle invänta avslutad exekvering av trådarna innan programmet avslutade tidmätningen.

```python
import time
import threading

def IO_dependent_task():
    #Simulates a task that needs two inputs to complete
    print('Sent request for new package')
    time.sleep(1)
    print('Recevied data package')

if __name__ == "__main__":
    start_time = time.perf_counter()
    x = threading.Thread(target=IO_dependent_task)
    y = threading.Thread(target=IO_dependent_task)
    z = threading.Thread(target=IO_dependent_task)
    x.start()
    y.start()
    z.start()
    x.join()
    y.join()
    z.join()
    finish_time = time.perf_counter()
    print(f'Finished in {round(finish_time-start_time,2)} seconds')
```

Listing 13: I/O-bunden kod körd på flera trådar

I listning 14 syns att programmet exekverades parallellt, där tre förfrågningar utfördes utan den inducerade fördröjningen. Den totala exekveringens tiden uppmättes till 1.04 sekunder.

```
>>>Sent request for new package
>>>Sent request for new package
>>>Sent request for new package
>>>Recevied data package
>>>Recevied data package
>>>Recevied data package
>>>Finished in 1.04 seconds
```

Listing 14: Utskrift genererad av programmet i listning 13

För att testa hur ett IO-bundet program påverkas när det exekveras i flera processer skapades programmet i listning 15. Funktionen *multiprocessing* användes för att skapa en ny tråd som exekverade funktionen IO_dependent_task. Tre trådar skapades, x, y och z. Funktionen *join* användes för att huvudprogammet skulle invänta avslutad exekvering av processerna innan programmet avslutade tidmätningen.

```
import time
import multiprocessing

def IO_dependent_task():
    #Simulates a task that needs two inputs to complete
    print('Sent request for new package')
    time.sleep(1)
    print('Recevied data package')

if __name__ == "__main__":
    multiprocessing.set_start_method('spawn')
    start_time = time.perf_counter()
    x = multiprocessing.Process(target=IO_dependent_task)
    y = multiprocessing.Process(target=IO_dependent_task)
    z = multiprocessing.Process(target=IO_dependent_task)
    x.start()
    y.start()
    z.start()
    x.join()
    y.join()
    z.join()
    finish_time = time.perf_counter()
    print(f'Finished in {round(finish_time-start_time,5)} seconds')
```

Listing 15: I/O-bunden kod körd på flera processer

I listning 16 syns att programmet exekverades parallellt, där tre förfrågningar utfördes utan den inducerade fördröjningen. Den totala exekveringens tiden uppmättes till 1.11 sekunder.

```
Sent request for new package
Sent request for new package
Sent request for new package
Recevied data package
Recevied data package
Recevied data package
Finished in 1.11 seconds
```

Listing 16: Utskrift genererad av programmet i listning 15

3.1.2 *Testning av processorbundet program*

För att testa hur exekveringstiden av program påverkas när det utför en processorbunden uppgift skapades ett enkelt program, se listning 11. Funktionen *cpu_heavy_task* belastar processorn genom att räkna ned ifrån 10^7 till 0 genom en loop subtraherar med 1 för varje passage.

```
import time
from math import pow

def cpu_heavy_task(id):
    print(f'CPU heavy task {id} started')
    i = pow(10,7)
    while(i>0):
        i = i - 1
    print(f'CPU heavy task {id} finished')

if __name__ == "__main__":
    start_time = time.perf_counter()
    cpu_heavy_task(1)
    cpu_heavy_task(2)
    cpu_heavy_task(3)
    finish_time = time.perf_counter()
    print(f'Finished in {round(finish_time-start_time,2)} seconds')
```

Listing 17: Processorbunden kod körd sekventiellt

Programmet i listning 17 generade utskriften i listning 18. Utskriften består av startad och avklarad beräkning i tre omgångar och avslutas sedan med tidsresultatet. Tiden det tog för programmet att exekvera sekvensiellt var 2.78 sekunder.

```
>>>CPU heavy task 1 started
>>>CPU heavy task 1 finished
>>>CPU heavy task 2 started
>>>CPU heavy task 2 finished
>>>CPU heavy task 3 started
>>>CPU heavy task 3 finished
>>>Finished in 2.78 seconds
```

Listing 18: Utskrift genererad av processorbunden kod körd på en tråd

För att testa hur ett processorbundet program påverkas när det exekveras i flera trådar skapades programmet i listning 19. Modulen *threading* användes för att skapa en ny tråd som exekverade funktionen *cpu_heavy_task*. Tre trådar skapades, x, y och z. Funktionen *join* användes för att huvudprogammet skulle invänta avslutad exekvering av trådarna innan programmet avslutade tidmätningen.

```
import time
from threading import Thread
from math import pow

def cpu_heavy_task():
    print('CPU heavy task started')
    i = pow(10,7)
    while(i>0):
        i = i - 1
    print('CPU heavy task finished')

if __name__ == "__main__":
    start_time = time.perf_counter()
    x = Thread(target=cpu_heavy_task)
    y = Thread(target=cpu_heavy_task)
    z = Thread(target=cpu_heavy_task)
    x.start()
    y.start()
    z.start()
    x.join()
    y.join()
    z.join()
    finish_time = time.perf_counter()
    print(f'Finished in {round(finish_time-start_time,2)} seconds')
```

Listing 19: Processorbunden kod körd på flera trådar

I listning 20 syns att programmet exekverades parallellt, där tre beräkningar startade innan någon tråd exekverade färdigt. Den totala exekveringens tiden uppmättes till 3.12 sekunder.

```
>>>CPU heavy task started
>>>CPU heavy task started
>>>CPU heavy task started
>>>CPU heavy task finished
>>>CPU heavy task finished
>>>CPU heavy task finished
>>>Finished in 3.12 seconds
```

Listing 20: Utskrift genererad av processorbunden kod körd på flera trådar

För att testa hur ett processorbundet program påverkas när det exekveras i flera processer skapades programmet i listning 21. Modulen *multiprocessing* användes för att skapa en nya processer som exekverade funktionen *cpu_heavy_task*. Tre trådar skapades, x, y och z. Funktionen *join* användes för att huvudprogammet skulle invänta avslutad exekvering av trådarna innan programmet avslutade tidmätningen.

```
import time
from multiprocessing import Process, set_start_method
from math import pow

def cpu_heavy_task():
    print('CPU heavy task started')
    i = pow(10,7)
    while(i>0):
        i = i - 1
    print('CPU heavy task finished')

if __name__ == "__main__":
    set_start_method('spawn')
    start_time = time.perf_counter()
    x = Process(target=cpu_heavy_task)
    y = Process(target=cpu_heavy_task)
    z = Process(target=cpu_heavy_task)
    x.start()
    y.start()
    z.start()
    x.join()
    y.join()
    z.join()
    finish_time = time.perf_counter()
    print(f'Finished in {round(finish_time-start_time,2)} seconds')
```

Listing 21: Processorbunden kod körd på flera processer

I listning 22 syns att programmet exekverades parallellt, där tre beräkningar startade innan någon tråd exekverade färdigt. Den totala exekveringenstiden uppmättes till 1.26 sekunder.

```
>>>CPU heavy task started
>>>CPU heavy task started
>>>CPU heavy task started
>>>CPU heavy task finished
>>>CPU heavy task finished
>>>CPU heavy task finished
>>>Finished in 1.26 seconds
```

Listing 22: Utskrift genererad av processorbunden kod körd på flera trådar

3.2 Test av parallell programmering i C

I detta avsnitt undersöks vilka prestandaeffekter parallell programmering i C kan ha på program som är IO-bundna respektive processorbundna. Testerna visar att både multitrådning och multiprogrammering ger stora prestandavinster i båda fallen.

3.2.1 *Testning av IO-bundna program i C*

En av de primära användningsområdena för multitrådning är i program som är IO-begränsade. För att visa hur ett sådant program kan minska sin exekveringstid genom användande av multitrådning har ett enkelt program skapats, se listning 23. Funktionen fetch_package ska simulera beteendet i ett system där en komponent behöver avvakta i väntan på svar från upp till tre andra långsamma komponenter före vidare exekvering. Därför är funktionens uppgift att skriva ut ett meddelande om att en förfrågan om pakethämtning har skickats, vila en sekund, och sedan avsluta funktionen med meddelandet att paketet har hämtats. Vilotiden representerar svarstiden från den långsamma komponenten. En svarstid på en sekund är naturligtvis orimligt lång väntetid i ett verkligt system, och vid praktiskt användande måste programmeraren individuellt bedöma om vilotiden är lång nog för att multitrådning ska vara ett lämpligt alternativ.

```c
void *fetch_package()
{
    /* Simulate fetching a packages of data from a slow transmitter */

    printf("Sent request for new package\n");
    sleep(1);
    printf("Received data package\n");

    return NULL;
}
```

Listing 23: Exempel på hämtning av datapaket

För att bedöma prestandaförbättringen skrevs ett program där hämtningsfunktionen först exekveras tre gånger sekvensiellt, och tre gånger parallellt. Den parallella exekveringen skedde genom dels genom skapandet av tre trådar som exekverar fetch_package en gång vardera, dels genom skapandet av tre processer som exekverar fetch_package en gång vardera. Programkoden i fråga visas i listning 24 och resultatet vid exekvering i listning 25. Man kan där observera att tiden det tog att utföra uppgiften sekvensiellt var 3.00 sekunder, parallellt med trådar 1.00 sekunder och parallellt med processer 1.00 sekunder. Den artificiella väntetiden på 1 sekund gör att man tydligt kan identifiera väntetiden till den begränsande faktorn i programmets exekvering. Det syns också tydligt att en parallell lösningsmetodik är lämpligt att utnyttja vid IO-bundna problem, även om det är svårt att utifrån detta experiment dra slutsatser om huruvida en multitrådad lösning är mer lämplig en än multiprogrammerad.

```c
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>


void *fetch_package()
{
```

```
    /* Simulate fetching a packages of data from a slow transmitter*/
    printf("Sent request for new package\n");
    sleep(1);
    printf("Received data package\n");

    return NULL;
}


/* Create child processes */
void make_worker()
{
    if (fork() == 0)
    {
        /* Is child process */
        fetch_package();
        exit(0);
    }
}


int main(int argc, char** argv)
{
    /* Setup for timing */
    struct timespec tval;
    double time_before, time_after;

    /* Sequential execution */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_before = tval.tv_sec + tval.tv_nsec/1e9;
    /* Fetch the packages */
    fetch_package();
    fetch_package();
    fetch_package();
    /* Measure time taken */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_after = tval.tv_sec + tval.tv_nsec/1e9;
    printf("Time taken with sequential execution: %fs \n\n",
                time_after-time_before);


    /* With threads */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_before = tval.tv_sec + tval.tv_nsec/1e9;
    /* Fetch the packages by making three (3) threads. */
    pthread_t thread1, thread2, thread3;
    pthread_create(&thread1, NULL, fetch_package, NULL);
    pthread_create(&thread2, NULL, fetch_package, NULL);
    pthread_create(&thread3, NULL, fetch_package, NULL);
    /* Stall execution in main until all other threads have finished */
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    /* Measure time taken */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_after = tval.tv_sec + tval.tv_nsec/1e9;
    printf("Time taken with multithreaded execution: %fs \n\n", time_after-time_before);
```

```c


    /* With processes */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_before = tval.tv_sec + tval.tv_nsec/1e9;
    /* Fetch the packages by making three (3) processes. */
    for (int i=0;i<3;i++)
        make_worker();
    /* Measure time taken */
    while(wait(NULL) > 0) /* Wait for child processes to finish */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_after = tval.tv_sec + tval.tv_nsec/1e9;
    printf("Time taken with multiprogrammed execution: %fs \n", time_after-time_before);


    return 0;
}
```

Listing 24: Parallell och sekventiell exekvering av IO-bundet program.

```
Sent request for new package
Received data package
Sent request for new package
Received data package
Sent request for new package
Received data package
Time taken with sequential execution: 3.000596s

Sent request for new package
Sent request for new package
Sent request for new package
Received data package
Received data package
Received data package
Time taken with multithreaded execution: 1.000427s

Sent request for new package
Sent request for new package
Sent request for new package
Received data package
Received data package
Received data package
Time taken with multiprogrammed execution: 1.000529s
```

Listing 25: Utdata från exekverad kod i listning 24

3.3 Testning av processorbundna program i C

Paralleliserings effektivitet vid processorbundna program har undersökts genom multitrådning och multiprogrammering. Koden som nyttjades vid undersökningen visas i listning 26. I exemplet exekveras den beräkningstunga uppgiften fem gånger sekventiellt, av fem olika trådar som var för sig utför uppgiften en gång samt slutligen av fem separata processer som var för sig utför uppgiften en gång. Utskriften av programmet visas i listning 27.

```c
#include <time.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

void* CPU_heavy_task()
{
    int i;
    for (int j = 0; j<100000000;j++)
        i=i*i*i*i*i;

    return NULL;
}


void make_worker()
{
    if (fork()==0)
    {
        /* is child process */
        CPU_heavy_task();
        exit(0);
    }
}


int main(int argc, char** argv)
{
    /* Timing setup */
    struct timespec tval;
    double time_before, time_after;

    /* Sequential execution of CPU-heavy-task */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_before = tval.tv_sec + tval.tv_nsec/1e9;
    for (int i=0;i<5;i++)
        CPU_heavy_task();
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_after = tval.tv_sec + tval.tv_nsec/1e9;
    printf("Time taken with sequential execution: %.6f s\n\n",
            time_after - time_before);

    /* With multiple threads */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_before = tval.tv_sec + tval.tv_nsec/1e9;
    /* Create threads and execute task */
    pthread_t thread[5];
    for (int i = 0; i<5; i++)
        pthread_create(&thread[i], NULL, CPU_heavy_task, NULL);
    for (int i=0; i<5; i++)
        pthread_join(thread[i], NULL);
    /* Clocking */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_after = tval.tv_sec + tval.tv_nsec/1e9;
    printf("Time taken with multiple threads: %.6f s\n\n",
            time_after - time_before);
```

```


    /* With multiple processes */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_before = tval.tv_sec + tval.tv_nsec/1e9;
    /* Create processes and execute task */
    for (int i = 0; i<5; i++)
        make_worker();
    while(wait(NULL) > 0) /* Wait for child processes to finish */
    clock_gettime(CLOCK_MONOTONIC, &tval);
    time_after = tval.tv_sec + tval.tv_nsec/1e9;
    printf("Time taken with multiple processes: %.6f s\n\n",
            time_after - time_before);

    return 0;
}
```

Listing 26: Processorbunden kod exekverad sekventiellt, med multitrådning och med multiprogrammering

```
Time taken with sequential execution: 2.553172 s

Time taken with multiple threads: 0.835787 s

Time taken with multiple processes: 0.847574 s
```

Listing 27: Utdata från 26

I listning 27 syns det att nyttjande av parallelisering minskar exekveringstiden av programmet från 2.4 sekunder till 0.8 sekunder. Exemplet visar även att den multitrådade metoden och den multiprogrammerade har ungefär lika lång exekveringstid. Således kan man avgöra att en parallelisering av problemet minskar exekveringstiden avsevärt, samt att testet inte pålyser någon stor skillnad mellan lösningsmetodikerna multitrådning och multiprogrammering.

4 DISKUSSION OCH SLUTSATSER

4.1 Python

Vid parallell programmering i Python blir det tydligt när det är lämpligt att använda trådar respektive processer. Som nämnt i avsnitt 2.2.1 så hindrar Pythons *"Global interperter lock"* trådar från att exekvera parallellt på flera kärnor i processorn. Därav finns det ingen nytta med att multitråda program i Python som är processorbegränsade. Däremot finns det mycket att vinna på att multitråda program som är IO-begränsade. Det såg vi i undersökningen av IO-begränsade program i avsnitt 3.1.1 där exekveringstiden för sekventiell, multitrådad och multiprogrammerad körning blev 3.05, 1.04 respektive 1.11 sekunder. Vi kan dra slutsatsen att det största delen exekveringstiden spenderade på att vänta på IO. I det här fallet var skillnaden mellan multitrådad och multiprogrammerad exekvering endast 0.07 sekunder, alltså för liten för att dra någon slutsats om vilken som är snabbast. Enligt 2.2.3 går det att skapa nya processer med metoden *fork* som kräver få resurser och då bör skillnaderna bli ännu mindre. Men för stora program kan det tänkas att trådar kan vara snabbare på att hantera IO-begräsningar än processer eftersom att kopiera stora processer eller skapa nya kan vara tids- och resurskrävande.

Vid test av processorbunda program syns konsekvensen av *"Global interperter lock"* tydligt. Exekveringstiden för sekventiell, multitrådad och multiprogrammerad körning blev 2.78, 3.12 och 1.26 sekunder. Att skapa trådar och växla mellan trådar skapade en flaskhals som gjorde programmet långsammare än vid sekventiell körning. Via multiprogrammering kunde vi utnyttja flera kärnor i processorn och därmed blev multiprogrammerad $1,52$ och $1,86$ sekunder snabbare än sekventiell respektive multrådad körning.

Slutsatsen blir att vid parallellprogrammering av rent IO-bundna program är det oftast lämpligast att använda trådar. Medan vid processorbunda program blir trådar istället en flaskhals och flera processer bör istället skapas.

4.2 C

I programspråket C kan både trådar och processer exekveras på flera kärnor. Skillnaderna mellan multitrådning och multiprogrammering blir därmed inte lika tydliga som för Python. I avsnitt 3.2.1 undersöktes effekterna av parallell programmering på ett IO-bundet program. Exekveringstiderna för sekventiell, multitrådad och multiprogrammerad körning blev 3.000596, 1.000427 respektive 1.000529 sekunder. Vi kan avläsa att stora prestandavinster görs med både multitrådning och multiprogrammering. Ett motsvarande resultat erhölls vid undersökning av det processorbundna programmet i avsnitt 3.3. Exekveringstiderna för sekventiell, multitrådad och multiprogrammerad körning blev 2.414844, 0.780055 respektive 0.770522 sekunder. Återigen erhölls en stor prestanda ökning för både multitrådning och multiprogrammering. Både för det IO-bundna och det processorbundna programmet kan ingen nämnvärd prestanda skillnad mellan multitrådning och multiprogrammering observeras.

I avsnitt 2.1.3 och 2.1.4 beskrivs multitrådning och multiprogrammering och dess skillnader. En viktig skillnad som lyfts fram är att varje tråd delar på alla resurser, medan processer erhåller sina egna resurser. Slutsatsen är således att vid parallell programmering i C så finns inga betydande prestandaskillnader mellan multitrådning och multiprogrammering. Val av lämplig metod bör istället avgöras av hur programmeraren vill att resurserna ska fördelas.

REFERENSER

[1] B. Barney, "Posix threads programming," 2021, accessed 2021-02-23. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/

[2] E. L. Lamie, "Mutual exclusion challenges and considerations," in *Real-time embedded multithreading*. Manhasset, NY, USA:CMP Books, 2015.

[3] F. S. Foundation, "Creating a pipe," accessed 2021-04-07. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Creating-a-Pipe.html

[4] F. S. Foundation, "Creating a pipe," acessed 2021-04-07. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/FIFO-Special-Files.html

[5] P. S. Foundation, *Glossary*, 2021. [Online]. Available: https://docs.python.org/release/3.9.1/glossary.html

[6] D. Beazley, "The python gil visualized," 2010. [Online]. Available: http://dabeaz.blogspot.com/2010/01/python-gil-visualized.html

[7] P. S. Foundation, *Initialization, Finalization, and Threads*, 2021. [Online]. Available: https://docs.python.org/3/c-api/init.html

[8] A. Jesse, "Grok the gil: How to write fast and thread-safe python," 2017. [Online]. Available: https://opensource.com/article/17/4/grok-gil

[9] P. S. Foundation, *multiprocessing — Process-based parallelism*, 2021. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html

[10] P. S. Foundation, *threading — Thread-based parallelism*, 2021. [Online]. Available: https://docs.python.org/3/library/threading.html

[11] P. S. Foundation, *concurrent.futures — Launching parallel tasks*, 2021. [Online]. Available: https://docs.python.org/3/library/threading.html

[12] W. Gay, "CPU," in *Advanced Raspberry Pi: Raspbian Linux and GPIO Integration*, 2:nd ed. New York City, NY, USA:Apress, 2018, sec. Threads and Mutexes.

