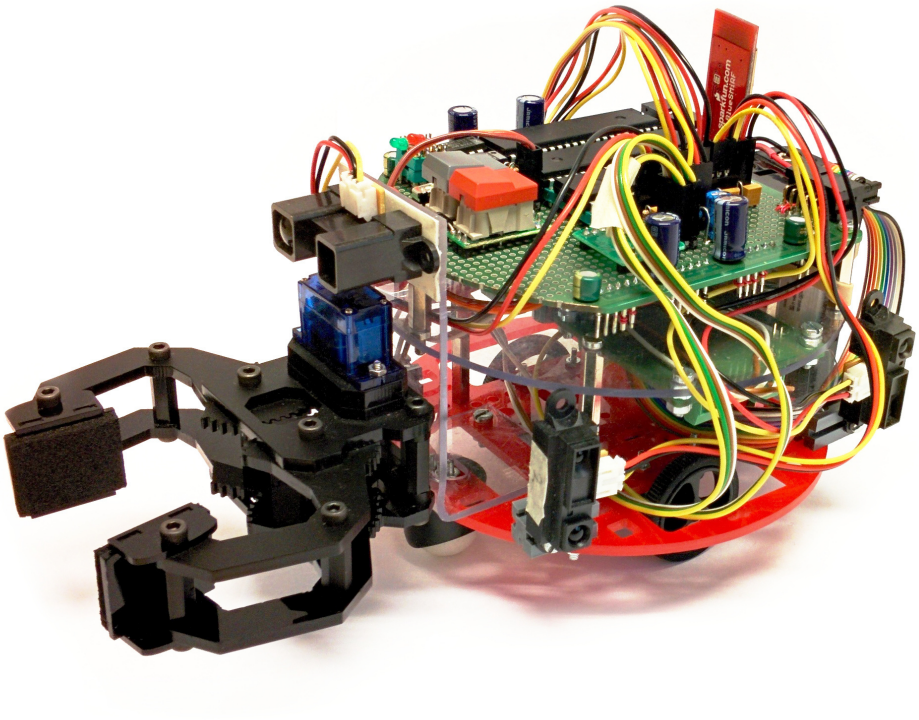
Teknisk dokumentation

Roboten Ronny

Jesper Otterholm

Version 1.1



Status

Granskad	Grupp 1	2015-05-27
Godkänd		

PROJEKTIDENTITET

2015/VT, Grupp 1
Tekniska högskolan vid Linköpings universitet, ISY

Gruppdeltagare

Namn	**Ansvar**	**Telefon**	**LiU-id**
Jesper Otterholm	Projektledare (PL)	073 800 03 17	jesot351
Lage Ragnarsson	Dokumentansvarig (DOK)	073 972 36 35	lagra033
Erik Sköld		073 905 43 43	erisk214
Emma Söderström		073 396 21 72	emmso236
Matilda Östlund Visén		073 817 15 90	matos000
Filip Östman		072 203 33 07	filos433

E-postlista för hela gruppen: jesot351@student.liu.se

Kund: Institutionen för systemteknik, Linköpings universitet
Kontaktperson hos kund: Kent Palmkvist, 3B:502, 013-28 13 47, kentp@isy.liu.se
Kursansvarig: Thomas Svensson, 3B:528, 013-28 13 68, thomass@isy.liu.se
Handledare: Olov Andersson, 3B:504, 013-28 26 58, olov@isy.liu.se

Dokumenthistorik

Version	Datum	Utförda förändringar	Utförda av	Granskad
1.0	2015-05-27	Första versionen	Grupp 1	Grupp 1
1.1	2015-06-03	Redaktionella ändringar, JTAG	JO	

Sammanfattning

Roboten är konstruerad med ett nyutvecklat chassi och tre moduler: en styrmodul, en sensormodul och en kommunikationsmodul. Modulerna är sammankopplade med en I^2C-buss för intern kommunikation. Till roboten medföljer programvara till PC. Denna programvara benäms som PC-modulen vilket gör att produkten består av totalt fyra moduler.

Styrmodulen består huvudsakligen av en mikrokontroller kopplad till motorer och gripklo. Den ansvarar för att styra dessa på ett lämpligt sätt för att utföra uppdraget. Styrmodulen ansvarar även för kartering av och navigering.

Sensormodulen består av en mikrokontroller, sensorer och tillhörande elektronik och har till uppgift att läsa av och tolka sensorvärden från robotens IR-sensorer, pulsgivare och reflexsensor. När sensorvärdena har hanterats skickas lämpliga meddelanden till de andra modulerna via bussen.

Kommunikationsmodulen är länken mellan roboten och PC-modulen. Den skickar vidare kommandon mottagna från PC-modulen till styr- och sensormodulen samt sensorvärden och kartinformation från styr- och sensormodulerna till PC-modulen.

PC-modulen är programvara för att visa sensorvärden och karta under autonom körning och skicka styrkommandon under manuell körning. Kommunikationen mellan PC-modulen och roboten sker via Bluetooth.

För framtida utveckling kan det vara intressant med ny funktionalitet så som att möjliggöra autonom upplockning av förnödenheter och generellt arbete med bättre tillförlitlighet.

Innehåll

1 Inledning

Detta dokument beskriver konstruktionen av roboten framtagen i kursen *TSEA56 - Kandidatprojekt i elektronik* vid Linköpings universitet vårterminen 2015. Roboten består av tre moduler: en styrmodul, en sensormodul och en kommunikationsmodul. Ett program till PC tillkommer för att visa telemetri under körning samt för att styra roboten i manuellt läge. Hårdvaran beskrivs detaljerat, både gällande den mekaniska och den elektriska konstruktionen. Mjukvarans övergripande struktur presenteras tillsammans med utvalda detaljer i syfte att vägleda vid vidarekonstruktion och felsökning.

Dokumentet inleds med en översikt över den färdiga produkten och vad den används till. Därefter följer ett avsnitt om den teori som ligger till grund för de tekniska lösningar som tagits fram i konstruktionen av roboten, vilka sedan beskrivs i detalj i de efterföljande avsnitten. Slutligen presenteras gruppens idéer kring fortsatt arbete och vidareutveckling av produkten.

1.1 Definitioner

Nödställda Tänkt nödställda i en labyrint som behöver undsättning. Kommer i verkligheten endast vara en speciellt markerad plats i labyrinten. Ibland refererat till som *målet*, *målruta* eller liknande.

Labyrint Den omgivning roboten ska utforska. Ibland kallad *omgivning*, *bana*, *grottsystem* eller liknande.

Tillstånd Robotens uppdrag är indelat i ett antal mindre uppgifter kallade tillstånd. Tillstånden definerar robotens beteende i en viss situation.

Rutt En rutt anger hur roboten ska navigera mellan två rutor i labyrinten.

Flood fill En algoritm för kortaste vägenproblem.

I^{2}C En databuss som används för att kommunicera mellan modulerna.

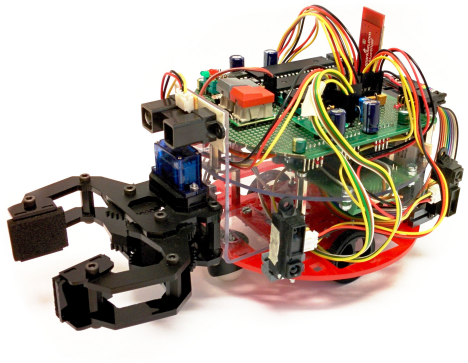
Master Mikrokontrollern som för tillfället bestämmer över bussens trafik.

Slave Den eller de mikrokontrollers som blir adresserade av *mastern*.

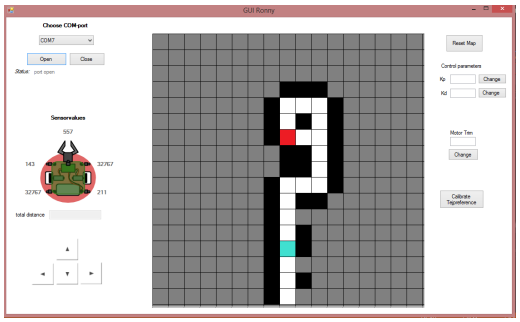
2 Produkten

Produkten utgörs av en robot för både autonom och manuell körning samt av ett program till PC. Roboten är byggd på ett cirkulärt chassi och drivs framåt av två hjul medan två stödkulor säkerställer robotens balans. Fem IR-sensorer används för avståndsmätningar till omgivningen. Optiska pulsgivare monterade på motoraxlarna används för att beräkna avlagd sträcka och rotation. Framtill på roboten finns en gripklo monterad för transport av förnödenheter. En reflexsensor sitter monterad på

robotens undersida för att detektera målmarkeringen på marken. Den medföljande programvaran används för att visa kart- och sensorinformation samt för manövrering av roboten i manuellt läge.



(a) Den konstruerade roboten.



(b) En skärmdump av den medföljande programvaran.

Figur 1: Produkten med dess ingående komponenter.

Roboten kan autonomt köra och navigera i en labyrint enligt banspecifikationen [1] för att lokalisera de nödställda och leverera förnödenheter till denna ruta. Roboten reglerar mot labyrintens väggar och finner den kortaste vägen mellan start och mål, markerade med svarta tejpremsor. En MER-förpackning symboliserar förnödenheter och leveras med hjälp av gripklon till de nödställda.

Under körning visar programvaran robotens position och det utforskade området i labyrinten tillsammans med sensorvärden. I manuellt läge används programvaran för att skicka styrkommandon. Programvaran kan även justera robotens reglering genom att skicka nya reglerparametrar.

3 Teori

I detta avsnitt beskrivs några av de metoder och algoritmer som roboten använder för att utföra sitt uppdrag. Syftet är att ge bakgrundsinformation för att möjliggöra läsning om den specifika implementationen som följer under respektive moduls avsnitt.

3.1 Reglering

För att hålla roboten i mitten av korridoren under körning används ett återkopplat reglersystem. Vi har valt att använda skillnaden i avstånd mellan vänster och höger vägg som reglerfel, $e[n]$, till en PD-regulator. Detta reglerfel kan ses som robotens avstånd till korridorens mitt då roboten är ungefär vinkelrät mot väggarna. Det innebär att en styrsignal, $u[n]$, kan räknas ut enligt ekvation (1).

$$
\begin{aligned}
e[n] &= x_v[n] - x_h[n] \\
u[n] &= K_P e[n] + K_D \frac{e[n] - e[n-1]}{T_s},
\end{aligned}
\tag{1}
$$

Där T_s är samplingsperiodtiden för IR-sensorerna och K_P och K_D är reglerparametrar som tas fram experimentellt för att roboten ska få ett önskat beteende. x_v och x_h är avstånden till vänster respektive höger vägg.

3.2 Flood fill

Flood fill är en algoritm för kortaste vägenberäkningar. Den går ut på att man tilldelar varje koordinat ett avståndsvärde som motsvarar antalet steg det tar att förflytta sig dit. Algoritmen låter en ”vågfront” börja i önskad startruta växa utåt i labyrinten. Eftersom vågfronten växer räknas en avståndsräknare upp och när vågfronten passerar en ruta tilldelas denna avståndsvärdet. Vågfrontens utbredning begränsas av väggar och ibland även av outforskade rutor. När labyrinten är tillräckligt fylld kan kortaste vägen beräknas genom att från målrutan hela tiden gå till en ruta som har ett lägre avståndsvärde. På så vis kommer man alltid ett steg närmare starten och går garanterat den kortaste vägen. Denna rutt måste sedan reverseras för att få rutten från start till mål.

Vi har implementerat tre olika varianter av denna algoritm för olika delar av uppdraget. Dessa varianter beskrivs i avsnitt 5.1.7. Alla algoritmer har den gemensamma strukturen:

1. Sätt alla rutor till ett initialvärde.
2. Sätt avståndsräknaren till noll.
3. Lägg till koordinaten rutten ska börja från i vågfronten.
4. För varje element i vågfronten:
   - 4.1. Ge rutan värdet på avståndsräknaren.
   - 4.2. Om stoppvillkoret[1] uppfyllts: beräkna en rutt och avsluta.
   - 4.3. Lägg till varje tillåten[2] granne i den nya vågfronten.
5. Öka avståndsräknaren med ett.
6. Låt den nya vågfronten bli den aktuella vågfronten och gå tillbaka till steg 4.

[1] Olika varianter av algoritmen har olika stoppvillkor. Dessa beskrivs i avsnitt 5.1.7 och handlar alltid om att man har fyllt så pass mycket att man hittat en ruta man vill färdas till.

[2] Vilka rutor som är tillåtna att lägga till i vågfronten varierar med vilken varianter av algoritmen som används, se avsnitt 5.1.7.

3.3 Tillståndsmaskiner

Följande är ett utdrag ur *Förstudie, Reglering och kartering för en autonom robot* [2]:

> En tillståndsmaskin är en abstrakt modell för att beskriva ett beteende utifrån olika tillstånd och villkor för tillståndsövergångar. Modellen lämpar sig väl för att beskriva ett beslutsfattande system som både beror på aktuell information och på historiska beslut, såsom en autonom robot. En fördel med att konstruera beslutsfattandet som en tillståndsmaskin är att det ger en överskådlig kod där varje tillstånd enbart behöver beakta den delmängd beslutsdata som är relevant för just det tillståndet. En annan fördel är att det blir enkelt med testning och verifiering då man kan testa att varje enskilt tillstånd fungerar korrekt samt att tillståndsövergångarna sker på ett korrekt sätt.
>
> En avvägning som behöver göras vid realiseringen av en tillståndsmaskin är vad som ska utgöra ett eget tillstånd. Med för många tillstånd blir systemet svåröverskådligt och med för få och komplexa tillstånd går tillståndsmodellens fördelar förlorade.

