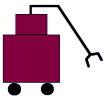


Teknisk Dokumentation

TSEA29 – Konstruktion med mikrodatorer

Grupp 8
Redaktör: Matilda Kaalhus
2018–12–15

v. 1.1

Status		
Granskad	Hugo Hörnquist	2018–12–15
Godkänd		

Projektidentitet

Namn	Ansvar	Epost
Kimberley Andersson	Integrationsansvarig	kiman080@student.liu.se
Adrian Byström	Projektledare	adrby928@student.liu.se
Hugo Hörnquist	Gitansvarig	hugho389@student.liu.se
Matilda Kaalhus	Dokumentansvarig	matka216@student.liu.se
Ellen Kavéus	Leveransansvarig	ellka591@student.liu.se
Juliette Winneroth	Modultestansvarig	julwi075@student.liu.se

Innehåll

Dokumenthistorik

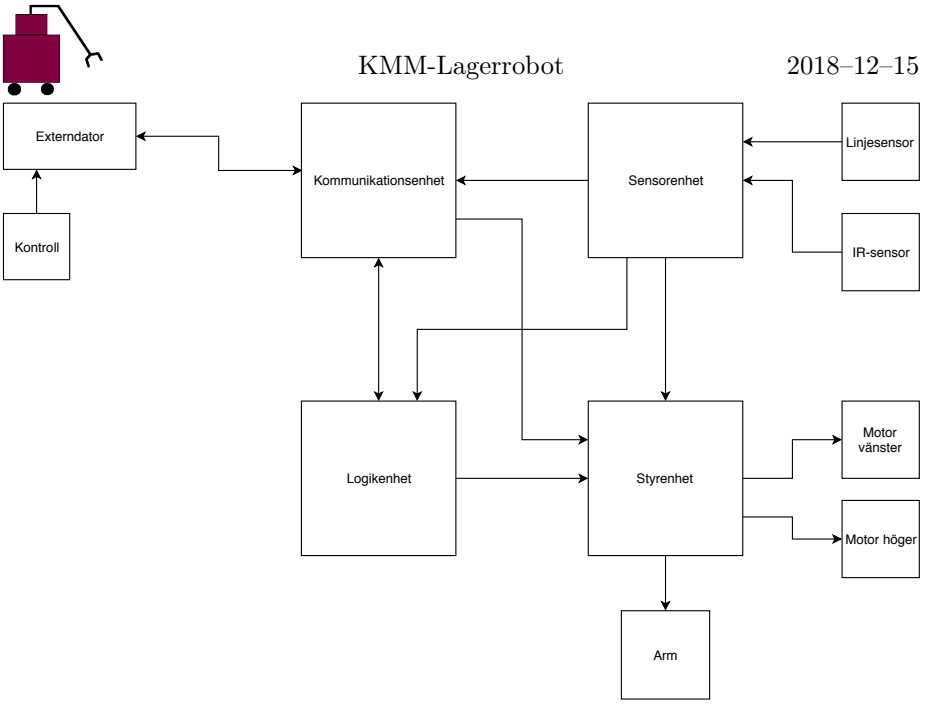
v.	Datum	Utförda Ändringar	Av	Granskad
1.1	2018–12–15	Uppdaterade kopplingsschemat för Sensormodul, då fel version var inlagd innan.	Grupp 8	Hugo Hörnquist
1.0	2018–12–13	Slutgiltiga versionen.	Grupp 8	Matilda Kaalhus

Kapitel 1

Inledning

Detta dokument är den tekniska dokumentationen av en lagerrobot som utvecklats under höstterminen 2018 vid Linköpings Universitet i kursen TSEA29 Konstruktion med mikrodatorer. Denna robot består av tre hårdvarumoduler, i form av två virkort och en Raspberry Pi 3 (se 2.1) som tillsammans fyller fyra ”roller”, beskrivna nedan. Utöver roboten så används en extern dator med ett grafiskt användargränssnitt för att bland annat styr roboten manuellt.

Roboten hanterar direkta kommandon från ett användargränssnitt på en extern dator. Kommandona skickas trådlöst via en wifi-länk till en Raspberry Pi 3, där de tas emot av kommunikationsenheten och sedan skickas vidare till relevanta moduler i systemet. Roboten kan även autonomt ta sig till en ”vara” som är definierad i det rutnät som roboten rör sig i. Detta rutnät definieras genom användargränssnittet på den externa datorn. Vid autonom styrning används sensordata från sensorenheten i styrenheten till att reglera robotens körning. Logikenheten använder behandlad sensordata för att bestämma hur roboten ska ta sig fram i lagret och skickar därefter kommandon till styrenheten. Under tiden tar kommunikationsenheten emot styr-och-sensordata från de relevanta enheterna för att kunna uppdatera användaren om vad roboten gör i nuläget. Robotens arm kan sen styras manuellt för att plocka upp ”varan”, därefter tar sig roboten autonomt till utlämningsstationen och autonomt lämnar ”varan”. Vid manuell upplockning skickar kommunikationsenheten indata från kontrollen, som används för armens styrning, till styrenheten som rör och reglerar armen.



Figur 1.1: Övergripande blockschema för systemet

I *Figur 1.1* ovan visas ett blockschema över hur de olika enheterna, sensorer och motorer kommer att kommunicera med varandra. Denna kommunikation beskrivs mer detaljerat i kapitel 4.

Roboten använder sig av en linjesensor och en IR-sensor för att ta sig fram i lagret och upptäcka hinder på vägen. Linjesensorn är placerad på undersidan av roboten, något förskjuten i ordinarie färdriktning. IR-sensorn är placerad på framsidan av robotens basplatta, framför robotens arm sådant att sensorn pekar framåt i robotens färdriktning.

Det finns två drivkretsar till hjulen för att röra roboten framåt och svänga. Den ena drivkretsen har uppgiften att styra motorerna på de vänstra hjulen och den andra styr motorerna på de högra hjulen. Dessa drivkretsar är placerade på den robotplattform som projektet utgår ifrån intill deras respektive hjulpar. Armen styrs av sina inbyggda servon och ingår även den i robotplattformen.

1.1 Syfte

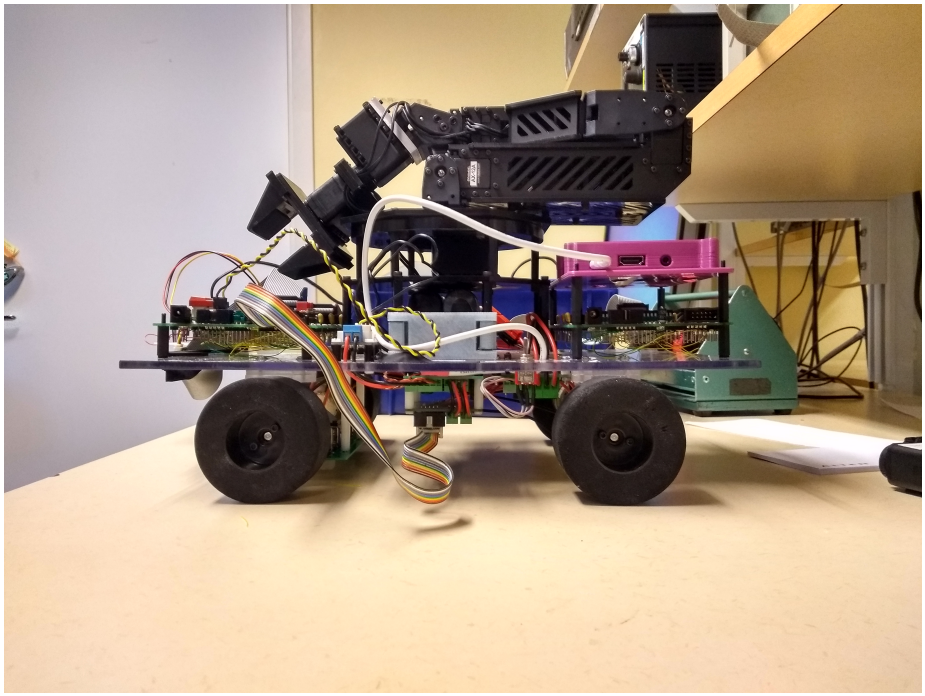
Syftet med detta projekt är att skapa en funktionell laggerrobot som autonomt kan röra sig på en given bana samt manuellt plocka upp varor för att sedan autonomt avlämna dessa på en utsatt avlämningsstation. Detta dokument har i syfte att beskriva hur denna robot är konstruerad i form hård- och mjukvara. Systemet är uppbyggt av fyra huvudmoduler, sensor-, logik-, kommunikation- och styrmodulen som alla finns dokumenterade i detta dokument.

Kapitel 2

Produkten

Den allmänna funktionaliteten av produkten som beskrivs i detta dokument är den av en lagerrobot. Den kan styras både genom ett användargränssitt på en extern dator via en wifi-länk och denna styrning kan vara både fri enligt användarens instruktioner eller i autonomt läge där systemet själv kan navigera i ett rutnätslager som specificerats genom det externa gränssnittet. En specificerad punkt i rutnätet kommer definieras som upplockningsstation dit systemet kommer navigera för att gå in i manuellt armstyrningsläge. Där kommer användaren styra armen tillhörande systemet för att plocka upp ”vara”. Vid signal från användaren kommer systemet återgå till autonomt läge för att navigera till fördefinierad avlämningsstation i rutnätet och lämna av varan.