[13] F. S. Foundation, "Creating a process," acessed 2021-04-07. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Creating-a-Process.html

H DESIGNSPECIFIKATION

Designspecifikation

Henrik Ahlinder
Edvin Johansson
Mikael Josefsson
Sebastian Andersson
Martin Dahl
Charbel Youssef

7 juni 2021

Version 1.02

Status

Granskad	Projektgruppen	2021-03-11
Godkänd	Peter Johansson	2021-03-12

Projektidentitet

Grupp E-post: TSEA56_2021VT_O2_G10@groups.liu.se

Beställare: Anders Nilsson, Linköpings universitet
Tfn: 013-28 26 35
E-post: anders.p.nilsson@liu.se

Handledare: Peter Johansson
Tfn: +4613281345
E-post: peter.a.johansson@liu.se

Kursansvarig: Mattias Krysander
Tfn: 013-28 21 98
E-post: mattias.krysander@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Henrik Ahlinder	Projektledare (PL)	henah490@student.liu.se
Mikael Josefsson	Kommunikationsmodulansvarig, GUI-ansvarig	mikjo695@student.liu.se
Sebastian Andersson	Testansvarig	seban218@student.liu.se
Martin Dahl	Dokumentansvarig, mjukvaruansvarig	marda545@student.liu.se
Charbel Youssef	Sensormodulansvarig	chayo642@student.liu.se
Edvin Johansson	Styrmodulansvarig	edvjo404@student.liu.se

INNEHÅLL

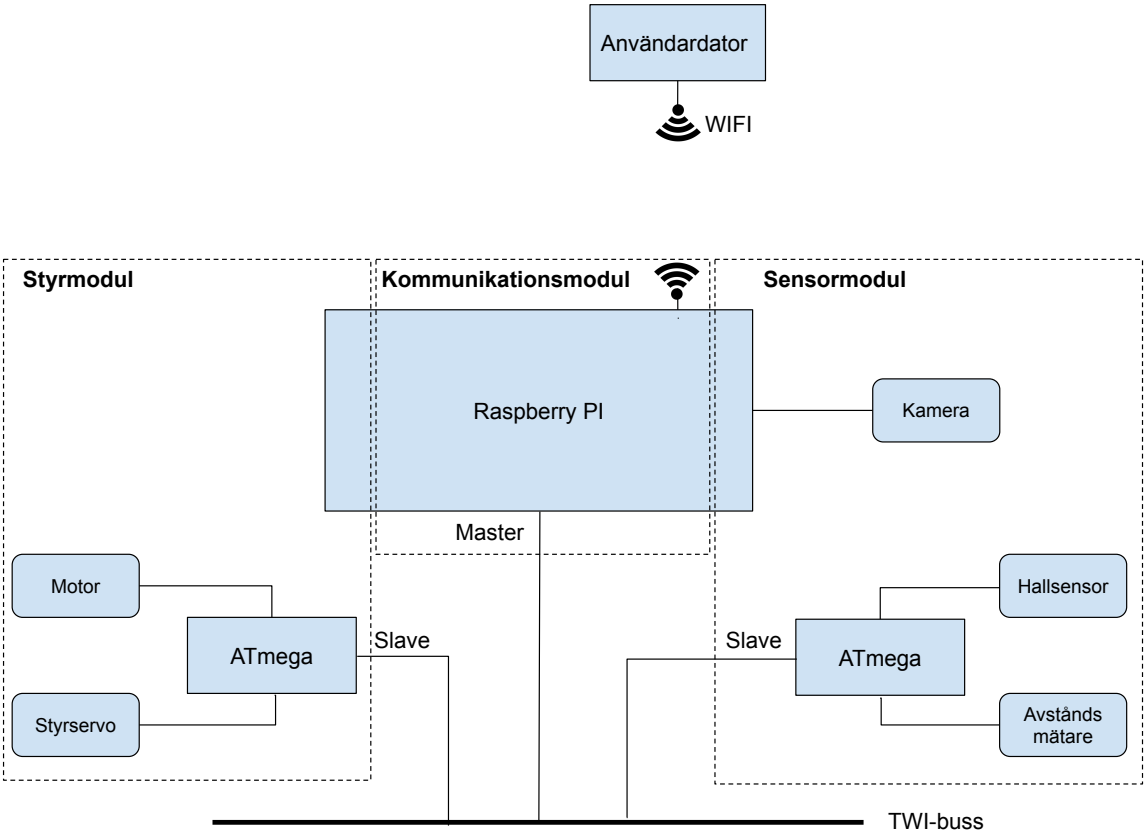
1 Sammanfattning 1
2 Delsystem 2
2.1 Kommunikationsmodulen . . . 2
2.2 Styrmodul . . . 4
2.3 Sensormodul och kamera . . . 11
3 Intermodulär kommunikation 17
3.1 Kommunikation mellan mikroprocessorerna . . . 17
3.2 Kommunikation mellan kommunikationsmodul och användardatorn . . . 17
3.3 Data . . . 19
4 Användardator 20
5 Implementationsstrategi 21
5.1 Testning . . . 21
5.2 Sampling . . . 21
Referenser 22

DOKUMENTHISTORIK

Version	Datum	Utförda förändringar	Utförda av	Granskad
0.1	2021-03-03	Första utkast	Alla	Alla

1 SAMMANFATTNING

En taxibil som ska köras i tre olika lägen (manuellt, semi-autonomt och helautonom) ska konstrueras. För att underlätta konstruktionsarbetet delas konstruktionen upp fyra moduler. En översiktsbild av hur dessa moduler är distribuerade ges i figur 1. Sensormodulen, kommunikationsmodulen och styrmodulen sitter fysiskt monterade på bilen medan modulen användardator beskriver användargränssnittet på användarens bärbara dator. På taxibilen finns tre processorer som används i olika mån av de olika modulerna. Vilka processorer som nyttjas i de olika modulerna visas i figuren.



Figur 1: Översiktbild av systemet

Bilen konstrueras i ett förbyggt chassi med förmonterade styrservon, motorer och halleffektsensorer. Dessa placeras alltså inte av projektgruppen. Utöver detta nyttjas en ultraljudssensor och en kamera för att generera information som används av styrmodulen i regleringen av bilen. Sensorernas placering och funktion beskrivs i kapitel 2.3. Reglering och styrning av bilen beskrivs i kapitel 2.2. För att möjliggöra kommunikation mellan tre övriga moduler introduceras kommunikationsmodulen i kapitel 2.1. Kommunikationsmodulens syfte är att upprätta och tillhandahålla kommunikationsmöjligheter mellan de övriga modulerna. Kommunikationsprotokoll och övriga detaljer om kommunikation mellan moduler beskrivs i kapitel 3.

Sammantaget ger moduluppdelningen en bra överblick för konstruktionen och delar upp projektet i tydliga delar och milstolpar som ger en god möjlighet att på ett bra vis se status i projektet samt punkter som potentiellt kan skapa flaskhalsar. Vidare i dokumentet beskrivs de olika modulerna i detalj med hårdvaruschema, mjukvara och funktionalitet. Dokumentet avslutas med en beskrivning av implementationsstrategin i kapitel 5.

2 DELSYSTEM

Nedan följer en detaljerad beskrivning av systemets uppbyggnad och funktion. För att underlätta läsandet delas detta avsnitt in i beskrivningar av de olika modulerna, följt av ett avslutande kapitel där kommunikation mellan modulerna beskrivs.

2.1 Kommunikationsmodulen

Kommunikationsmodulens funktion i systemet är att kommunicera datainformation mellan övriga delsystem genom att agera Master i TWI-systemet och access point för webbservern, se kapitel 3. Kommunikationsmodulen drivs av en mikrodator av typ Raspberry Pi 3B+. Notera att det är samma mikrodator som även driver kamera och utför bildbehandling, dessa delar tillhör inte kommunikationsmodulen utan beskrivs under kapitel 2.3.

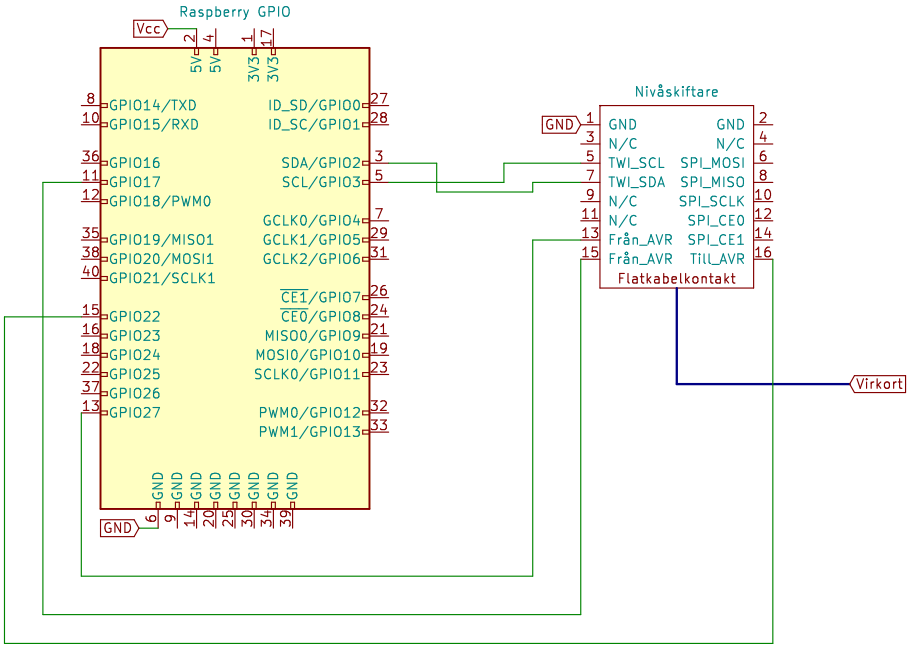
2.1.1 *Hårdvara och kopplingsschema*

I figur 2 visas hur GPIO-pinnarna i mikrodatorn Raspberry Pi ska sammankopplas med övriga systemet via en nivåskiftare. Nivåskiftaren krävs eftersom GPIO-pinnar drivs av 3.3V-logik på mikrodatorn, men 5V-logik på övriga mikroprocessorer. AVR-RPI-anslutningar redovisas i tabell 1.

RPI-pin	Nivåskiftare pin	Destination
GPIO17	15	Styrmodul
GPIO27	13	Sensormodul
GPIO22	16	LED

Tabell 1: Sammanslutning mellan AVR och nivåskiftare, se figur 2.

Notera att JTAG-anslutning saknas på denna modul eftersom mikrodatorn programmeras med hjälp av USB-anslutet tangentbord samt HDMI-ansluten skärm. Mikrodatorn drivs av ett operativsystem som installeras på ett microSD-kort som ansluts till datorn. Komponentbudgeten visas i 2.



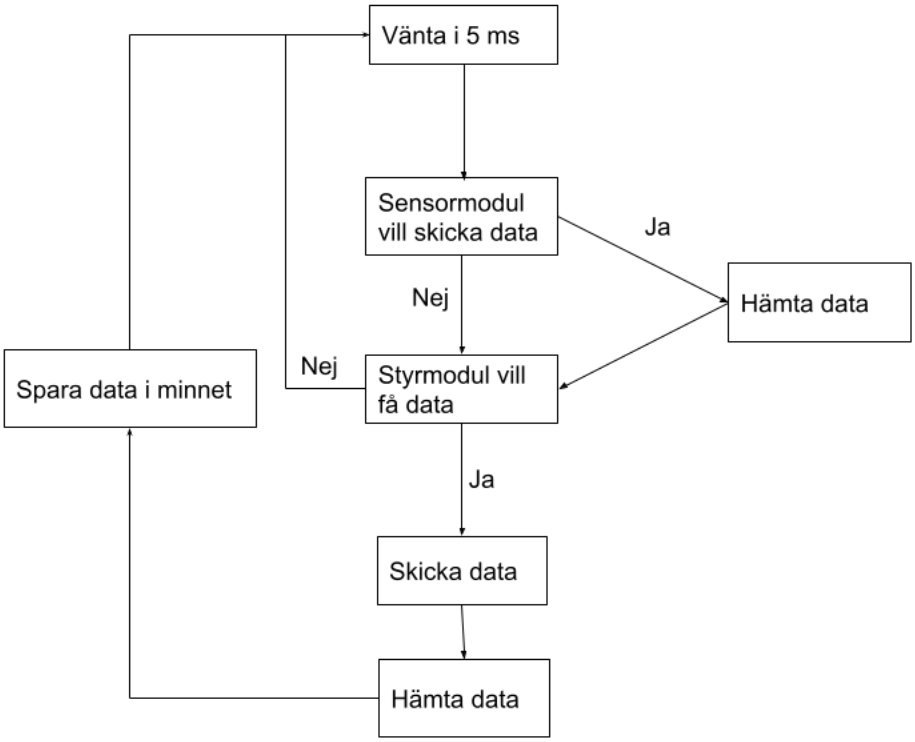
Figur 2: GPIO-kontakter på kommunikationsmodulen

Användningsområde	Antal (st)	Komponent
Programmering	1	Datorskärm med HDMI-kontakt
	1	HDMI-kabel
	1	USB-anslutet tangentbord
	1	USB-ansluten datormus
	1	microSD-kort (minst 8GB)
Exekvering	1	Nivåskiftare
	1	16-pin Flatkabel
	1	Strömkälla 5V
	1	Raspberry Pi 3B+

Tabell 2: Komponentbudget kommunikationsmodul

2.1.2 *Mjukvara*

Kommunikationsmodulen kommer att exekvera en huvudloop där den väntar ett fixt antal millisekunder innan den försöker initiera kommunikation mellan modulerna. Data kommer att hämtas och skickas när sensor- och styrmodul signalerar via IRQ, att de är redo för att skicka/motta data över TWI-bussen. Flödet i huvudloopen visas i figur 3.



Figur 3: Flödesschema huvudloop i kommunikationsmodulen

2.2 Styrmodul

Styrmodulens uppgift är att givet sensordata styra bilen till dess destination inom ett väldefinierat vägnät. Detta utförs med olika grad av autonomi beroende på vilken körläge bilen befinner sig i :

- **Manuellt läge** : Bilens motorer och styrservo styrs helt av en användare på användardatorn. Styrmodulen hanterar dock motorer och styrservo på låg nivå: Användaren behöver enbart önska ett gaspådrag och ett styrutslag, styrmodulen hanterar pulsbreddmodulering.
- **Semi-Autonomt läge** : Användaren kan önska att bilen kör fram till nästa korsning, sväng eller stopplinje. Styrmodulen beräknar själv lämpliga gaspådrag och styrutslag.
- **Helautonomt läge** : Användaren kan önska komplicerade köruppdrag, till exempel köra snabbaste vägen till en viss stopplinje. Styrmodulen beräknar då den snabbaste vägen samt de gaspådrag och styrutslag som krävs för att ta sig dit.

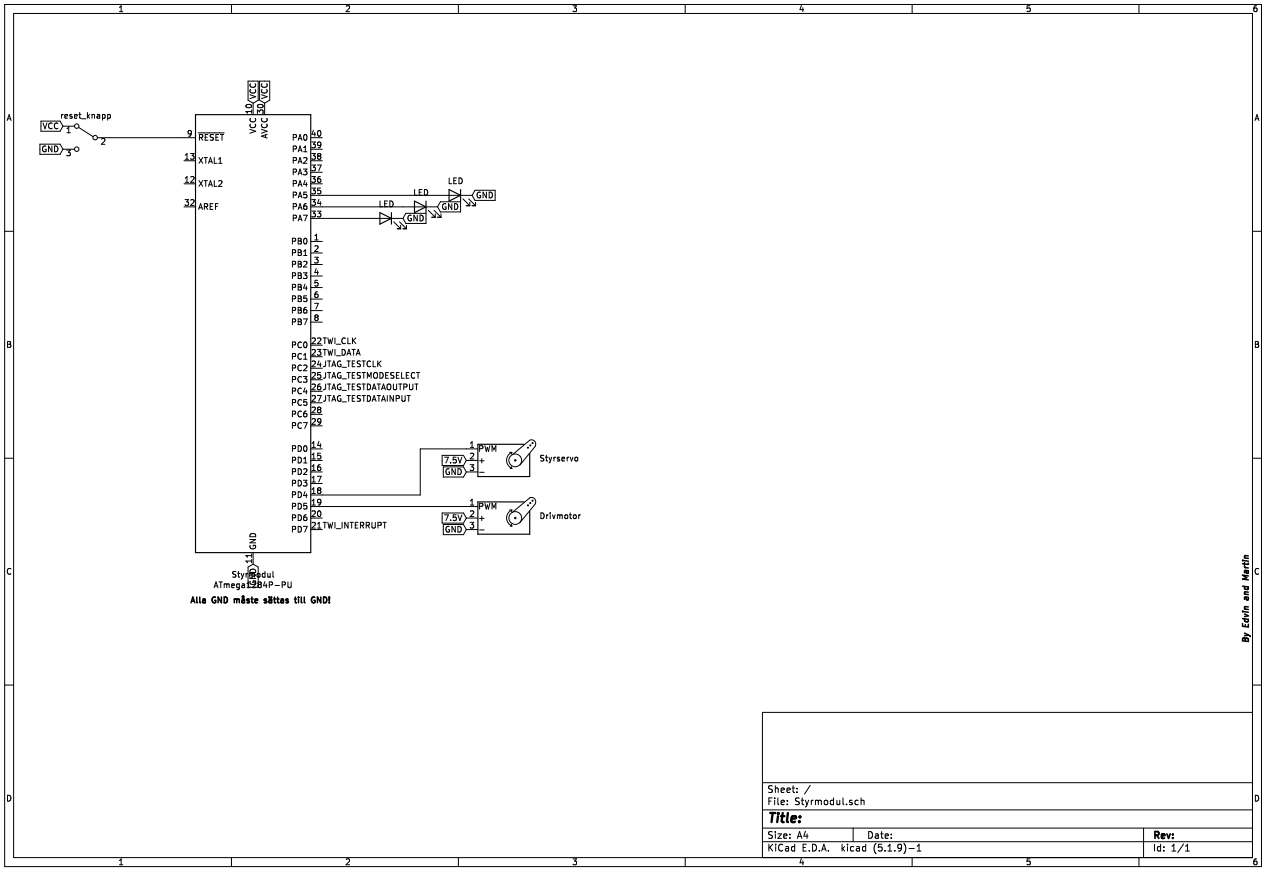
Styrmodulen omfattar alltså system för att hantera motorer, autonom styrning längs med väglinjer samt beslutsfattning för navigation.

2.2.1 *Hårdvara*

Följande komponenter krävs för att realisera styrmodulen:

- **ATmega1284P** : Den mikroprocessor som används.
- **Styrservo** : Det styrservo som används för att realisera bilens styrutslag, som är en av de två utsignaler som beräknas i styrmodulens regulator.
- **Motor** : Den motor som används för att realisera bilens gaspådrag, som är den andra av de två utsignaler som beräknas i styrmodulens regulator.
- **Tre lysdioder** : Tre lysdioder används för att indikera vilken körläge som bilen befinner sig i.
- **Resetknapp** : En knapp som används för att starta om programmet i styrmodulen.

Vid val av mikroprocessor i styrenheten är val av regulator den största påverkande faktorn. Beroende på hur avancerad den använda regulatorn är så ställs olika höga krav på mikroprocessorns beräkningskraft. Olika regulatorer kan kräva olika uppdateringsfrekvens och det är även stor skillnad i hur många beräkningar som krävs per iteration. I skrivande stund finns tre regulatorer som efter en litteraturstudie visat sig lovande. Det är svårt att på förhand avgöra vilken av dem som är bäst lämpad, så implementation och testning krävs för att avgöra vilken som slutligen ska användas. Därför används mikroprocessorn ATmega1284P i styrmodulen, som bedöms ha tillräckligt med beräkningskraft för att kunna hantera alla regulatorer med god uppdateringsfrekvens. ATmega1284P har även mer än tillräckligt med minne och I/O-portar. Portarnas användning visualiseras i figur 4 nedan.

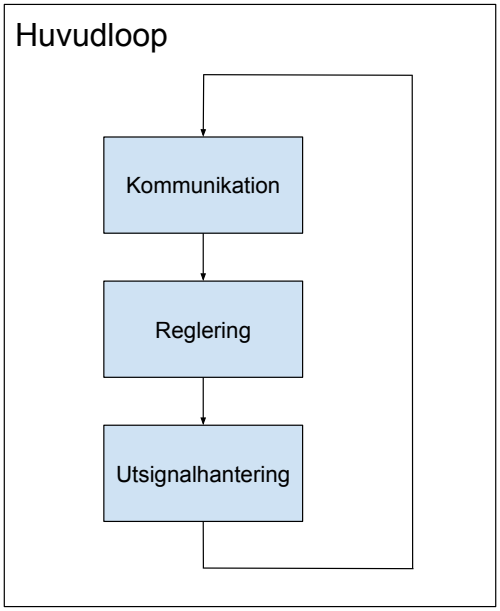


Figur 4: Kopplingsschema för styrmodulen

2.2.2 *Mjukvara*

Mjukvaran implementeras i första hand på Atmega1284P-kortet men en del av programmet implementeras på kommunikationsmodulens Rapsberry Pi. En översiktlig skiss över programmets flöde visas i figur 5, där de ingående blocken beskrivs mer detaljerat under respektive figur.