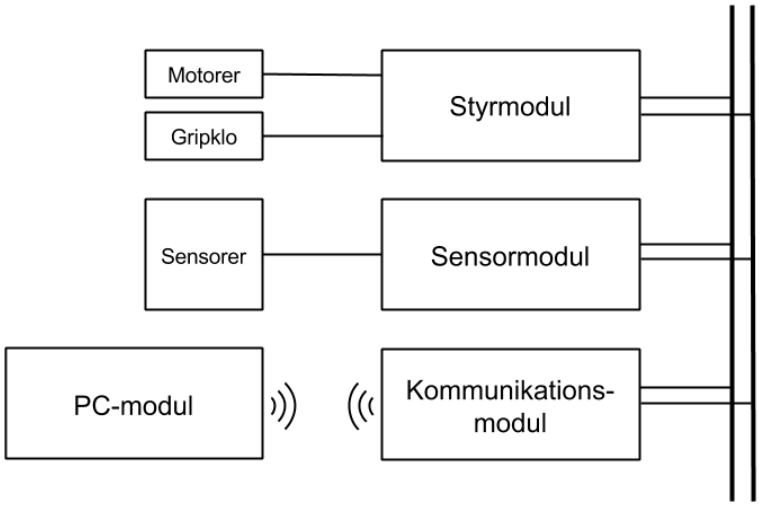
3.4 I^2C

I^2C är en tvåtrådsbuss som moduler kan anslutas med och skicka data. En av ledarna innehåller klockpulser, *SCL*, och den andra innehåller data, *SDA*. Mikrokontrollerna som används har gott stöd för I^2C. Vid multimaster-implementation kan samtliga enheter på bussen ta rollen som *master* och därmed ta kontroll över bussen. Vid överföring av data skickar sändaren ett startkommando på bussen genom att *SDA* går låg när *SCL* är hög. Därefter skickar *mastern* adressen till önskad mottagare på bussen. Den adresserade enheten blir då *slave* och svarar genom att dra klockan låg, *ACK*, om den är redo att ta emot data alternativt behålla klockan hög, *NACK*, om den inte kan ta emot data för tillfället. Registrerar *mastern* ett *ACK* följs detta upp av en byte med data som mottagaren svarar på med antingen *ACK* eller *NACK* på samma sätt som vid adresseringen. Om fler bytes ska sändas fortsätter det sista steget tills det att samtliga bytes är skickade och besvarade med *ACK*. Då sänder *mastern* ett stoppkommando på bussen genom att *SDA* går hög när *SCL* är hög. Ett flödesschema beskrivs nedan:

1. Sänd startkommando om bussen är ledig.
2. Sänd adressen till önskad modul/moduler. Om *ACK* mottas fortsätt till 3. Om *NACK* gör om steg 1.
3. Sänd en byte data. Om *ACK* mottas och samtliga bytes är skickade fortsätt till 4, finns det fler bytes att skicka gör om steg 3 med nästa byte. Om *NACK* mottas kan sändningen starta om från steg 1.
4. Sänd stopkommando.

4 Systemet

Systemet består av fyra moduler: styrmodulen, sensormodulen, kommunikationsmodulen och PC-modulen. En översikt över modulerna och deras inbördes relationer ses i figur 2. Kommunikation mellan modulerna sker via en I^2C-buss och data mellan PC-modulen och kommunikationsmodulen skickas via Bluetooth. För detaljerad information om de olika modulerna se avsnitt 5.



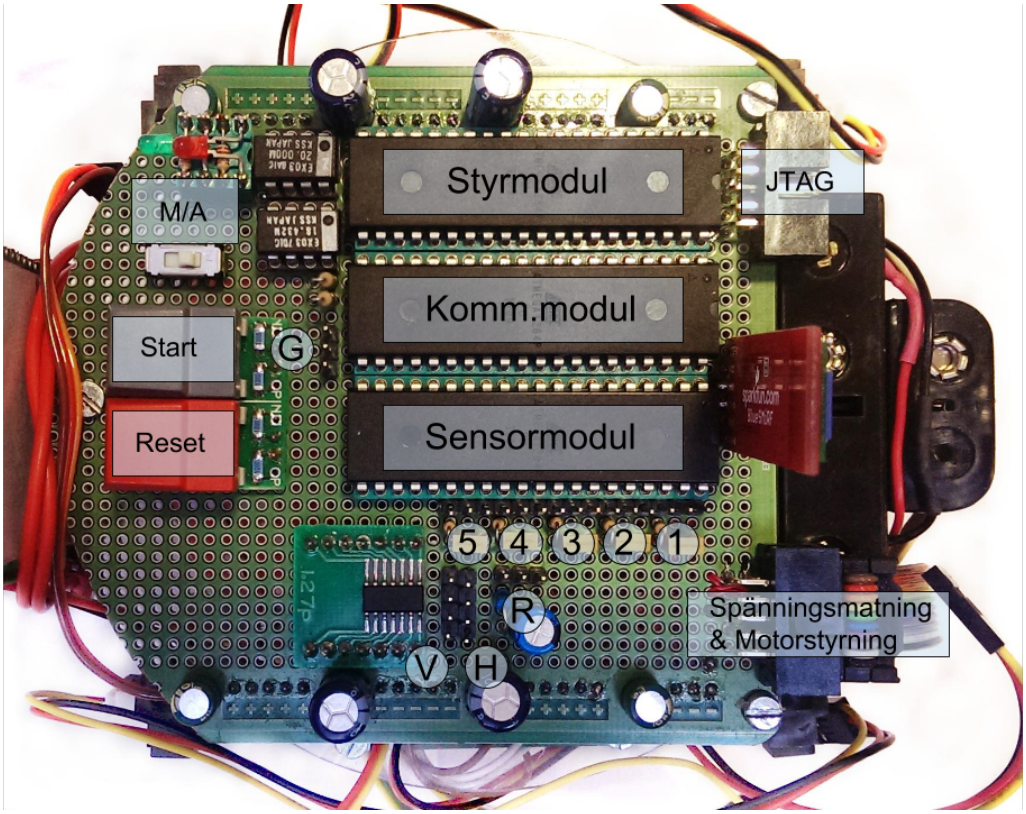
Figur 2: En översikt av hur de olika modulerna samverkar med varandra och övriga komponenter.

4.1 Mekanisk konstruktion

Alla moduler, med undantag för PC-modulen, är monterade på ett och samma kretskort för att minimera storlek och vikt. Roboten är designad för att vara så liten och smidig som möjligt och är därför konstruerad med detta detta i åtanke. Konstruktionen vilar på en rund bottenplatta som är 12, 7 cm i diameter. På bottenplattans undersida är två stycken DC-motorer med tillhörande 50:1 växellådor monterade. På dem sitter 32 mm i diameter stora hjul. Motorerna roterar 625 varv per minut vid 6 V vilket ger en teoretisk maxfart på ungefär 1 m/s. Motorernas motoraxel sticker ut 4,5 mm baktill och på dessa har optiska pulsgivare bestående av ett tretandat plasthjul och två IR-sensorer monterats. Två stödkulor är monterade på undersidan av bottenplattan för att förhindra att roboten tippar. Roboten stöder dock endast på en stödkula i taget.

För att montera IR-sensorer och gripklo har extra fästen konstruerats i plast. Gripklon har också modifierats för att spara plats och vikt genom att ta bort ett servo för lyftande rörelser. Ett batteripack för sex stycken laddningsbara AA-batterier har byggts och är fastspänt med gummiband på robotens baksida för att uppnå jämnare viktfördelning och ge bättre servicemöjligheter. Framåtsensorn är monterad högt upp så att den inte ska blockeras av de upplockade förnödenheterna.

En överblick över layouten av de elektriska komponenterna och anslutningarna ges i figur 3.



Figur 3: Robotens elektriska komponenter och anslutningar. (1)-(5) anger kontakter till IR-sensorerna, se avsnitt 5.2.2, vardera med tre anslutningar: signal, spänning och jord (från vänster till höger i bilden). (R) anger kontakt för reflexsensorn, se avsnitt 5.2.4, med tre anslutningar: signal, spänning, jord (från vänster till höger i bilden). (G) anger kontakt för gripklon, med tre anslutningar: signal, spänning, jord (uppifrån och ner i bilden). (V) och (H) anger kontakter för vänster respektive höger pulsgivare, se avsnitt 5.2.3, vardera med fyra anslutningar: spänning, signal-a, signal-b och jord (nerifrån och upp i bilden). Skjutomkopplaren markerad M/A *ställer roboten i manuellt (vänster) respektive autonomt (höger) läge.*

4.2 JTAG-kedja

Robotens moduler är sammankopplade i en JTAG-kedja med ordningen (i) styrmodul, (ii) kommunikationsmodul och (iii) sensormodul. Placering av systemets JTAG-kontakt ses i figur 3.

4.3 Intern kommunikation

Modulerna är sammankopplade via en I^2C-buss med multimaster-funktion. Därmed har samtliga moduler lika stort inflytande över bussen. Om bussen är ledig och

någon av modulerna vill skicka data så tar den kontroll över bussen och bli *master*. Busskommunikationen blir därmed effektiv eftersom meddelanden kan skickas direkt, utan att någon modul först frågar efter datan. Meddelanden hanteras sedan med avbrott både under sändning och mottagning. Detta medför att modulerna kan hantera andra uppgifter under tiden en sändning pågår. Bussens hastighet är cirka 400 kb/s vilket är tillräckligt snabbt för den mängden data som skickas under ett uppdrag.

4.3.1 Bussgränssnitt

De tre modulerna har tilldelats en unik intern I^2C-adress som används för att adressera mottagaren för meddelandet som ska skickas. Sensormodulen vill skicka meddelanden som ska till både styr- och kommunikationsmodulen. Istället för att skicka meddelandet separat till dessa moduler finns även en adress, *General call*, som adresserar samtliga moduler som är konfigurerade för att motta denna. De olika adresserna kan ses i tabell 1. Dessa adresser är endast 7 bitar då den efterföljs av en bit som beskriver om det ska ske en skrivning (0) eller läsning (1). Vi utnyttjar dock bara skrivning eftersom bussen är implementerad som multimaster.

Till	Adress
Samtliga moduler (*General call*)	0000 000
Styrmodul	0001 001
Sensormodul	0001 010
Kommunikationsmodul	0001 011

Tabell 1: Modulernas SLA-adresser.

Varje meddelande består av en header som är en byte lång följt av ett antal bytes med data. Det är känt hur många byte data som följer varje header. Headern är på formen `XXYYZZZZ`, där `XX` står för sändarens förkortning, `YY` är mottagarens förkortning och `ZZZZ` beskriver vad meddelandet innehåller. Modulernas adresser visas i tabell 2.

Modul	Förkortning
Styrmodulen	00
Sensormodulen	01
Kommunikationsmodulen	10
PC-modulen	11

Tabell 2: Modulernas förkortningar i headern.

I tabell 3 listas samtliga headers som används samt längd på meddelandena.

Header	Beskrivning	Frekvens	Antal bytes inklusive header
0011 0000	Styrmodulen skickar absolut position i form av x- och y-koordinater till PC-modulen.	< 5 Hz	3
0011 0001	Styrmodulen meddelar PC-modulen att roboten är i autonomt läge.	-	1
0011 0010	Styrmodulen meddelar PC-modulen att roboten är i manuellt läge.	-	1
0011 0011	Styrmodulen meddelar PC-modulen att körbar ruta är utforskad samt koordinater.	< 5 Hz	3
0011 0100	Styrmodulen meddelar PC-modulen att vägg är funnen samt koordinater för denna.	< 5 Hz	3
0011 0101	Styrmodulen meddelar PC-modulen att nödställd är funnen samt dess koordinater.	-	3
0100 0000	Sensormodulen skickar data från de fem avståndssensorerna till styrmodulen samt kommunikationsmodulen. 2 bytes per sensor, med avstånd i mm, i ordningen: vänster bak, vänster fram, höger fram, höger bak samt framåt.	25 Hz	11
0100 0001	Sensormodulen skickar förflyttad sträcka och relativ vinkel från senaste sändningen.	< 1 kHz	3
0100 0010	Sensormodulen meddelar övriga moduler att tejpmarkering är funnen.	-	1
0111 0000	Sensormodulen meddelar PC-modulen vad det nya referensvärdet för reflexsensorn är.	-	2
1011 0000	Kommunikationsmodulen skickar sensorvärden till PC-modulen.	5 Hz	11
1100 0001	PC-modulen meddelar styrmodulen att köra rakt framåt.	20 Hz	1
1100 0010	PC-modulen meddelar styrmodulen att rotera höger.	20 Hz	1
1100 0011	PC-modulen meddelar styrmodulen att rotera vänster.	20 Hz	1
1100 0100	PC-modulen meddelar styrmodulen att köra bakåt.	20 Hz	1
1100 0101	PC-modulen meddelar styrmodulen att köra framåt höger.	20 Hz	1
1100 0110	PC-modulen meddelar styrmodulen att köra framåt vänster.	20 Hz	1

1101 0000	PC-modulen meddelar sensormodulen att kalibrera tejpsensorn.	-	1
1100 0111	PC-modulen meddelar styrmodulen att ändra K_P-värdet.	-	2
1100 1000	PC-modulen meddelar styrmodulen att ändra K_D-värdet.	-	2
1100 1001	PC-modulen meddelar styrmodulen att ändra värdet för motortrim.	-	2

Tabell 3: Beskrivning av busstrafiken.