Det allmänna användningsområdet för denna produkt är främst som lagerrobot vid ett lager som är konstruerat enligt banspecifikationen.



Figur 2.1: Bild av roboten.

Kapitel 3

Teori

3.1 Datastrukturer

3.1.1 Kart-nod

I logikenheten så representeras kartan av lagret som ett antal noder med x- och y-koordinater och vägar som leder ut ur noden. Varje nod är alltså en korsning i lagret och har ett unikt sett koordinater. Den har även information om vägarna upp, ner, vänster och höger, där värdet 1 innebär en tillgänglig väg, och värdet 0 innebär att väg där inte finns, eller att roboten upptäckt ett hinder på denna väg. Till nod struct:en tillhör de nedanstående fält som initialiseras vid skapandet av en ny nod:
self.xCoord
self.yCoord
self.wayUp
self.wayRight
self.wayDown
self.wayLeft

Till Node klassen tillhör även en funktion addObstacle som sätter en av nodens fyra vägar till 0 beroende på vilken input den får.

3.1.2 TCP

Både ”server”-delen som sitter på den externa datorn och ”client”-delen på RP3:an är skriven i python, och är implementerad som ett TCP-bibliotek. Biblioteket är uppbyggt som två klasser, server och client, som innehåller wrapper funktioner för att underlätta kommunikationen mellan enheterna. Klasserna innehåller varsin socket medlem, så det går att skriva egna funktioner via denna. I kommunikationsenheten så importeras ”client”-delen som ett bibliotek, men servern på den externa datorn är sitt egna program, och pratar via en unix-socket.

3.2 Algoritmer

3.2.1 PD-reglering

PD-reglering används av styrmodulen för att korrigera sin styrning med den sensordata den får angående dess position i relation till den tejp den ska följa. PD-reglering är reglering som tar hänsyn till det proportionella felet, det vill säga det avstånd mellan önskad position och nuvarande position, och derivatan av styrning, det vill säga skillnaden mellan det nuvarande proportionella felet och det förra proportionella felet.

Den enklaste typen av reglering är den som endast tar hänsyn till det proportionella felet eftersom det kommer indikera vilken generell korrigering behövs. Problemet med att endast använda proportionellt fel för att avgöra korrigering är att vid mindre responsiva eller fördröjda system kommer överslaget från den önskade positionen hindra en effektiv styrning. Denna reglering kan vidareutvecklas genom att lägga till ännu ett element i korrigeringsalgoritmen, en integralskonstant, som tar hänsyn till konstanta fel som skillnader mellan drivkretseffektivtet eller liknande. Detta skulle resultera i en så kallad PID-reglering.

3.2.2 Pathfinding

Robotens väg igenom lagret fungerar på så sätt att pathfinderfunktionen först kallas. Där i letas först upp vilken nod som roboten ska ha som sin slutnod. Detta bestäms efter vilken av de två noder som utlämnings stationen ligger emellan som är närmast till robotens nuvarande position. Sedan räknas närmaste vägen ut med hjälp av en egenskriven bredden-först sökalgoritm. Då roboten ska kunna hantera hinder i vårt lager är algoritmen anpassad för detta.

När en väg hittats så kommer roboten börja köra. För varje ny nod som roboten kommer till så kommer den vända sig i nästa nods riktning och sedan kolla ifall det finns ett hinder på denna väg. Om det inte finns något hinder kommer det fortsätta till noden som planerat annars tas vägen bort som en möjlig transportväg och en ny väg till slutnoden räknas ut från den nuvarande positionen.

3.3 Protokoll

För att kunna kommunicera mellan de olika enheterna och för att de ska förstå varandras meddelanden, har vi tagit fram en del protokoll. Dessa protokoll är en anvisning på hur meddelande ska byggas för att de ska vara tolkningsbara.

Ett antal olika protokoll används mellan enheterna. Anledningen till olika är olika krav på mängden data, diversiteten av data, samt tillgänglig hårdvara.

3.3.1 Kommando-protokoll

Detta visar formen på alla kommandon som skickas till styrenheten. Detta protokoll ska följas hela vägen från den externa datorn, via logikenheten och till styrenheten. Alla kommandon respresenteras av 8 karaktärer anting 1:or eller 0:or. Fram till den process som hanterar kommunikationen mellan kommunikations- och styrenheten så är dessa kommandon strängar innan de omvandlas till heltal. "0000—-"

Kommando lista —-:
0001: stop
0010: Tangent pil upp - kör rakt fram
0011: följ linjen framåt
0100: Tangent pil ner - backa
0101: följ linjen bakåt
0110: Tangent pil vänster - sväng vänster
0111: Tangent pil höger - sväng höger
1000: Tangent PgUp - 45 vänster
1001: Tangent PgDn - 45 höger

3.3.2 Arm-protokoll

Arm protokollet visar hur kommandon till armen ska skrivas. Dessa ska skickas från den externa datorn vid tangenttryck. Om roboten befinner sig i manuellt läge kan dessa skickas när som, annars behöver roboten vara framme vid en upplockningsstation. Formatet på dessa kommandon fungerar som ovan
"1111—-"
Kommando lista —-:
0001: Tangent 1 - Medurs vridning klo
0010: Tangent 2 - Moturs vridning klo
0011: Tangent q - IN
0100: Tangent w - UT
0101: Tangent a - UPP
0110: Tangent s - NER
0111: Tangent z - VÄNSTER
1000: Tangent x - HÖGER
1001: Tangent , - KLÄM
1010: Tangent .- SLÄPP

3.3.3 Kart-protokoll

Detta protokoll ska följas vid meddelande innehållande lagret och varor. Denna typen av meddelande skickas från den externa datorn då användaren fyllt i storlek på lagret, och valt varors position och nummer. ”X,Y,x,y” Där

`X` är storleken på lagret i X-led, i antal rutor.

`Y` är storleken på lagret i Y-led, i antal rutor.

`x` är utlämningsstationens x-koordinat

`y` är utlämningsstationens y-koordinat.

Då utlämningsstationen alltid ligger i ett av lagrets fyra hörn har den 4 möjliga positioner: $(0,\,0)$, $(0,\,Y)$, $(X,\,0)$ och $(X,\,Y)$.

3.3.4 Varu-protokoll

Varu-protokollet är det protokoll som följs när man skickar information om de varor som finns i lagret, från den externa enheten till kommunikationsenheten och sedan vidare till logikenheten. ”v1,x1,y1,s1;v2,x2,y2,s2;...” Där för varje vara i lagret det anges:

`v` varonumret/namnet.

`x` är x-koordinaten på rutan som varans upplockningsstation beffiner sig på.

`y` är y-koordinaten på rutan

`s` är vilken sida på rutan som varan befinner sig på. Sidorna benäms 0-3 och är namngedda klockvis där 0 är vänstra sidan på rektangeln och 3 nedre sidan.

3.3.5 Paket

Data som skickas via UART (kommunikationsenheten till styr- och sensorenheten) paketeras i paket, vilka är implementerade genom C strukturer. All data följer samma grundstruktur. Alla paket börjar med ett pakethuvud som berättar vad paketet kommer att innehålla, samt hur långt (i UART frames) det kommer vara.

Strukturen på dessa paket är följande:

$$
\begin{aligned}
\textit{struct}\ \text{packet} &\rightarrow \{\text{header}, \text{payload}\} \\
\textit{struct}\ \text{header} &\rightarrow \{\texttt{uint8\_t}\ \textit{id} \\
&\quad\ \texttt{command\_type}\ \textit{type} \\
&\quad\ \texttt{uint8\_t}\ \textit{length}\} \\
\textit{union}\ \text{payload} &\rightarrow \{\texttt{struct handshake} \\
&\quad\ \texttt{uint8\_t distance} \\
&\quad\ \texttt{char steercomm} \\
&\quad\ \texttt{char steerdata} \\
&\quad\ \texttt{enum linedata}\}
\end{aligned}
$$

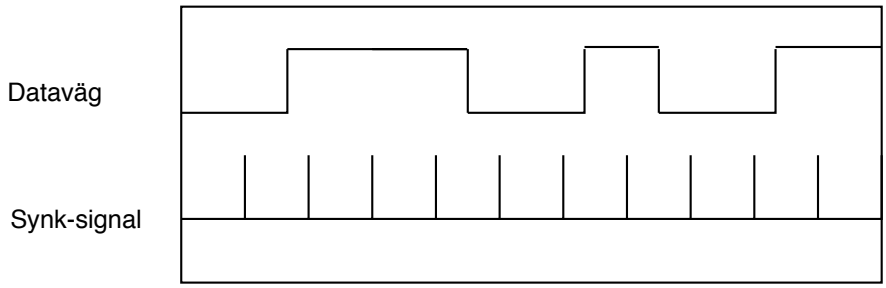
UART handskakning

AVR:en sänder ut en handshake signal i form av tidigare beskrivet packetprotokoll som då väntar på ett svar på samma form. Sensormodulen skickar ut synksignal SYN och tar sedan emot ett acknowledge-signal ACK från RP3:an som då bekräftar att kommunikationen mellan dem är uppsatt och paket kan börja sändas från Sensormodulen.