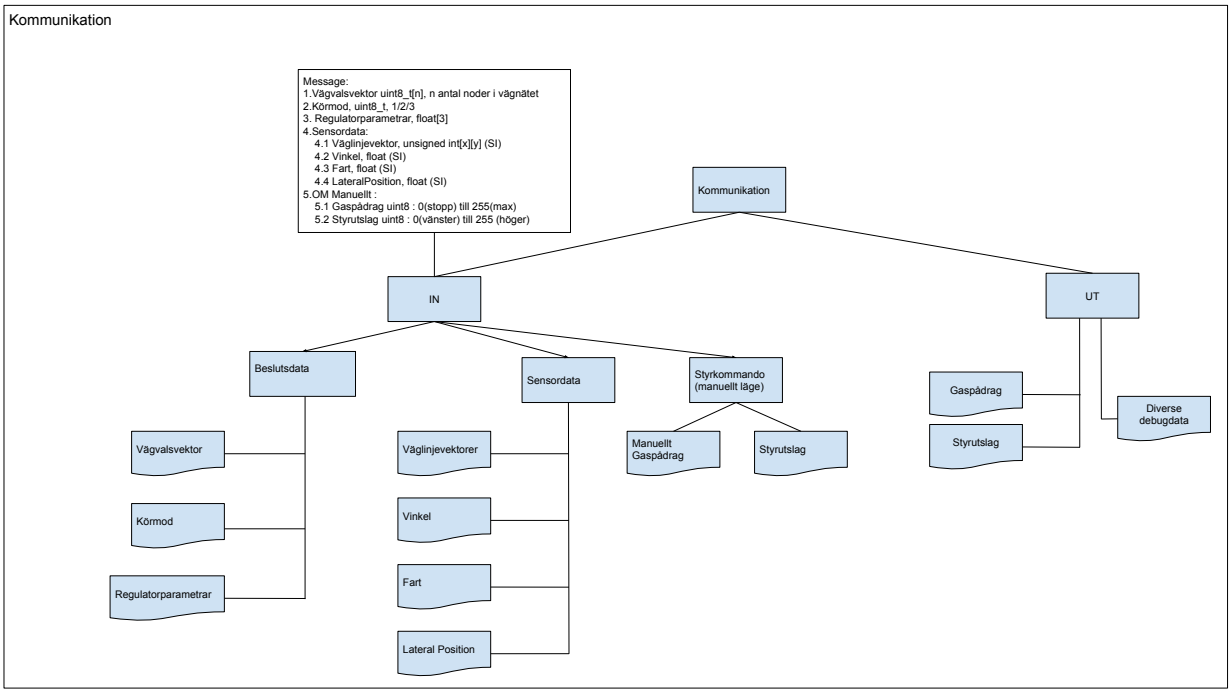
Huvudloop



Figur 5: Styrmodulens huvudloop

- **Huvudloop** : Hanterar hela styrmodulens programflöde.
- **Kommunikation** : Styrmodulens kommunikationsblock.
- **Reglering** : Styrmodulens reglerblock, implementerar en tillståndsmaskin, styr- och beslutstagningsalgoritmer.
- **Utsignalshantering** : Konverterar önskat gaspådrag och servoutslag till PWM-signaler och skickar dessa till motor och styrservo.
- **Loop** : Slutligen återstartas loopen med att hämna in ny data via kommunikationblocket.

Kommunikation

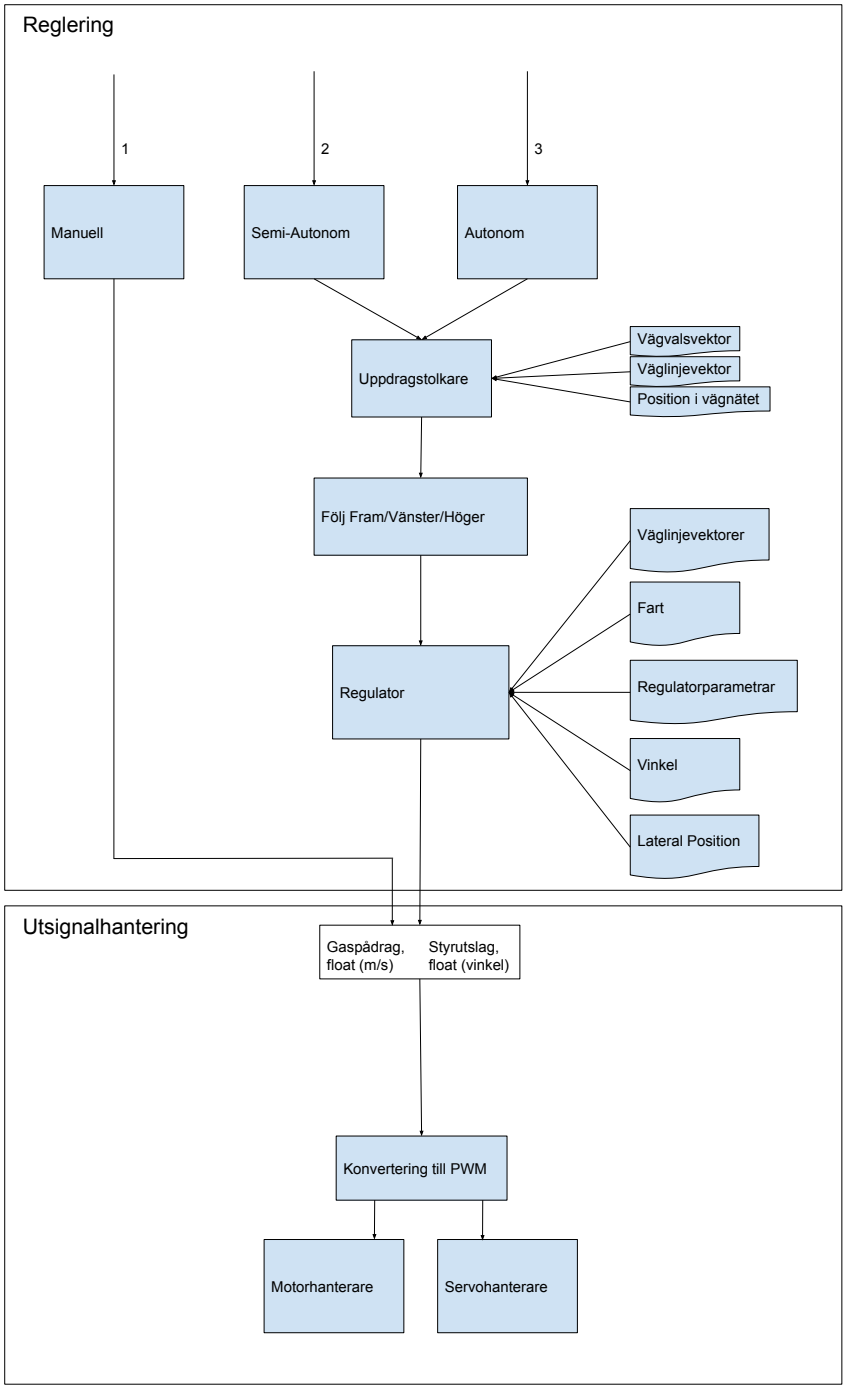


Figur 6: Styrmodulens kommunikationsflöde

I figur 6 illustreras ordningen för kommunikationsflödet.

- **IN/UT** : Uppdelning av det som kommer in i styrmodulen mot det som går ut ur styrmodulen.
- **Beslutsdata** : Värden från kommunikationsmodulen (som vissa har sitt ursprung i användardatorn) för att ställa in körmod, parametrar till regulatorer och information om hur bilen ska navigera i vägnätet.
- **Sensordata** : Värden från sensormodulen.
- **Styrkommando (manuellt läge)** : Denna information används när bilen körs i manuellt läge.
- **UT** : Styrmodulen rapporterar med vilket gaspådrag, styrutslag den har satt, samt eventuell debug-data.
- **Message** : Beskrivning av vilka datatyper som används i kommunikationsflödet och i vilken ordning de tas emot.

Reglering och utsignal



Figur 7: Styrmodulens regler och beslut flöde

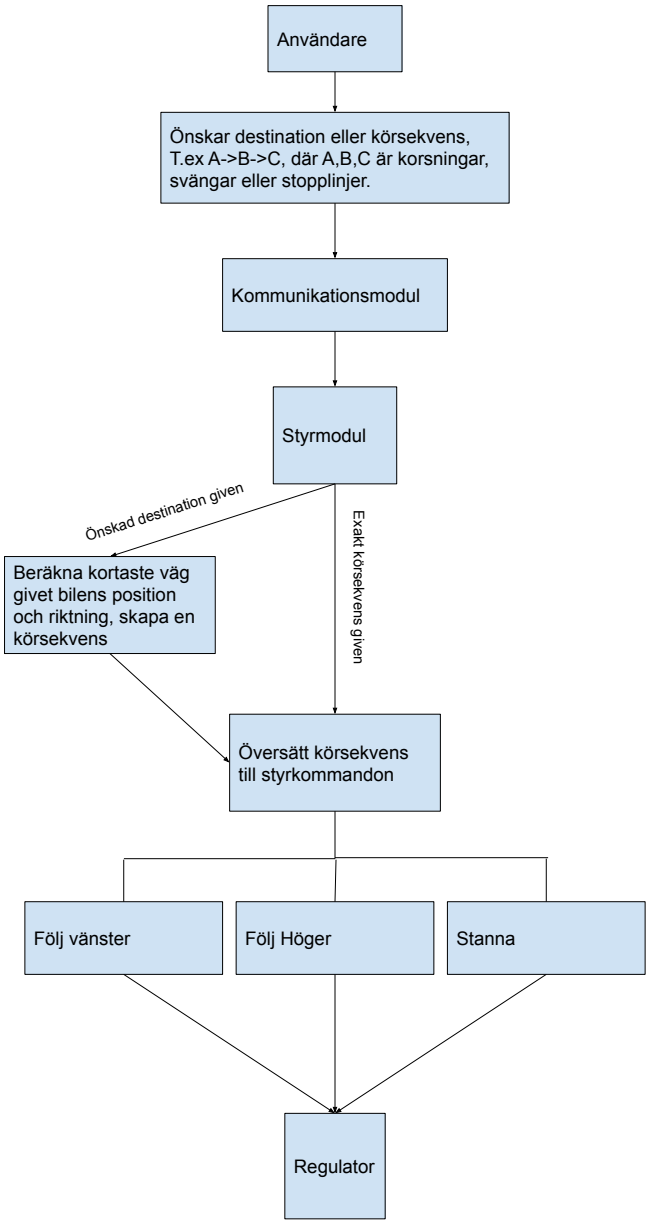
- **Manuellt/Semi-Autonomt/Helautonomt** : Demonstrerar vilket flöde som sker för respektive körläge.
- **Uppdragstolkare** : Omvandlar en start- och en slutdestination till en sekvens av beslut som kan tolkas av regulatorn, t.ex. ”följ vägen framåt”, ”följ vägen vänster” eller ”stanna vid nästa stopplinje”. Denna del kan komma att implementeras på kommunikationsmodulens Raspberry Pi för att få tillgång till färdiga bibliotek som kan utföra t.ex. Dijkstra’s algoritm på vägnätet.
- **Regulator** : Abstraktion av regulatorn och den sensordata som den behöver ses till höger om blocket. Regulatorn är den del av styrmodulen som ser till att bilen följer den önskade referenslinjen. Regulatorn beskrivs i mer detalj nedan, bland annat med Figur 9.

- **Utsignalshantering** : Konverterar önskat gaspådrag och styrutslag till PWM-signaler som skickas till motor och styrservo.

Vägval utifrån uppdrag

I Figur 8 beskrivs flödet från användarens val av destination till det styrbeteende som skickas till i regulatorn.

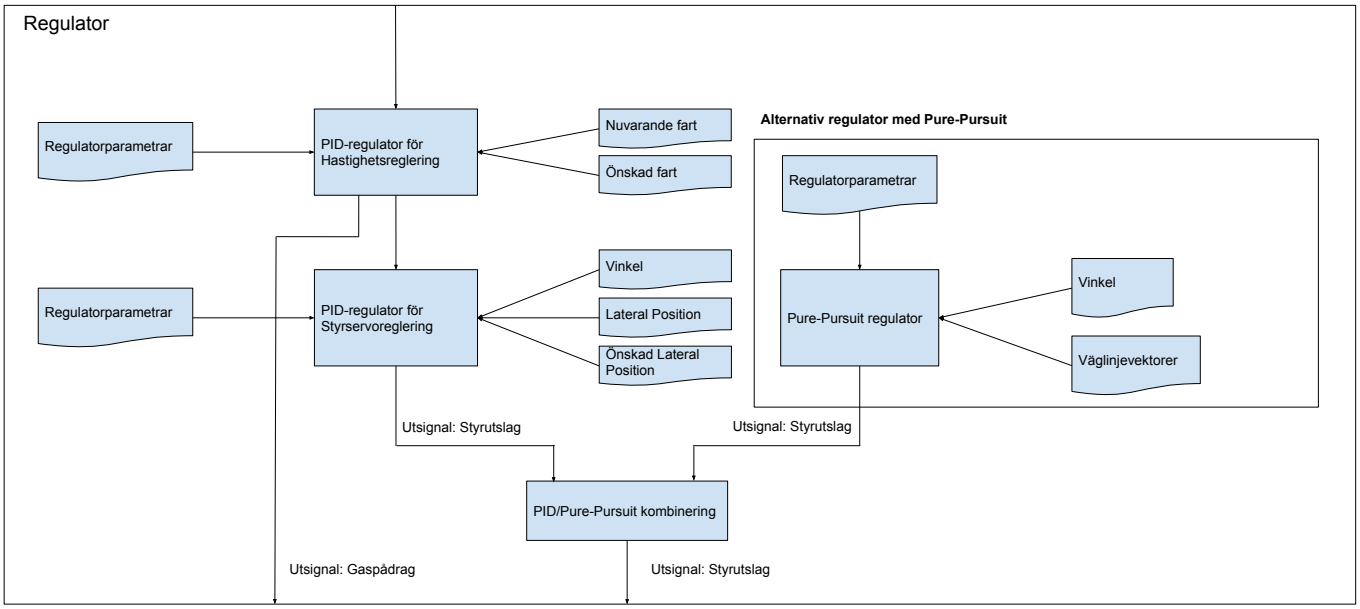
Om användaren anger en slutdestination, vilket den gör i styrläget ”helautonomt”, behöver bilen beräkna kortaste möjliga väg dit. Det kan göras snabbt med till exempel Dijkstra’s Algoritm. Sekvensen som sedan fås omvandlas till ett körbeteende som går ut på att i varje korsning följa antingen vänster eller höger väglinje, och att slutligen stanna.



Figur 8: Skiss över hur användarens val av slutdestination ger ett körbeteende

Regulator

Regulatorn består av två delar, en del för att reglera gaspådraget och en del för att reglera styrutslaget. Båda regulatorerna implementeras i form av PID-regulatorer. Ifall PID-regulatorn som reglerar styrutslaget visar sig ha för dålig prestanda kan en alternativ regulator av *Pure-pursuit*-typ komma att användas enskilt eller i kombination med PID-regulatorn. Hur dessa regulatorer fungerar och hur de kan implementeras finns beskrivet i dokumentet ”Förstudie - Regler” [1].



Figur 9: Styrmodulens regulator

2.3 Sensormodul och kamera

Sensormodulens funktion är att ta in data från de sensorer som är monterad på bilen och omvandla mätdatan till ett format som de andra modulerna kan tolka. De sensorer som används är två halleffektssensorer, även benämnda hallsensorer, och en ultraljudssensor. Hallsensorerna är till för att mäta bilens fart, genom att mäta vinkelhastigheten på bilens bakhjul. Ultraljudssensorns funktion är att läsa av avståndet till potentiella hinder som kan förekomma framför taxibilen. Den insamlade datan sparas och kan hämtas genom en förfrågan från kommunikationsmodulen. Sensordatan uppdateras hela tiden med ny mätdata och skriver över den gamla datan.

Sensormodulen använder sig av en kamera som monteras på Raspberry Pi:en. Där utförs även bildbehandling. Bildbehandlingen utför beräkningar på kamerans bildström av vägen framför bilen för att beräkna koordinater för vänster- och höger vägkant. Bildbehandlingen beräknar även vinkeln som taxibilen har relativt vägens mittlinje och bilens laterala avvikelse från mittlinjen.

Sensormodulen utför också beräkning av bilens position i vägnätet. Det görs med data från bildbehandlingen tillsammans med information om bilens hastighet, se figur 13. Bildbehandlingen används för att avgöra om bilen befinner sig i en sväng, korsning eller raksträcka; om bilen svänger och vilken vinkel den har relativt vägens mittlinje. Med bilens hastighet avgörs om bilen rör sig eller står stil.

2.3.1 *Hårdvara*

Följande komponenter ingår i sensormodulen:

- **ATmega1284P** : Mikroprocessorn som används för att ta emot sensordata.
- **2 st A1120** : Halleffektssensorer som ger data för att kunna avgöra bilens fart (endast en kommer användas).
- **SRF04** : Ultraljudssensor för att kunna se eventuella hinder och avståndet till dem. Den har en räckvidd på 3-300 cm.
- **Raspberry Pi Camera Module v2** : Kamera som tar bilder till bildbehandling. Kameran ska sitta i bilens front och vara vinklad neråt med en viss vinkel. Denna vinkel behöver experimentellt tas fram för kameran ska få en bra avvägning mellan att se vägen nära bilen och att se vägen långt fram.
- **Raspberry Pi** : Mikrodator för bildbehandling samt positioneringen (delas med kommunikationsmodulen).

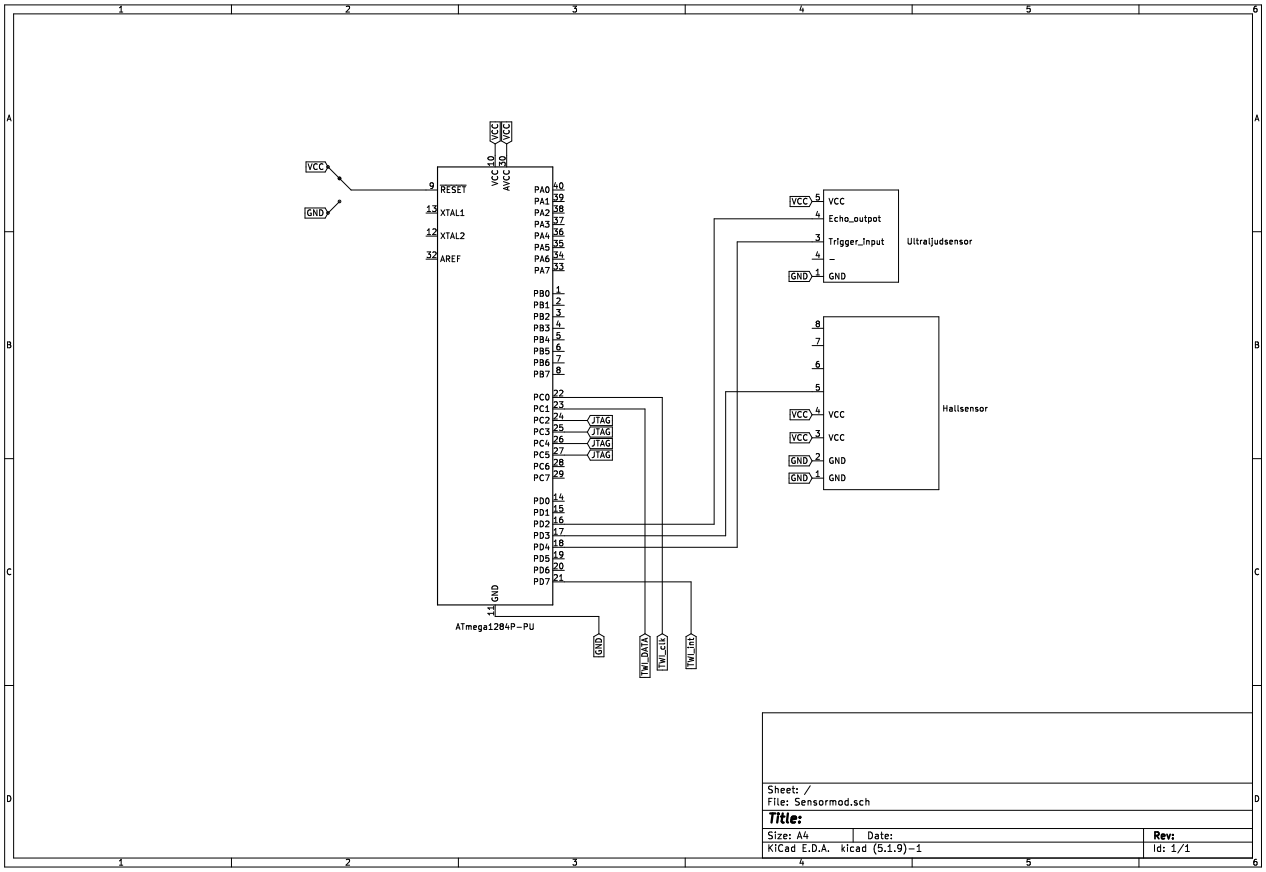
2.3.2 *Kopplingsschema för sensormodul*

Kopplingsshemat för sensormodulen (Raspberry Pi-delen är exkluderad) visas i figur 10. Funktionen hos de pinnar på ATmega1284P som används är följande:

- **PD7:** PD7 används som avbrottssignal från slaven (sensormodulen) till mastern (utifrån TWI-protokollet).
- **PC0:** PC0 är kopplad till klockan som ligger på bussen (enligt TWI-protokollet).
- **PC1:** PC1 är kopplad till databussen (enligt TWI-protokollet).
- **PC2-5:** PC2 fram till PC5 är kopplade till JTAG. Dessa pinnar har funktionalitet för att hantera JTAG.
- **PD2:** PD2 är kopplad till pinne 4 på ultraljudssensorn. Denna pinne är en avbrottspinne som aktiveras när utsignalen från ultraljudsensorn mottas.
- **PD3:** PD3 är kopplad till pinne 5 på hallsensorerna. Denna pinne är en avbrottspinne som kommer att starta ett avbrott när hallsensorn skickar en signal (negativ flank).