När ett meddelande ska skickas anger sändaren vem som är mottagaren, meddelandet samt längden av meddelandet. Via bussen fylls en buffer på med data hos mottagarmodulen. Denna får sedan baserat på den första byten i buffern, alltså headern, identifiera meddelandetypen och sedan läsa av önskat antal efterföljande bytes beroende på meddelandets storlek. Funktionerna nedan tillhör implementeringen av I^2C och används av samtliga på roboten tillsammans med ett antal mindre funktioner:

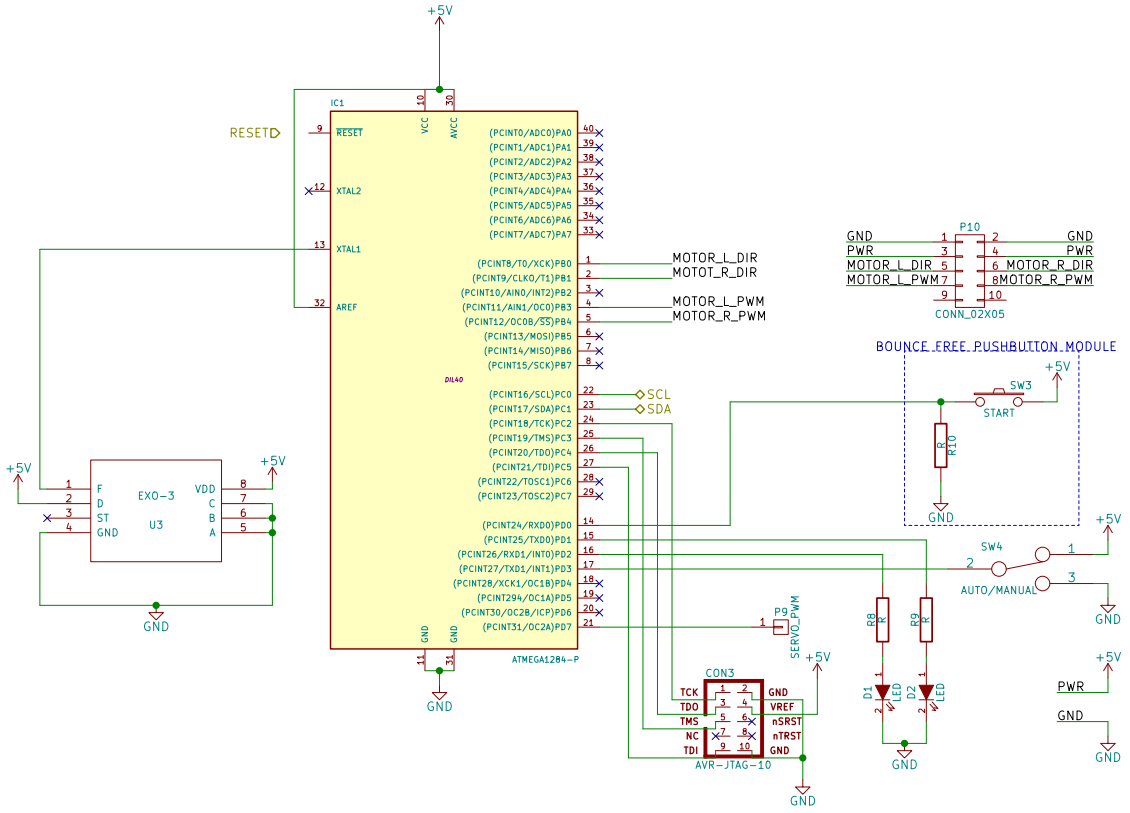
uint8_t i2c_write(uint8_t address, uint8_t* data, uint8_t length) Denna funktion inleder en sändning av ett meddelande till önskad modul. Adressen som anges är modulberoende och hittas i tabell 1. Funktionen returnerar 1 om bussen var ledig och meddelandet sändes och 0 om den egna modulen redan skickar ett meddelande. Detta får sedan avsändaren hantera genom att avgöra om det är viktigt att meddelandet går fram och i så fall försöka skicka igen.

void handle_received_message() Samtliga moduler på roboten har denna funktion som anropas när ett meddelande är korrekt mottaget. Funktionen implementeras på varje modul för att meddelanden ska omhändertas på önskat sätt.

5 Modulerna

Robotens funktionalitet är uppdelad i ett antal moduler. På själva roboten sitter styrmodulen, sensormodulen och kommunikationsmodulen. På en dator körs PC-modulen. PC-modulen är alltså endast en mjukvarumodul. Modulerna på roboten kommunicerar sinsemellan med en I^2C-buss i multimaster-konfiguration. På grund av utrymmesskäl är alla modulerna monterade på samma kretskort på roboten men de är logiskt separerade och har ett väl definierat bussgränssnitt.

Nedan beskrivs var modul för sig och de viktigaste aspekterna och funktionerna beskrivs i detalj.



Figur 4: Kopplingsschema för styrmodulen.

5.1 Styrmodulen

Styrmodulen har en central roll för roboten och ansvarar för att ta autonoma beslut och styra motorerna. De två motorerna och gripklon styrs direkt av styrmodulen. Styrmodulen tar emot sensorvärden från sensormodulen och besluten som fattas görs utifrån dessa och robotens aktuella uppgift. Under utforskningen skapar styrmodulen en intern representation av omgivningen i syfte att kunna navigera till och från de nödställda.

5.1.1 Konstruktion

Styrmodulens beräkningar utförs på en ATmega1284P mikrokontroller. Mikrokontrollern klockas med en extern 20 MHz klocka istället för den interna klockan på 8 MHz. Då styrmodulen ska utföra många beräkningar eftertraktas högsta möjliga klockfrekvens.

Kopplingsschemat återfinns i figur 4 samt i Appendix A. Motorstyrningen sker med två ledare per motor, en för PWM och en för vilken riktning motorn ska gå i. Dessa kopplas till motordrivaren som sitter på chassit. Kontakten SERVO_PWM kopplas till gripklons servo.

Styrmodulen är ansluten till bussen för kommunikation med övriga moduler. Den har en knapp med vilken autonomt läge startas samt en omkopplare där antingen

autonomt- eller manuellt läge väljs. De två lysdioderna kan användas för *debug* av programmet under körning.

Ingående komponenter

1 x ATmega1284P
1 x EXO3/20,000 MHz (extern klocka)
1 x studsfri knapp
1 x omkopplare (autonomt/manuellt läge)
2 x 120 Ω resistor (LED)
2 x LED

5.1.2 Kodstruktur

Koden för styrmodulen är relativt omfattande. För att få en hanterbar programkod har koden delats upp i flera filer. Nedan beskrivs i vilka filer olika delar av programmet ligger.

control_unit.c Denna fil utgör ingångspunkten till programmet. Här finns alla tillstånd och mekanismen för att byta tillstånd implementerade.

control_system.c Här finns funktioner för motorstyrning samt en avbrottsrutin för att hantera acceleration och beräkning av styrsignal för motorstyrningen. Beräkningen av reglerfel görs dock i *bus_communication.c*

map.c Här finns kartrepresentationen samt hjälpfunktioner för att uppdatera kartan och robotens koordinater.

flood_fill.c Här implementeras olika varianter av kortaste vägenberäkningar utifrån datan i *map.c*.

bus_communication.c Här implementeras funktionen *handle_received_message* som kallas på från *I2C.c* när ett meddelande kommer på bussen. Här beräknas även reglerfelet då sensorvärden mottas från bussen.

I2C.c Samma fil som används av alla moduler. Hanterar sändning och mottagning av meddelanden på bussen.

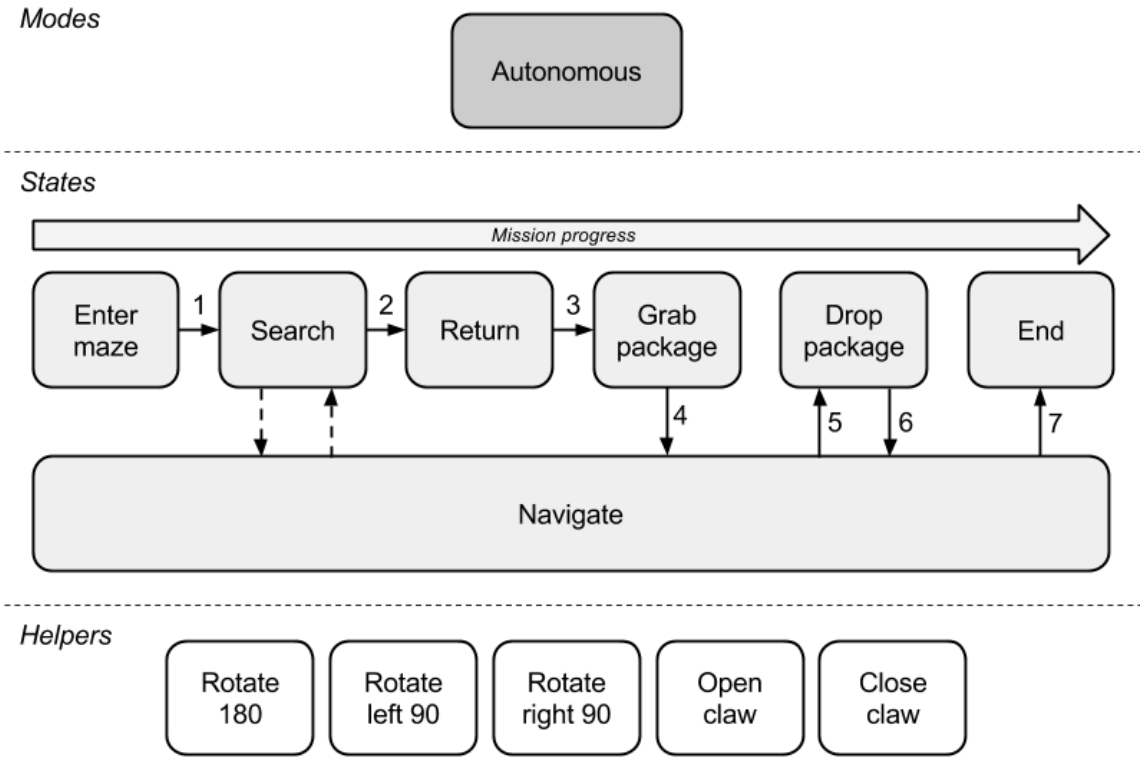
mult16x16.h Ett bibliotek för att göra effektiva multiplikationer med 16-bitars tal [3] på en AVR-processor.

5.1.3 Tillståndsmaskinen

Tillståndsmaskinen som utgör styrmodulens beslutsfattande är implementerad med 14 stycken C-funktioner där två av dem är huvudloopar för det autonoma respektive manuella läget. För det autonoma läget finns det sju stycken tillståndsfunktioner som styr programflödet under ett uppdrag. Till sin hjälp finns det även fem hjälpfunktioner

för att hantera robotens rotation och styrning av gripklo. Se figur 5 för en översikt av de olika tillstånden och deras tillståndsövergångar under loppet av ett uppdrag.

Övergången mellan två tillstånd sker genom att det tillstånd man lämnar sätter en eller två globala variabler: *next_state* och *follow_up_state*, med funktionspekare till det eller de tillstånd som ska köras härnäst. Därefter returnerar tillståndet man lämnar och programmets huvudloop kör den funktion som *next_state* pekar på. De tillstånd som drar nytta av att man kan sätta ett *follow_up_state* kopierar detta till *next_state* innan den returnerar till huvudloopen.



Figur 5: Illustration av tillstånd i autonomt läge och deras övergångar. De heldragna pilarna illustrerar de huvudsakliga tillståndsövergångarna för ett uppdrag och numreringen anger ordningen. De sträckade pilarna visar övergångar som kan ske godtyckligt många gånger innan nästa numrerade övergång sker.

De sju tillståndsfunktionerna beskrivs mer ingående nedan:

Enter maze Roboten åker från startområdet in i labyrinten till startkoordinaten.

Search Roboten utforskar labyrinten enligt följande algoritm:

1. Om möjligt: kör rakt fram.
2. Annars, om möjligt: sväng vänster och gå tillbaka till nummer 1.
3. Annars, om möjligt: sväng höger och gå tillbaka till nummer 1.
4. Annars, gör en *flood_fill_to_unmapped*, växla till *Navigate* för att följa rutten och gå sedan tillbaka till nummer 1.

Under själva utforskningen uppdateras den interna kartrepresentationen. Detta görs då roboten passerar mitten av varje ruta och då markeras denna som körbar och utifrån sidosensorer och framåtsensor uppdateras kartan med eventuella väggar.

Return Roboten börjar med att göra en optimistisk skattning (*flood_fill_home_optimistic*, se avsnitt 5.1.7) av kortaste vägen till start och jämför denna skattning med hur långt det är via redan utforskade delar. Om det är lika långt att köra längs den redan utforskade vägen som för den optimistiska skattningen kör roboten tillbaka till starten längs den utforskade vägen. Annars börjar roboten köra längs den optimistiskt skattade vägen. I varje ruta uppdateras kartan med information om omgivningen varefter den nuvarande rutten utvärderas på nytt. Detta görs genom att jämföra den påbörjade optimistiska vägen med en ny optimistisk skattning från mål till start. Om den nya optimistiska skattningen är kortare kör roboten tillbaka mot mål tills den befinner sig på en ruta som ingår i den nya vägen och fortsätter därifrån mot starten.

Grab package Roboten kör in i startområdet, öppnar klon, stänger klon och kör sedan tillbaka ut i labyrinten.