3.3.6 ExDe

ExDe är en synkron, enkelriktad, databuss, och är ett alternativ till UART. Den består av 3 pin, vilka är:

- Data
- Sync
- Retursync / handskakning



Figur 3.1: Exempel på överföring via ExDe.

Bussen körs genom att varje bit läggs under ett intervall på data-pinnen, och i ”mitten” av intervallet går synk-signalen en mycket kort tid hög. I figur 3.1 visas sändning av sekvensen $0, 1, 1, 1, 0, 0, 1, 0, 0, 1$.

Notera att ingen standard angående storlek på paket finns, samt ingen form av paket start/stop. Det är upp till användaren att sätta upp om sådant behövs.

ExDe handskakning

Den tredje linan används för handskakning. Vid start ska den sändande enheten skicka ut en synk-signal ”nu och då”. När den mottagande enheten plockar upp den ska den skicka tillbaka en egen synksignal på motsvarande form[1]. När sändaren fått tillbaka signalen anses handskakningen komplett, och överföring av data kan påbörjas.

3.3.7 GaG & Robert

På den externa datorn kör två program. Användargränssnittet (GaG), samt servern som vidarebefodrar meddelanden mellan GaG och kommunikationsmodulen. Serverns, vilken nedan (för enkelhetens skull) kallas Robert, finns dokumenterad i sektion 3.1.2.

GaG och Robert delar meddelanden med varandra över en sockel. Alla meddelanden utgår från att Robert skickar någonting till GaG, och GaG svarar enligt protokollet.[2]

Allmänna deklarationer

$$\begin{aligned}
\langle \text{cm} \rangle &::= \langle \text{int} \rangle \ \#\text{centimeter är heltal} \\
\langle \text{id} \rangle &::= \langle \text{int} \rangle \\
\langle \text{prodid} \rangle &::= \langle \text{id} \rangle \\
\langle \text{direction} \rangle &::= \langle \text{left} \rangle \mid \langle \text{up} \rangle \mid \langle \text{right} \rangle \mid \langle \text{down} \rangle
\end{aligned}$$

Kartdata

- Första koordinaten än kartans storlek.

[1] Det här är enda gången den mottagande enheten skickar någonting
[2] Protokollet är på BNF form, kort introduktion finns i appendix A

- Andra koordinaten är utlämningsstationens plats (var inte den alltid i origo?)

$$\text{`}karta'\ \langle\text{kart-data}\rangle \to \text{`}karta'\ \langle\text{kart-data}\rangle$$
$$\langle\text{kart-data}\rangle ::= \langle\text{coord}\rangle\ \langle\text{coord}\rangle$$
$$\langle\text{coord}\rangle ::= \langle\text{int}\rangle\ \langle\text{int}\rangle$$

Sidlängd

Längden på en kvadrat i banan i cm.

$$\text{`}sidelength'\ \langle\text{cm}\rangle \to \text{`}sidelength'\ \langle\text{cm}\rangle$$

Plockstationer

Lista över var plockstationer är belägna. Svaret börjar med ett heltal vilket är antalet plockstationer, fölt av en lista av dem. ⟨id⟩ är ett godtyckligt (unikt) numeriskt id (av plockstationen $\iff$ prodkuten).

Sen är det en koordinat för vilken rektangel produkten står i, och sedan vilken riktning man ska komma åt produkten från.

$$\text{`}prodlist'\ \langle\text{int}\rangle\ \{\ \langle\text{product}\rangle\ \}$$
$$\langle\text{product}\rangle ::= \langle\text{prodid}\rangle\ \langle\text{coord}\rangle\ \langle\text{side}\rangle$$
$$\langle\text{side}\rangle ::= \langle\text{direction}\rangle$$

Hämta Prodkut

$$\text{`}fetchprod'\ \langle\text{prodid}\rangle \to \text{`}fetchprod'\ \langle\text{prodid}\rangle$$

Armkommandon

$$\text{`}armcomm'\ \langle\text{armcomm}\rangle \to \text{`}armcomm'\ \langle\text{armcomm}\rangle$$

⟨armcomm⟩ är ett heltal $\in [F1_{16}, FA_{16}]$, där de olika värdena representerar:

Nr	value	Tangenent	kommand
F1	11110001	1	CW
F2	11110010	2	CCW
F3	11110011	q	IN
F4	11110100	w	UT
F5	11110101	a	UPP
F6	11110110	s	NER
F7	11110111	z	VÄNSTER
F8	11111000	x	HÖGER
F9	11111001	,	KLÄM (-check)
FA	11111010	.	SLÄPP

Styrkommando

$$\text{`}styrcomm\text{'} \rightarrow \text{`}styrcomm\text{'}\ \langle\text{styrcomm}\rangle$$

$\langle$styrcomm$\rangle$ är ett heltal $\in [1, 9]$, där de olika värdena representerar:

Nr	value	Tangenent	kommand
1	00000001	None	Stop
2	00000010	Pil Up	Kör rakt fram
3	00000011	None	Följ linje bak
4	00000100	Pil Ner	Backa
5	00000101	None	Följ linje fram
6	00000110	Pil Vän	Sväng vänster
7	00000111	Pil Hög	Sväng höger
8	00001000	PgUp	45 grad vänster
9	00001001	PgDown	45 grad höger

Telemetri

$\emptyset$ innebär att ingenting returneras.

$$\text{`}thought\text{'}\ \langle\text{any string}\rangle \rightarrow \emptyset$$
$$\text{`}set\_robot\text{'}\ \langle\text{coord}\rangle \rightarrow \emptyset$$
$$\text{`}set\_heading\text{'}\ \langle\text{direction}\rangle \rightarrow \emptyset$$

3.4 Datadelare

Datadelare används på Rasberry Pi:en för att inkommande data ska delas med både kommunikations- och logikenheten. Datadelaren skötar UART handskakningarna och sedan fork:as för att starta processer som sköter styr- respektive sensorenhetens kommunikation med RP3:an.

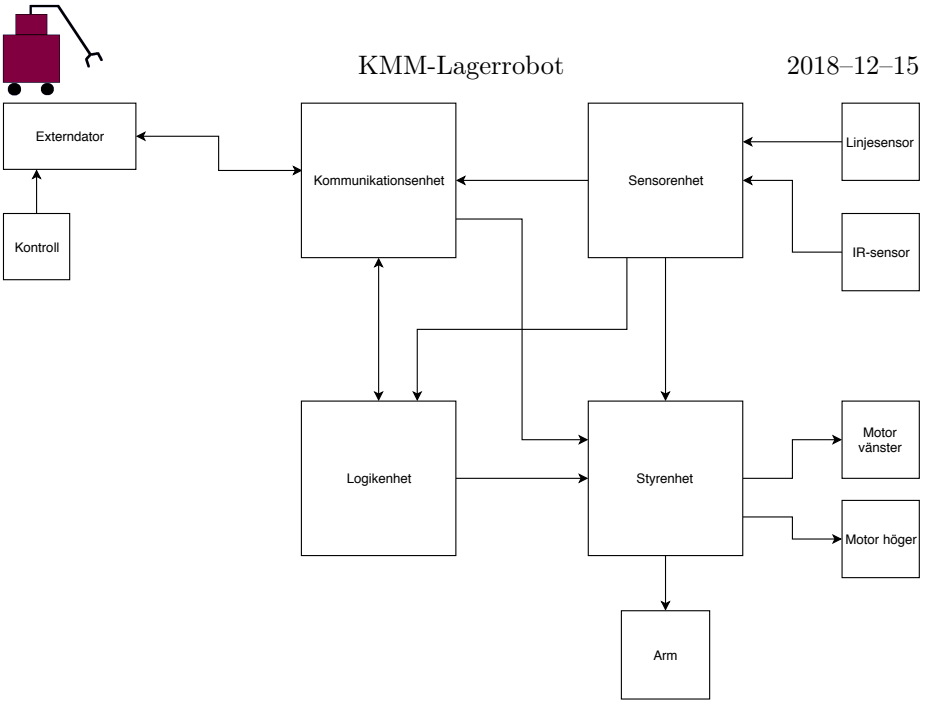
Kapitel 4

Systemet

Detta kapitel syftar att beskriva hur systemet hänger samman och har stort fokus på att beskrivat kommunikationen mellan modulerna. Mer modulspecifik information finns att hitta i respektive delkapitel i nästkommande kapitel 5.

4.1 Kommunikationen mellan enheter

Fördelningen av arbete mellan modulerna kräver god kommunikation mellan modulerna för att systemet ska fungera som önskat. Denna kommunikation sker på fyra olika typer av protokoll; TCP-sockets, buffertfiler, UART-kommunikation och eget EXDE-protokoll. Bilden nedan 4.1 visar i vilka riktningar datavägarna går i och mellan vilka moduler som de går mellan. Utförligare beskrivningar av de olika protokollen finns att finna i 3.3 samt nedan för modul-till-modul-specifik beskrivning av deras kommunikation.



Figur 4.1: Kommunikationen mellan produktens enheter.

.

4.1.1 Logik- och Kommunikationsenhet

Mellan logik och kommunikationsenhet så sköts kommunikationen via bufferfiler. Enheterna har hårdkodade platser i map-strukturen som de läser/skriver till för att prata med varandra.