- **RESET**: RESET pinnen är kopplad till en knapp för att det ska vara möjligt att starta om programmet i sensormodulen.

Övriga pinnar används ej.



Figur 10: Kopplingsschema för Sensormodulen

2.3.3 *Mikroprocessor*

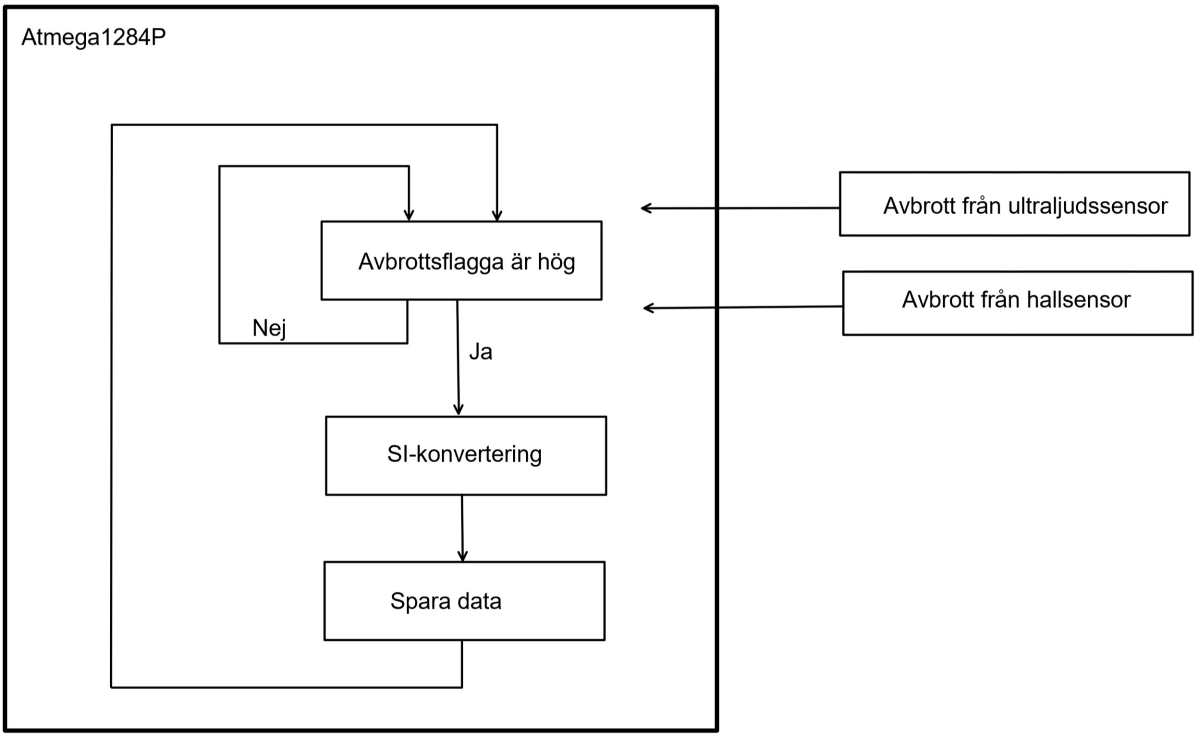
I sensormodulen ingår mikroprocessorn ATmega1284P som består av 32 programmerbara portar och har ett 128kB-flashminne. Enligt kopplingschema i figur 10 räcker antalet pinnar som finns i microprocessorn till de ingående signalena från de inkopplade sensorerna. 128kB-flashminne är tillräckligt för att kunna spara datan och köra det program som ska finnas på mikroprocessorn.

2.3.4 *Mjukvara*

Strukturen för mjukvaran för ATMega1284P illusterars i figur 11 och för Raspberry Pi:en i figur 12 och figur 13. Dessa figurer är flödesscheman som visar i vilken sekvens saker utförs.

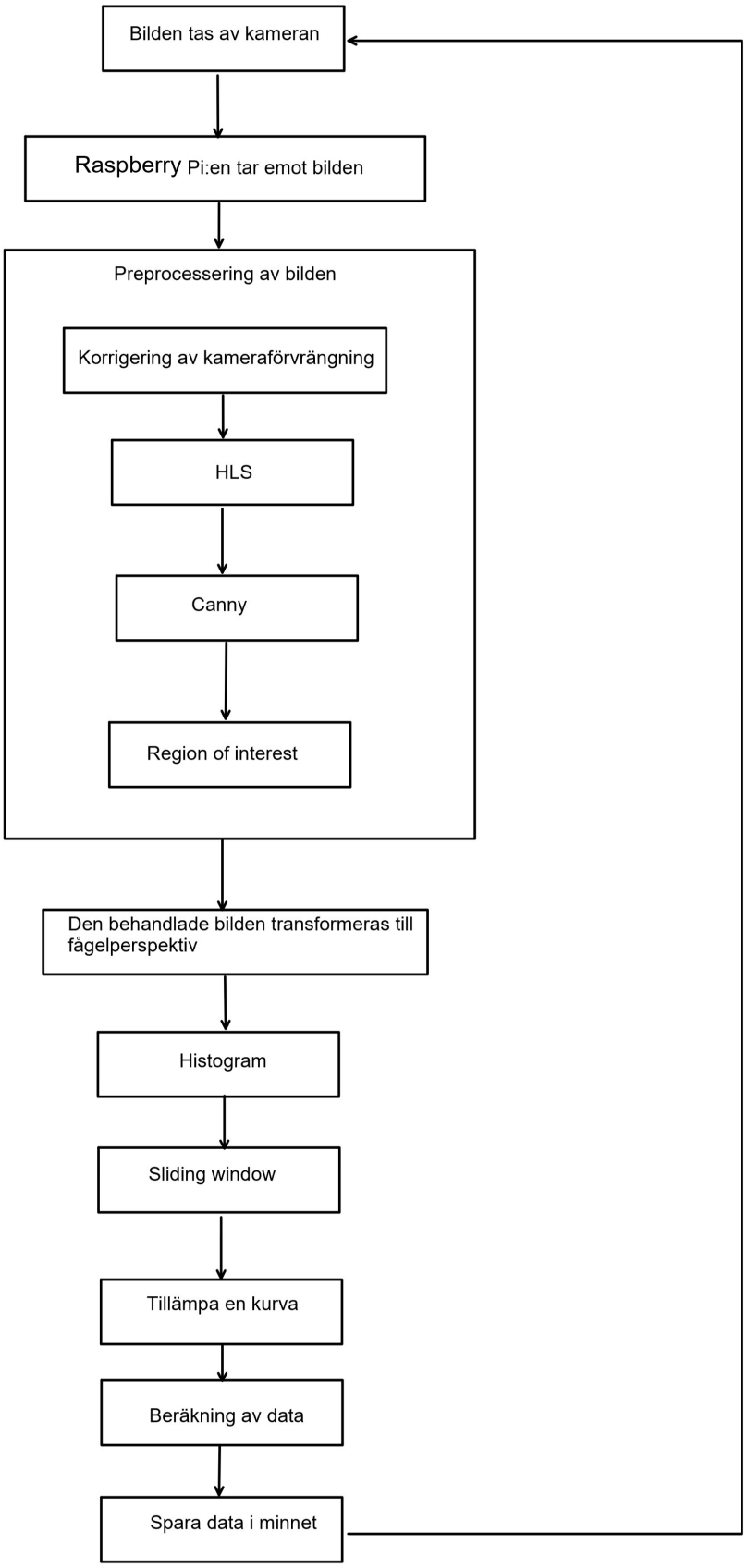
Programmet på ATmega1284P sköter sensorhantering. Programmet består av en huvudloop och ett antal avbrottsrutiner som indikerar när mätningar är slutförda. Varje gång bakhjulet har roterat ett visst antal grader så passerar en magnet hallsensorn, vilket gör att den skickar en puls. Dessa pulser aktiverar en avbrottsrutin som mäter tiden sedan det senaste avbrottet. Tiden omvandlas sedan till en hjulhastighet som lagras i minnet. Den andra avbrottsrutinen aktiveras då ultraljudssensorn skickar en signal att den är klar med en avståndsmätning. I den avbrottsrutinen sätts en flagga för att indikera att ultraljudssensorn har ny mätdata. Den tredje avbrottsrutinen är kopplad till TWI-kanalen och sätter en flagga som indikerar att kommunikationsmodulen vill ha sensordata.

I huvudloopen testas ifall någon av flaggorna är satta. Ifall flaggan från ultraljudssensorns avbrottsrutin är satt så läses ultraljudssensorna mätdata av, konverteras till SI-enhet, och lagras i minnet. Ifall inget hinder upptäcks inom ultraljudssensorns räckvidd (3-300 cm) sätts avståndet till NaN. Här startas även en ny mätning för ultraljudssensorn. Efterföljande mätningar från ultraljudssensorn sparar över den föregående mätningen så att endast den senaste mätningen finns sparad. Ifall flaggan från kommunikationsmodulen är satt så skickas den nuvarande farten och det senast uppmätta avståndet från ultraljudssensorn. Ifall ingen flagga är satt så görs ingenting i huvudloopen.



Figur 11: Flödesschema för Sensormodulens mikropocessor.

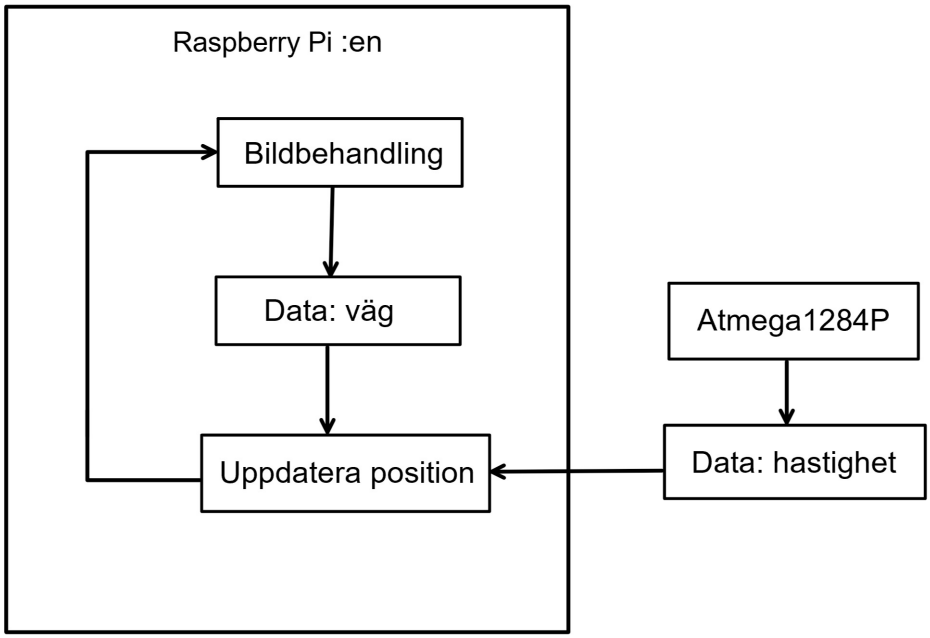
Programmet på Raspberry Pi:en hanterar bildbehandling, se figur 12, och positionering, se figur 13. Huvudloopen i detta program går i stora drag ut på att ta emot en bild från kameran, förprocessera bilden och till sist detektera vägkanter i bilden. När detta är utfört så beräknas vinkelfelet och den laterala positionen. Detta sparas sedan i minnet (gammal data skrivs över). Postioneringen använder denna data och data från ATmega1284P för att uppdatera bilens position i vägnätet.



Figur 12: Flödesschema för bildbehandling.

Trender i flödesschemat (figur 12)

- *Korrigering av kameraförvrängning*: Korrigera för förvrängning som uppstår från kamerans lins.
- *HLS*: (Nyans, ljusstyrka och mättnad) Specifika färger kan isoleras genom applicering av olika filter.
- *Canny*: Hitta stora förändringar i pixlarnas intensitet.
- *Region of interest*: Klipp ut ett område i bilden där vägen med stor sannolikhet finns. Detta sänker beräkningsmängden.
- *Fågelperspektiv*: En transformering av bilden till en vy som representerar vägen ovanifrån.
- *Histogram*: Ett histogram för antalet pixlar. Används för att hitta en startpunkt på ett fönster.
- *Sliding window*: Hitta vägkanterna i bilden genom att sätta ut en startpunkt på ett fönster och beroende på antalet pixlar med en viss intensitet bestäms nästa position för fönstret. Fönstret går uppåt tills hela bilden söks.



Figur 13: Flödesschema över positioneringsalgorithm

Sammantaget kommer sensormodulen alltså att bestå av två processorer. I ATmega1284p utförs:

- Beräkning av fart och avstånd till närmaste hinder, givet i SI-enheter.
- Beräkning av hur långt bilen har färdats.

I Rasperryn Pi:en utförs:

- Bildhämtning och bildbehandling.
- Positionering av bilen på kartan.

Eftersom bildbehandling kräver komplexa beräkningar kommer biblioteket *OpenCV*, som är specialiserat på bildbehandling, att användas. Bildbehandlingen sker kontinuerligt utan hänsyn till avbrott eller övriga processer som exekveras på mikrodatorn.

3 INTERMODULÄR KOMMUNIKATION

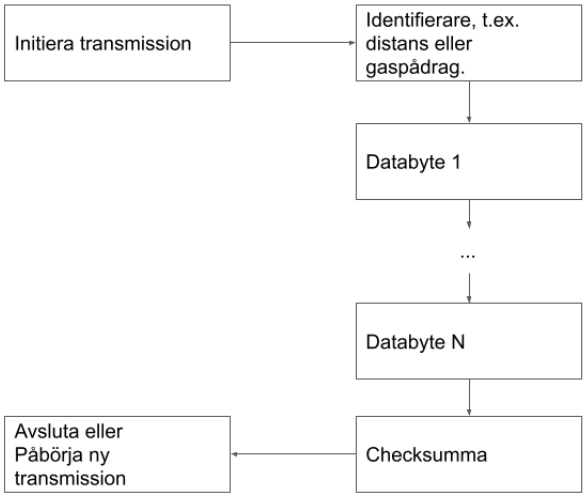
I detta kapitel beskrivs hur kommunikation mellan de olika delsystem sker. De tre delsystemen styrmodulen, kommunikationsmodulen och sensormodulen är fastmonterade på taxibilen och ska därför anslutas till varandra via en hårdvarubuss. Hur kommunikationen längs bussen sker beskrivs i kapitel 3.1. Utöver det ska det finnas en trådlös kommunikationskanal mellan kommunikationsmodulen och användardatorn. Denna trådlösa kommunikationskanal beskrivs i kapitel 3.2. Slutligen beskrivs vilken data som skickas mellan modulerna översiktligt i 3.3, samt vilken väg som datan färdas.

3.1 Kommunikation mellan mikroprocessorerna

Varje delsystem på bilen, d.v.s styrmodulen, sensormodulen och kommunikationsmodulen innehåller en mikroprocessor vardera. Detta avsnitts syfte är att utveckla och förklara hur kommunikationen mellan dessa tre delsystem sker.

De tre delsystem ansluter till varandra via en buss av TWI-typ med en enskild master. Master i systemet är kommunikationsmodulen och de övriga processorerna agerar som slavar. Detta innebär att all datainformation som ska transmitteras mellan de olika delsystem går via kommunikationsmodulen. Utöver denna buss kopplas även en extra avbrottskanal in mellan vardera slavprocessorer och master, se pin markerad TWI_int i bilderna på delsystemen. Denna extra kanal används av slaven för att signalera till master om en transmission ska initieras eller ej. Hur denna kanal nyttjas visas i figur 3, i kapitel 2.1.

När transmissionen har inletts kommer Varje sekvens av data kommer att föregås av en identifierare och avslutas med en checksumma. Varje datapaket som saknar identifierare eller har en felaktig checksumma kommer mottagaren be sändaren att skicka om genom att sända en NACK. För att genera checksumman utförs operationen XOR på identifieraren och samtliga databytes. Sekvens illustreras i figur 14.



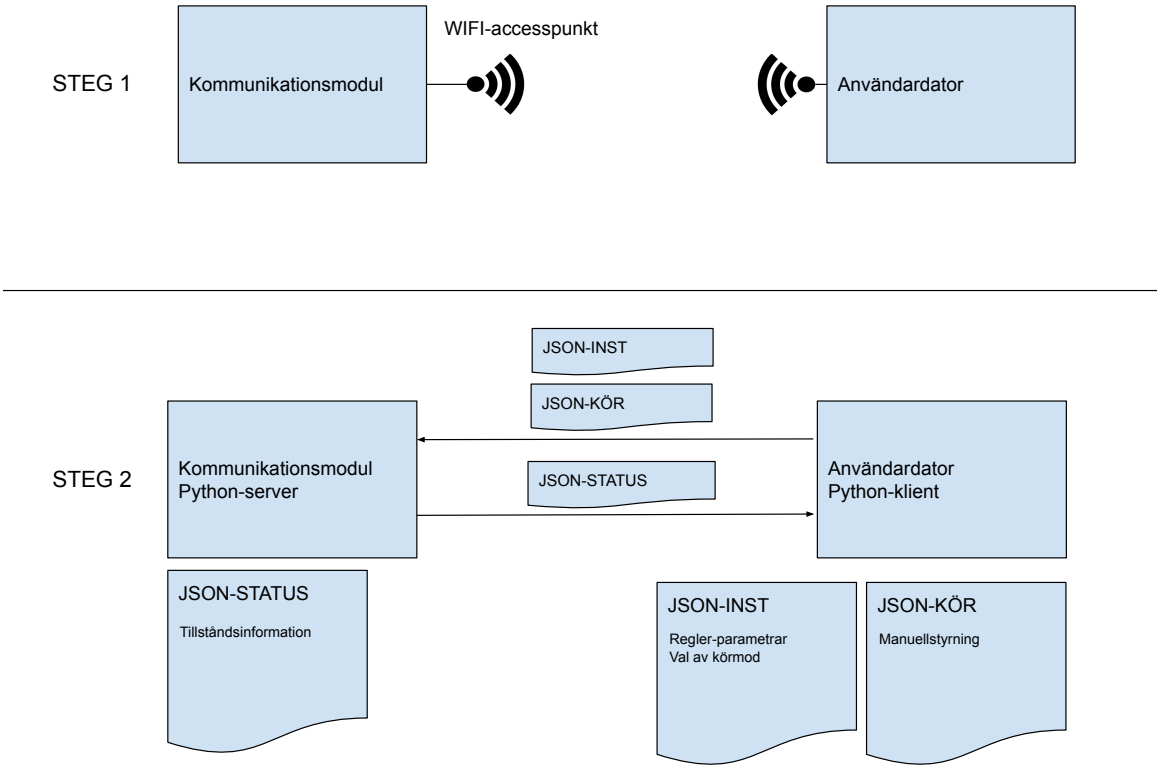
Figur 14: Flödesschema för kommunikationsprotokollet

Skulle Master inleda en transmission, men slaven inte svarar då den är upptagen, behandlas kommunkationen som icke-önskvärd och en ny transmission inleds först vid nästa cykel i kommunkationsmodulens huvudloop.