Drop package Roboten backar 10 cm för att positionera klon i målrutans mitt, öppnar klon, backar ytterligare 30 cm för att hamna i mitten av första rutan på vägen tillbaka till start.

End Roboten kör in i startområdet och stannar.

Navigate Ett centralt tillstånd som används i flera olika sammanhang för att navigera utifrån en på förhand framtagen rutt. Tillståndet använder hjälpfunktionerna för rotation för att ställa roboten i det väderstreck som rutten anger och kör sedan framåt tills dess att en ny rotation krävs eller att rutten är slut. Tillståndet innehåller inte någon logik för tillståndsövergångar utan tillståndet som övergår till *Navigate* sätter *follow_up_state* till den tillståndsfunktion som ska komma efter *Navigate*.

Tillståndsmaskinen är konstruerad så att de huvudsakliga tillstånden följer efter varandra utan möjlighet att gå tillbaka. Det finns alltså inget sätt att börja om uppdraget utan att starta om roboten.

Om robotens skjutomkopplare är satt till manuellt läge körs en annan huvudloop som inte byter mellan olika tillstånd utan enbart styr motorerna utifrån styrkommandon som kommer från PC-modulen.

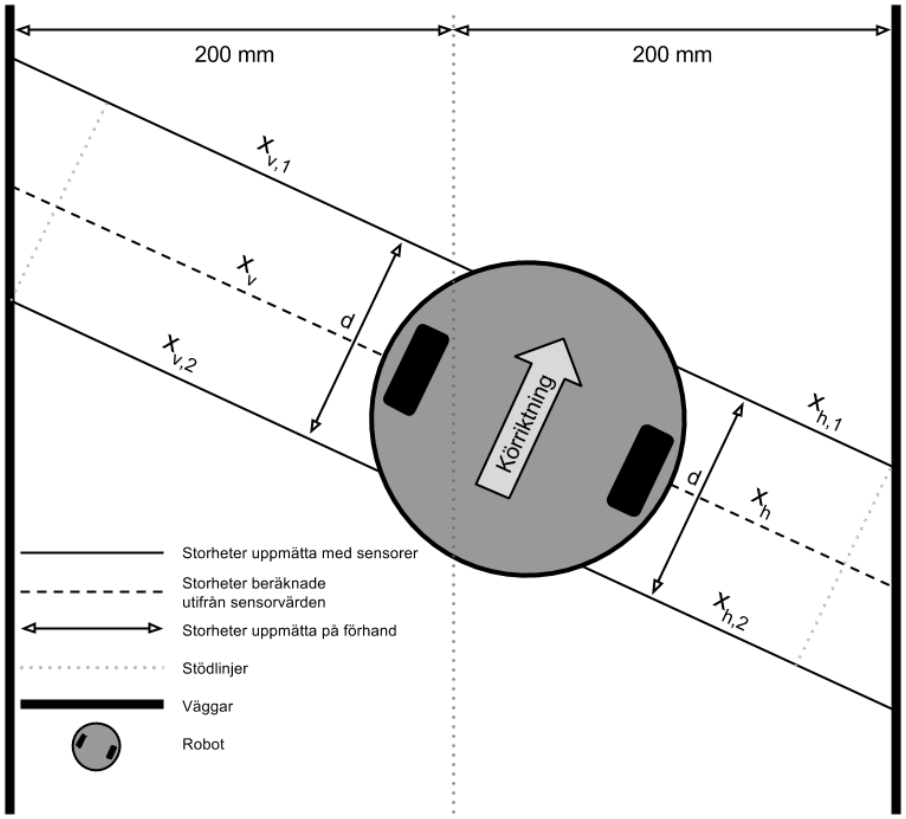
5.1.4 Reglering

För att realisera ett reglersystem likt det som beskrivs i avsnitt 3.1 behöver ställning tas till när och hur regleringen ska användas. Till exempel måste systemet kunna hantera att det bara finns en vägg på ena sidan eller ingen alls. Detta görs genom att undersöka alla sensorvärdens rimlighet. Se figur 6 för tillgängliga sensorvärden.

Om en enskild sidsensor anger ett värde större än 300 mm eller att dess derivata är större 350 mm/s anses värdet vara oanvändbart för reglering. I detta läge utesluts den sidans sensorvärden helt från uträkningen av reglerfelet. Med derivatans värde kan hörn upptäckas innan sensorvärdet har hunnit bli för stort när roboten kommer in till en korsning. Beräkning av derivatan av en godtycklig signal $y(t)$ görs enligt

$$\left.\frac{\partial y(t)}{\partial t}\right|_{t=nT_s} \approx \frac{(y[n+1]-y[n])}{T_s} \equiv \dot{y}[n], \tag{2}$$

där $T_s = 1/f_s$ och där f_s är sensorernas samplingsfrekvens, 25 Hz.



Figur 6: Roboten i en korridor med avstånd uppmätta av sensorerna utmarkerade. x_v och x_h är genomsnitten av avståndsvärdena på respektive sida och d är en på förhand uppmätt konstant.

Om alla sensorer visar rimliga värden och har rimliga derivator så räknas avstånden x_v och x_h ut som två genomsnitt enligt ekvation (3).

För sidor som inte uteslutits för reglering räknas avstånd till vägg ut enligt

$$\begin{aligned} x_v[n] &= \frac{x_{v,1}[n] + x_{v,2}[n]}{2}, \\ x_h[n] &= \frac{x_{h,1}[n] + x_{h,2}[n]}{2}. \end{aligned} \tag{3}$$

Reglerfelet $e[n]$ och dess derivata $\dot{e}[n]$ räknas ut utifrån tillgängliga avståndsvärden och avståndsderivator. Om det finns oegentligheter på båda sidorna sätts reglerfelet och dess derivata till 0. De olika fallen sammanfattas i ekvationer (4) och (5).

$$e[n] = \begin{cases} x_v - x_h, & \text{om båda sidor OK,} \\ x_v - 140, & \text{om endast vänster sida OK,} \\ 140 - x_h, & \text{om endast höger sida OK,} \\ 0, & \text{annars.} \end{cases} \tag{4}$$

$$\dot{e}[n] = \begin{cases} \dot{x}_v[n] + \dot{x}_h[n], & \text{om båda sidor OK,} \\ 2\dot{x}_v[n], & \text{om endast vänster sida OK,} \\ 2\dot{x}_h[n], & \text{om endast höger sida OK,} \\ 0, & \text{annars.} \end{cases} \tag{5}$$

Värdet 140 mm i ekvation (4) är sträckan från robotens periferi till närmaste vägg då roboten befinner sig i mitten av en korridor. Hur styrsignalen beräknas och används beskrivs i kapitel 5.1.5.

5.1.5 Motorstyrning

Motorstyrningen sker med två ledningar per motor, en pulsbreddsmodulerad signal för hastighetsstyrning och en signal för vilken riktning motorn ska gå i. Pulsbreddsmoduleringen har 8 bitars upplösning och hastigheten anges därför som ett 8 bitars tal. Riktningssignalen är binär.

Hastigheten för motorn sätts aldrig momentant utan en önskad hastighet, *desired_engine_speed*, anges och en avbrottsrutin som körs i 500 Hz accelererar eller retarderar sedan *current_engine_speed* som är det värde som faktiskt används som pulsbredd. I samma avbrottsrutin räknas styrsignalen $u[n]$ ut enligt ekvation (6) där v representerar *current_engine_speed*.

$$u[n] = \begin{cases} K_P e[n] + K_D \dot{e}[n], & \text{om } |K_P e[n] + K_D \dot{e}[n]| < v/2, \\ v/2, & \text{annars.} \end{cases} \tag{6}$$

När styrsignalen har beräknats undersöks dess tecken. Är värdet positivt subtraheras det från vänstermotorns hastighet och om det är negativt så subtraheras dess belopp från högermotorns hastighet.

5.1.6 Gripklo

Robotens gripklo styrs genom att ange pulsbredd för dess servo. Det finns två funktioner för att ange klons läge, *close_claw* och *open_claw*, som är anpassade för att bära och släppa en MER-förpackning.

5.1.7 Kartering och kortaste vägen

Som intern kartrepresentation används en 33×33-matris. Denna är initialt fylld med värdet UNMAPPED. Dessa värden kan sedan utifrån vad roboten utforskar ändras till NOT_WALL eller WALL. För lagring av avståndsvärden som genereras vid en flood fill-beräkning används en lika stor matris.

För kortaste vägenberäkningar används modifierade varianter av vanligt förekommande flood fill-algoritmer då de även fungerar i delvis outforskade miljöer. Vilka rutor som tillåts att fyllas samt när fyllningen ska avbrytas varierar mellan de olika versionerna som har implementerats:

flood_fill_to_destination(coordinate destination) börjar på robotens position och fyller rutor som roboten har satt till *NOT_WALL* tills koordinaten *destination* har givits ett avståndsvärde. Detta används till exempel om man vill ta sig från start till mål när omgivningen är känd.

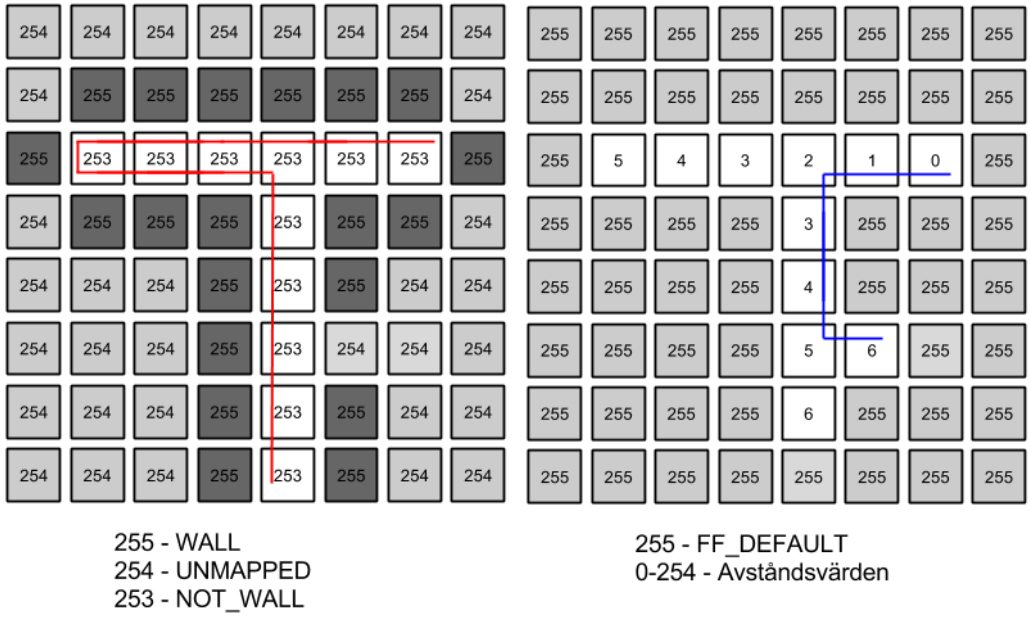
flood_fill_to_unmapped() börjar på robotens position och fyller rutor som roboten satt till *NOT_WALL* tills en outforskad ruta har givits ett avståndsvärde. Detta används under utforskningen då roboten har kört in i en återvändsgränd eller kommit tillbaka till en redan utforskad ruta och vill hitta ett nytt område att fortsätta avsökningen i.

flood_fill_home_optimistic(coordinate origin) börjar i koordinaten *origin* och fyller alla rutor, även de som är outforskade, med avståndsvärden tills dess att labyrintens startruta har fått ett avståndsvärde. Denna funktion används när roboten har hittat målet och vill söka upp kortaste vägen tillbaka till starten.

Alla flood fill-algoritmer avslutas med att funktionen *calculate_route* anropas. *calculate_route* tar den avståndsdata som skapats samt den koordinat man vill färdas till och skapar en rutt för hur man ska ta sig dit. Denna rutt är en global lista av väderstreck som roboten ska följa. Rutten skapas genom att baklänges iterera från målet till starten och lista i vilket väderstreck man ska åka för att ta sig till denna ruta. I lägen då det finns flera alternativa vägar av samma längd kommer roboten att försöka göra så få svängar som möjligt. Denna optimering är dock girig och skapar inte alternativa rutter som den sedan jämför. Det är alltså inte säkert att den rutt som skapas är den med minst antal svängar men ett enkelt försök att minska antalet svängar görs. Själva ruttföljningen görs av tillståndet *Navigate*. Detaljer om detta tillstånd finns i avsnitt 5.1.3.