4.1.2 Logik- och Sensorenhet

Kommunikationen mellan dessa enheter sköts via en UART-USB kabel, med ett protokoll skrivet av gruppen, med handskakning så att de olika fysiska enheterna "vet vem som är vem". Nerskalad sensordata skickas från sensorenheten till logikenheten via data-delaren på kommunikationsenheten. Den nedskalade sensordata innebär att det enda som logikenheten får (och behöver få) är om roboten är på linjen, på linjen lite till vänster, på linjen lite till höger, vid en t-korsning åt höger, vid en t-korsning åt vänster, vid en fyr-vägs korsning, vid upp-plockningsstationen, eller ej vid en linje. Detta för skicka kommandon utefter den väg som roboten räknat ut. Logikenheten kommer även få avståndsdata från IR-sensorn i form av metern. Detta så att logikenheten kan hantera hinder och upphämtning av varor.

4.1.3 Kommunikations- och Sensorenhet

Använder sig av samma teknik som logikenheten, och tar emot samma information. Detta sköts av en extern process, en data-delare.

4.1.4 Logik- och Styrenhet

Mellan logik- och styrenheten går det en USB UART. De kommandon som logikenheten vill skicka till styrenheten sparas på en buffer. Den tråd som hanterar skickandet och mottagandet av meddelanden mellan enheterna hämtar sedan information från buffern för att kunna skicka vidare till styrenheten. Denna tråden sköter även hämtningen av styrdata över UART:en och sparar datan på en fil som logikenheten sedan läser ifrån. Kommandona skickas enligt det protokoll som gruppen skrivit, se 3.3.1 och 3.3.2.

4.1.5 Kommunikations- och Styrenhet

Då kommunikationsenheten och logikenheten båda två ligger på samma kort så använder de samma kommunikationsvägar. Kommunikationen mellan kommunikations- och styrenheten fungerar därför på samma sätt som den mellan logik- och styrenheten. All denna kommunikation går på samma UART.

4.1.6 Kommunikationsenheten och Externa datorn

Kommunikationen mellan den externa datorn och RP3:an sköts av en TCP-sockets som öppnas när roboten start/resetas. Värt att nämna är att den externa datorn mest kommer att ligga och lyssna på att roboten behöver något i autonomt läge, då den inte kommer att behöva indata mer än i början, då kartan och varan som roboten ska hämta kommer skickas över. Om nu roboten slår över i manuellt läge så kommer den fråga den externa datorn efter nästkommande kommando. Den externa datorn kommer då behöva bygga upp en kö av kommandon om den skickar dessa snabbare än roboten kan utföra de, då kommunikationsenheten endast kommer be om ett meddelande i taget.

4.1.7 Sensorenhet- och Styrenhet

Sensorenheten packeterar sensordata från IR-sensor samt Linjesensorn och skickar över dessa till Styrenheten via ett protokoll som vi valt att kalla 'exde'. Dessa aktiveras via en setup samt en sändingsrutin som görs från båda hållen. Linjedatan skickas till Styrmodulen från Sensormodulen i paket dock består datan inom paketet av råa"11-bitssekvenser istället för den nedskalade abstraherade värden som logikmodulen får. En 1:a indikerar att sensorn är på banan (tejpen) och en nolla indikerar att den är utanför. Styrmodulen använder sedan denna data för att kunna PD-reglera sin styrning för att följa tejpen.

Kapitel 5

Modulerna

5.1 Logikmodul

Logikenheten är den enhet som sköter allt logiskt tänkande. Denna är alltså inte aktiv i manuellt läge då kommandon skickas direkt från kommunikationsenheten till styrenheten. De enda funktionerna som utförs i manuellt läge är att kolla ifall reset knappen eller switch knappen tryckts ner och utför i sådant fall rätt åtgärder. Dessa funktioner utförs alltid, alltså även i autonomt läge.

Vid start av roboten frågar kommunikationsenheten användaren efter lagrets rutors sidolängd. För att logikenheten ska kunna starta behövs denna längd och denna loopar alltså tills ett meddelande med storleken tagits emot.

I autonomt läge behöver logikenheten en del information för att veta vart roboten ska och sedan leda den dit.

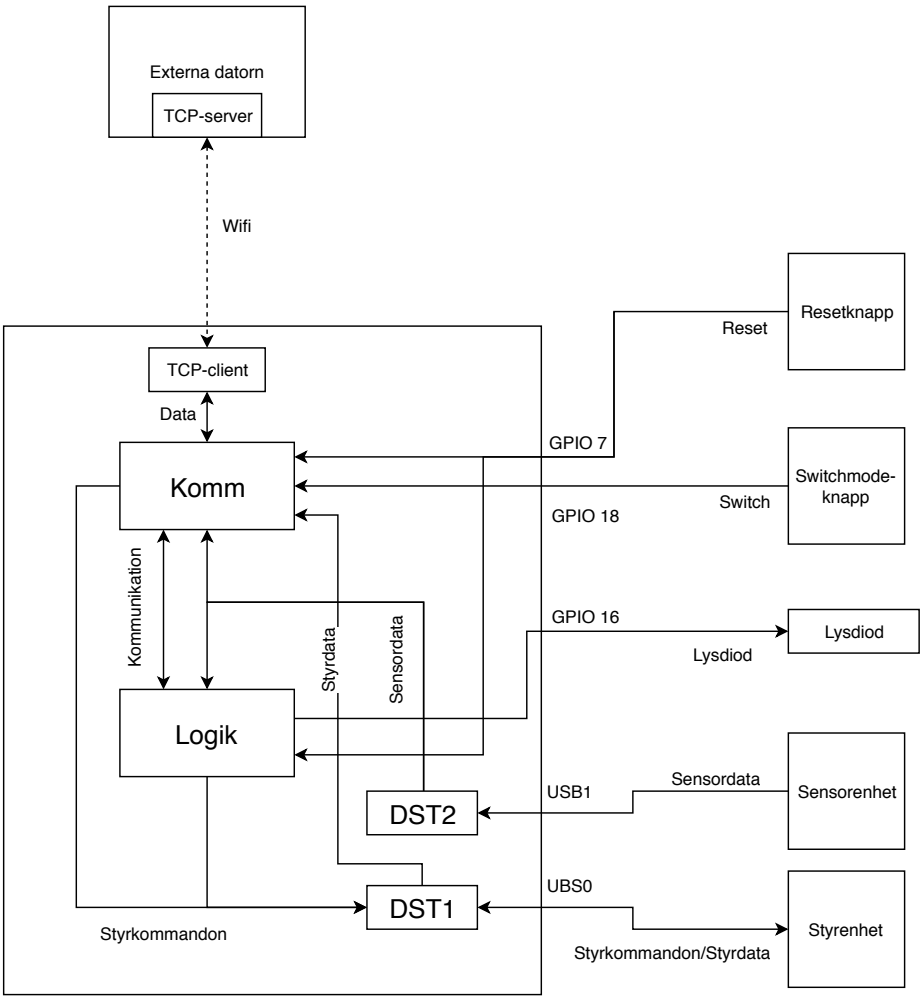
Den informationen logikenheten behöver är:

- Lagrets utseende
- Var vilka varor finns i lagret
- Var varor ska lämnas
- Vilken vara som ska plockas upp

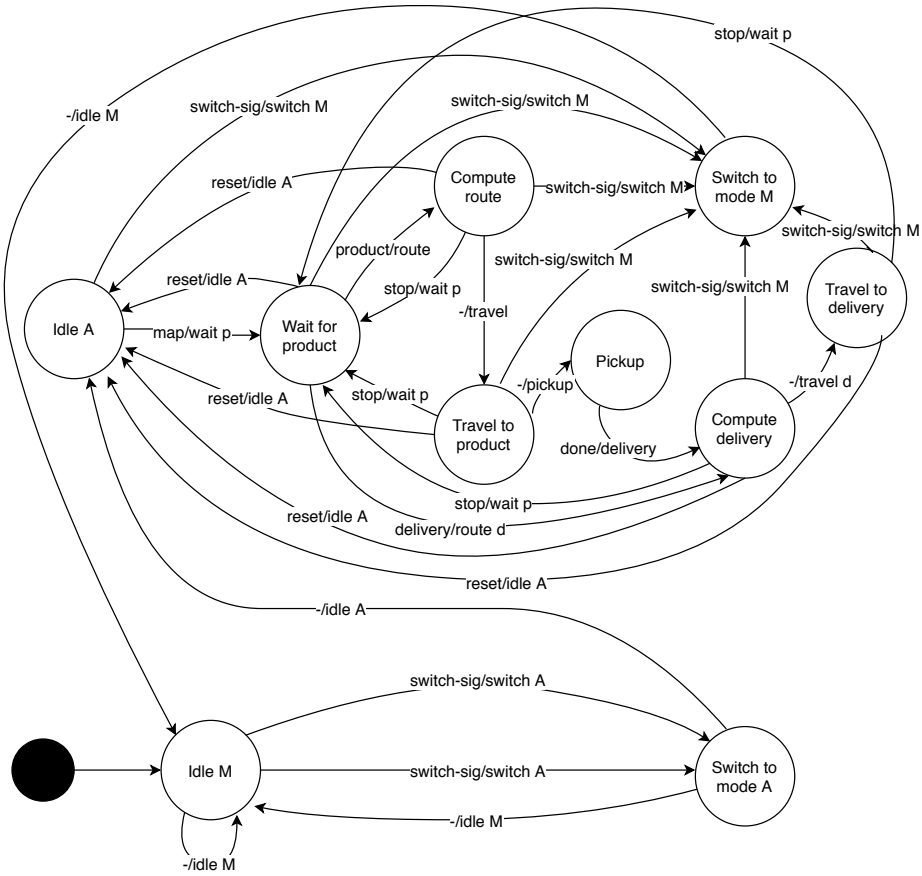
Logikmodulen påbörjar sitt program med att vänta på ovanstående data från kommunikationsmodulen. Där efter letar modulen upp en lämplig väg till sitt mål, som antingen är en korsning intill en plockstation, eller utlämningsstationen.