3.2 Kommunikation mellan kommunikationsmodul och användardatorn

Kommunikationsmodulen agerar accesspunkt och skapar ett WIFI-nätverk som användardatorn kan ansluta till. Kommunikationsmodulen kommer att ha en server skriven i Python som användardator sedan kan ansluta sig till. Användardatorn

interagerar med taxibilen via en klient skriven i Python som kommunicerar med servern. Datat mellan server och klient kommer att skickas i JSON-format.



Figur 15: Skiss av intermodulär kommunikation

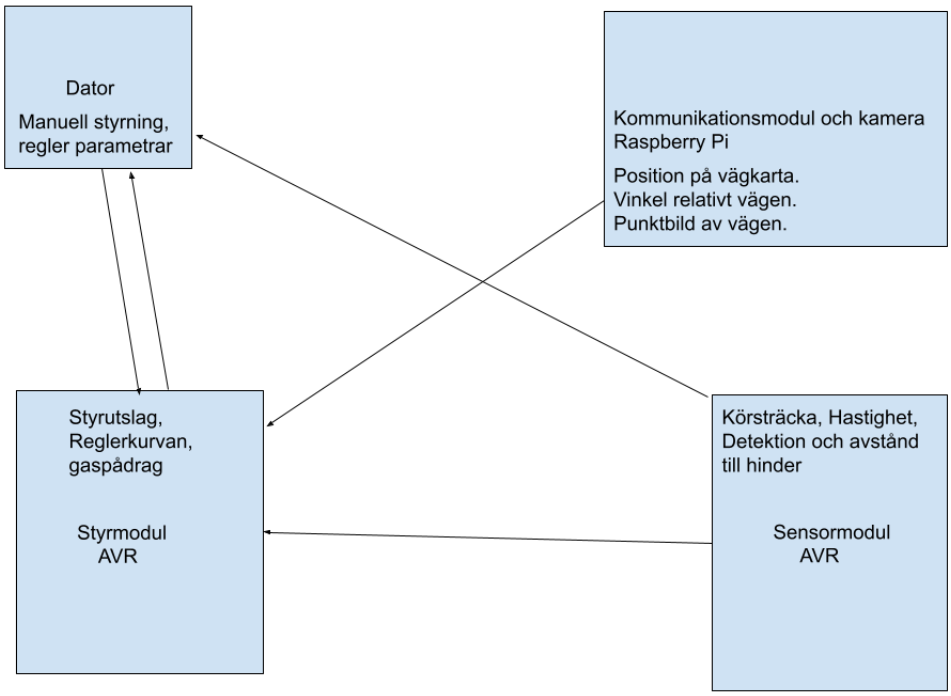
3.3 Data

Kommunikation mellan de olika processerna i kommunikationsmodulen, såsom kommunikation mellan servern och bussen sker via att läsa och skriva till en fil i en ramdisk.

Data som skickas mellan delsystem på taxibilen visas översiktligt i figur 16. Eftersom all datainformation går via kommunikationsmodulen, se kapitel 3.1, symboliserar pilarna vart datan hamnar i slutändan och speglar inte vägen som datan går för att ta sig dit. Datan som varje modul genererar visas i modulblocket, och det är denna data som ska skickas på bussen.

I figuren ser vi att det finns två övergripande dataströmmar. En dataström från moduler som generar information till styrmodulen, där styrreglering sker, samt en dataström från bilens tillståndsvariabler till användardatorn, där visualisering sker.

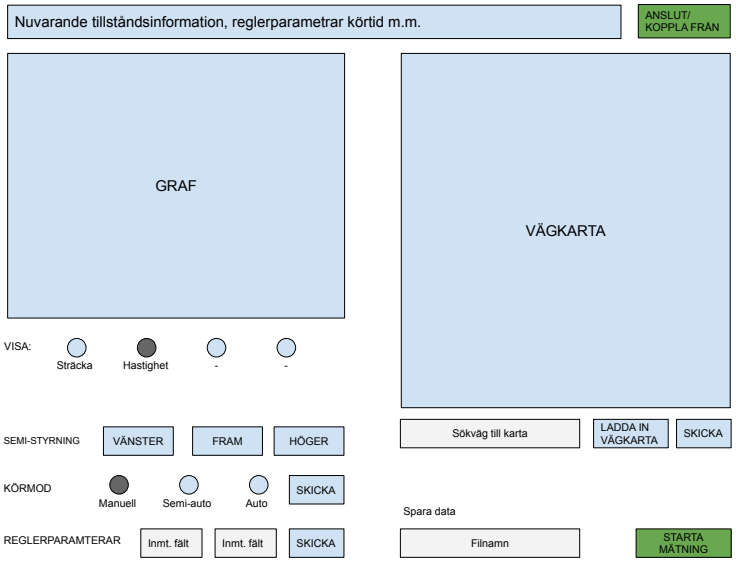
Notera att all data som skickas från blocket ”kommunikationsmodul” är ett resultat av behandlad bilddata, se kapitel 2.3. Denna data visas i blocket med Raspberry Pi:n eftersom det är där kameran ansluts, men är en del av sensormodulen. Kommunikationsmodulens uppgift är således enbart att kommunicera information mellan de övriga modulerna.



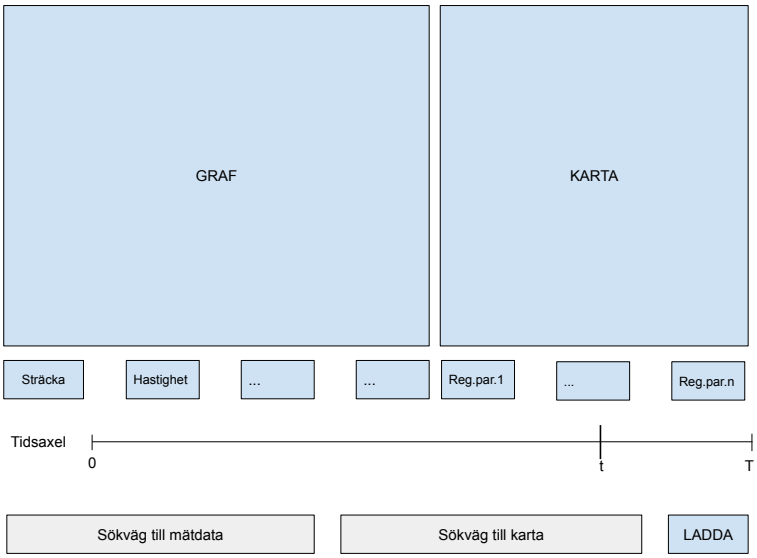
Figur 16: Skiss av intermodulär kommunikation

4 ANVÄNDARDATOR

Användardatorn interagerar med taxibilen via ett program skrivet i Python. Det grafiska gränsnittet är byggt med hjälp av biblioteket tkinter. För att kommunicera med servern som körs på taxibilen har ett klient-API konstruerats som sedan integretats i det grafiska gränsnittet. All data som skickas från server till klienten sparas i en CSV-fil. Under körning plottas data kontinuerligt och bilens position visas på kartan, se figur 17. För att plotta data i efterhand används en separat flik i användagränssnittet, se figur 18. Genom att flytta markören på tidsaxlen ritas bilens position ut för den aktuella tidpunkten och motsvarande data i grafen markeras.



Figur 17: Skiss över fliken som används under körning



Figur 18: Skiss över fliken som används för att analysera data

5 IMPLEMENTATIONSSTRATEGI

Strategin är att följa tidplanen med alla aktiviteter som beskrivs i projektplanen. Det vill säga börja från början, ta steg framåt, avsluta när produkten är levererad.

5.1 Testning

Testningsstrategin är att bygga delarna utifrån aktvivterna i projketplanen och sedan testa de i stort när den delen är färdigbyggd (gäller både mjukvara och hårdvara). Testning av mjukvara görs via simulering samt plottning av grafer innan det testas på hårdvaran. Testerna dokumenterats så de går att se vid senare tillfälle vad som funkade då om problem har uppstått. Återkoppling från systemen fås genom att utnyttja mätverktyg som är givna, som då kan vitalisera vad systemet ger ut för signaler.

5.2 Sampling

Samplingen hos systemet ska ske så att styrservot kan reagera på ny data vid varje puls. Styrservot kan inte ändra utslagen mer än en gång per 20 ms, vilket innebär att ny styrsignal ska vara tillgänglig då för bästa prestanda. Eftersom sensornmodulen kontinuerligt uppdaterar sin data så kommer det alltid finns ny data att hämta med styrservots hastighet. Mjukvaran förväntas implementeras på ett sådant vis att denna uppdateringsfrekvens kan hållas.

REFERENSER

[1] E. Johansson and M. Dahl, “Förstudie - regler,” 2021.

I PROJEKTPLAN

Projektplan

Henrik Ahlinder
Edvin Johansson
Mikael Josefsson
Sebastian Andersson
Martin Dahl
Charbel Youssef

7 juni 2021

Version 1.2

Status

Granskad	Henrik Ahlinder	2021-02-18
Godkänd		

Projektidentitet

Grupp E-post: TSEA56_2021VT_O2_G10@groups.liu.se

Beställare: Anders Nilsson, Linköpings universitet
Tfn: 013-28 26 35
E-post: anders.p.nilsson@liu.se

Handledare: Peter Johansson
Tfn: +4613281345
E-post: peter.a.johansson@liu.se

Kursansvarig: Mattias Krysander
Tfn: 013-28 21 98
E-post: mattias.krysander@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Henrik Ahlinder	Projektledare (PL)	henah490@student.liu.se
Mikael Josefsson	Kommunikationsmodulansvarig, GUI-ansvarig	mikjo695@student.liu.se
Sebastian Andersson	Testansvarig	seban218@student.liu.se
Martin Dahl	Dokumentansvarig, mjukvaruansvarig	marda545@student.liu.se
Charbel Youssef	Sensormodulansvarig	chayo642@student.liu.se
Edvin Johansson	Styrmodulansvarig	edvjo404@student.liu.se

INNEHÅLL

1 Beställare 1
2 Översiktlig beskrivning av projektet 1
2.1 Syfte och mål 1
2.2 Leveranser 1
2.3 Begränsningar 1
3 Organisationsplan 2
3.1 Villkor för samarbete inom projektgruppen 2
3.2 Definition av arbetsinnehåll och ansvar 3
4 Dokumentplan 4
5 Utvecklingsmetodik 5
6 Utbildningsplan 5
6.1 Egen utbildning 5
7 Rapporteringsplan 5
8 Mötesplan 6
9 Resursplan 6
9.1 Personer 6
9.2 Material 7
9.3 Lokaler 7
9.4 Ekonomi 7
10 Milstolpar och Beslutspunkter 8
10.1 Milstolpar 8
10.2 Beslutspunkter 8
11 Aktiviteter 9
11.1 Användardatorn 9
11.2 Styrmodul 9
11.3 Sensormodul 10
11.4 Kommunikationsmodul 10
11.5 Dokument 10
11.6 Hårdvara 11
11.7 Övrigt 11
12 Tidplan 12
13 Kvalitetsplan 12
13.1 Granskningar 12
13.2 Testplan 12
14 Prioriteringar 12
15 Projektavslut 13
Referenser 14

Appendix **15**
A Gruppkontrakt 15

DOKUMENTHISTORIK

Version	Datum	Utförda ändringar	Utförd av	Granskad
0.1	2021-02-10	Första utkast	Alla	Alla
0.2	2021-02-16	Ändringar av material under rubrik 11; Aktiviteter	Alla	Alla
1.0	2021-02-17	Korrigering av tidplan	EJ, SA	SA, HA
1.1	2021-04-12	Uppdatering av tid för milstolpar	HA	Alla
1.2	2021-04-26	Uppdaterat aktivitetslistan	SA	Alla

1 BESTÄLLARE

Beställare är Linköpings Universitet, via Anders Nilsson.

2 ÖVERSIKTLIG BESKRIVNING AV PROJEKTET

Projektet går ut på att utföra ett konstruktionsprojekt där produkten som ska utvecklas är en autonom taxibil. Projektet är även grunden till medlemmarnas kandidatarbete och därför ska även en kandidatrapport som beskriver projektet skrivas.

Projektet utförs via LIPS-modellen [1], och syftet med detta dokument är att utgöra en grund för planeringen av projektets utförande.

2.1 Syfte och mål

Syftet med projektet är att konstruera en taxibil som uppfyller kraven listade i kravspecifikationen [2]. Utöver detta har projektgruppen som mål att lära sig effektiv bildbehandling samt få erfarenhet av konstruktionsprojekt med praktiskt applicerad elektronik och kretsteori. Dessutom är förhoppningen att lära sig om parallellprogrammering och hur detta kan utnyttjas för att utföra beräkningar snabbare och på så vis bygga en effektivare maskin.

2.2 Leveranser

I tabell 1 listas de leveranser som ingår i projektet.

Nr	Beskrivning	Datum
1	Godkännande av kravspecifikation, beslut att påbörja förberedelser	2021-02-03
2	Godkännande av projektplan, tidplan och systemskiss [3], beslut att påbörja utförande	2021-02-22
5a	Godkännande av produktens basfunktionalitet, beslut att definiera leveranskrav	2021-05-05
5b	Godkännande av produktens funktionalitet, beslut att leverera	2021-05-19

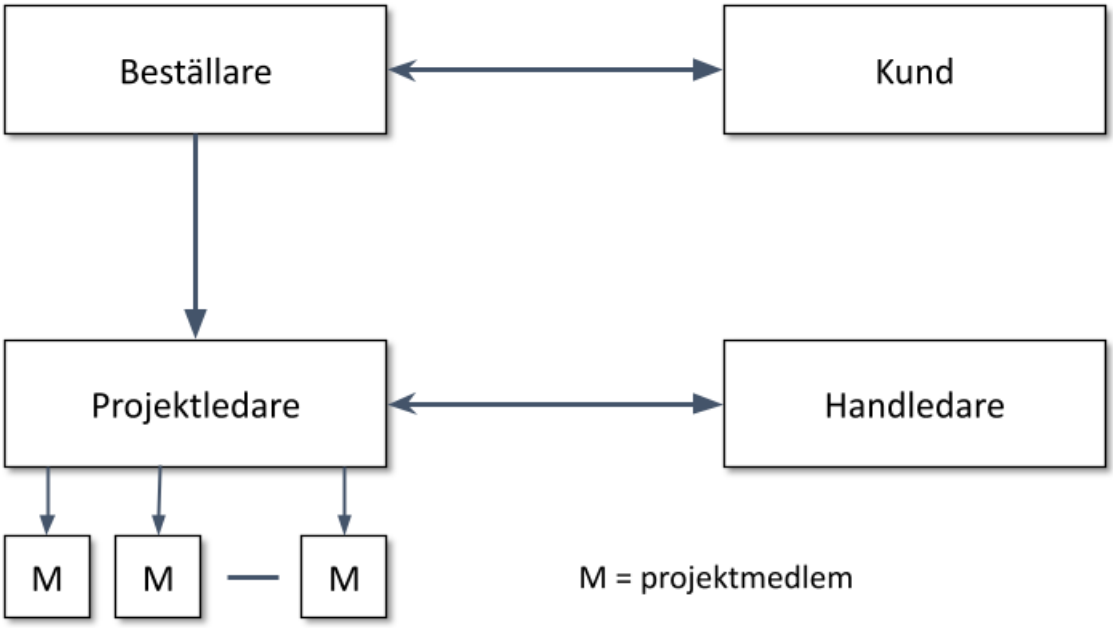
Tabell 1: Leveranser

2.3 Begränsningar

Bilen kör endast uppdrag på den väldefinierade vägbanan och kan endast styras från användardatorn.

3 ORGANISATIONSPLAN

Den övergripande organisationsplanen för projektet visas i figur 1. Däri visas vilken roll kund, beställare, projektledare, handledare respektive projektmedlemmar har i organisationen samt tydliggör hur kommunikationsvägarna inom projektet är.



Figur 1: Organisationsplan

3.1 Villkor för samarbete inom projektgruppen

Det fullständiga gruppkontraktet som skrevs vid projektets start finns i Appendix A. En kort sammanfattning av de viktigaste punkterna är:

Möteskultur	Projektmedlemmar förväntas komma förberedda till möten.
Arbetsansvar	Ett gemensamt arbetspass schemaläggs 2h per vecka där projektmedlemmar förväntas delta. Utöver det ansvarar projektmedlemmar för att anpassa sin arbete efter arbetsbördan.
Samordning	Alla ansvarar för att kommunicera problem och frågor med övriga projektmedlemmar. Vid de tillfällen där det krävs bär projektledare ansvaret för att samordna arbete inom gruppen.

3.2 Definition av arbetsinnehåll och ansvar

Nedan listas de roller som projektets medlemmar och övriga parter har.

Namn	Roll	Organisation
Anders Nilsson	Beställare	ISY
Henrin Ahlinder	Projektledare, leveransansvarig, utvecklare	Y-Sektionen
Martin Dahl	Dokumentansvarig, mjukvaruansvarig, utvecklare	Y-Sektionen
Charbel Youssef	Sensormodulansvarig, utvecklare	Y-Sektionen
Mikael Josefsson	Kommunikationsmodulansvarig, GUI-ansvarig, utvecklare	Y-Sektionen
Sebastian Andersson	Testansvarig, utvecklare	Y-Sektionen
Edvin Johansson	Styrmodulansvarig, utvecklare	Y-Sektionen
Peter Johansson	Handledare och teknisk expert	ISY
Olov Andersson	Teknisk expert	ISY
Svante Gunnarsson	Teknisk expert	ISY
Mattias Krysander	Teknisk expert	ISY

- **Beställare** : Projektets kund/beställare.
- **Projektledare** : Har ett övergripande ansvar för projektet utvecklas i rätt riktning.
- **Leveransansvarig** : Ansvarar för att leveranser sker samt kontrollerar och följer upp tidsplanen.
- **Modulansvarig** : Ansvarar för att ha uppdaterad information om sin modul och ha en djupare förståelse av modulen i fråga. Modulansvarig ska även identifiera förseningar i utvecklingen av modulen som kan påverka projektet i stort.
- **GUI-ansvarig** : Ansvarig för användarprogrammets utformning och funktion.
- **Mjukvaruansvarig** : Ansvarig för git, struktur och ordning i projektets kodbas.
- **Dokumentansvarig** : Ansvarig för kvalitet och ordning av levererade dokument.
- **Testansvarig** : Se till att tester är gjorda och dokumenterade enligt en enhetlig modell.
- **Utvecklare** : Bygger tekniska system för att uppnå projektets krav.
- **Handledare** : Bistår utvecklingsgruppen med råd om tekniska detaljer samt projektstyrning.
- **Teknisk expert** : Expert inom ett särskilt område som berör projektets tekniska system.

4 DOKUMENTPLAN

Tabell 2: Dokument som skall produceras

Dokument	Godkänns av	Syfte	Distribueras till	Färdig-datum
Kravspecifikation	Beställare	Entydig beskrivning av vad som ska produceras. Utgör underlag för systemskiss, konstruktionsdokument och projektplan.	Beställare	2021-02-03
Projektplan	Beställare	Beskriver en tolkning av hur krav och andra behov ur projektdirektivet ska uppfyllas samt villkor för samarbete inom grupp och beställare	Beställare	2021-02-22
Systemskiss	Beställare	Ger en översiktlig bild på hur produkten ska utformas	Beställare	2021-02-22
Designspecifikation	Handledare	Beskriver hur produkten ska konstrueras.	Handledare	2021-03-12
Förstudier	Beställare	Version 1.0 av tre förstudier inom reglerteknik, sensorteknik respektive kommunikation som förberedelser inom projektet	Beställare och Skrivuppgiftshandledare	2021-04-08
Teknisk dokumentation	Projektgrupp	Konstruktionsunderlag för produkten, samt beskrivning av underhåll, felsökning och av olika lösningar som berör både hård- och mjukvara.	Beställare	2021-05-19
Användarhandledning	Projektgrupp	Beskriver hur produkten ska användas	Beställare	2021-05-19
Efterstudie	Beställare	Sammanfattning av erfarenheter från, och utvärderingar av, projektarbetet. Utgör ett underlag för utveckling av genomförande av framtida projekt.	Beställare	2021-06-07
Kappa	Examinator	Ger en komplett överblick av det genomförda arbetet i projektet.	????	2021-06-07

5 UTVECKLINGSMETODIK

Utvecklingsgruppen arbetar med ett testdrivet tankesätt och med mål att alltid underhålla ett fungerande system. Programspråk kan väljas efter utvecklarens individuella bedömning samt efter vad som specificerats i designspecifikationen, främst ska val av programspråk hållas konsekvent. Utvecklare avgör själva om arbetet sker individuellt eller gruppvis, tidplanen baseras på gruppvis ansvarsindelning som därför kan anses vara mest lämplig. Utvecklingsgruppen kommer överens om kodkonventioner som ska hållas konsekvent under projektets gång samt en nivå av läsbarhet på kod.