5.2 Sensormodulen

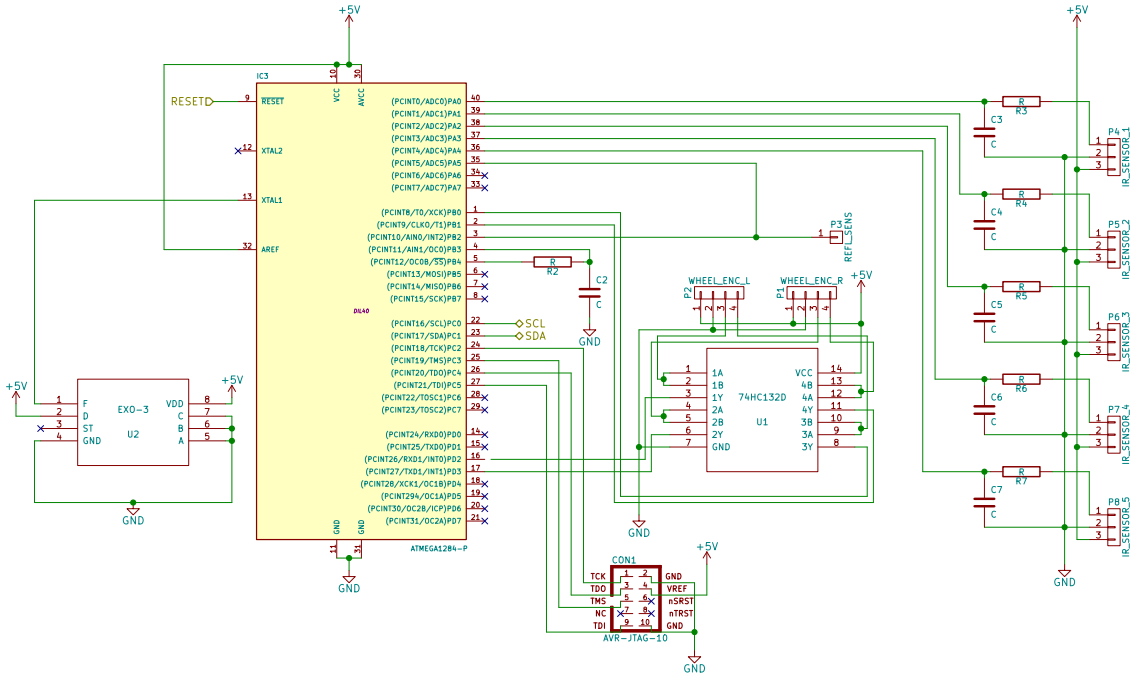
Sensormodulen är den del av roboten som ansvarar för att läsa av och behandla data från systemets samtliga sensorer. Tre sensortyper utnyttjas: IR-sensorer för avståndmätningar till omgivningen, optiska pulsgivare för mätning av avlagd sträcka och rotation, samt en reflexsensor för att upptäcka svarta tejpremsor på underlaget. Behandlad sensordata skickas på bussen till övriga moduler som beskrivet under avsnitt 4.3.



Figur 7: Kartrepresentation till vänster och data från flood fill till höger. Roboten har kört enligt det röda strecket och anropar sedan funktionen flood_fill_to_unmapped. *Roboten beräknar sedan en rutt längs det blåa strecket till den outforskade rutan.*

5.2.1 Konstruktion

Beräkningarna i sensormodulen utförs på en ATmega1284P med en extern klocka på 20 MHz. I kopplingsschemat i figur 8 illustreras hur modulens olika komponenter är sammankopplade och vilka portar på mikrokontrollern som används. Kopplingsschemat återfinns även i Appendix A.



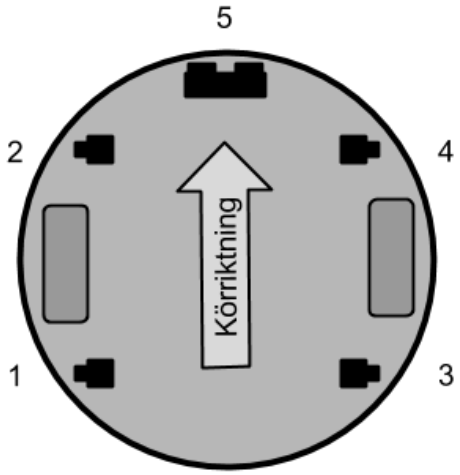
Figur 8: Kopplingsschema för sensormodulen.

Ingående komponenter

1 x ATmega1284P
1 x EXO3/20,000 MHz (extern klocka)
5 x 18 kΩ resistor (lågpassfilter IR-sensor)
5 x 100 nF kondensator (lågpassfilter IR-sensor)
4 x GP2Y0A21 (IR-sensor 10-80 cm)
1 x GP2Y0A02YK (IR-sensor 20-150 cm)
1 x 4,7 kΩ resistor (lågpassfilter PWM)
1 x 100 μF kondensator (lågpassfilter PWM)
1 x reflexsensor
1 x 74HC132D (quad 2-input NAND schmittrigger)
2 x Pololu 5 V optisk pulsgivare

5.2.2 IR-sensorer

För avståndsmätning till omgivningen används fyra sensorer för mer exakt mätning av korta avstånd och en sensor för längre och mindre kritiska avstånd. De olika sensorernas montering och numrering illustreras i figur 9. Denna numrering matchar den i kretsschemat och sensorernas identifiering i mjukvaran. Sensorer 1-4 mäter inom intervallet 71-304 mm och normalt inom en felmarginal på 5 mm. Då avläst värde går under och över det givna intervallet anges aktuell sensors värde som 0 respektive 65536. Sensor 5 mäter inom intervallet 148-891 mm och normalt inom en felmarginal på 15 mm. Även denna sensors värde anges som 0 och 65536 då



Figur 9: IR-sensorernas montering och numrering.

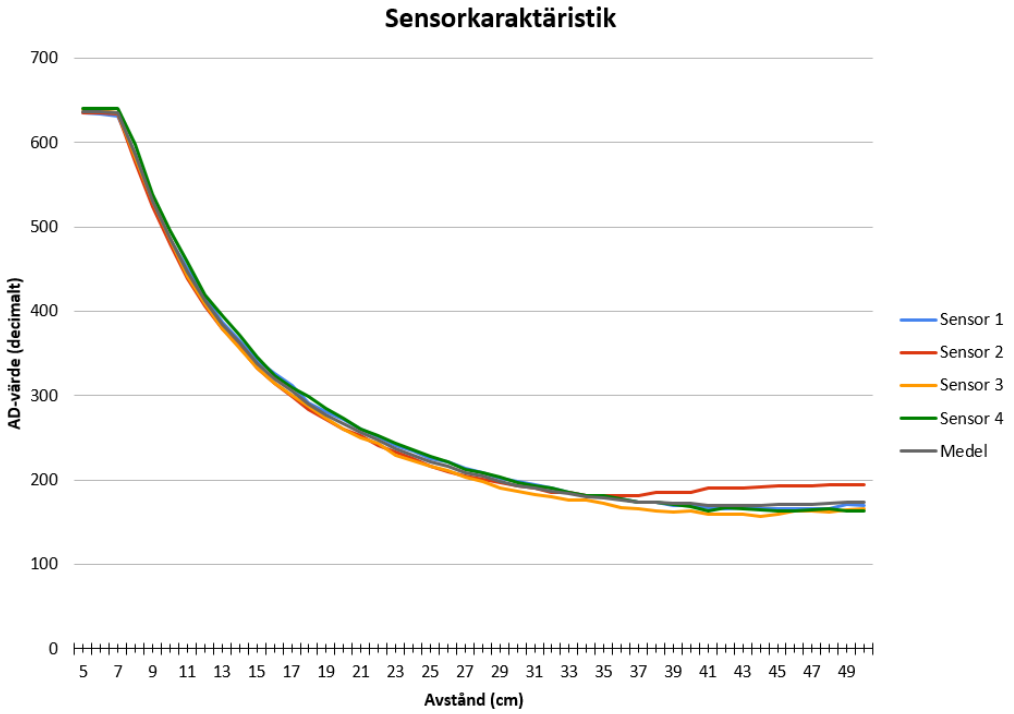
mätningar hamnar under respektive över det givna intervallet. För konvertering från AD-omvandlat spänningsvärde till avstånd i millimeter används *look-up-tables* framtagna utifrån följande polynom anpassade efter mätvärden:

$$\begin{aligned} y_{1-4} &= a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \\ a_4 &= 0,0000000013804522 \\ a_3 &= -0,0000027237335169 \\ a_2 &= 0,002027049404442 \\ a_1 &= -0,699803588682573 \\ a_0 &= 107,088548372709 \end{aligned} \tag{7}$$

$$\begin{aligned} y_5 &= b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \\ b_5 &= -0,0000000000387658 \\ b_4 &= 0,000000072590202 \\ b_3 &= -0,0000541266871615 \\ b_2 &= 0,020268372823463 \\ b_1 &= -3,94414786764439 \\ b_0 &= 366,792867978953 \end{aligned} \tag{8}$$

där x är det AD-omvandlade spänningsvärdet. Mätningar från sensorer 1-4 samt en graf av det anpassade polynomet i ekvation (7) illustreras i figur 10 respektive 11.

Sensorerna samplas i sekvens från ett avbrott med en frekvens på 25 Hz vilket överensstämmer med sensorernas uppdateringsfrekvens [4], [5]. De avlästa värdena läggs i cirkulära buffrar varefter medelvärden av de två senaste mätningarna beräknas och skickas på bussen som *Generall call*.



Figur 10: Sensorkaraktäristik för sensorer 1-4.

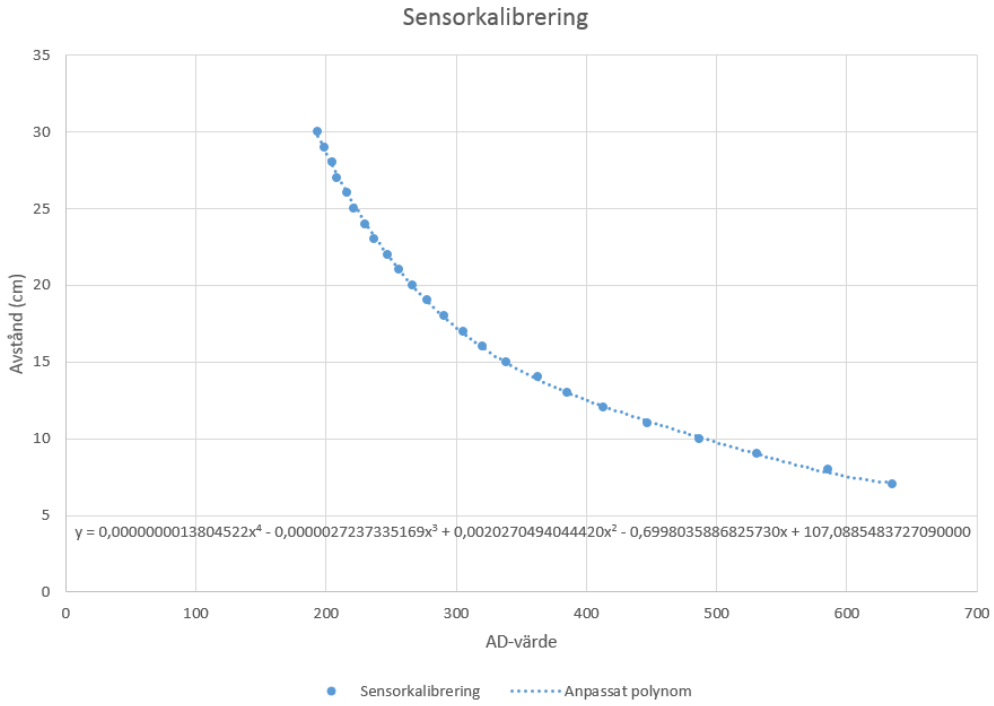
5.2.3 Pulsgivare

För att bestämma robotens avlagda sträcka och rotation används optiska pulsgivare monterade på motoraxlarna. Var pulsgivare genererar två sinusliknande vågformer 90° ur fas med varandra. Signalerna omvandlas till pulståg via schmittriggrar varefter en signal per givare används som avbrottskälla. I respektive avbrott läses motsvarande andra signal av för att avgöra rotationens riktning. Då pulsgivarna är monterade väldigt nära och riktade mot varandra finns en överhängande risk att de stör ut varandra. För att undvika detta har en svart avskiljare placerats mellan sensorerna.

Pulsgivarna räknar tre tick per varv, vilket med motorernas utväxling 50:1 resulterar i 150 tick per hjulrotation. För att inte överbelasta bussen och dessutom skicka onödigt högupplöst mätdata inväntas fem tick från någon av givarna innan omvandling från antal tick till avlagd sträcka i millimeter samt rotation i grader görs och skickas till styrmodulen. Omvandlingen görs med upplösningen 256:e delars millimeter respektive grader per bit. Beräkningen utförs därför i 16 bitars utrymme varefter resultatet trunkeras och skiftas ner till 8 bitar, med upplösningen 1 millimeter respektive grad per bit, för att slutligen skickas på bussen som *Generall call*. Trunkeringen sparas och adderas till nästa omvandling.

5.2.4 Reflexsensor

En reflexsensor är monterad på robotens undersida för att detektera svarta tejpmarkeringar på underlaget. Mikrokontrollerns analoga komparator används för att jämföra sensorns analoga utsignal med en kalibrerbar referenssignal. Referenssigna-



Figur 11: Det anpassade polynomet i ekvation (7).

len utgörs av en kraftigt lågpassfiltrerad pulsbreddsmodulerad signal vars tillslagstid (eng. *duty cycle*) bestäms vid kalibrering och sätts till medelvärdet av sensorns aktuella utsignal och 5 V. Nivån på sensorns utsignal ökar med minskande reflektans så komparatorn triggar då signalen går över referenssignalen. Då detta sker genereras ett avbrott varifrån övriga moduler meddelas via bussen.

5.3 Kommunikationsmodulen

Kommunikationsmodulen fungerar som en länk mellan PC-modulen och robotens moduler. Vid autonom körning skickas bland annat kartdata och sensorvärden från styr- respektive sensormodulerna via I^2C till kommunikationsmodulen som skickar dessa vidare till PC-modulen. I manuellt läge skickas styrkommandon från PC:n vidare till styrmodulen. Se avsnitt 4.3 för en komplett beskrivning av robotens samtliga meddelandetyper.