Om målet är en plockstation åker roboten från korsninngen mot plockstationen tills den hittar den. Därefter svänger den in och ställer sig i position för att plocka upp. Logikenheten meddelar sedan kommunikationsenheten att en vara skall plockas upp, och väntar på att den signalerar till baka att varan är upplockad. Slutligen sätts utlämningsstationen som nytt mål.

Om målet istället är utlämningsstationen åker roboten dit. Där ska den följa den väg som ”lämnar lagret”. När den är framme vid den t-korsning som signifierar positionen för avlämning så skickas ett kommando till styrenheten att lämna varan. Logikenheten meddelar kommunikationsenheten att varan är levererad och kommer sedan vänta på att kommunikationsenheten meddelar om en ny vara som ska hämtas.

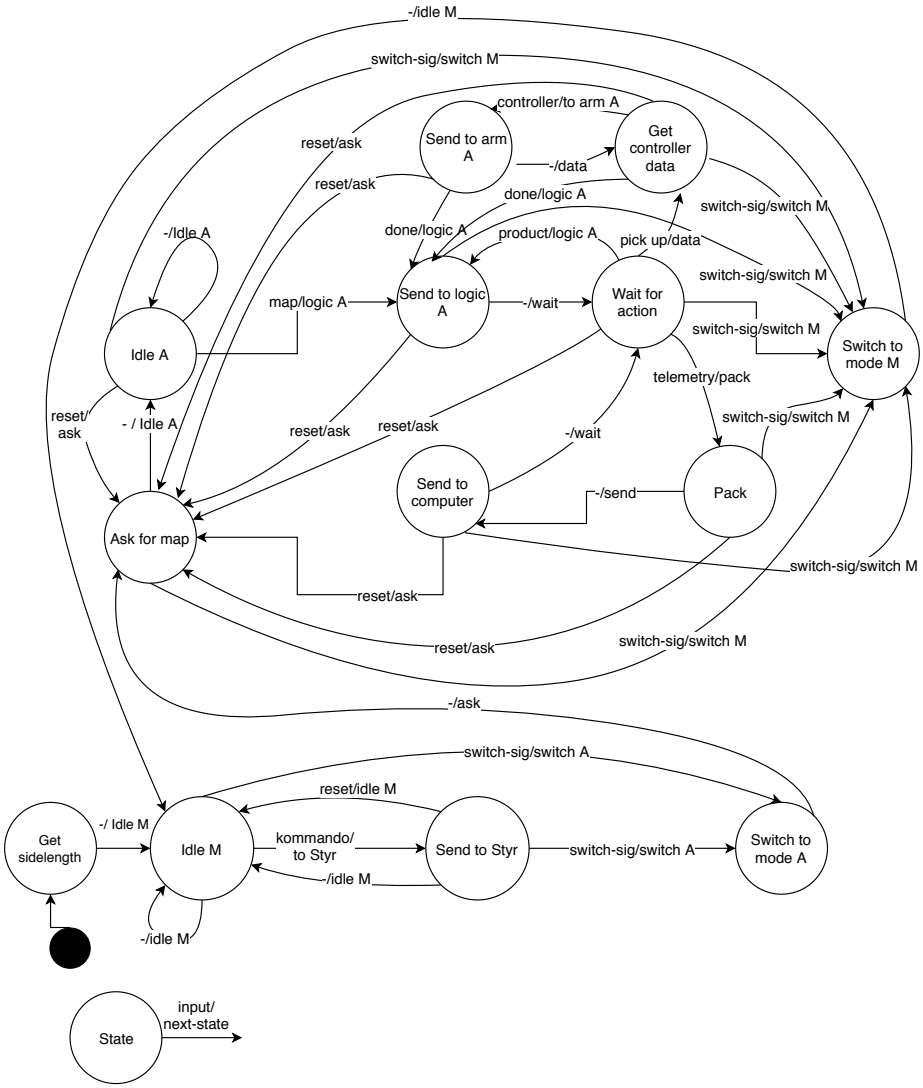


Figur 5.1: Kopplingsschema för RPi3:an, där logikmodulen och kommunikationsmodulen ligger på.



Figur 5.2: Flödesschema för logikenheten

5.2 Kommunikationmodul



Figur 5.3: Flödesschema för kommunikationsenheten

Vid kommunikationsenhetens uppstart begärs storleken av varje ruta (i cm) från den externa datorn. Därefter skickar den en signal till logikenheten, och väntar på ett ”ready” svar innan dess huvud-loop kör igång.

Kommunikationsenhetens huvud-loop är uppdelad i två delar, en som alltid gör samma sak, och en som gör olika saker beroende på om den är i autonomt eller manuellt läge.

I den generella delen kollas om RESET- eller SWITCH-knappen tryckts ner, och utför deras rutiner vid behov.

Manuellt I manuellt läge vidarebefodrar kommunikationsmodulen kommandon från externa datorn till styrenheten. Detta sker genom att kommunikationsenheten kallar på klient.

När ett kommando tagits emot läggs detta på en buffer av datadelaren. Den tråd som tar hand om kommunikationen mellan styrenheten och de enheter som ligger på RP3:an kommer sedan hämta information från buffern och skicka vidare över UART till styrenheten. I manuellt läge gör inte kommunikationsenheten mer än så.

Autonomt Formen på kartan begärs ut om den inte redan är känd, och skickas sedan vidare till logikenheten.

På samma sätt hämtas en lista av varor och deras positioner, samt slutligen vilken vara som ska hämtas.

Kommunikationsenheten väntar sedan på en signal från logikenheten att roboten är redo för att plocka upp en vara. Då begär Kommunikationsenheten armkommandon från den externa datorn, och utför dem, tills att en ”pickup done”-signal kommer.

Kommunikationsenheten kollar varje loop om denna har fått något nytt meddelande från logikenheten. Möjliga meddelanden är:

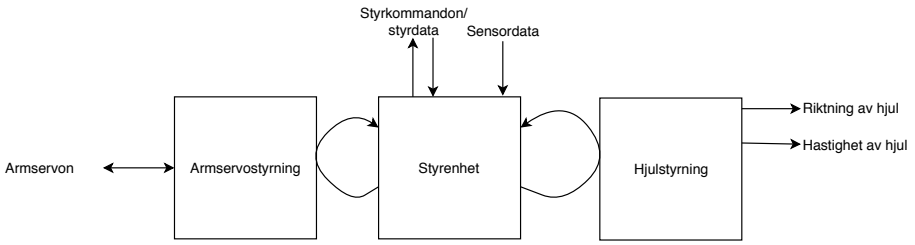
- In position
- Getting Item
- Item Delivered
- Next node

Vid ett ”Next Node” meddelande så skickar kommunikationsenheten vidare denna nod till externa datorn för att användaren ska kunna se hur roboten beter sig och var den är på väg. De andra meddelandena sätter flaggor för att main loopen ska fungera korrekt.

5.3 Styrmodulen

Styrmodulen ansvarar för den fysiska styrningen av robotplattformen, alltså styrning av de drivkretsar kopplade till hjulen samt styrningen av de servon som utgör armen. Denna modul är därför strikt beroende av de kommandon och sensordata som den får från andra moduler för att utföra sina uppgifter på önskat sätt. Eftersom styrmodulen agerar som mellanhanden mellan besluten av styrning och den hårdvaruimplementationen av styrningen så prioriterades abstraktion och läsbarhet vid utvecklandet av modulen. Detta innebär att modulen är främst uppdelad i tre ansvarsområden: huvudområde, hjulstyrning och armstyrning.

Bilden nedan illustrerar det informationsflöde och kontrollflöde som finns i modulen. De rundade pilarna mellan styrenhet och de andra delarna av modulen indikerar hur styrenheten kallar på metoder för att utföra tidsbegränsade uppgifter.