6 UTBILDNINGSPLAN

6.1 Egen utbildning

Tid för egen utbildning är inkluderad i alla aktiviteter, för vissa aktiviteter kommer det läggas mer tid för utbildning än andra. Utbildningstiden uppskattas gå jämnt ut över alla aktiviteter och utvecklare.

Två av gruppens utvecklare utbildas även inom I^2C genom en laboration tillhörande kursen TSEA56.

7 RAPPORTERINGSPLAN

Under projektets gång ska rapporterna beskrivna i tabell 3 skrivas. En version av varje rapporttyp ska distribueras till beställare senast första helgfria arbetsdag 16.00 vari föregående veckas arbete visas. Leveransdatum för första rapporten är 29/3-2021 och för sista rapporten 7/6-2021.

Typ	Beskrivning	Ansvarig	Distribueras till
Tidrapporter	En tidrapport över senaste veckan tillsammans med uppdaterad tidplan	PL	Beställare
Statusrapporter	Beskriver framsteg som gjorts och problem som stötts på sedan senaste tidrapporten samt vad som ska utföras kommande vecka	PL	Beställare

Tabell 3: Rapporter

Tidrapporter upprättas kontinuerligt av samtliga projektmedlemmar som förväntas fylla i veckans arbetstid i gruppens tidplan. Projektmedlemmar förväntas ha uppdaterat tidplanen med veckans information före veckomötet efterföljande vecka, eller om mötet uteblir söndag 20.00 samma vecka. Därefter åligger det projektledare att distribuera tidrapporten och den tillhörande tidplanen till beställaren.

Statusrapporter skrivs under veckomötet av veckans utsedda sekreterare. Därefter åligger det projektledare att distribuera rapporten. I händelse av att en projektmedlem inte kan närvara på mötet där statusrapporten skrivs förväntas att medlemmen skriver en egen kort statusrapport i god tid före mötet. Denna individuella statusrapport ska sedan under mötet inkorporeras i statusrapporten för hela gruppen. Skulle veckomötet utebli ska statusrapporten antingen skrivas under närmast tidigare föreliggande sammankomst av projektmedlemmar. Saknas ett lämpligt möte för att skriva statusrapporten ska varje projektmedlem skriva en individuell statusrapport senast senast söndag 20.00 veckan innan distribution. Projektledare ansvarar sedan för att sammanfoga de individuella rapporterna till en rapport från hela gruppen som distribueras till beställare.

8 MÖTESPLAN

Projektgruppen har ett uppföljningsmöte varje måndag, schemalagt till en timme. Under detta möte ska gruppens arbete synkroniseras och projektets status ska utvärderas. Utöver detta uppföljningsmöte strävar gruppen efter att minimera antalet möten, och väljer istället att utnyttja minst ett schemalagt arbetspass per vecka där strukturen är mindre strikt och fokus är produktivitet.

9 RESURSPLAN

I avsnitten i detta kapitel listas de resurser som finns att tillgå för projektet.

9.1 Personer

Arbetstid för projektmedlemmar är vardagar 8-17, och inom denna tid förväntas projektmedlemmar vara kontaktbara. Frågor som en majoritet av projektmedlemmarna behöver ta ställning till ska i huvudsak tas upp på veckomötet eller arbetspassen, medan mindre frågor kan lyftas och besvaras i gruppens Discord-kanal.

Projektmedlemmarna kommer också utnyttja handledare för projektet, Peter Johansson, för tekniskt stöd. Utöver projektmedlemmar och handledare finns även handledning av följande experter att tillgå:

Område	Person	Arbetstid
LIPS projektmodell	Mattias Krysander	8-17
Analog elektronik	Peter Johansson	8-17
Mekanik	Peter Johansson/Olov Andersson	8-17
Reglerteknik	Alla handledare	8-17
Mikroprocessorer	Alla handledare	8-17

De veckor som vissa projektmedlemmar förväntar sig ha modifierade arbetstider och eventuell minskad arbetsbörda anges nedan. Det förväntas att projektmedlemmar som önskar modifierad arbetstid strukturerar sitt arbete så att projektet inte påverkas av den förändrade arbetstiden.

Person	Vecka	Förväntad arbetstid/vecka (h)	Anledning
Alla	11-12	4	TentaP
Henrik	7, 13	2	Skidresa

9.2 Material

Projektet utförs inom ramarna av kursen TSEA56 som tillhandahåller projektet med de komponenter som behövs för att bygga den autonoma taxibilen. Möjliga komponenter listas bland annat på webbsidan *http://www.da.isy.liu.se/vanheden/*. Ingen brist på komponenter förväntas uppstå samt skicket på komponenterna från vanheden förväntas vara i gott tillstånd. Om en situation uppstår med brist på fungerande komponenter kommer en anhållan om att beställa nya komponenter göras till kursledningen.

9.3 Lokaler

Projektgruppen har en reserverad labbsal i Muxen. Utöver muxen har projektmedlemmarna möjlighet att boka tillgängliga lokaler på LiU, särskilt ISY's lokaler som Visionen.

9.4 Ekonomi

I projektet ingår totalt 1380 arbetstimmar distribuerat på 230 h per projektmedlem.

10 MILSTOLPAR OCH BESLUTSPUNKTER

Milstolpar och bestlutspunkter som ska uppfyllas under projektets gång listas nedan i tabell 4 respektive tabell 5.

10.1 Milstolpar

Nr	Beskrivning	Datum
1	Designspecifikation är klar för inlämning	2021-03-11
2	Kommunikation mellan moduler samt mellan bil och användardatorn är etablerad.	2021-04-16
3	Bilen kan köras i manuellt läge	2021-04-16
4	Bilen kan köras i semi-autonomt läge	2021-04-23
5	Bilen kan köras i autonomt läge	2021-04-28
6	Baskrav är uppfyllda	2021-05-03
7	Slutkrav är uppfyllda	2021-05-11

Tabell 4: Milstolpar

10.2 Beslutspunkter

Nr	Beskrivning	Datum
1	Godkännande av kravspecifikation, beslut att påbörja förberedelser	2021-02-03
2	Godkännande av projektplan, tidplan och systemskiss, beslut att påbörja utförande	2021-02-22
3	Godkännande av designspecifikation, beslut att fortsätta utförande	2021-03-12
4	Godkännande av designval i konstruktionen, beslut att påbörja konstruktion	2021-04-16
5a	Godkännande av produktens basfunktionalitet, beslut att definiera leveranskrav	2021-05-05
5b	Godkännande av produktens funktionalitet, beslut att leverera	2021-05-19
6	Inlämning av kandidatrapport, efterstudie samt återlämning av nycklar. Beslut att lösa upp projektgrupp	2021-06-07

Tabell 5: Beslutspunkter

11 AKTIVITETER

De aktiviteter som kommer att genomföras i projektet listas nedan. Aktiviteterna är huvudsakligen uppdelade efter modul, men detta är ingen designspecifikation; placering och modultillhörighet av aktiviteter kan komma att ändras, och aktiviteter som är multimodulära kan vara placerade under vilken som helst av de relaterade modulerna.

Uppdateringar under projektets gång.

- Nolla aktivitet 'konstruktion av krets' då detta inte behövs göras.
- Flyttat timmar från aktivitet '2D->3D transform' till 'detektion av kantlinjer' då detta genomfördes som i mindre stadie av den sist nämnda aktiviteten.

11.1 Användardatorn

Nr	Aktivitet	Beroende av aktivitet	Beräknad tid
1.1	Implementera ett grundläggande GUI	-	20
1.2	Implementera funktionalitet för inmatning av körbana i användardatorn	1.1	15
1.3	Implementera användardatorns konfigurering av uppdrag	1.1	10
1.4	Implementera användardatorns tangentavläsning för manuell styrning	1.1	10
1.5	Implementera inskrivning av kommandon för semi-autonoma läget i användargränssnittet	1.1, 1.9	10
1.6	Implementera funktionalitet för ändring av regulatorparametrar i bilens styrmodul	1.1, 4.1, 2.1	20
1.7	Implementera sparande och visualisering av sensordata	1.1, 3.2, 3.3, 3.4, 3.10, 3.11, 4.2, 4.3	30
1.8	Gör användargränssnittet kapabelt att byta kör-läge på taxibilen	1.1, 2.6, 4.2, 4.3	10
1.9	Implementera ett fint GUI	1.1	20

Tabell 6: Användardatorn

11.2 Styrmodul

Nr	Aktivitet	Beroende av aktivitet	Beräknad tid
2.1	Implementera funktionalitet för att kontrollera styrutslag och gaspådrag.	2.6, 6.1	10
2.2	Implementera reglering av styrutslag	2.1, 2.3, 2.6, 3.4, 6.1	25
2.3	Implementera reglering av gaspådrag	3.6, 6.1	30
2.4	Hitta rätt parametrar hos regulatorn för styrreglering	1.6, 1.8, 2.2	15
2.5	Hitta rätt parametrar hos regulatorn för gaspådrag	1.6, 1.8, 2.3	15
2.6	Bygg en grundläggande tillståndsmaskin för hantering av olika körlägen, hinderhantering och vägsegment i styrmodulen	-	40
2.7	Implementera hantering av inbromsning vid detektion av hinder	2.3, 2.6, 3.3	10
2.8	Implementera hantering av inbromsning vid destination	3.2, 3.7	10
2.9	Implementera beräkning av det kortaste avstånd till destination (optimering)	1.2	10
2.10	Hastighetsoptimering	2.4, 2.5, 2.7, 2.8, 3.7	30

Tabell 7: Styrmodulen

11.3 Sensormodul

Nr	Aktivitet	Beroende av aktivitet	Beräknad tid
3.1	Implementera detektion av kantlinjer	-	85
3.2	Implementera detektion av stopplinjer	3.1	20
3.3	Implementera detektion av hinder	-	20
3.4	Implementera beräkning av riktning relativt vägen	3.1	20
3.5	Implementera 2D -> 3D kamera transform	-	0
3.6	Implementera sträcka- och hastighetsberäkning från hallsensorer	-	22
3.7	Implementera positionering relativt vägnät.	3.4, 3.5, 3.6, 3.10	50
3.8	Implementera detektion av korrekt destination i hel-autonoma läget	3.2, 3.7	25
3.9	Implementera detektion av avklarat semi-autonomt uppdrag	3.7	10
3.10	Implementera detektion av svängar	3.1	15

Tabell 8: Sensormodulen

11.4 Kommunikationsmodul

Nr	Aktivitet	Beroende av aktivitet	Beräknad tid
4.1	Implementera en trådlös länk mellan användardatorn och kommunikationsmodulen	-	20
4.2	Implementera mjukvara/server som skickar information till/från användardator	4.1	30
4.3	Implementera intermodulär kommunikation	-	30

Tabell 9: Kommunikationsmodulen

11.5 Dokument

Nr	Aktivitet	Beroende av aktivitet	Beräknad tid
5.1	Skriva designspecifikation	-	100
5.2	Skriva kappa	-	110
5.3	Skriva teknisk dokumentation	-	70
5.4	Skriv användarhandledning	-	70
5.5	Skriv efterstudie	-	25

Tabell 10: Dokument

11.6 Hårdvara

Nr	Aktivitet	Beroende av aktivitet	Beräknad tid
6.1	Montering av hårdvara på bilchassi	-	10
6.2	Konstruera krets	-	0

Tabell 11: Hårdvara

11.7 Övrigt

Nr	Aktivitet	Beroende av aktivitet	Beräknad tid
7.1	Sammanfogning av system och systemtest	-	150

Tabell 12: Hårdvara

12 TIDPLAN

Projektets tidplan beskrivs i detalj i dokumentet ”Tidplan” [4]. Tidplanen är konstruerad med hänsyn till avsnitt 11 och aktivteternas inbördes beroenden.

13 KVALITETSPLAN

13.1 Granskningar

Utvecklingskod genomförs på branches så att master har en hög sannolikhet att fungera. Anser man sig klar med en feature gör man en mergerequest till master. När en projektmedlem har hunnit granska mergen godkänner han den och koden läggs till master.

13.2 Testplan

Varje nyutvecklat tillägg ska testas och funktionskontrolleras. Tester ska skrivas och dokumenteras kontinuerligt av samtliga utvecklare och denna tid finns med i tidplanen. Behov och utformning av en mer detaljerad testplan och dokumentationsprotokoll utreds av testansvarig. En veckovis kontroll kommer att göras mot kravspecifikationen.

14 PRIORITERINGAR

Projektgruppen strävar i helhetspersktiv efter att uppfylla krav med prioritering Basförst, och därefter Slut". Detta åstadkoms genom att i största möjliga mån prioritera aktiviteter som krävs för att nå nästkommande milstolpe utan hänsyn för framtida milstolpar. Skulle en milstolpe riskera att inte bli uppfylld åligger det alla projektmedlemmar att lyfta problemet omgående. Därefter kommer gruppen på ett möte gemensamt fram till en handlingsplan för att lösa situationen på bästa vis.

Skulle någon i projektgruppen ha anledning att tro att det föreligger en risk att projektgruppen inte skulle klara av att uppfylla alla krav i tid åligger det medlemmen i fråga att lyfta detta på nästkommande projektmöte där situationen sedan diskuteras.

Anser en majoritet av mötesmedlemmarna att problemet inte går att lösa så har projektledaren i uppgift att inom 3 arbetsdagar schemalägga ett möte med beställare för att omförhandla kravet. Skulle projektgruppen under detta möte inte hitta en lösning kontaktar projektledaren omgående beställare och examinator och anpassar projektet efter deras föreslagna lösningar.

15 PROJEKTAVSLUT

Projektet avslutas och projektgruppen upplöses efter att samtliga beslutspunkter i avsnitt 10.2 har blivit uppfyllda. Det innebär i skrivande stund efter inlämning av kandidatrapporten, erfarenhetsrapport och återlämning av material och nycklar som sker den 7:e juni, varefter projektgruppen löses upp permanent.

REFERENSER

[1] T. Svensson and C. Krysander, *Projektmodellen LIPS*, 1st ed. Studentlitteratur, 2011, iSBN:9789144075259.

[2] M. Dahl, H. Ahlinder, S. Andersson, E. Johansson, C. Youssef, and M. Josefsson, "Kravspecifikation," 2021.

[3] ——, "Systemskiss," 2021.

[4] ——, "Tidplan," 2021.

Appendix

A GRUPPKONTRAKT

Frågeställning	Gruppens åsikt
Deltagarna i gruppen ska gemensamt göra upp ordningsregler för gruppen om tider, närvaro, förberedelser etc.	Ett möte per vecka där alla förväntas delta. Därefter upp till medlemmar att göra fler möten där inte alla måste delta. Vi har även ett gemensamt pass på 2h per vecka där vi arbetar.
Alla deltagare i gruppen ska delta vid de tillfällen som gruppen kommit överens om.	Alla förväntas delta vid veckomötet och arbetspasset. Undantag till detta kan godkännas av pluralitet inom gruppen för, men ej begränsat till, ledighet eller sjukdom. På övriga pass som bokas in föreligger inget deltagandekrav.
Alla deltagare i gruppen ska komma väl förberedda till sammankomsten.	Till veckomötet åligger det medlemmar att ha läst igenom dagordningen, men till övriga pass finns inget sådant krav.
Det är viktigt med en ”samordnare” i gruppen.	I huvudsak förväntas medlemmar att kommunicera mellan varandra. I övrigt agerar projektledaren samordnare vid de tillfällen som det krävs.
Arbetsuppgifterna ska fördelas likvärdigt mellan gruppmedlemmarna, så att arbetsinsatsen blir ungefär lika stor.	Det är okej att jobba olika tider olika veckor, men i slutändan bör alla ha jobbat ungefär lika många timmar (+- 20h).
Samarbetet i gruppen måste när som helst kunna diskuteras öppet, även om det innebär obehag för någon.	Det är viktigt att alla är bekväma med att ge konstruktiv kritik och vågar ta upp konflikter. Tycker någon medlem att det är jobbigt att lyfta en fråga kan hen ta upp det med projektledaren som sedan lyfter frågan anonymt.
Den som inte bidrar aktivt ska inte heller dra nytta av gruppens gemensamma arbete.	Om det har gått väldigt långt med avvikande arbetstimmar så bör man skiljas från gruppen. (Worst case scenario)
Det är viktigt att ge varandra återkoppling, såväl positiv som negativ.	Ja, ge varandra så mycket positiv återkoppling som möjligt, ge också varandra negativ återkoppling.
Varje veckomöte avslutas med en utvärdering, där var och en belyser hur arbetet i gruppen fungerat.	Medlemmar kan under veckomötet lyfta eventuella problem i samarbetet. Lyfts inget anses samarbetet fungera bra och därmed sker ingen utvärdering.
Vår grupps ambitionsnivå är att arbetet ska leda till att vi krossar i tävlingen eller bara blir godkända.	Det ställs inget krav på projektmedlemmar att arbeta på projektet för att vinna tävlingen, men det är okej att satsa individuellt om man vill.

Tabell 13: Gruppkontrakt

J SYSTEMSKISS

Systemskiss

Martin Dahl
Henrik Ahlinder
Charbel Youssef
Edvin Johansson
Mikael Josefsson
Sebastian Andersson

7 juni 2021

Version 1.0

Status

Granskad	Henrik Ahlinder	2021-02-18
Godkänd		

Projektidentitet

Grupp E-post: TSEA56_2021VT_O2_G10@groups.liu.se

Beställare: Anders Nilsson, Linköpings universitet
Tfn: 013-28 26 35
E-post: anders.p.nilsson@liu.se

Handledare: Peter Johansson
Tfn: +4613281345
E-post: peter.a.johansson@liu.se

Kursansvarig: Mattias Krysander
Tfn: 013-28 21 98
E-post: mattias.krysander@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Henrik Ahlinder	Projektledare (PL)	henah490@student.liu.se
Mikael Josefsson	Kommunikationsmodulansvarig, GUI-ansvarig	mikjo695@student.liu.se
Sebastian Andersson	Testansvarig	seban218@student.liu.se
Martin Dahl	Dokumentansvarig, mjukvaruansvarig	marda545@student.liu.se
Charbel Youssef	Sensormodulansvarig	chayo642@student.liu.se
Edvin Johansson	Styrmodulansvarig	edvjo404@student.liu.se

INNEHÅLL

1 Inledning 1
2 Översikt av systemet 3
3 Delsystem 4
3.1 Användargränsnittet 4
3.2 Sensormodulen 5
3.3 Styrmodulen 7
3.4 Kommunikationsmodulen 9

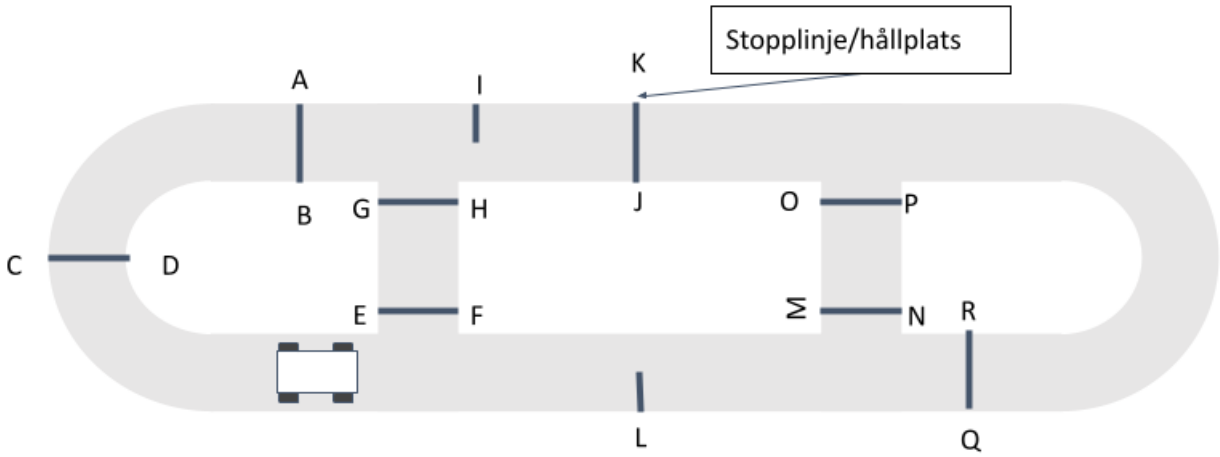
DOKUMENTHISTORIK

Version	Datum	Utförda ändringar	Utförd av	Granskad
0.1	2021-02-16	Första utkast	Alla	Alla
1.0	2021-02-18	Godkänd med minimala ändringar	Alla	Alla

1 INLEDNING

Produkten som ska konstrueras är en ”taxibil” som ska kunna styras manuellt, semi-autonomt och hel-autonomt. I det manuella läget ska bilen kontrolleras som en vanlig RC-bil, via instruktioner givna genom ett användargränssnitt. I detta läge ska bilen kunna köras i alla miljöer med en plan, greppbar yta.