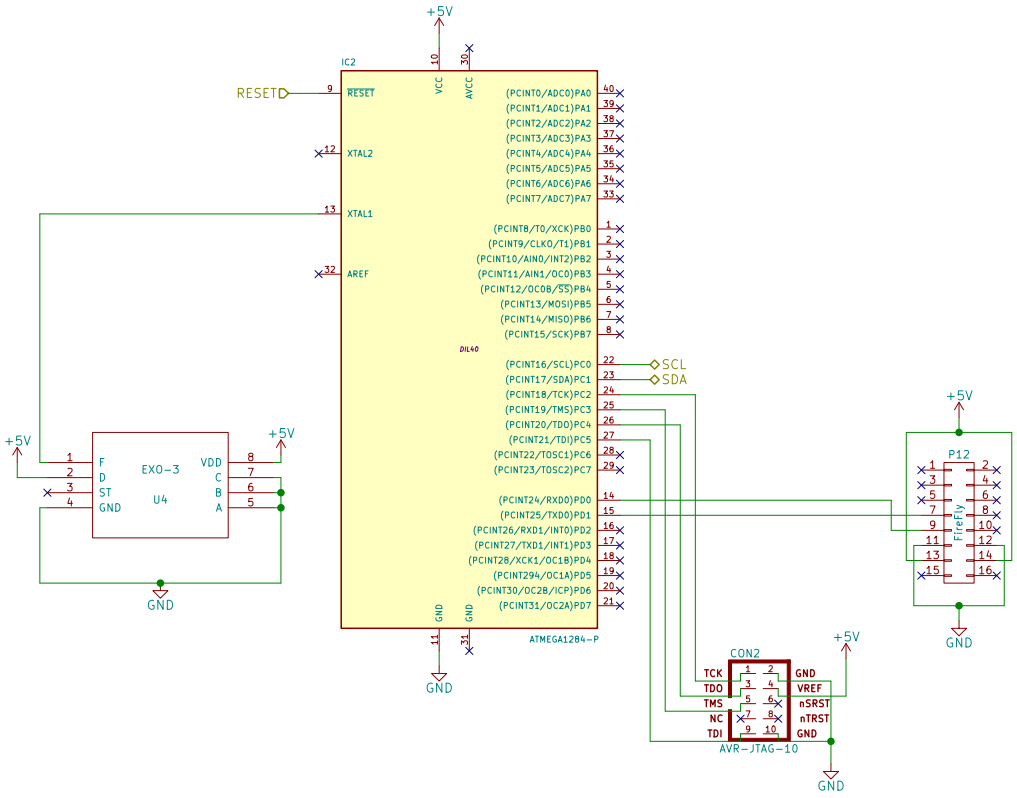
5.3.1 Konstruktion

Kommunikationsmodulens beräkningar görs på en ATmega1284P. Till denna mikrokontroller är en Bluetoothmodul ansluten via UART med en överföringshastighet (eng. *Baud Rate*) på 115200 b/s. Mikrokontrollern klockas av en extern klocka med frekvensen 18,432 MHz. Valet av denna klockfrekvens grundas i att detta är den högsta hastighet för vilken mikrokontrollerns UART uppnår önskad överföringshastighet utan att fel introduceras under sändningen som konsekvens av icke överensstämmande

klockfrekvenser. Kommunikationsmodulens kopplingsschema återfinns i figur 12. Kommunikationsmodulen är ansluten till robotens övriga moduler via bussen.

Ingående komponenter

1 x ATmega1284P
1 x EXO3/18,432 MHz (extern klocka)
1 x BlueSMiRF Gold (Bluetoothmodul)
2 x 1 kΩ resistor (pull-up för buss)



Figur 12: Kopplingsschema för kommunikationsmodulen.

5.3.2 Meddelandehantering

Kommunikationsmodulens uppgift är att vidarebefordra inkommen data till korrekt modul. Meddelanden från styr- och sensormodulerna skickas direkt till PC-modulen via Bluetooth. Detta görs genom att inspektera headern på meddelandet och sedan skicka vidare korrekt antal bytes från en buffer som bussen fyllt upp. Denna process sker så fort mottagningen av meddelandet är slutförd genom att I^2C-hårdvaran triggar ett avbrott vid genomförd sändning. Meddelanden som kommer från PC-modulen via Bluetooth hanteras på ett annat vis. Vid genomförd mottagning triggar UART-hårdvaran ett avbrott varifrån en flagga sätts för att sedan kontrolleras i kommunikationsmodulens huvudloop. Denna huvudloop innehåller en tidsfördröjning vars funktion är att styra frekvensen på data som skickas från PC-modulen. Dessa

meddelanden är huvudsakligen styrkommandon då roboten befinner sig i manuellt läge. För att styrmodulen ska hantera det manuella läget på ett förutsägbart sätt krävs att styrkommandon inkommer med en lagom frekvens.

5.4 PC-modulen

PC-modulen hanterar inkommande data från roboten samt visar den behandlade datan i ett grafiskt användargränssnitt. Styrkommandon och reglerparametrar går även att skicka från PC-modulen. Programmet är skrivet i Visual C++/CLI med .NET-framework där klasser för användargränssnitt och serieport finns fördefinierade.

5.4.1 Serieport

Data som skickas från roboten kommer till PC-modulen via en serieportsanslutning. Ett meddelande består av en header, som beskriver vilken data som följer, se tabell 2 för detaljer.

Programmet körs i två trådar, en för serieporten och en för användargränssnittet. Detta tillåter serieportstråden att sätta en flagga vid mottagen data och skicka datan vidare utan att behöva vänta på tidskrävande utritning. När data tas emot läses alla tillgängliga bytes in till en global array som hanteras av en funktion, *receive_data*, som arbetar i användargränssnittets tråd. Det finns två huvudfunktioner som *receive_data* använder sig av: *handle_header* avgör vilken header som tagits emot och hur många bytes som förväntas och *handle_byte* hanterar efterföljande bytes på rätt vis. Om arrayen inte innehåller tillräckligt många bytes hämtas ytterligare bytes från serieporten tills det fullständiga meddelandet har mottagits.

Styrkommandon, förfrågan om kalibrering av reflexsensorn samt nya reglerparametrar skickas skickas med headers enligt tabell 2.

5.4.2 Kartutritning

Karteringen av labyrinten sköts av styrmodulen. PC-modulen tar emot data för att utföra själva visualiseringen i ett grafiskt användargränssnitt. Detta sker genom att PC-modulen tar emot kartinformation i form av x- och y-koordinater för väggar och körbara rutor. Information om roboten och målets positioner tas också emot och ritas ut. Kartinformationen sparas i en 33×33-matris och visualiseras med olika färger i det grafiska gränssnittet. Körbar ruta ritas ut med vit färg, vägg med svart, nödställd med röd och nuvarande position markeras med en blå ruta. Dessa visas sedan i ett 17×17-rutnät i det grafiska användargränssnittet, då detta räcker för att täcka in hela labyrintens storlek, se banspecifikationen [1].

För utritningen av användargränssnittets 17×17-rutnät görs ett utsnitt ur den större 33×33-matrisen. Utifrån utsnittets övre vänstra hörn ritas 17 rutor i både x- och y-led ut. Eftersom utformningen av labyrinten från början är okänd är utsnittet flexibelt så att all aktuell kartdata kan visas i användargränssnittet. Detta steg sköts av funktionen *move_grid*.

För att uppdatera kartan används funktionen *update_map* som dels sköter uppdateringen av robotens position och dels anropar själva utritningsfunktionen *fill_square*. Utritningsfunktionen *fill_square* ritar ut avsedd färg för aktuell ruta.

6 Vidareutveckling

För att hantera läsning och skrivning av längre meddelanden på bussen har två linjära buffrar implementerats. Detta medför att endast en skrivning kan initieras åt gången och att denna måste slutföras innan efterföljande skrivningar inleds. På motsvarande sätt måste ett mottaget meddelande omhändertas innan ett nytt meddelande kan börja tas emot. Detta ställer oönskade krav på den som utnyttjar busskommunikationen. En vidareutveckling av systemet kan involvera cirkulära buffrar för både läsning och skrivning för att undgå några av dessa begränsningar.

Robotens autonoma körning regleras av en PD-regulator. Identifiering av olikheter i motorernas styrka indikerat att en integrerande del i regulatorn kan vara till nytta. Reglering med hjälp av motorernas pulsgivare är också något som kan utvärderas för att även möjliggöra reglering i situationer där närliggande väggar saknas, som till exempel i en fyrvägskorsning. Mer pålitliga IR-sensorer med högre uppdateringshastighet kan underlätta i utvecklandet av en mer robust regulator som även kan hantera högre hastigheter.

Roboten saknar i dagsläget funktionalitet för att autonomt lokalisera och plocka upp förnödenheter. Diskussioner har förts kring att utgå ifrån en fix placering av förnödenheterna i startrutan och sedan använda väggarna för att positionera roboten i rätt läge. Den svarta tejpmarkeringen som definierar ingången till labyrinten [1] kan då eventuellt användas för att avgöra hur långt roboten ska köra tills förnödenheterna kan plockas upp. Med denna metod undviks behovet av ytterligare sensorer.

En möjlighet att göra roboten mer tillförlitlig rent mekaniskt är att införskaffa större hjul så att motorerna kan monteras på ovansidan av bottenplattan. Detta skulle skydda motorerna och pulsgivarna bättre från damm och smuts på underlaget och därmed öka komponenternas livslängd. För att ytterligare minska robotens storlek kan ett eget kretskort tas fram med CAD-verktyg. Vidare kan en betydligt mindre motordrivare införskaffas. Detta skulle kunna halvera robotens höjd och göra roboten betydligt mer stabil.

Programvaran erbjuder i dagsläget endast visning av de nuvarande sensorvärdena roboten mäter upp. För att kunna följa robotens beteende vore det önskvärt att visa tidigare sensorvärden i en tabell alternativt i en graf. Kartutritningen sker för tillfället med en viss fördröjning vilket kan undersökas vidare.

Referenser

[1] Grupp 1-6, ”Banspecifikation”, Version 1.0, febr. 2015.

[2] L. Ragnarsson och E. Sköld, ”Förstudie, reglering och kartering för en autonom robot”, Version 1.2, april 2015.

[3] N. Pozar, *Arduino - avr gcc multiplication*, `https://github.com/rekka/avrmultiplication`, 2012.

[4] *Gp2y0a21yk optoelectronic device*, GP2Y0A21YK, Reference Code SMA05008, SHARP, 2005.

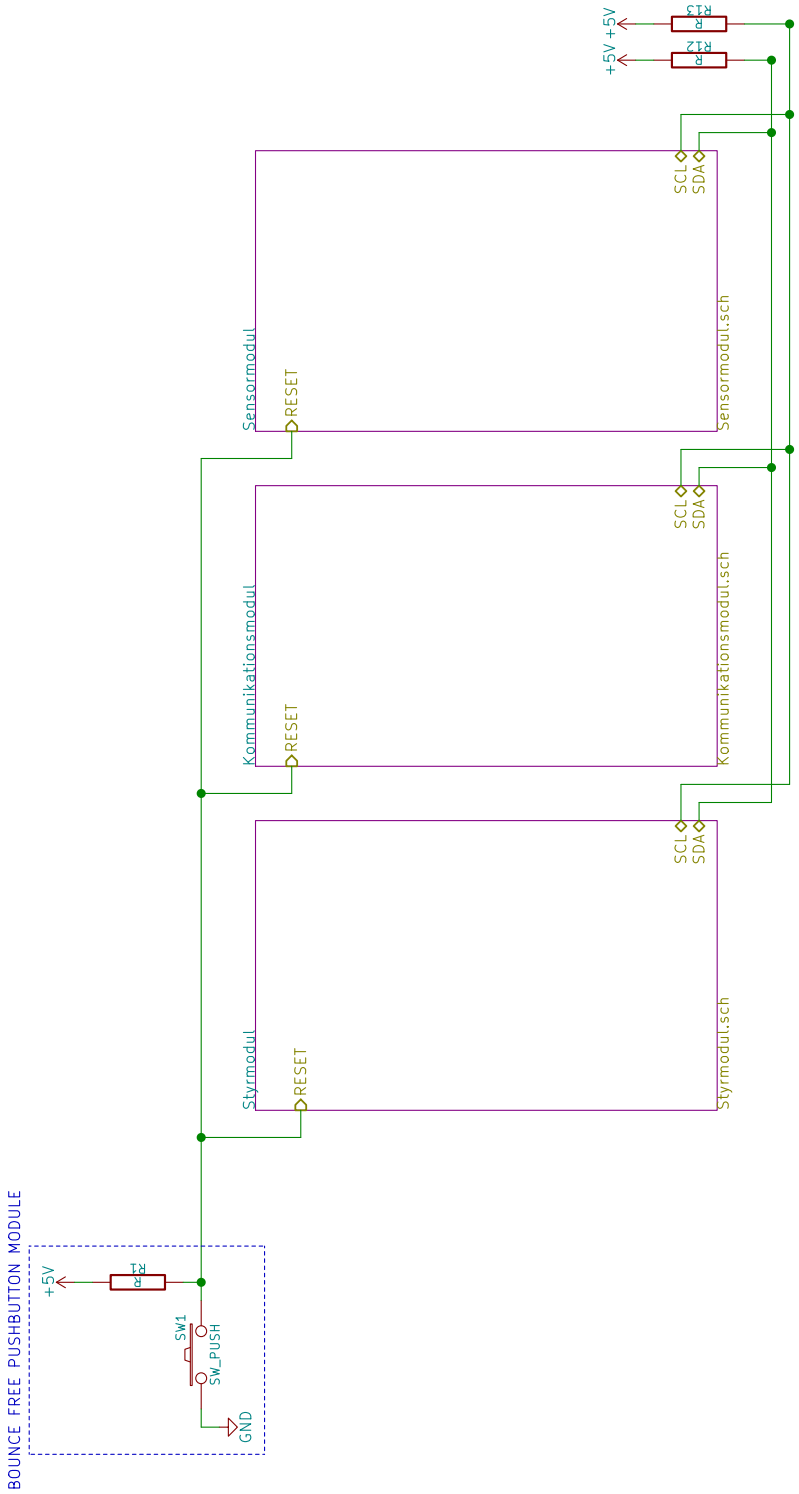
[5] *Gp2y0a02yk long distance measuring sensor*, GP2Y0A02YK, SHARP.