Figur 5.4: Beskrivande bild över koduppbyggnaden av styrmodulen

5.3.1 Styrenhet

Huvudområde är den fil styrenhet.c som hanterar all kommunikation till andra moduler och delegerar ut CPU-tid åt utförande av den önskade funktionaliteten. Det är här main-funktionen ligger som börjar med att initialisera och konfigurera all kommunikation till och från modulen för att sedan gå in en evig for-loop. Denna eviga for-loop återvänder kontrollflödet till efter en uppgift har utförts till sitt slut eller efter ett läge har brutits genom signal från extern modul. Här uppdateras de register som används för att kontrollera om extern modul har signalerat för avbrytning, vilket innebär att den nya signalen som skrivits in i den interna globala variabeln ”command register” skriver över det tidigare värdet i den interna globala variabeln ”copy register”. Tanken med detta är att externa signaler inte ska interagera med funktionaliteten direkt utan ska slussa genom dessa register för att undvika brytning av utförande vid olämplig tidpunkt. Flaggan som återställs används för att signalera om ”command register” och ”copy register” skiljer sig åt. Därefter används värdet i ”copy register” som det kommando som ska utföras och väljer mellan om det är arm- eller hjulrelaterat baserat på den inledande kodningen av kommadon.

Armrelaterade kommandon som skickas från logik/kommunikationsmodulerna består inledningsvis av fyra 1:or och hjulrelaterade kommandon som skickas består inledningsvis av fyra 0:or enligt armkommando-protokoll 3.3.2. Detta ska underlätta för särskiljning mellan de två olika typerna av kommandon. De båda kommandohanterarna är väldigt lika som två switch-satser som matchar kommandot till det som ligger i ”copy register” till ett av sina fall och delegerar ut uppgift att utföras i respektive områdesspecifika fil. Skillnaderna är att i armkommandohanteraren så kontrolleras först ett armlås som endast låses upp om robotplattformen är i standby-läge och vissa av fallen i hjulkommandohanteraren inte går direkt in i en metod i sin områdesspecifika fil. Istället går dessa ”lägen” genom metoder i huvudfilen för att behålla det främsta av den kontrollen som sedan anropar metoder i områdesspecifika filen. Dessa lägen är de som använder sig av flaggan för att bryta sig ut ur och återvända till den for-loop i main funktionen.

5.3.2 Hjulstyrning

Hjulstyrningsområdet är istället den hårdvarunära implementationen av funktionalitet kring hjulstyrning. Det är alltså där pinspecifika instruktioner och liknande implementeras för att underlätta läsbarheten och eventuellt underhåll som kräver att byta pinnar eller mikrokontroller eller liknande. Drivkretsarna

använder sig av pulsbreddsmodulering för att kontrollera hastigheten de driver hjulen. Detta arbete sköter mikrokontrollern genom att sätta inställningar i initialiseringen av drivkretsarna för att aktivera den alternativa inbyggda funktionaliteten till de kopplade pinnarna. Efter detta måste endast kontrollregistrarna som bestämmer arbetscykeln sättas till önskat värde. Detta är i nästan alla fall den definierade konstanten PWM_STANDARD som används för att lätt kunna anpassas om högre eller lägre hastighet skulle önskas. Drivkretsarna ska alltid vara PWM_STANDARD eller 0 förutom när roboten ska följa tejpen. Då kommer PD-regleringen styra hur de två drivkretsarna ska minska eller öka sin arbetscykel för att korrigera efter tejpen. Värdet är dock centrerat på PWM_STANDARD.

Riktningarna på hjulen sätts istället genom att sätta pinnarna kopplade till DIR (se Kopplingsschema) till 1 eller 0. Eftersom drivkretsarna är spegelvända kommer förflyttning från punkt kräva att riktningarna är inverterade av varandra, alltså att en DIR är satt till 0 och den andra till 1. Förflyttning på plats, alltså att rotera på plats, kräver istället att de båda riktningarna är satta till samma värde. För att förenkla vidareutveckling av andra mer komplexa kommandon så finns en metod som sätter passande riktningar, help_dir, som ändrar värdena på pinnarna till passande vid given riktning.

5.3.3 Armstyrning

Armstyrningsfilen är istället fokuserad på att vara ett lättförståeligt och användbart gränssnitt till det strikta format som armservona tillhörande robotplattformen kräver. Tanken är att vid byte av armservon eller implementation av nya armrelaterade kommandon ska läsbarheten och modulariteten av uppdelningen av koden kunna underlätta det arbetet.

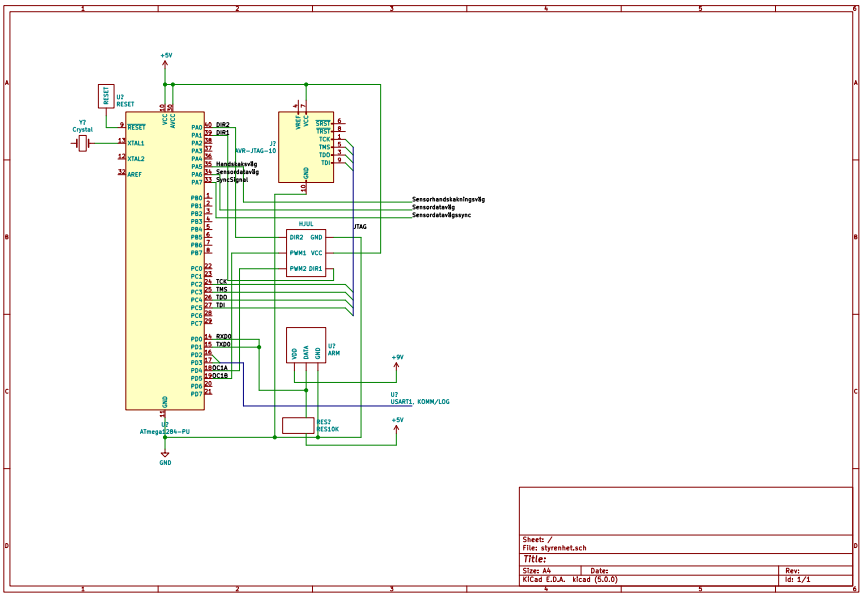
Armservona är AX-12or kommunicerar via en halvduplex UART dataväg. Detta, kombinerat med att dess önskade bits per second ligger på 1M bits/s, gör det lämpligare att kommunicera med armen via en hårdvaru-UART. Detta är anledningen att kommunikationen mellan sensor- och styrmodul inte sker genom en hårdvaru-UART som planerat utan genom en mjukvaruimplementerad lättviktig dataväg, då mikrokontrollern endast har två hårdvaru-UARTs. Då datavägen är enkelriktad men data måste åt båda håll så är det kritiskt att hindra data från att krocka för att armservona ska fungera som önskat. Det kräver även att mikrokontrollerns hårdvaru- UARTs pinnar ska kopplas samman med en resistor för att undvika flytande värden på pinnen enligt specifikation i AX-12s datablad. Detta skulle även kunna implementeras med en intern pull-up resistor inom mikrokontrollern, men under projektets gång så användes extern resistor för att eliminera felkälla.

Armservona har fördefinierat kommunikationsprotokoll som finns beskrivet i deras datablad, och den främsta uppgiften som finns i armstyrningsfilen är att etablera ett mer tillgängligt gränssnitt som sköter konverteringen till önskat format. För att skriva till armservona används globala interna arrays som parametrar och metoden transmit packet. Transmit packet tar vissa värden som parametrar, som ID på det servo man vill styra, vilken instruktion man vill utföra samt hur många parametrar som ska användas i paketet. Genom transmit packet kan man då ha ett flexibelt antal parametrar som används beroende på vad man vill utföra som nås genom den globala arrayn. Det instruktionspaket som skickas till servot kommer i vissa fall generera ett statuspaket som ska re-

turneras till mikrokontrollern. Detta accepteras av receive_packet-metoden för att hindra kollision i datavägen.

5.3.4 Kopplingar på virkort

Virkorten som styrmodulen är implementerad på är rimligtvis så nedskalad och begränsad. De komponenter som finns är ”atmega” kopplad till systemklockan, resistor, resetknapp samt de datavägar till och från modulen. För att underlätta så valdes virsladdarnas att vara färgkodade. De svarta virsladdarna är till jord eller konstanta 0:or, de röda är till ström eller konstanta 1:or, de gröna är för data in mot ”atmega” och de gula för data ut från ”atmega”. Nedan finns även kopplingsschemat som stöd till att förstå kopplingarna på virkortet.



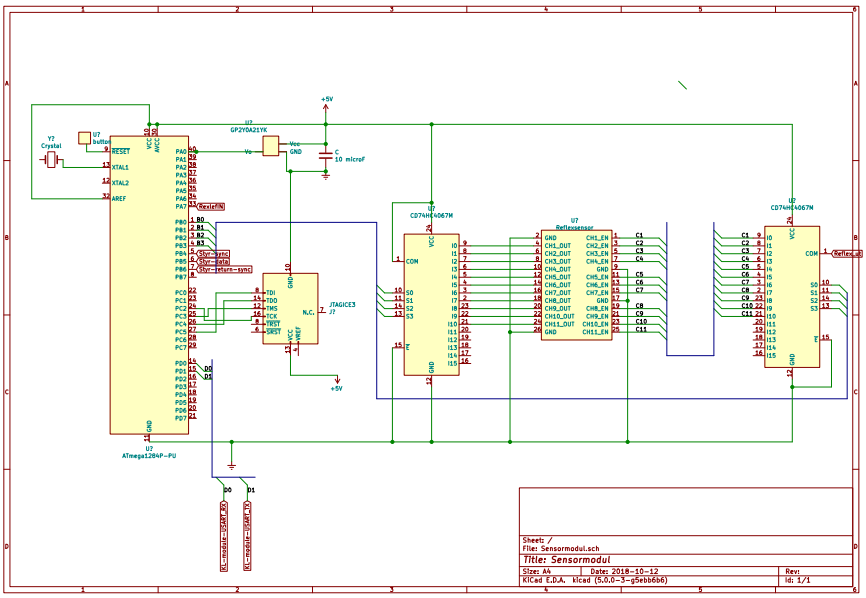
Figur 5.5: Kopplingsschema över styrmodulens på virkortet.