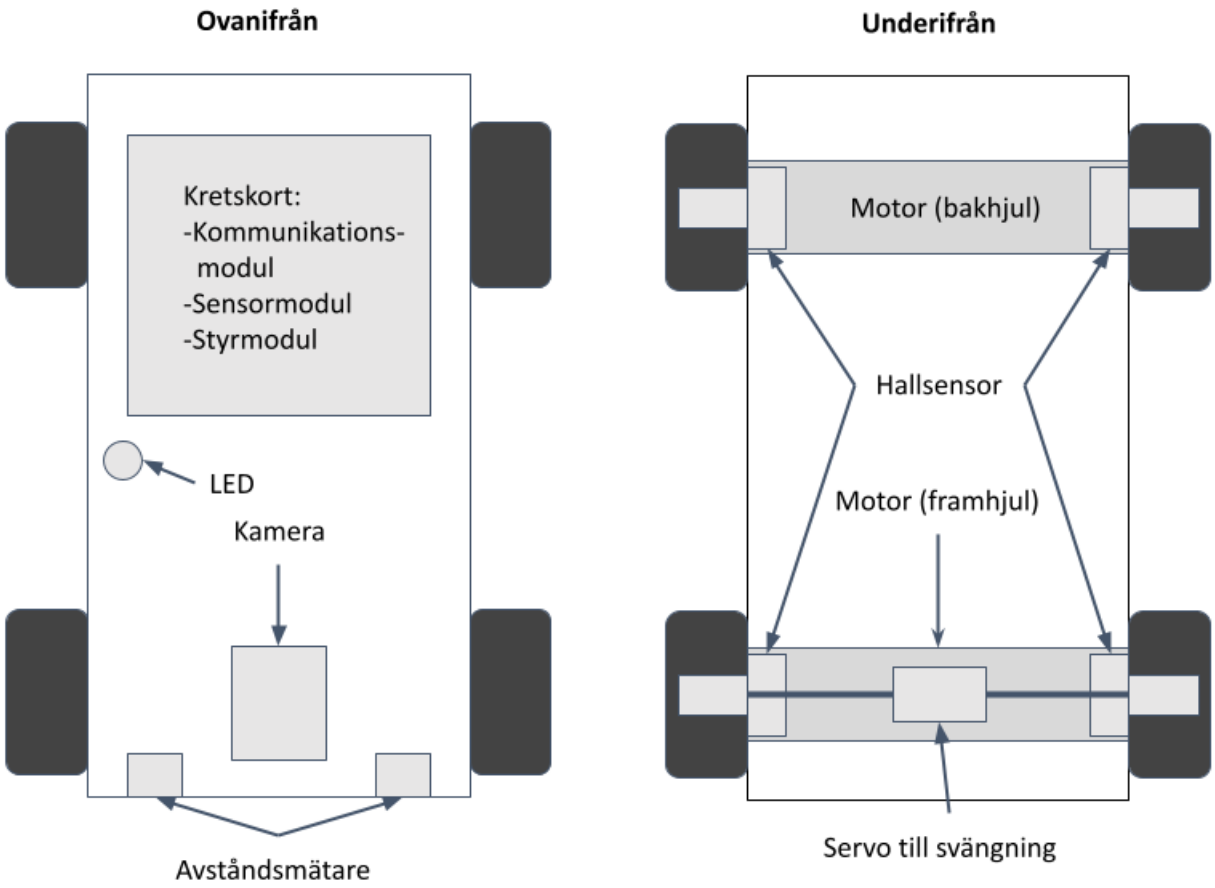
I de båda autonoma lägena, ska bilen navigera i ett givet vägnät med stopplinjer/hållplatser. Ett exempel på en sådan miljö visas i figur 1.



Figur 1: Vägnätsexempel

I det enklare av de autonoma lägena, det semi-autonoma, kommer produkten att kunna styras autonomt då användaren ger korta och enkla instruktioner. I det andra autonoma läget, det helautonoma, ska produkten kunna styras autonomt givet en karta av vägnätet och en slutdestination från användaren. För att åstadkomma detta beteende krävs de övergripande komponenter i bilen som visas i figur 2

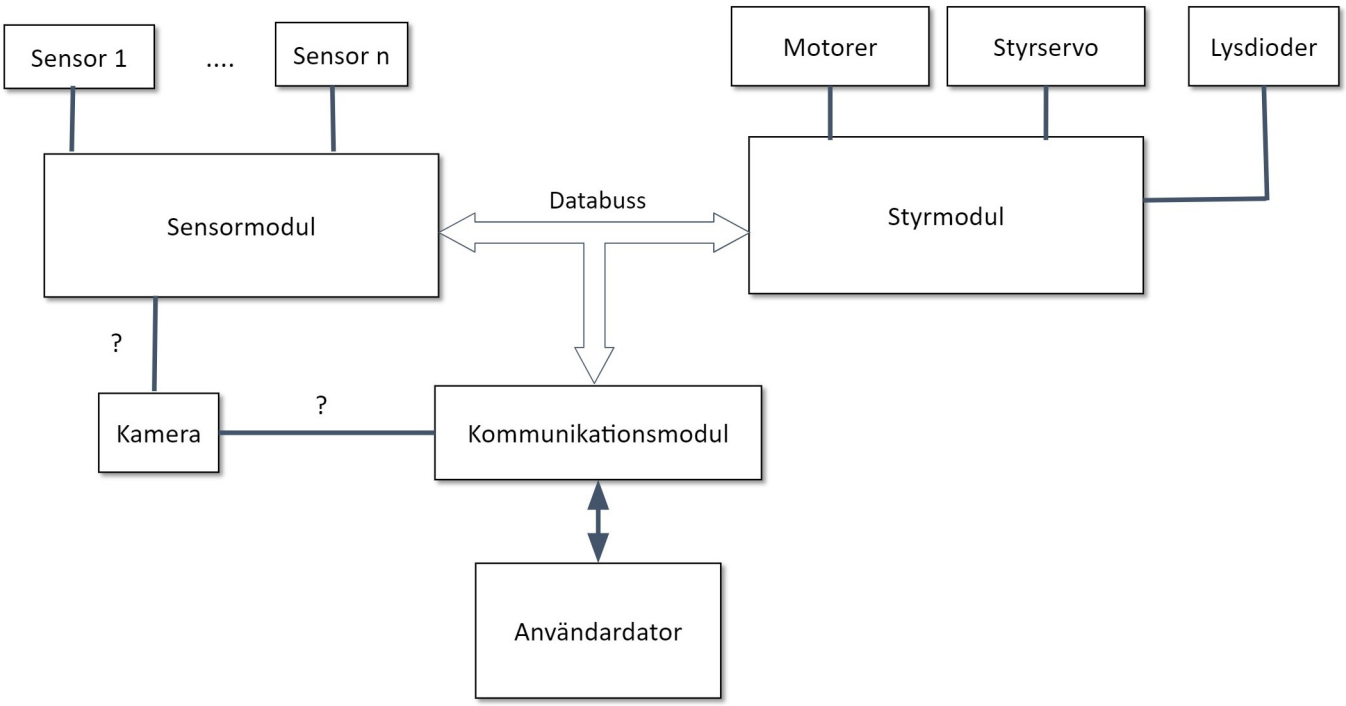
Utöver detta krävs även ett användargränsnitt där användaren kan interagera med de olika komponenterna i bilen. Denna modul visas kort i kapitel 2, och beskrivs i detalj i avsnitt 3.1



Figur 2: Översikt av bilkomponenter

2 ÖVERSIKT AV SYSTEMET

Produkten innehåller tre övergripande moduler, en sensormodul, en styrmodul och en kommunikationsmodul, som visas i figur 3. Dessa moduler beskrivs i detalj i avsnitt 3.2, 3.3 respektive 3.4.



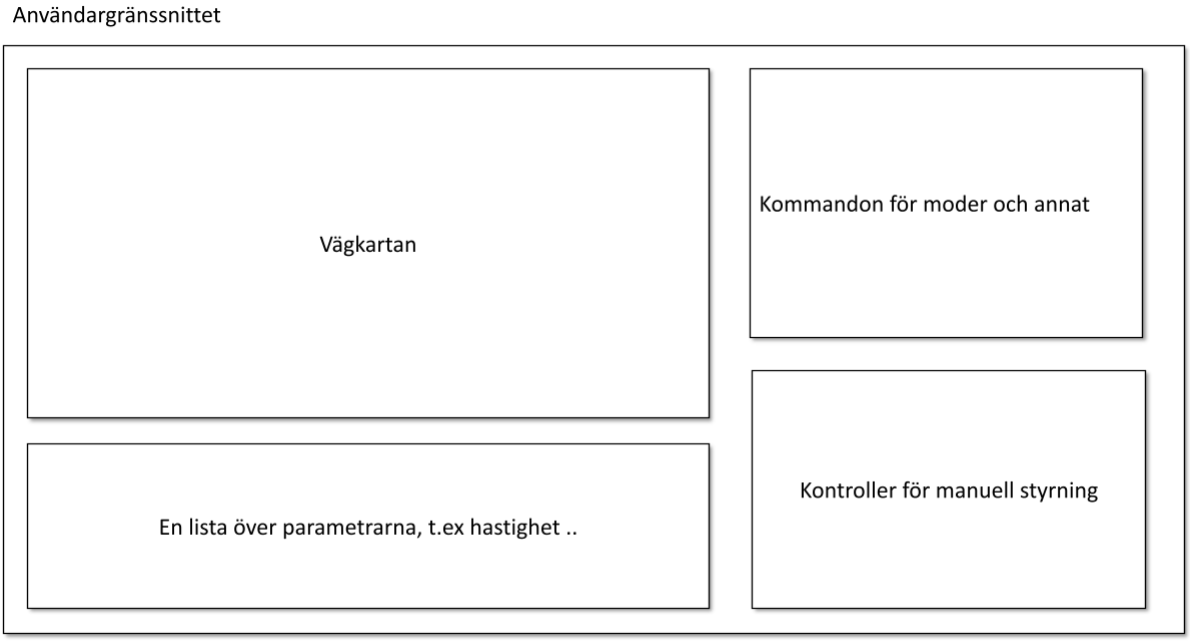
Figur 3: Översikt av delsystem

3 DELSYSTEM

3.1 Användargränsnittet

Användargränssnittets olika block beskrivs i figur 4. Notera att detta blockschema är preliminärt och att storlek och placering av blocken kan komma att förändras.

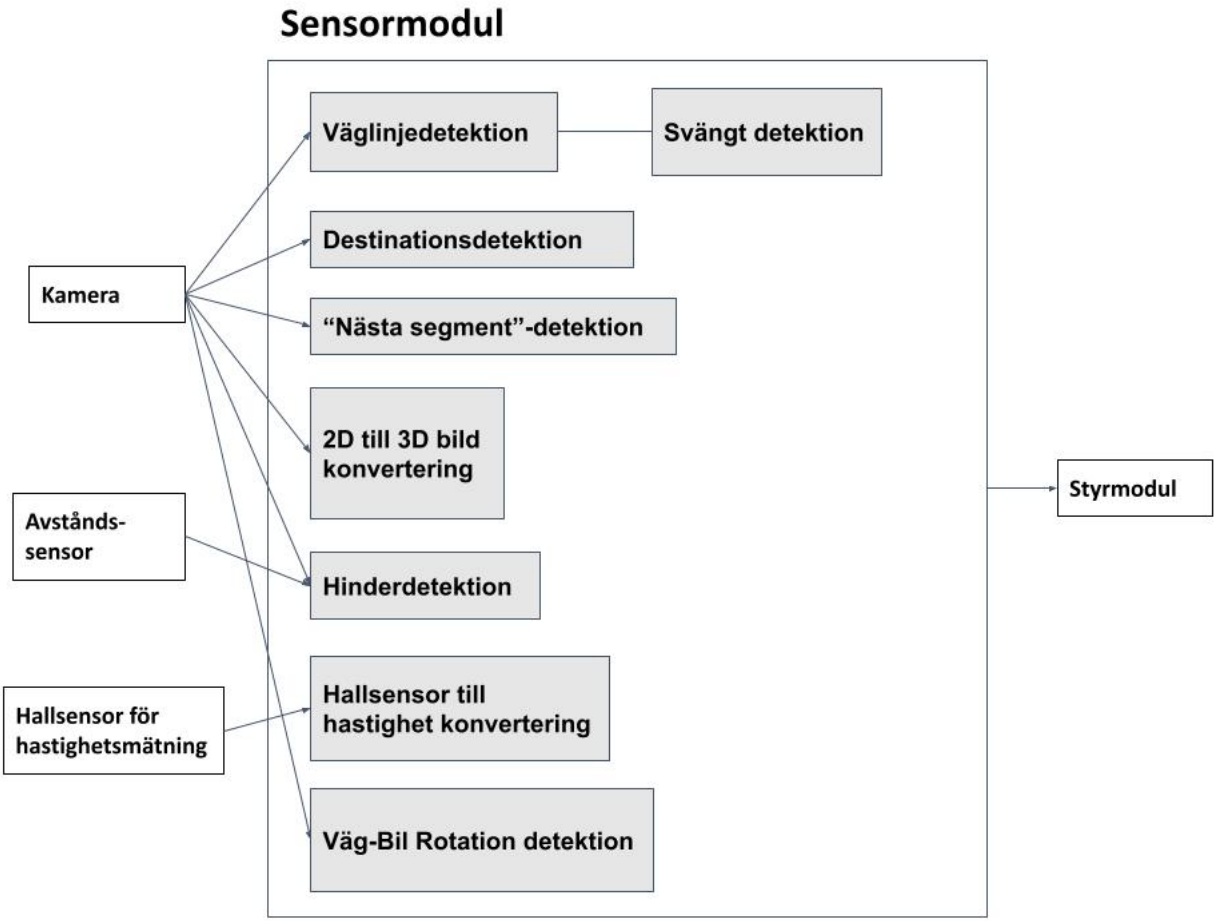
- **Vägkartan:** här syns vägbanan och bilens aktuella position.
- **Kontroller för manuell styrning:** Innehåller knappar och beskrivning angående manuell styrning, t.ex att köra fram eller svänga.
- **Datalista:** en lista över data som beskriver bilens rörelse, t.ex hastighet, acceleration, gaspådrag m.m.
- **Kommandon för moder:** Här väljer användaren vilken mod bilen ska styras enligt.



Figur 4: Användargränssnitt

3.2 Sensormodulen

Sensormodulens olika block beskrivs i figur 5. Sensormodulen läser bilens hastighet, avstånd till eventuella hinder och har en kamera för maskinseende. Hastigheten läses m.h.a. halleffektsensorer och magneter i bakjulen. Avstånd till hinder läses m.h.a. avståndsmätare (laser/ultraljud). Systemet kräver en låg ”färdad sträcka per frame” men då detta mätvärde beror av både bilens hastighet samt kamerans inspelningshastighet utelämnas ett mer exakt värde till förmån för designspecen. Dock behöver detta värde vara under ca 10 cm per frame för att projektet överhuvudtaget ska gå att utföra.



Figur 5: Användargränssnitt

- **Väglinjedetektion, Destinationsdetektion, Nästa segment-detektion och väg-bil rotation detektion:** De olika mätdata som erhålls via kameran och som bilen behöver för önskad lämplig körning.

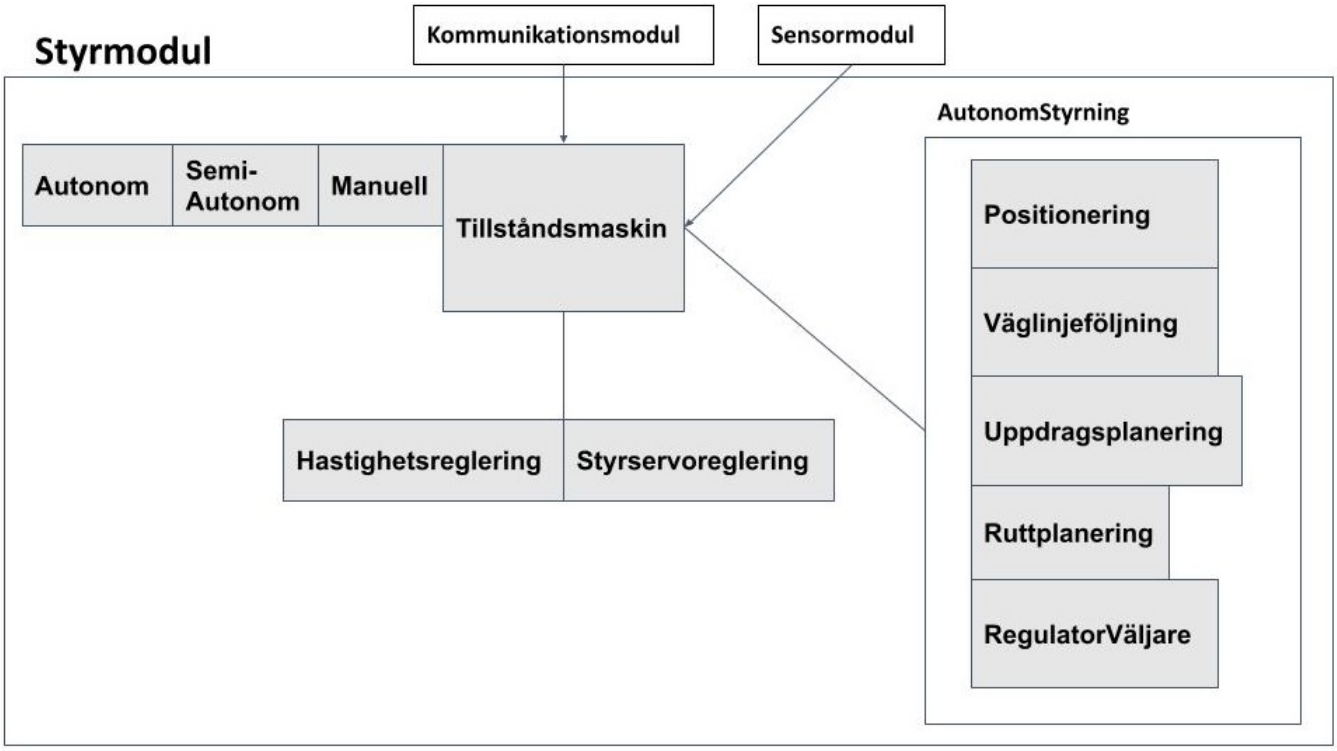
- **Hinderdetektion**: En avståndsmätare används för att upptäcka eventuella hinder.

- **Hallsensor till hastighet konvertering:** Principen används för att mäta bilens aktuella hastighet.

- **2D till 3D bild konvertering:** Omvandla bild som fångas av kameran till data för vidare hantering.

3.3 Styrmodulen

Styrmodulens uppgift är att ta emot styrsignaler och få bilen att utföra en önskade rörelse. Svängning åstadkoms via ett styrservo kopplat till framaxeln, hastigheten regleras via ett inbyggt styrreglage. All denna kontroll regleras med hjälp av PD-reglering



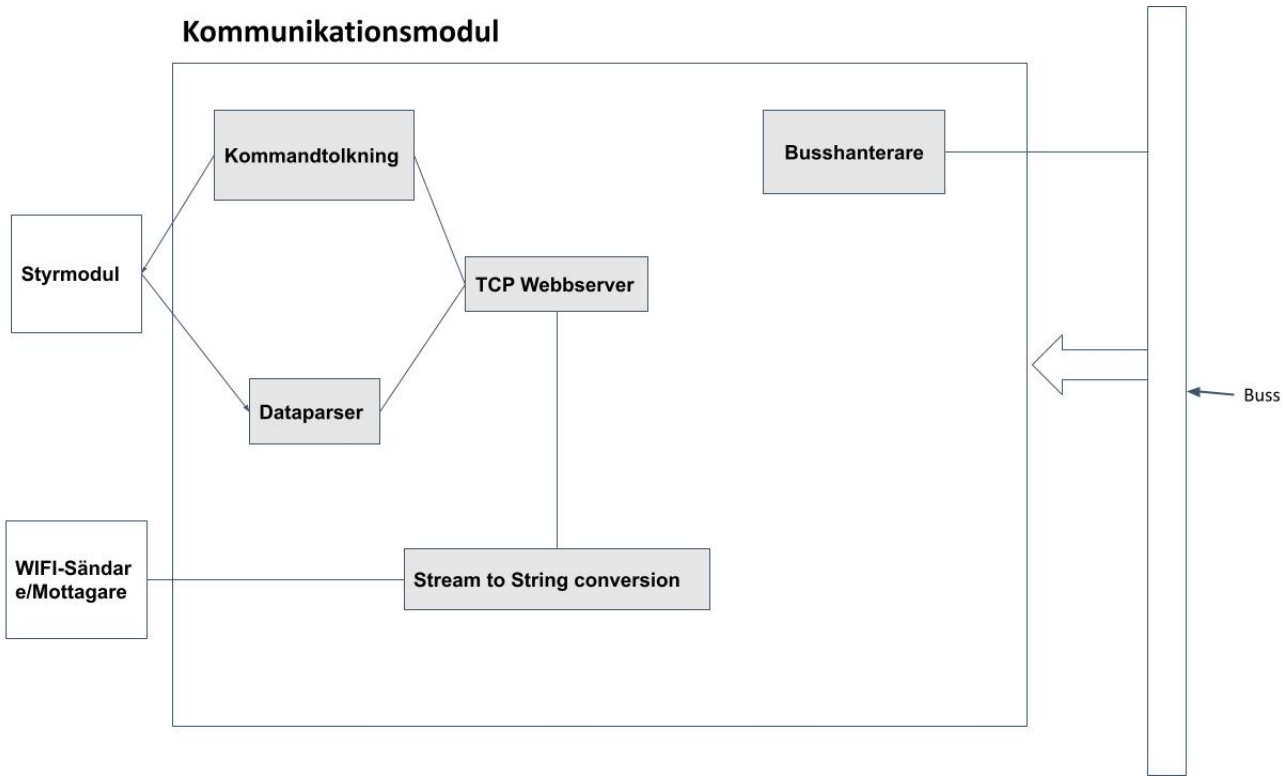
Figur 6: Användargränssnitt

- **Tillståndsmaskin** : Försätter bilen i olika lägen beroende på om autonomt, semi-autonomt eller manuellt läge används. Den hanterar även tillstånd som uppstår då bilen måste bromsa för hinder.
- **Hastighetsreglering** : Är systemets brygga mellan önskad hastighet och bilens motorer.
- **Styrservoreglering** : Är systemets brygga mellan önskad styrriktning och bilens styrservon.
- **Autonom Styrning** : Hanterar alla delsystem som sköter bilens uppförande i autonoma lägen.
  - **Positionering** : Spårar bilens position på vägbanan.