[6] J. Otterholm, L. Ragnarsson, E. Sköld, E. Söderström, M. Östlund Visén och F. Östman, *Ronny-the-robot*, `https://github.com/lragnarsson/Ronny-the-robot`, 2015.

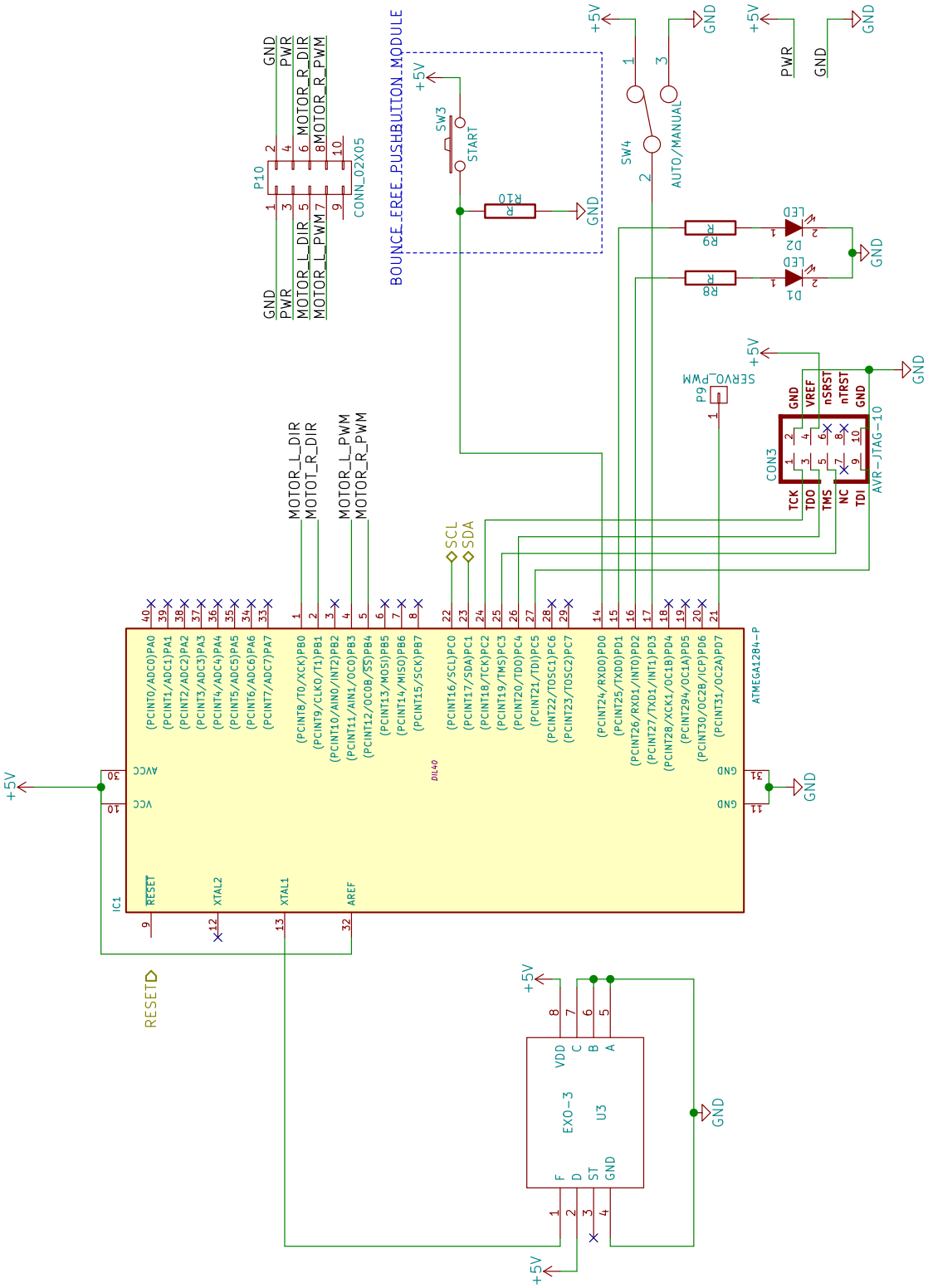
Appendix

A Kopplingsscheman

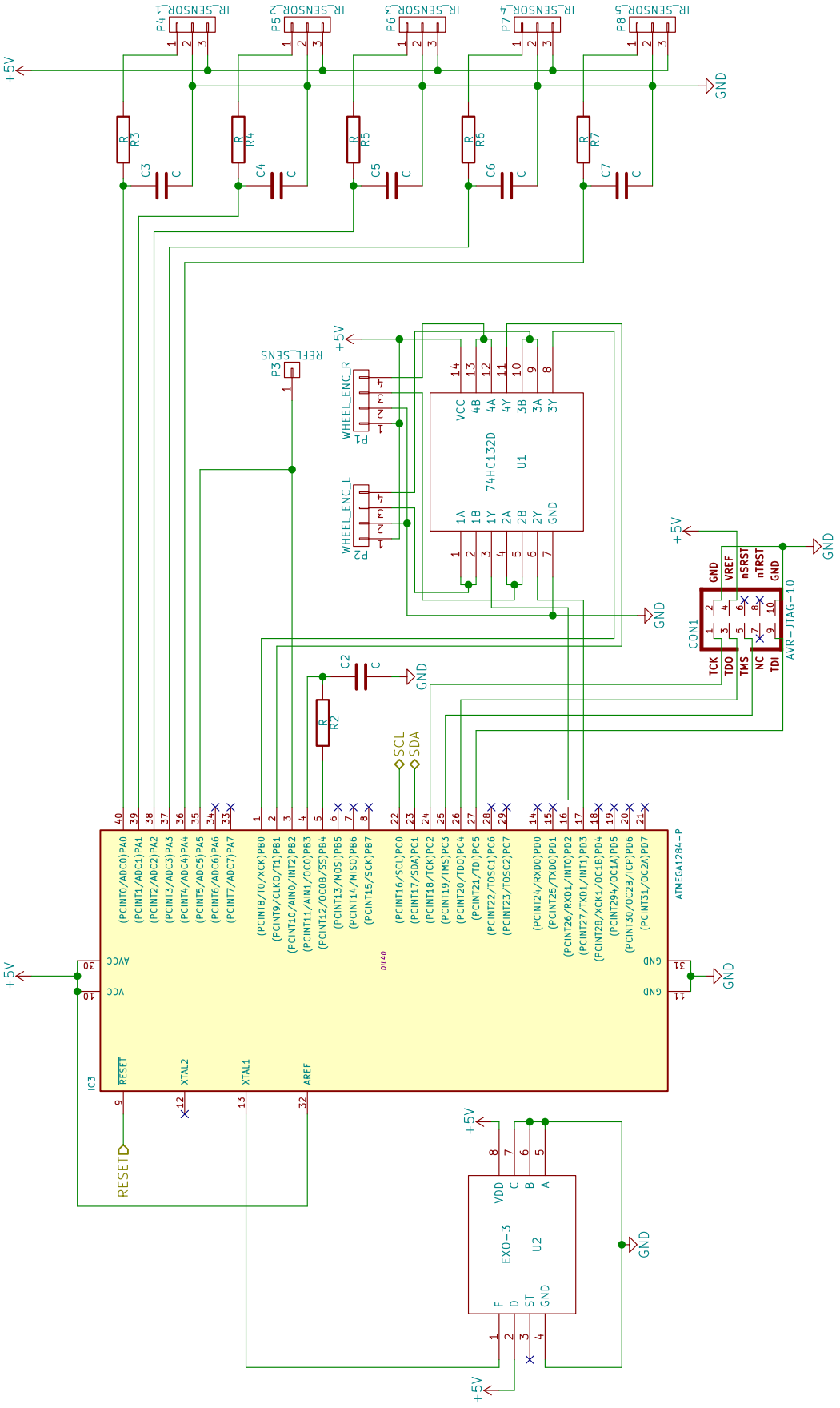
A.1 Modulernas sammankoppling



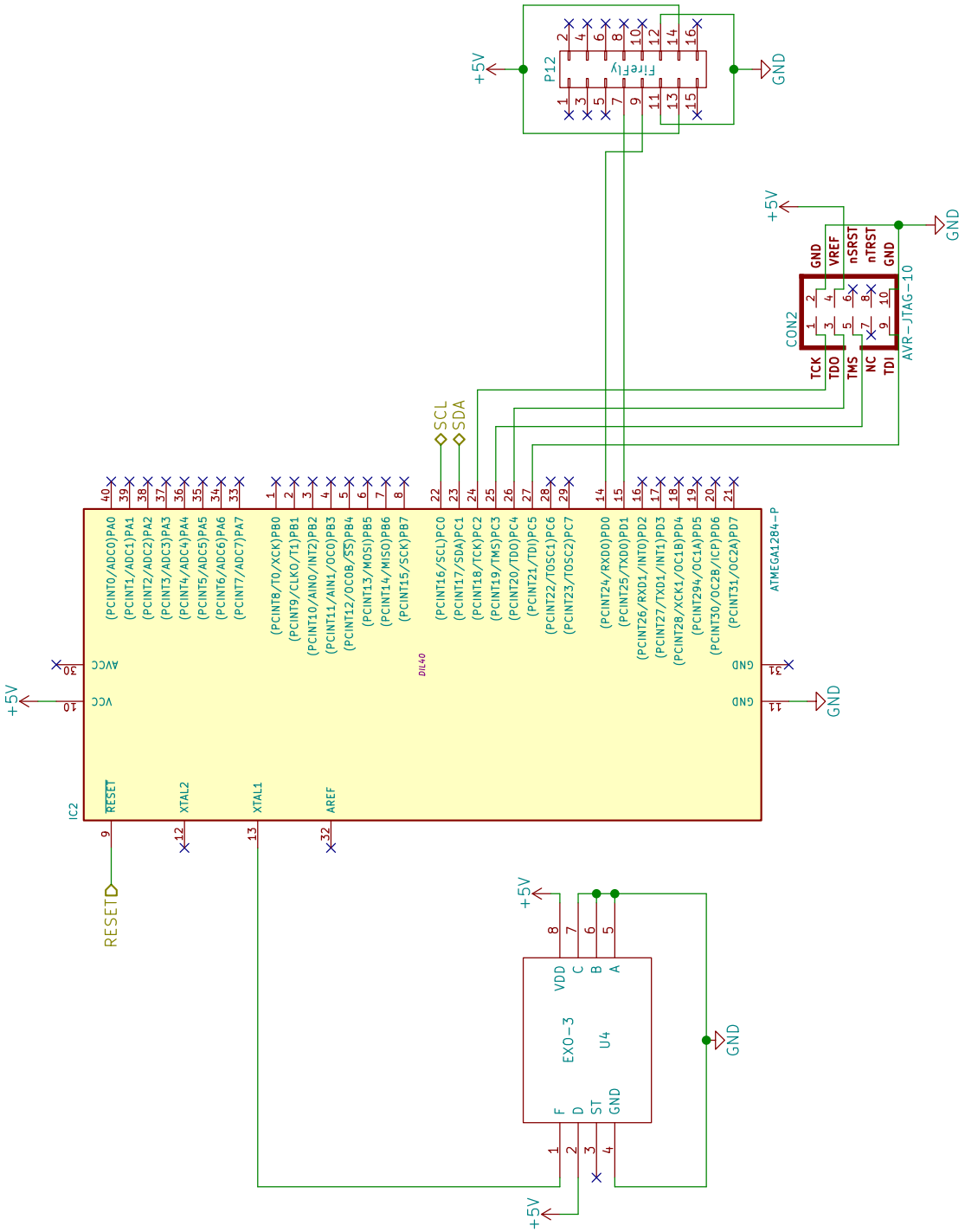
A.2 Styrmodulen



A.3 Sensormodulen



A.4 Kommunikationsmodulen



B Utdrag ur källkoden

Nedan finns ett utdrag ur källkoden för roboten. Detta är inte den fullständiga koden utan är till för att ge en bild av kodens struktur, språkkonventioner och läsbarhet. Den fullständiga källkoden finns på GitHub [6].