5.4 Sensormodul

Sensormodulens uppgift är att samla in all mätdata till roboten, och vidarebefodra datan till de övriga modulerna på lämpligt format.

Roboten har två sensorer: en IR-sensor som ”tittar” framåt och hittar hinder, samt en linjesensor som berättar hur bra vi håller oss på banan.

IR-sensorn IR-sensorn ger en analog signal motsvarande hur långt bort ett hinder är. För inläsning nyttjas ATmega1284p:ns ADC (Analog Digital Converter). De inlästa värdena konverteras sedan tillbaka till cm.



Figur 5.6: Kopplingsschema över Sensormodulen

Linjesensorn Linjesensorn har ett antal ljus-sensorerer sida vid sida. Läsning sker av dem sekventiellt, och ger oss en lista av digitala värden.

5.4.1 Sändning av data

Till Logik-/Kommunikationsmodulen skickas data via UART-paketen beskrivna i stycke 3.3.5. Den data som skickas är avståndet till hindret framför i cm, samt var vi är på linjen.

Styrmodulen får data via ExDe systemet, och får enbart rå linjedata.

5.4.2 Huvudrutin

Programmets struktur kan ses i figur 5.7.

Programmet börjar med att initialisera all hårdvara, följt av att det väntar på handskakningar från de andra modulerna. Det påbörjar sedan en loop där den:

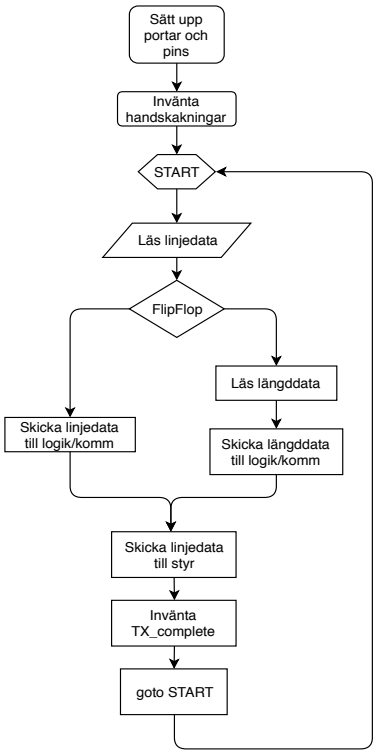
1. Läser linjedata
2. Antingen:
   - Påbörja asynkron överföring av *linjedata* till logik-/kommunikationsenheten **ELLER**
   - Läs längdata med IR-sensorn.
   - Påbörja asynkron överföring av *längddata* till logik-/kommunikationsenheten.
3. Skickar linjedata till styr

4. Inväntar TX_complete

5. Upprepar

Till logik- och kommunikationsenheten skickas data asynkront över UART. Det låter oss samtidigt skicka data till styrenheten över ExDe, vilket är mer CPU-krävande. FlipFlop är en ”flipfloppande” variabel som varannan gång är sann, varannan falsk. Detta för att logik- och kommunikationsenheterna ej behöver data lika ofta som styrenheten.

När allt det är klart ser vi till att alla paket skickats klart, och efter det upprepar hela processen.



Figur 5.7: Flödesschema för sensormodulen

5.4.3 Koppling från Sensormodul

På brädet så från Sensormodulen så finns det 4 portar som i den här uppsättningen används på följande sätt:

- Port 1 alternativ ström in + data pins, ger ej ström när annan ström används
- Port 2 JTAG port
- Port 3 buss till andra brädet
- Port 4 UART och IR-sensor-kopping.

Kapitel 6

Slutsatser

Det finns en del förbättringar som kan göras på produkten. Autonom uppplockning av varan, samt ditskruvandet av en skärm för att visa information på själva roboten är båda saker som, i mån om tid, kan göras. Vidar så kan, i logikenheten, en kö av varor implementeras, som roboten sedan ska plocka upp en och en för att sedan lasta av på avplockningsstationen.

Det grafiska användargränssnittet kan förbättras,så att robotens nuvarande position visas i realtid. En annan förbättring till det grafiska användargränssnittet varans position ska även visas på datorns gränssnitt.

Bilaga A

BNF

BNF står för Backus Naur Form, och är ett formellt språk för att beskriva andra formella språk. Vi har en kort genomgång av det här då vi använder några små förlängningar till det.

⟨**a**⟩ ::= ‘a' innebär att ⟨a⟩ är en symbol som alltid är strängen ”a”.

⟨**b**⟩ ::= ⟨**a**⟩ | ⟨**c**⟩ innebär att ⟨b⟩ är en symbol som antingen är symbolen ⟨a⟩ eller symbolen ⟨c⟩.

⟨**query**⟩ → ⟨**response**⟩ visar en förfrågan mellan enheterna. *OBS* Ej standard BNF.

⟨**s**⟩ ::= ⟨**a**⟩ ⟨**b**⟩ innebär ⟨a⟩ följt av ⟨b⟩ *med* whitespace mellan.

⟨**s**⟩ ::= ⟨**a**⟩⟨**b**⟩ innebär ⟨a⟩ följt av ⟨b⟩ *utan* whitespace mellan.

⟨**l**⟩ ::= { ⟨**a**⟩ } innebär en eller flera ⟨a⟩ objekt, sepparerade med whitespace.

Bilaga B

Kodlistningar

Nedan följer ett antal kodlistningar vi tyckte var av extra stor vikt. För komplett källkod, kontakta gruppen eller se `https://gitlab.ida.liu.se/kmm-lagerrobot`.

B.1 UART packets

```
#ifndef PACKET_H
#define PACKET_H

#include <stdint.h>
#include <stddef.h>

/*
https://stackoverflow.com/questions/19995440/c-cast-byte-array-to-struct
http://digitalvampire.org/blog/index.php/2006/07/31/why-you-shouldnt-use-__att
 */

/*
 * Each package should end with a set value. Mostly for error
 * checking. The actual value of this field should never be checked
 * numerically, and could be changed at any moment.
 */
typedef uint8_t end_type;
#define PKG_END 0xFF

/*
 * List of possible commands.
 * Extend this list to add more command.
 * No more than 16 different command types are supported with the
 * current package format
 */
typedef enum command_type {
        handshake,
        linedata,
        distance,
```

```
        raw_linedata,
        steercomm,
        steerdata
} command_type;

/*
 * High level information about the robots position on the line.
 * Each state referents a possible state, and should each be self
 * explanatory.
 */
enum linedata {
        on_line, not_on_line,
        left_of_line, right_of_line,
        station_to_left, station_to_right,
        four_way_intersection,
        drop_station,
};

/*
 * The header of each package.
 * - `id` should be a incrementing number, where each uart connection
 *   has its own counter. Dropped packages can be detected if
 *   `pkg.id != last_pkg.id + 1` (mod 16)
 * - `type` is the expected contents of the package. See command_type
 *   for possible values.
 * - `length` in bytes of package. Max 0xFF
 */
typedef struct header {
        uint8_t id      : 4;
        command_type type: 4;
        uint8_t length  : 8;
} __attribute__((packed)) header;

/*
 * Callsigns for different units. Used for handshake packages.
 */
typedef enum callsign {
        cs_err, cs_raspi, cs_sensor, cs_styr
} callsign;

/*
 * SYN/ACK packages. Used for handshake.
 * (Note that these SYN and ACK messages doesn't use the appropriate
 * ASCII values, this is to save space).
 */
typedef enum synack { SYN, ACK } synack;

/*
 * Payload of a handshake package.
 */
```

```
struct handshake {
        synack synack : 2;
        callsign sign : 6;
} __attribute__((packed));

/*
 * Union of different payloads. Appropriate payload should be
 * documented along with each command type.
 */
/*
 * The sizeof a union is the sizeof its largest element. In this case
 * that would be sizeof(void*) == 8, but could be something else if
 * the structure changes, or the code is compiled towards a system
 * with less than 64 bits.
 */
union payload {
        struct handshake handshake;
        uint8_t distance;
        char steercomm[8];
        char steerdata[8];
        enum   linedata  linedata;
};

/*
 * Package struct. Contains a header, a data and a pkg_end field.
 * The user should only ever manually look at the first 2 fields.
 *
 * TODO check how this struct packs with the union type of unclear
 * size in the middle.
 *
 * the 'end' field should never be used, bu ensures that enough data
 * is allocated for the object.
 */
typedef struct packet {
        struct header header;
        union payload data;
        end_type end; /* Never use this field for anything */
} __attribute__((packed)) packet;

// ---------------- Functions ----------------

/*
 * Sets some default values for packets. Should be called on all newly
 * allocated packages that are to be sent.
 */
void setup_packet (packet* p);


/*
 * Returns the size of a given package in bytes.
```

```
 * Includes header, payload and stop frame.
 */
size_t packet_size(packet* p);

#ifdef __unix__
void print_packet(packet* p);
#endif

#endif /* PACKET_H */
```