- **Väglinjeföljning** : Reglerar bilens läge i förhållande till väglinjer.
- **Uppdragsplanering** : Laddar in uppdrag i bilen för att sedan användas till ruttplanering.
- **Ruttplanering** : Beräknar bästa väg mellan två punkter och sätter igång körningen för detta.
- **Regulatorväljare** : Justerar bilens reglersystem beroende på i vilket typ av vägsegment bilen befinner sig i, t.ex rak väg jämfört med en sväng.

3.4 Kommunikationsmodulen

Kommunikationsmodulens syfte är främst att ta emot och tolka instruktioner som ges trådlöst av användaren via användargränssnittet samt i ett senare läge distribuera tillbaka sensorinformation för att visa i användardatorn. De huvudsakliga komponenterna för att åstadkomma detta beskrivs i figur 7.



Figur 7: Användargränssnitt

- **Kommandotolkning**: Tolkar kommandon från servern och översätter det till format som styrmodulen kan uppfatta. Kommandot skickas vidare till styrmodulen.

- **Dataparser**: Tolkar data från styrmodulen och skickar tolkningen till servern

- **Stream to String konverterare**: Tar emot en ström från WIFI-sändaren/motagaren och konverterar den till sen sträng som skickas till servern. Kan även ta emot en sträng från servern och konvertera den till en ström som skickas till WIFI-sändare/mottagare.

- **TCP Webbserver**: Hanterar uppkopplingen mellan bilen och användardator och hanterar datan som går där i mellan.

- **Busshanterare**: Hanterar databussen så datan från de olika modulerna begärs och skickas rätt mellan varandra.

K EFTERSTUDIE

Efterstudie

Martin Dahl
Edvin Johansson
Mikael Josefsson
Charbel Youssef
Henrik Ahlinder
Sebastian Andersson

7 juni 2021

Version 1.0

Status

Granskad	Alla	2021-05-19
Godkänd		

Projektidentitet

Grupp E-post: TSEA56_2021VT_O2_G10@groups.liu.se

Hemsida:

Beställare: Anders Nilsson, Linköpings universitet
Tfn: 013-28 26 35
E-post: anders.p.nilsson@liu.se

Kund:
Tfn:
E-post:

Handledare: Peter Johansson
Tfn: +4613281345
E-post: peter.a.johansson@liu.se

Kursansvarig: Mattias Krysander
Tfn: 013-28 21 98
E-post: mattias.krysander@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Henrik Ahlinder	Projektledare	henah490@student.liu.se
Mikael Josefsson	Kommunikationsmodulansvarig, GUI-ansvarig	mikjo695@student.liu.se
Sebastian Andersson	Testansvarig	seban218@student.liu.se
Martin Dahl	Dokumentansvarig, mjukvaruansvarig	marda545@student.liu.se
Charbel Youssef	Sensormodulansvarig	chayo642@student.liu.se
Edvin Johansson	Styrmodulansvarig	edvjo404@student.liu.se

INNEHÅLL

1 Tidsåtgång 1
- 1.1 Arbetsfördelning . . . 1
- 1.2 Tidsåtgång jämfört med planerad tid . . . 1

2 Analys av arbete och problem 2
- 2.1 Reflektioner av de nya laborationerna . . . 2
- 2.2 Vad hände under de olika faserna? Och hur använde vi projektmodellen? . . . 2
- 2.3 Hur arbetade vi tillsammans? . . . 3
- 2.4 Hur fungerade relationen med beställaren? . . . 3
- 2.5 Hur fungerade relationen med handledaren? . . . 3
- 2.6 Distansläge . . . 3
- 2.7 Tekniska framgångar och problem . . . 4

3 Måluppfyllelse 4
- 3.1 Vad har uppnåtts? . . . 5
- 3.2 Hur fungerade leveransen? . . . 5
- 3.3 Hur har studiesituationen påverkat projektet? . . . 5

4 Sammanfattning 5
- 4.1 De tre viktigaste erfarenheterna . . . 5
- 4.2 Råd till de som ska utföra ett liknande projekt . . . 5

DOKUMENTHISTORIK

Version	Datum	Utförda förändringar	Utförda av	Granskad
1.0	2021-06-07	Första version	Alla	Alla

1 TIDSÅTGÅNG

1.1 Arbetsfördelning

Projektets arbetsfördelning har till största del fungerat väl och har bidragit till gott resultat. Bildbehandlingensgruppen blev dock isolerad från resten av projektgruppen. Dels på grund av att bildbehandlingen var till stor del fristående från övriga delar på bilen, dels för att övriga projektmedlemmar som inte gjort en förstudie i ämnet hade svårt att bistå bildbehandlingsgruppen i deras arbete. Tidigt i projektet var reglergruppens möjlighet att arbeta begränsad eftersom det tog längre tid att upprätta en användbar kommunikation i bilen än tänkt. Möjligtvis hade det varit bättre att närmare knyta samman kommunikationsgruppen med reglergruppen för att tillåta en mer jämn arbetsbörda i början av projektet.

1.2 Tidsåtgång jämfört med planerad tid

Sammanlagt har alla projektmedlemmar uppfyllt kraven på 230 timmar per person, och en del till och med mer än så. Det var dock inte många aktiviter som utfördes på planerad tid, utan nästintill samtliga aktiviterer tog antingen längre eller kortare tid än planerat. Ett dynamiskt omplanerande gjorde dock att en balans i arbestbörda i gruppen uppnåddes mot slutet av projektet, och alla medlemmar har arbetat ungefär lika mycket.

2 ANALYS AV ARBETE OCH PROBLEM

2.1 Reflektioner av de nya laborationerna

De nya laborationerna upplevdes som bra komplement i kursen och i huvudsak var omfattningen lagom. Fokuspunkterna på framförallt de två första laborationerna var mycket god och var till stor hjälp under projektets gång.

2.1.1 *AVR-laboration*

Labb-PM är ett bra och användbart verktyg att ha med sig i projektet. Framförallt nyttjas instruktionerna för Atmel studio frekvent. Till nästa år önskas ännu tydligare dokumentation för hur man designar timers i AVR, eftersom det krävs i alla projekt och är någorlunda knepigt. Vi önskar även att labben hade utförts på en mer realistisk nivå, t.ex. med de AVR-enheter som förmodligen kommer att användas i projektet (1284p) och med liknande uppsättning av hårdvara.

2.1.2 *Mätteknik*

Mycket bra labb! Logikanalysatorn har nyttjas frekvent under projektet för att felsöka problem som annars hade varit svåra att lösa. Det är bra att det finns med ”avancerade” funktioner såsom hur man läser av buss-signaler utöver vanliga pinnar; det är primärt denna typ av funktionalitet som nyttjas sedan.

2.1.3 *AVR-laboration med I2C*

Tanken med laborationen är god, men den hade behövt vara lite mer genomarbetad. Det var nyttigt att få en introduktion till hur man skriver kommunikationsprotokoll i AVR med avbrottstyrning. Synpunkter följer nedan.

Positiva synpunkter:

- Mycket bra sammanfattande appendix
- Bra att få praktisk erfarenhet av logikanalystorn för att felsöka kommunikation
- Bra att få en smak av hur ett avbrottstyrt kommunikationssystem kan konstrueras.

Negativa synpunkter: Labbskelettet var tyvärr oöverskådligt och det var svårt att få en överblick på vad som fanns förimplementerat och vad man själv förväntades skriva in. Dessutom var det krävande att förstå vad som hände i det som var förimplementerat. Det hade varit lämpligare att istället bryta isär labbskelettet till två övergripande delar: en där koden är färdigskriven och en med funktioner som ska skrivas; tänk på att det är svårt att utröna en struktur i program skrivna i ett ovant programspråk såsom C.

2.2 Vad hände under de olika faserna? Och hur använde vi projektmodellen?

LIPS-modellen följdes under hela projektet.

- **Före-fasen**: Gruppen etablerades, varje gruppmedlem fick ett ansvarsområde och ett gruppkontrakt togs fram. Gruppen delades in i tre mindre grupper med två medlemmar i varje. Varje par fick i uppdrag att skriva en förstudie som handlar om en av projektets tre huvuddelar, reglering, bildbehandling och kommunikation. Därefter utformades de dokument som beskriver projektets struktur och arbetsgång, bl.a en kravspecifikation, en projektplan och en tidsplan.

- **Under-fasen**: Denna fas inleddes med att skriva en designspesifikation. Projektgruppen erhöll en djupare insyn i vilka tekniska komponenter som skulle behövas och tidsplanen reviderades. Under projektets gång har både designspecen och tidsplanen reviderats kontinuerligt när förändringar uppstått. Efter designspecen var skriven påbörjades monteringen av bilens hårdvara och utvecklingen av bilens mjukvara. Testning av de implementerade programeringsbitarna skedde frekvent under arbetetsgång. Ytterligare dokumentation levererades bl.a. tekniska-dokumentation och användarhandledning. Denna fas avslutades med ett beslutpunktsmöte som gav möjlighet att omförhandla krav inför slutleveransen.
- **Efter-fasen**: Denna fas infattade en del förbättringar och utveckling av bilen och avslutades med en godkänd leverans. Denna efterstudie ingår i den här fasen.

2.3 Hur arbetade vi tillsammans?

Bra. Ett möte i veckan på ungefär 45 minuter har varit bra för att förankra gruppen och internt synkronisera arbete.

Projektuppdelning i olika grupper och att parallellt pipelinea projektet har gått bra och effektiviserat arbetet. Dock har kombinationen av distansläget tillsammans med paralleliseringen orsakat att olika grupper har löpt in i ”varandras” problem. Gruppen ser ingen självklar lösning som hade kunnat förhindra dessa problem eftersom arbetet utförs under en relativt kort period. Kanske hade problemen kunnat minskas vid en mer testdriven lösningsmetodik, med större tillträde till bilen. Eventuellt hade man kunnnat nyttja log-funktionen i git för att tydligare dokumentera ändringar i kod som har skett.

Vissa ändringar av den ursprunglig planen utfördes (främst p.g.a att lösningarna i planen var orealistiska) utan att kommuniceras till gruppen, vilket orsakad en viss röra för övriga medlemmar.

Kommunikationen har i huvudsak gått fort och varit tydlig. Dock har en central Facebook-messenger tråd nyttjats nästan enbart för kommunikation, vilket gjort det rörigt och i viss mån stressande att kommunicera med gruppen. En kommunikationstjänst som tillåter flera trådar, såsom Discord, är att föredra.

2.4 Hur fungerade relationen med beställaren?

Mycket bra. Beställaren har varit trevlig, tillmötesgående och enkel att kommunicera med.

2.5 Hur fungerade relationen med handledaren?

Mycket bra. Man fick enkelt kontakt med honom och han svarade fort och var allmänt behjälplig. I början av projektet var han ofta tillgänglig i labbet så att man kunde få hjälp fort - det var väldigt uppskattat och tacksamt.

2.6 Distansläge

Eftersom gruppen tilläts vara på campus märktes detta inte av i lika stor mån som i andra kurser.

Distansmöten har försvårat kommunikationen eftersom det är lätt att prata i mun på varandra. Dock underlättar det, eftersom man slipper boka sal, cykla till campus osv. Korta möten är enklare att uppnå p.g.a. detta. Det upplevdes som

lättare att fokusera på dokumentarbetet eftersom man enkelt kan ”skärma av” sina arbetskamrater och gå in i sin egen produktivitetsbubbla.

Överlag har det gått bra och smärtfritt.

2.7 Tekniska framgångar och problem

Det vi har åstadkommit kan sammanfattas enligt följande:

- Bilen kör runt banan.
- Bildbehandling (och datagenomströmning) i 30 FPS.
- Topphastighet av bilen i upp t.o.m. 1 m/s.
- Robust kantlinje- och stopplinjedetektion tillåter säker och rak körning även i kurvor.

Problem och utmaningar:

- Flera gånger under projektet har FPS:n flaskhalsats under 30 FPS men gruppen har vid alla tillfällen lyckats åtgärda detta. Ofta har låga batterinivåer varit orsaken till minskad FPS, vilket var ett problem som var svårt att finna; en del timmar spenderades på felsökning innan detta upptäcktes. Lägre FPS leder till oscillativt körbeteende.
- Artefakter som introduceras i bildbehandlingen ger upphov till spikar i kantavläsningen vilket tyvärr ger reglerfel som bilen i höga hastigheter inte klarar av att återhämta sig från.
- Kamerastativet som nyttjades är inte optimalt då det både är vint och skevt, samt ej tillåter till en bra vinkel mot vägen. Dessutom skakar kameran under körning vilket ger udda beteende i bildbehandlingen.
- Det har varit problem med versionhanteringsverktyget Git. Det parallella arbetandet har resulterat i att legacy-buggar har propagerats genom projektet på olika branches. Det har också varit svårt att upprätthålla en stabil branch för testningar eftersom ett system för detta utvecklades sent under projektet. En noggrann planering av hur Git ska nyttjas är en bra idé att utföra i planeringsfasen.
- Hårdvaruproblem har varit frekventa under projektet. Bland annat var ett virkort trasigt i både strömkontakten (som såg ut att ha varit avlödd) och en etsningsbro var avbruten. Detta upptäcktes först efter allting virats upp på kortet och en ansenlig mängd tid lagts på debugging.

3 MÅLUPPFYLLELSE

Projektgruppen har läst igenom och reflekterat över kursmålen. Samtliga medlemmar i projektgruppen känner att projektet har gett de möjligheter som krävs för att kunna uppfylla målen. Utöver kursmålen har samtliga bas- och slutkrav i kravspecifikationen avklarats.

3.1 Vad har uppnåtts?

Projektgruppen har byggt en bil som kan köras i manuellt, semi-autonomt och autonomt läge. Samtliga bas- och slutkrav som togs fram i början av projektet har klarats av.

3.2 Hur fungerade leveransen?

Den fungerade utmärkt och problemfritt trots distansläget.

3.3 Hur har studiesituationen påverkat projektet?

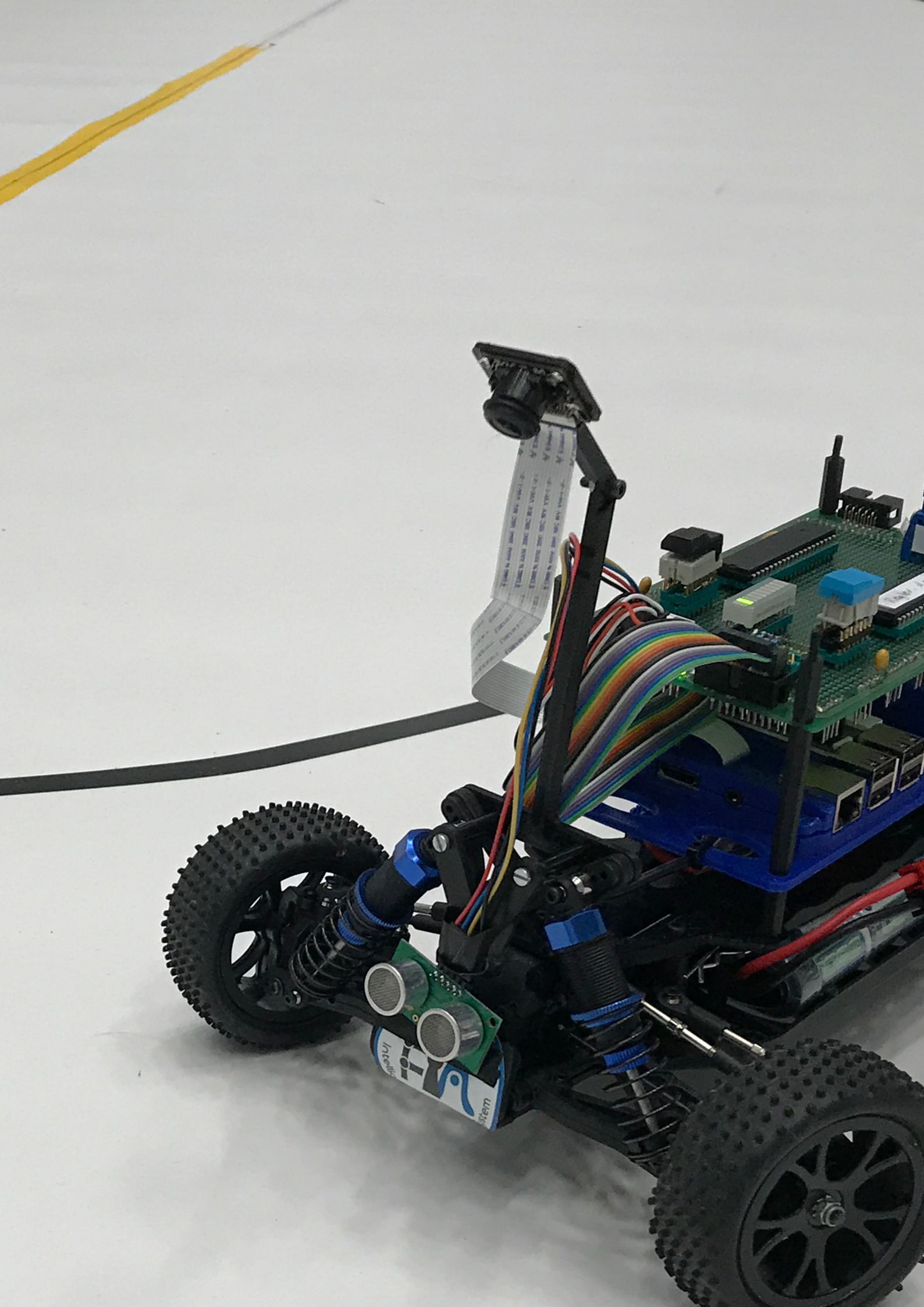
Samtliga medlemmar tycker att studiesituationen har varit acceptabel. Det har varit intensivt att läsa en kurs samtidigt som projektet, men inte för intensivt.

4 SAMMANFATTNING

4.1 De tre viktigaste erfarenheterna

1. Git - Svårt att arbeta i grupp parallellt och nyttja versionshantering på ett vettigt vis.
2. Projektplanering - vikten av att ha gjort en planering och bryta ned ett projekt i lösbara delar. Även nyttigt att förstå hur svårt det att konstruera rimliga tidplaner och komplexitetsestimering.
3. Vikten av tydligt kommunicera ändringar i projektplanering samt vad man arbetar med till resterande projektmedlemmar. Annars är det lätt hänt att flera projektmedlemmar arbetar med samma sak helt i onödan.

4.2 Råd till de som ska utföra ett liknande projekt

- Ha en tydlig plan för versionshantering i Git.
- Designspecen kommer att ändras under projektet, men en riktigt bra designspec kommer att göra arbetet mycket enklare.
- Maximera er testtid i Visionen - tiden går fortare än ni tror. Tänk igenom noggrant vad som ska testas och i vilken ordning. Prioritera det som endast kan testas i Visionen, det är lätt att hamna i sidospår och lägga fokus på ”fel” saker.

Denna sida är avsiktligt lämnad tom.