B.1 control_unit.c

```
/*
 * control_unit.c
 * Ronny-the-robot/control_unit
 * ------------------------------
 * This file is the entry point an backbone for the control unit.
 * The state machine for the completion of the mission is
   implemented here.
 * ------------------------------
 * Author: L. Ragnarsson, E. Sköld
 */

#include "control_unit.h"
#include "flood_fill.h"

int main(void)
{
    next_state = enter_map_state;
    init_map();
    init_bus_communication();
    init_control_system();
    DDRD |= (1<<DDD1)|(1<<DDD2); // LED output
    PORTD &= ~((1<<DEBUG_LED_GREEN)|(1<<DEBUG_LED_RED));
    sei();

    /* Wait for start button press */
    while (PIND & (1<<PIND0))
    {
        _delay_ms(1);
    }

    /* Switch in autonomous or manual? */
    if (PIND & (1<<PIND3))
    {
        current_mode = AUTONOMOUS;
        i2c_write_byte(COMMUNICATION_UNIT, AUTONOMOUS_MODE);
    }
    else
    {
        current_mode = MANUAL;
        i2c_write_byte(COMMUNICATION_UNIT, MANUAL_MODE);
    }

    _delay_ms(100);

    while (1) {
```

```
45          if(current_mode == MANUAL)
46              manual_mode();
47          else if(current_mode == AUTONOMOUS)
48              autonomous_mode();
49          else if (current_mode == TEST)
50              test_mode();
51      }
52  }
53  /*
54   * Searches the maze until the goal is found.
55   */
56  void search_state()
57  {
58      PORTD &= ~((1<<DEBUG_LED_GREEN)|(1<<DEBUG_LED_RED));
59      PORTD |= (1<<DEBUG_LED_GREEN)|(0<<DEBUG_LED_RED);
60
61      set_desired_engine_direction(ENGINE_DIRECTION_FORWARD);
62
63      /* Handle unawareness of initial surroundings */
64      set_walls(wall_front, wall_left, wall_right);
65      set_current_sqaure_not_wall();
66
67      /* We might end up in tape square directly from navigate */
68      if (tape_square)
69      {
70          set_desired_engine_speed(0);
71
72          uint8_t msg[] = { MAPPED_GOAL, current_position.x,
    current_position.y};
73          i2c_write(COMMUNICATION_UNIT, msg, sizeof(msg));
74          _delay_ms(10);
75
76          goal_position = current_position;
77          next_state = return_state;
78
79          return;
80      }
81
82      /* Or we might end up in an intersection... */
83      if (wall_front)
84      {
85          if (!wall_left)
86              rotate_left_90();
87          else if (!wall_right)
88              rotate_right_90();
89          else
90          {
91              set_desired_engine_speed(0);
92
93              flood_fill_to_unmapped();
94              if (current_route[0] == ROUTE_END)
95              {
96                  /* No route found */
97                  next_state = end_state;
98              }
99              else
```

```
100             {
101                 next_state = navigate_state;
102                 follow_up_state = search_state;
103             }
104 
105             return;
106         }
107     }
108 
109     /* Start searching */
110     int16_t last_distance_travelled = distance_travelled;
111 
112     for (;;)
113     {
114         int16_t sqaure_diff = distance_travelled -
    last_distance_travelled;
115 
116         if (intersection)
117             set_desired_engine_speed(INTERSECTION_SPEED);
118         else
119             set_desired_engine_speed(MAPPING_SPEED);
120 
121         _delay_ms(1);
122 
123         if (sqaure_diff > 400 || (sqaure_diff > 100 &&
    front_wall_distance < 230))
124         {
125             PORTD ^= (1<<DEBUG_LED_GREEN)|(1<<DEBUG_LED_RED);
126             is_sending = 0;
127 
128             /* Update map stuff */
129             move_map_position_forward();
130             set_walls(wall_front, wall_left, wall_right);
131 
132             /* Decide what to do */
133             if (tape_square)
134             {
135                 set_desired_engine_speed(0);
136                 set_current_sqaure_not_wall();
137 
138                 uint8_t msg[] = { MAPPED_GOAL, current_position.x,
    current_position.y};
139                 i2c_write(COMMUNICATION_UNIT, msg, sizeof(msg));
140                 _delay_ms(10);
141 
142                 goal_position = current_position;
143                 next_state = return_state;
144 
145                 return;
146             }
147             else if (current_sqaure_not_wall())
148             {
149                 set_desired_engine_speed(0);
150 
151                 flood_fill_to_unmapped();
152                 if (current_route[0] == ROUTE_END)
```

```
153                 {
154                     /* No route found */
155                     next_state = end_state;
156                 }
157                 else
158                 {
159                     next_state = navigate_state;
160                     follow_up_state = search_state;
161                 }
162 
163                 return;
164             }
165             else
166             {
167                 set_current_sqaure_not_wall();
168 
169                 if (wall_front)
170                 {
171                     if (!wall_left)
172                         rotate_left_90();
173                     else if (!wall_right)
174                         rotate_right_90();
175                     else
176                     {
177                         set_desired_engine_speed(0);
178 
179                         flood_fill_to_unmapped();
180                         if (current_route[0] == ROUTE_END)
181                         {
182                             /* No route found */
183                             next_state = end_state;
184                         }
185                         else
186                         {
187                             next_state = navigate_state;
188                             follow_up_state = search_state;
189                         }
190 
191                         return;
192                     }
193                 }
194             }
195 
196             /* Reset */
197             last_distance_travelled = distance_travelled;
198         } // end if
199     } // end for
200 }
```

B.2 flood_fill.c

```
/*
 * flood_fill.c
 * Ronny-the-robot/control_unit
 * ----------------------------
 * This file contains the algorithms for finding the shortest paths.
 * ----------------------------
 * Author: L. Ragnarsson, E. Sk ld
 */

#include <avr/io.h>
#include <util/delay.h>

#include "flood_fill.h"
#include "I2C.h"

uint8_t current_wavefront_size;
uint8_t new_wavefront_size;
coordinate wavefront_1[128];
coordinate wavefront_2[128];
coordinate* current_wavefront = &wavefront_1[0];
coordinate* new_wavefront = &wavefront_2[0];

/*
 * Swaps the pointer new_wavefront and current_wavefront.
 */
void swap_wavefronts(coordinate **wavefront1, coordinate **wavefront2)
{
    coordinate *temp = *wavefront1;
    *wavefront1 = *wavefront2;
    *wavefront2 = temp;
}

/*
 * Resets all the values in ff_map.
 */
void reset_flood_fill_values()
{
    for (int i = 0; i < MAP_SIZE; ++i)
        for(int j = 0; j < MAP_SIZE; ++j)
            ff_map[i][j] = FF_DEFAULT;
}

/*
 * Updates the current_rout variable with a route to the destination.
 * Calculates this route from data generated by one of the flood fill algotithms below.
 */
void calculate_route(coordinate destination)
{
    coordinate current_coordinate = destination;
```

```
50      current_route[ff_map[destination.x][destination.y]] = ROUTE_END
    ;
51      for(uint8_t i = ff_map[destination.x][destination.y] - 1; i !=
    255; --i)
52      {
53          /* Run straight ahead if possible */
54          switch(current_route[i+1])
55          {
56              case NORTH:
57                  if(ff_map[current_coordinate.x + 1][
    current_coordinate.y] == i)
58                  {
59                      current_route[i] = NORTH;
60                      ++current_coordinate.x;
61                      continue;
62                  }
63                  break;
64              case EAST:
65                  if(ff_map[current_coordinate.x][current_coordinate.
    y - 1] == i)
66                  {
67                      current_route[i] = EAST;
68                      --current_coordinate.y;
69                      continue;
70                  }
71                  break;
72              case SOUTH:
73                  if(ff_map[current_coordinate.x - 1][
    current_coordinate.y] == i)
74                  {
75                      current_route[i] = SOUTH;
76                      --current_coordinate.x;
77                      continue;
78                  }
79                  break;
80              case WEST:
81                  if(ff_map[current_coordinate.x][current_coordinate.
    y + 1] == i)
82                  {
83                      current_route[i] = WEST;
84                      ++current_coordinate.y;
85                      continue;
86                  }
87                  break;
88              default:
89                  break;
90          }
91
92          if(ff_map[current_coordinate.x + 1][current_coordinate.y]
    == i) // Go NORTH?
93          {
94              current_route[i] = NORTH;
95              ++current_coordinate.x;
96          } else if(ff_map[current_coordinate.x][current_coordinate.y
     - 1] == i) // etc.
97          {
```

```
98              current_route[i] = EAST;
99              --current_coordinate.y;
100         } else if(ff_map[current_coordinate.x - 1][
    current_coordinate.y] == i)
101         {
102             current_route[i] = SOUTH;
103             --current_coordinate.x;
104         } else if(ff_map[current_coordinate.x][current_coordinate.y
    + 1] == i)
105         {
106             current_route[i] = WEST;
107             ++current_coordinate.y;
108         }
109     }
110 }
111
112 /*
113  * Floods the maze on known paths until a destination is reached.
114  */
115 void flood_fill_to_destination(coordinate destination) {
116     reset_flood_fill_values();
117     *current_wavefront = current_position;
118     current_wavefront_size = 1;
119
120     uint8_t distance = 0;
121     while(1)
122     {
123         new_wavefront_size = 0;
124         for(int i = 0; i < current_wavefront_size; ++i)
125         {
126             uint8_t x_coord = (current_wavefront + i)->x;
127             uint8_t y_coord = (current_wavefront + i)->y;
128             ff_map[x_coord][y_coord] = distance;
129             if(x_coord == destination.x && y_coord == destination.y
    )
130             {
131                 calculate_route(destination);
132                 return;
133             }
134
135             if(map[x_coord - 1][y_coord] == NOT_WALL && ff_map[
    x_coord - 1][y_coord] == FF_DEFAULT) // NORTH
136             {
137                 *(new_wavefront + new_wavefront_size) = (coordinate
    ){x_coord - 1, y_coord};
138                 ++new_wavefront_size;
139             }
140             if(map[x_coord][y_coord + 1] == NOT_WALL && ff_map[
    x_coord][y_coord + 1] == FF_DEFAULT) // EAST
141             {
142                 *(new_wavefront + new_wavefront_size) = (coordinate
    ){x_coord, y_coord + 1};
143                 ++new_wavefront_size;
144             }
145             if(map[x_coord + 1][y_coord] == NOT_WALL && ff_map[
    x_coord + 1][y_coord] == FF_DEFAULT) // SOUTH
```

```
146             {
147                 *(new_wavefront + new_wavefront_size) = (coordinate
    ){x_coord + 1, y_coord};
148                 ++new_wavefront_size;
149             }
150             if(map[x_coord][y_coord - 1] == NOT_WALL && ff_map[
    x_coord][y_coord - 1] == FF_DEFAULT) // WEST
151             {
152                 *(new_wavefront + new_wavefront_size) = (coordinate
    ){x_coord, y_coord - 1};
153                 ++new_wavefront_size;
154             }
155         }
156         /* Route to destination not found */
157         if(new_wavefront_size == 0)
158         {
159             current_route[0] = ROUTE_END;
160             return;
161         }
162         ++distance;
163         current_wavefront_size = new_wavefront_size;
164         swap_wavefronts(&current_wavefront, &new_wavefront);
165     }
166 }
```

B.3 sensor_unit.c

```
1  /*
2   * sensor_unit.c
3   * Ronny-the-robot/sensor_unit
4   * ------------------------------
5   * This file contains all functionality available in the sensor
      unit,
6   * including reading sensor values as well as sending and
7   * receiving messages to and from other modules.
8   * ------------------------------
9   * Author: J. Otterholm
10  */
11
12 #include "sensor_unit.h"
13 #include "I2C.h"
14
15 int main(void)
16 {
17     init_ir();
18     init_reflectance();
19     init_wheel_encoder();
20     i2c_init(BITRATE_20MHZ, PRESCALER_20MHZ, SENSOR_UNIT);
21     DDRD = (0<<DDD0);
22
23     sei();
24
25     while(1) { }
26 }
27
28 /*
```

```
 * Send distance travelled since last time this message was sent.
 */
uint8_t send_odometry_readings()
{
    static uint8_t distance_trunc = 0;
    int16_t scaled_distance = (encoder_left + encoder_right) *
   ENCODER_DISTANCE_SCALE + distance_trunc;
    distance_trunc = (uint8_t)scaled_distance;
    int8_t distance = (int8_t)(scaled_distance>>8);

    static uint8_t rotation_trunc = 0;
    int16_t scaled_rotation = (encoder_left - encoder_right) *
   ENCODER_ROTATION_SCALE + rotation_trunc;
    rotation_trunc = (uint8_t)scaled_rotation;
    int8_t rotation = (int8_t)(scaled_rotation>>8);

    int8_t msg[] = { MOVED_DISTANCE_AND_ANGLE, distance, rotation
   };
    return i2c_write(GENERAL_CALL, msg, sizeof(msg));
}
```

B.4 I2C.c

```
/*
 * I2C.c
 * Ronny-the-robot/communication_unit
 * ------------------------------
 * This file contains the functions required to use the I2C-bus.
 * ------------------------------
 * Author: F.  stman
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <string.h>

#include "I2C.h"

volatile uint8_t helpaddress;
volatile uint8_t helpdata[16];
volatile uint8_t startpointer;
volatile uint8_t endpointer;

/*
 *  Init I2C.
 */
void i2c_init(uint8_t bitrate, uint8_t prescaler, uint8_t address)
   {
    receiverstart = 0x00;
    receiverstop = 0x00;
    TWBR = bitrate;
    TWSR |= prescaler;
    TWAR = address | (1<<TWGCE);        //Set address and enable
   General Call
    TWCR = RESET;
    is_sending = 0;
```

```
}

/*
 * Write multiple bytes to I2C.
 */
uint8_t i2c_write(uint8_t address, uint8_t* data, uint8_t length) {
    if(!is_sending) {
        helpaddress = address | 0; // Write
        startpointer = 0x00;
        endpointer = 0x00;
        while(endpointer < length) {
            helpdata[endpointer] = data[endpointer];
            endpointer++;
        }
        is_sending = 0x01;
        i2c_start();
        return 1;
    }
    else
        return 0;
}

/*
 * Write one byte to I2C.
 */
uint8_t i2c_write_byte(uint8_t address, uint8_t data) {
    if (!is_sending) {
        helpaddress = address | 0; //Write
        startpointer = 0x00;
        endpointer = 0x01;
        helpdata[0] = data;
        is_sending = 0x01;
        i2c_start();
        return 1;
    }
    else
        return 0;
}
```