B.2 UART Connections

```
#ifndef UART_COMMON_H
#define UART_COMMON_H

#include "packet.h"
#include "uart.h"
#include "util.h"
#ifdef __unix__
#include <pthread.h>
#endif

callsign callsignbuf;


#ifdef __unix__
typedef pthread_t pack_ret;
#define receive_packet receive_packet_unix
#else
typedef int pack_ret;
#endif

pack_ret send_packet(packet* p, uart_connection* c);
callsign connection_handshake (uart_connection* conn, callsign sign);

typedef union { char* str; int num; } conn_data;
int create_connection (uart_connection* c, conn_data data);
#endif /* UART_COMMON_H */
```

B.3 Styrenhet

```
/*
 * styrenhet.h
 * Created: 11/25/2018 2:16:09 PM
 */


/* This is the header file for the central code in styrenhet.c. The intention o
is to hold the control flow and delegate it to the arm servos and the wheel eng
```

It connects the external input to the module to the steering functionality thro

```
#ifndef STYRENHET_H
#define STYRENHET_H

#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <math.h>
#include <avr/interrupt.h>
#include "avr.h"
#include "packet.h"
#include "uart_common.h"
#include "wheels.h"
#include "arm_servo.h"

/* Wheel engine specific commands. */
#define STANDBY                 0b00000001
#define DRIVE_MANUAL_FORWARD    0b00000010
#define ACTIVE_FORWARD          0b00000011
#define DRIVE_MANUAL_BACKWARD   0b00000100
#define ACTIVE_BACKWARD         0b00000101
#define DRIVE_MANUAL_LEFT       0b00000110
#define LEFT_PERP               0b00000111
#define LEFT_DIAG               0b00001000
#define DRIVE_MANUAL_RIGHT      0b00001001
#define RIGHT_PERP              0b00001010
#define RIGHT_DIAG              0b00001011

/* Arm specific commands. */
#define CLOCKWISE               0b11110001
#define COUNTERCLOCKWISE        0b11110010
#define SERVO_IN                0b11110011
#define SERVO_OUT               0b11110100
#define SERVO_UP                0b11110101
#define SERVO_DOWN              0b11110110
#define SERVO_LEFT              0b11110111
#define SERVO_RIGHT             0b11111000
#define SERVO_SQUEEZE           0b11111001
#define SERVO_RELEASE           0b11111010
#define SERVO_DROP              0b11111011

/* Directions of the wheels. */
#define FORWARD     0
#define BACKWARD    1
#define LEFT        2
#define RIGHT       3

/* Constants in the PD-regulation in active_mode. */
#define TAPE_TARGET 0
```

```
#define CONST_PROP  2
#define CONST_DER   2

/* Flag for change in the register of command. */
int flag;

/* Lock for arm use, only unlocked in standby_mode. */
int arm_lock = 1;

/* Get the value of bit in variable */
#define get_bit(var, bit) ((var & (1 << bit)) != 0)

/* Pointer to position in packet */
volatile uint8_t* rx_packet_bytes = &rx_pack;
volatile uint8_t* tx_packet_bytes = &tx_pack;

/* Number of frames received/transmitted in current package */
volatile int8_t rx_count, tx_count, tapesensor_count;

/* UART connection to RaspberryPi */
uart_connection pi_conn;

/* External UART baud rate */
#define external_baud_rate 103

/* Shift bit read into register */
#define SHIFT_IN(sreg, bit) do { (sreg) <<= 1; (sreg) |= (1 & bit);} while(0)

uint16_t sreg;

#endif /* STYRENHET_H */
```

B.4 Armstyrning

```
/* The arm servo header file details all the included
files and definitions used in the arm servo c file. It
also contains all the method declarations that other
files can call upon. */

#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdio.h>
#include "uart.h"
#define cbi(REG8,BITNUM) REG8 &= ~(_BV(BITNUM))
#define sbi(REG8,BITNUM) REG8 |= _BV(BITNUM)
typedef unsigned char byte;
typedef unsigned int word;
```

```
/* Abstraction of the arm servos control table. */
#define P_MODEL_NUMBER_L 0
#define P_MODEL_NUMBER_H 1
#define P_VERSION 2
#define P_ID 3
#define P_BAUD_RATE 4
#define P_RETURN_DELAY_TIME 5
#define P_CW_ANGLE_LIMIT_L 6
#define P_CW_ANGLE_LIMIT_H 7
#define P_CCW_ANGLE_LIMIT_L 8
#define P_CCW_ANGLE_LIMIT_H 9
#define P_SYSTEM_DATA2 10
#define P_LIMIT_TEMPERATURE 11
#define P_DOWN_LIMIT_VOLTAGE 12
#define P_UP_LIMIT_VOLTAGE 13
#define P_MAX_TORQUE_L 14
#define P_MAX_TORQUE_H 15
#define P_RETURN_LEVEL 16
#define P_ALARM_LED 17
#define P_ALARM_SHUTDOWN 18
#define P_OPERATING_MODE 19
#define P_DOWN_CALIBRATION_L 20
#define P_DOWN_CALIBRATION_H 21
#define P_UP_CALIBRATION_L 22
#define P_UP_CALIBRATION_H 23
#define P_TORQUE_ENABLE (24)
#define P_LED (25)
#define P_CW_COMPLIANCE_MARGIN (26)
#define P_CCW_COMPLIANCE_MARGIN (27)
#define P_CW_COMPLIANCE_SLOPE (28)
#define P_CCW_COMPLIANCE_SLOPE (29)
#define P_GOAL_POSITION_L (30)
#define P_GOAL_POSITION_H (31)
#define P_GOAL_SPEED_L (32)
#define P_GOAL_SPEED_H (33)
#define P_TORQUE_LIMIT_L (34)
#define P_TORQUE_LIMIT_H (35)
#define P_PRESENT_POSITION_L (36)
#define P_PRESENT_POSITION_H (37)
#define P_PRESENT_SPEED_L (38)
#define P_PRESENT_SPEED_H (39)
#define P_PRESENT_LOAD_L (40)
#define P_PRESENT_LOAD_H (41)
#define P_PRESENT_VOLTAGE (42)
#define P_PRESENT_TEMPERATURE (43)
#define P_REGISTERED_INSTRUCTION (44)
#define P_PAUSE_TIME (45)
#define P_MOVING (46)
#define P_LOCK (47)
```

```
#define P_PUNCH_L (48)
#define P_PUNCH_H (49)

/* Instruction abstraction. */
#define INST_PING 0x01
#define INST_READ 0x02
#define INST_WRITE 0x03
#define INST_REG_WRITE 0x04
#define INST_ACTION 0x05
#define INST_RESET 0x06
#define INST_DIGITAL_RESET 0x07
#define INST_SYSTEM_READ 0x0C
#define INST_SYSTEM_WRITE 0x0D
#define INST_SYNC_WRITE 0x83
#define INST_SYNC_REG_WRITE 0x84

/* Abstraction of arm controls. */
#define CLEAR_BUFFER receive_buffer_read_pointer = receive_buffer_write_pointer
#define DEFAULT_RETURN_PACKET_SIZE 6
#define BROADCASTING_ID 0xfe
#define INCREMENTAL_ANGLE 3
#define STANDARD_SPEED 0

typedef unsigned char byte;
typedef unsigned int word;

/* Methods declarations available for use in other files. */
void initialize_servos();
void dropping_mode();
void set_neutral_mode();
void left_right_servo(int dir);
void up_down_servo(int dir);
void out_in_servo(int dir);
void cw_ccw_servo(int dir);
void squeeze_release_servo(int dir);

/* Declaration of global arrays. */
volatile byte receive_interrupt_buffer[256];
byte parameter[128];
byte receive_buffer_read_pointer;
byte receive_buffer[128];
byte transmission_buffer[128];
volatile byte receive_buffer_write_pointer;

/* Baud rate for arm servo and AVR communication. */
#define arm_baud_rate 1

void Arm_UART_Init( unsigned int ubrr );
void Arm_UART_Transmit( unsigned char data );
unsigned char Arm_UART_Receive( void );
```

B.5 Hjulstyrning

```
/*
 * wheels.h
 * Created: 11/24/2018 5:07:32 PM
 */

/* This is the header file for the wheel functions. The intention of this file
is to gather all functionality pertinent to the wheels in one place and abstrac
wheel functionality to improve readability and maintainability.*/


#ifndef WHEELS_H_
#define WHEELS_H_

/* Abstraction of possible directions for the wheels. */
#define FRONT 0
#define BACK 1
#define LEFT 2
#define RIGHT 3

/* The standard speed of the wheels. */
#define PWM_STANDARD 100

/* The time necessary for a 360 degree turn at standard speed. */
#define STANDARD_THREESIXTY 12000

/* Clear and set bit instructions to change PORT indices. */
#define cbi(REG8,BITNUM) REG8 &= ~(_BV(BITNUM))
#define sbi(REG8,BITNUM) REG8 |= _BV(BITNUM)

/* The methods used for wheel steering. */
void initialize_engines();
void standby();
void active(int direction, int adjustment);
void forward();
void backward();
void left_manual();
void left_perp();
void left_diag();
void right_manual();
void right_perp();
void right_diag();

#endif
```